

**LAB RECORD**

***Course code:***

***19CSE401 – Compiler Design***

***Submitted by***

CH.EN.U4CSE22134

M.DEVA VIGNAN

**BACHELOR OF TECHNOLOGY**

IN

COMPUTER SCIENCE AND ENGINEERING

AMRITA VISHWA VIDYAPEETHAM AMRITA SCHOOL OF COMPUTING CHENNAI

October – 2025



**AMRITA VISHWA VIDYAPEETHAM AMRITA SCHOOL OF COMPUTING, CHENNAI**

**BONAFIDE CERTIFICATE**

This is to certify that the Lab Record work for 19CSE102 – Computer Programming Subject submitted by **CH.EN.U4CSE22134 M.DEVA VIGNAN** in “**Computer Science and Engineering”** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Computing, Chennai.

Dr. S.Suthir Dr. S Sountharrajan

Assistant Professor (Selection Grade) Asso. Prof.& Program Head

Dept. of CSE Dept. of CSE

|  |  |  |
| --- | --- | --- |
| Experiment No | Title | Page No |
| 1 | **Implementation of Lexical Analyzer Using Lex Tool** | 4-6 |
| 2 | To implement eliminate left recursion and left factoring | 7-10 |
| 3 | To implement LL(1) parsing using C program. | 11-13 |
| 4 | To write a program in YACC for parser generation. | 14-16 |
| 5 | To implement Symbol Table. | 16-20 |
| 6 | To implementation of intermediate code generation. | 21-23 |
| 7 | To implementation of Code Optimization Techniques | 24-26 |
| 8 | To write a program that implements the target code generation | 26-30 |

**Implementation of Lexical Analyzer Using Lex Tool**

**Experiment No. :** 1

**Aim :** To implement Lexical Analyzer Using Lex Tool

**Algorithm :**

1. Open gedit text editor from accessories in applications.
2. Specify the header files to be included inside the declaration part (i.e. between %{ and %}).
3. Define the digits i.e. 0-9 and identifiers a-z and A-Z.
4. Using translation rule, we defined the regular expression for digit, keywords, identifier, operator and header file etc. if it is matched with the given input then store and display it in yytext.
5. Inside procedure main(),use yyin() to point the current file being passed by the lexer.
6. Those specification of a lexical analyzer is prepared by creating a program lexp.l in the LEX language.
7. The Lexp.l program is run through the LEX compiler to produce an equivalent code in C language named Lex.yy.c .
8. The program lex.yy.c consists of a table constructed from the Regular Expressions of Lexp.l, together with standard routines that uses the table to recognize lexemes.
9. Finally, lex.yy.c program is run through the C Compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

**Code :**

1. **analyzer.l**

%{

#include<stdio.h>

int COMMENT=0;

%}

identifier [a-zA-Z][a-zA-Z0-9]\*

%%

#.\* {printf("\n%s is a preprocessor directive", yytext);}

int|float|char|double|while|for|struct|typedef|do|if|break|continue|void|switch|return|else|goto {printf("\n\t%s is a keyword",yytext);}

"/\*" {COMMENT=1;}{printf("\n\t%s is a COMMENT", yytext);}

{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s", yytext);}

\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}

\} {if(!COMMENT)printf("BLOCK ENDS ");}

{identifier}(\[[0-9]\*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}

\".\*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}

\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}

\( ECHO;

= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}

<=|>=|==|>|< {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%

int main(int argc, char \*\*argv)

{

FILE \*file;

file=fopen("var.c","r"); // This program is hardcoded to read "var.c"

if(!file)

{

printf("could not open the file");

exit(0);

}

yyin=file;

yylex();

printf("\n");

return(0);

}

int yywrap()

{

return(1);

}

1. **var.c**

#include<stdio.h>

#include<conio.h>

void main()

{

    int a,b,c;

    a=1;

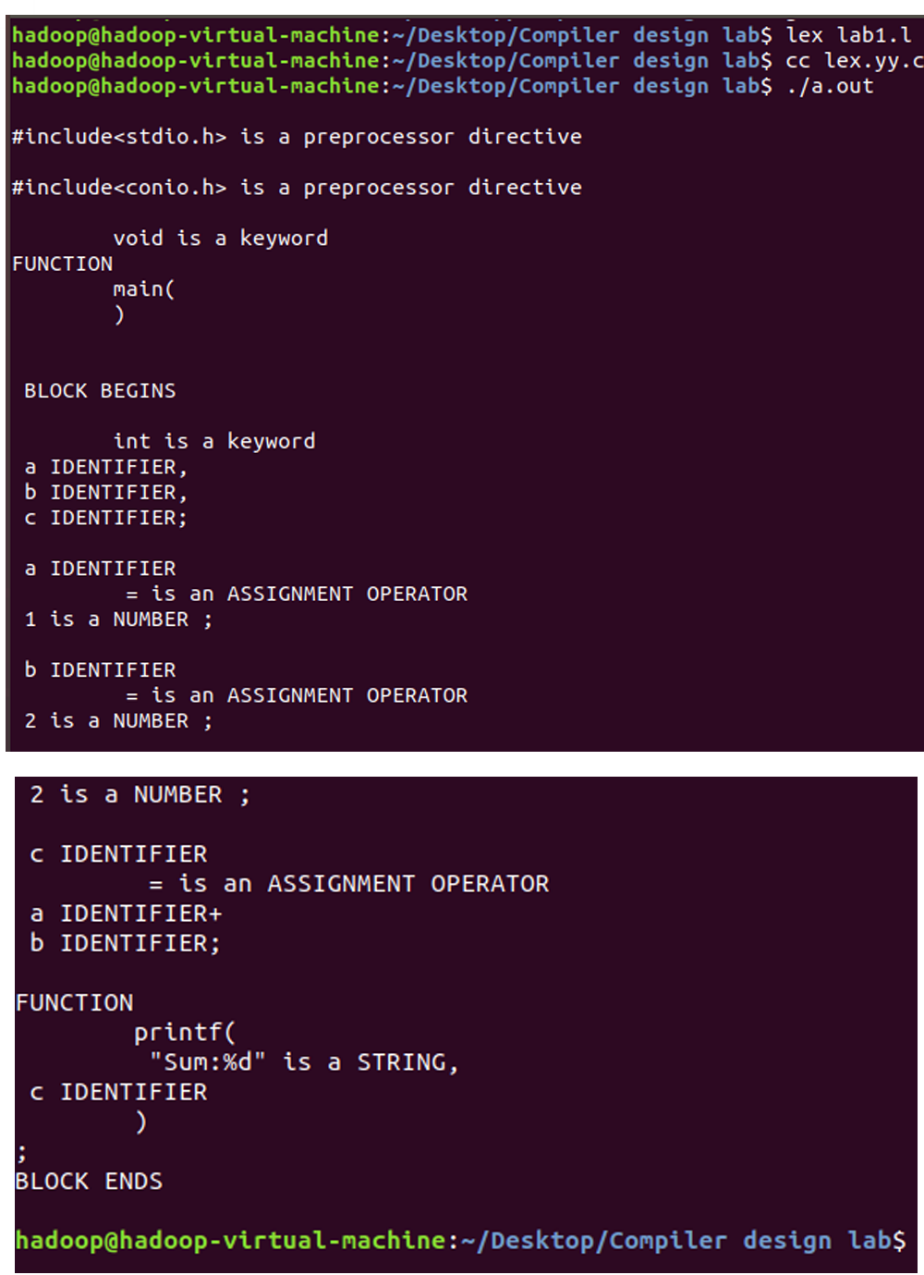
    b=2;

    c=a+b;

    printf("Sum:%d",c);

}

**Output :**



**Result :** The code has been executed and output displayed successfully.

**Program to eliminate left recursion and factoring from the given grammar**

**Experiment No. :** 2

**Aim :** To implement eliminate left recursion and left factoring from the given grammar using C program.

**Algorithm :**

**Left Factoring :**

* Start the processes by getting the grammar and assigning it to the appropriate variables
* Find the common terminal and non-terminal elements and assign them in a separate grammar
* Display the new and modified grammar.

**Code :**

**leftfactoring.c**

#include <stdio.h>

#include <string.h>

int main() {

    char gram[100], part1[50], part2[50], modifiedGram[50], newGram[50];

    int i, j = 0, k = 0, pos = 0;

    printf("Enter Production : A->");

    if (fgets(gram, sizeof(gram), stdin) == NULL) {

        printf("Error reading input.\n");

        return 1;

    }

    gram[strcspn(gram, "\n")] = '\0';

    char \*pipe\_pos = strchr(gram, '|');

    if (pipe\_pos == NULL) {

        printf("No '|' found in production.\n");

        return 1;

    }

    int pipe\_index = pipe\_pos - gram;

    strncpy(part1, gram, pipe\_index);

    part1[pipe\_index] = '\0';

    strcpy(part2, gram + pipe\_index + 1);

    for (i = 0; i < (int)strlen(part1) && i < (int)strlen(part2); i++) {

        if (part1[i] == part2[i]) {

            modifiedGram[k++] = part1[i];

            pos = i + 1;

        } else {

            break;

        }

    }

    modifiedGram[k++] = 'X';  // New non-terminal

    modifiedGram[k] = '\0';

    j = 0;

    for (i = pos; i < (int)strlen(part1); i++) {

        newGram[j++] = part1[i];

    }

    newGram[j++] = '|';

    for (i = pos; i < (int)strlen(part2); i++) {

        newGram[j++] = part2[i];

    }

    newGram[j] = '\0';

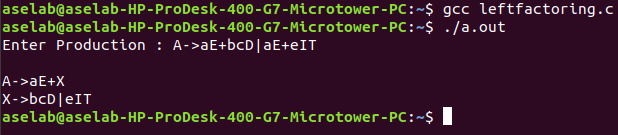
    printf("\nA->%s\n", modifiedGram);

    printf("X->%s\n", newGram);

    return 0;

}

**Output :**



**Left Recursion.c**

**Aim :** To implement left recursion using C.

**Algorithm :**

* Start the processes by getting the grammar and assigning it to the appropriate variables.
* Check if the given grammar has left recursion.
* Identify the alpha and beta elements in the production.
* Print the output according to the formula to remove left recursion

**Code :**

**recursion.c**

#include <stdio.h>

#include <string.h>

#define SIZE 10

int main() {

    char non\_terminal;

    char beta, alpha;

    int num;

    char production[10][SIZE];

    int index = 3;  // Starting index after "->"

    printf("Enter Number of Productions: ");

    scanf("%d", &num);

    printf("Enter the grammar as E->E-A :\n");

    for (int i = 0; i < num; i++) {

        scanf("%s", production[i]);

    }

    for (int i = 0; i < num; i++) {

        printf("\nGRAMMAR ::: %s", production[i]);

        non\_terminal = production[i][0];

        if (non\_terminal == production[i][index]) {

            alpha = production[i][index + 1];

            printf(" is left recursive.\n");

            while (production[i][index] != 0 && production[i][index] != '|') {

                index++;

            }

            if (production[i][index] != 0) {

                beta = production[i][index + 1];

                printf("Grammar without left recursion:\n");

                printf("%c->%c%c\'", non\_terminal, beta, non\_terminal);

                printf("\n%c\'->%c%c\'|E\n", non\_terminal, alpha, non\_terminal);

            } else {

                printf(" can't be reduced\n");

            }

        } else {

            printf(" is not left recursive.\n");

        }

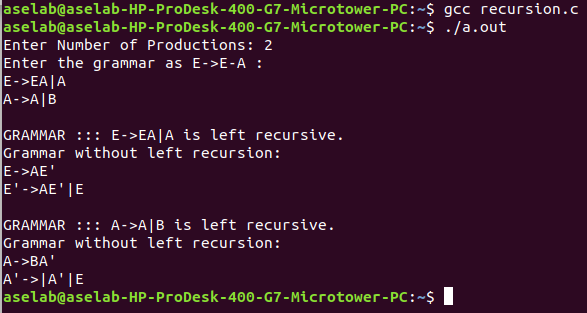
        index = 3;  // Reset index for next production

    }

    return 0;

}

**Output :**



**Result :** The program to implement left factoring and left recursion has been successfully executed.

**Implementation of LL(1) parsing**

**Experiment No. :** 3

**Aim :** To implement LL(1) parsing using C program.

**Algorithm :**

* Read the input string.
* Using predictive parsing table parse the given input using stack.
* If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $.

**Code :**

**ll.c**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char s[50], stack[50];

int main() {

    char m[5][6][5] = {

        {"tb",   "",     "",     "tb",   "",     ""},    // E

        {"",     "+tb",  "",     "",     "n",    "n"},   // B

        {"fc",   "",     "",     "fc",   "",     ""},    // T

        {"",     "n",    "\*fc",  "n",    "n",    "n"},   // C

        {"i",    "",     "",     "(e)",  "",     ""}     // F

    };

    int size[5][6] = {

        {2,0,0,2,0,0},

        {0,3,0,0,1,1},

        {2,0,0,2,0,0},

        {0,1,3,1,1,1},

        {1,0,0,3,0,0}

    };

    int i = 1, j = 0, k, n, str1 = 0, str2 = 0;

    printf("\nEnter the input string (only i,+,\*,(,) allowed): ");

    scanf("%s", s);

    strcat(s, "$");

    n = strlen(s);

    stack[0] = '$';

    stack[1] = 'e';

    printf("\nStack\tInput\n");

    printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

    while (stack[i] != '$' && s[j] != '$') {

        if (stack[i] == s[j]) {  // Terminal match

            i--;

            j++;

        }

        switch (stack[i]) {

            case 'e': str1 = 0; break;

            case 'b': str1 = 1; break;

            case 't': str1 = 2; break;

            case 'c': str1 = 3; break;

            case 'f': str1 = 4; break;

            default: str1 = -1; break;

        }

        switch (s[j]) {

            case 'i': str2 = 0; break;

            case '+': str2 = 1; break;

            case '\*': str2 = 2; break;

            case '(': str2 = 3; break;

            case ')': str2 = 4; break;

            case '$': str2 = 5; break;

            default: str2 = -1; break;

        }

        if (str1 == -1 || str2 == -1 || m[str1][str2][0] == '\0') {

            printf("\nERROR: Invalid string\n");

            exit(0);

        } else if (m[str1][str2][0] == 'n') {  // ε-production

            i--;

        } else {  // Expand production

            i--; // Pop non-terminal

            for (k = size[str1][str2] - 1; k >= 0; k--) {

                stack[++i] = m[str1][str2][k];

            }

        }

        for (k = 0; k <= i; k++) printf("%c", stack[k]);

        printf("\t");

        for (k = j; k < n; k++) printf("%c", s[k]);

        printf("\n");

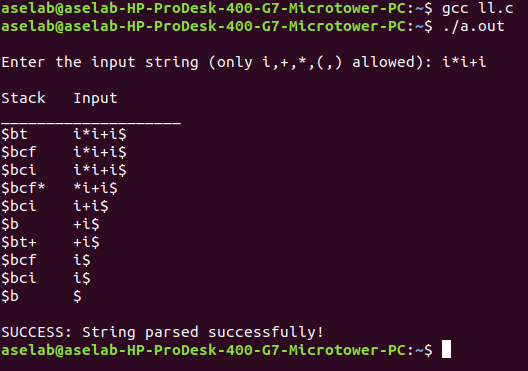
    }

        printf("\nSUCCESS: String parsed successfully!\n");

    return 0;

}

**Output :**



**Result :** Thus, the program to implement LL(1) has been successfully executed.

**Parser Generation using YACC**

**Experiment No. :** 4

**Aim :** To write a program in YACC for parser generation.

**Algorithm :**

1. Get the input from the user and Parse it token by token.
2. First identify the valid inputs that can be given for a program.
3. Define the precedence and the associativity of various operators like +,-,/,\* etc.
4. Write codes for saving the answer into memory and displaying the result on the screen.
5. Write codes for performing various arithmetic operations.
6. Display the possible Error message that can be associated with this calculation.
7. Display the output on the screen else display the error message on the screen

**Code :**

**calc.y**

%{

#include <stdio.h>

#include <ctype.h>

int yylex(void);

void yyerror(const char \*s);

#define YYSTYPE double

%}

%token NUMBER

%left '+' '-'

%left '\*' '/'

%right UMINUS

%%

lines:

      lines expr '\n'   { printf("Result: %g\n", $2); }

    | lines '\n'

    | /\* empty \*/

    ;

expr:

      expr '+' expr     { $$ = $1 + $3; }

    | expr '-' expr     { $$ = $1 - $3; }

    | expr '\*' expr     { $$ = $1 \* $3; }

    | expr '/' expr     {

                            if ($3 == 0) {

                                yyerror("Division by zero");

                                $$ = 0;

                            } else {

                                $$ = $1 / $3;

                            }

                         }

    | '(' expr ')'      { $$ = $2; }

    | '-' expr %prec UMINUS { $$ = -$2; }

    | NUMBER

    ;

%%

// Simple lexer

int yylex(void) {

    int c;

    // Skip whitespace

    while ((c = getchar()) == ' ' || c == '\t');

    // Handle numbers

    if (c == '.' || isdigit(c)) {

        ungetc(c, stdin);

        scanf("%lf", &yylval);

        return NUMBER;

    }

    return c;

}

// Error handling

void yyerror(const char \*s) {

    fprintf(stderr, "Error: %s\n", s);

}

int yywrap() {

    return 1;

}

int main() {

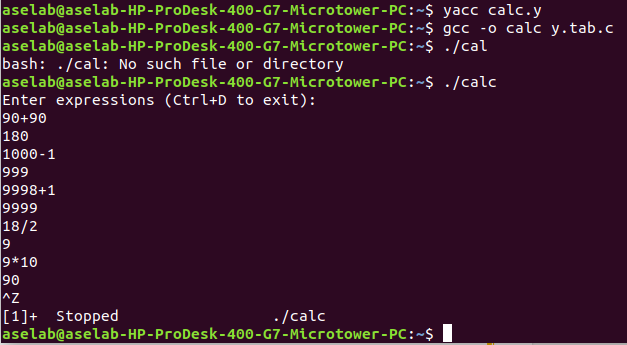
    printf("Enter expressions (CTRL+D to quit):\n");

    yyparse();

    return 0;

}

**Output :**



**Result :** Thus the program in YACC for parser generation has been executed successfully.

**Implementation of Symbol Table**

**Experiment No. :** 5

**Aim :** To implement Symbol Table.

**Algorithm :**

1. Start the Program.
2. Get the input from the user with the terminating symbol ‘$’.
3. Allocate memory for the variable by dynamic memory allocation function.
4. If the next character of the symbol is an operator then only the memory is allocated.
5. While reading, the input symbol is inserted into symbol table along with its memory address.
6. The steps are repeated till “$” is reached.
7. To reach a variable, enter the variable to the searched and symbol table has been.
8. Checked for corresponding variable, the variable along its address is displayed as result.
9. Stop the program

**Code :**

**symbol\_table.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

void main() {

    int x = 0, n, i = 0, j = 0;

    void \*mypointer, \*symbol\_address[100];

    char ch, search\_char, input\_expr[100], symbol\_list[100], c;

    printf("Input the expression ending with $ sign:\n");

    // Read characters until '$'

    while ((c = getchar()) != '$') {

        input\_expr[i++] = c;

    }

    n = i - 1;

    // Display given expression

    printf("\nGiven Expression: ");

    for (i = 0; i <= n; i++) {

        printf("%c", input\_expr[i]);

    }

    // Display symbol table

    printf("\n\nSymbol Table\n");

    printf("Symbol\tAddress\t\tType\n");

    while (j <= n) {

        c = input\_expr[j];

        // If it's an alphabet, it's an identifier

        if (isalpha(c)) {

            mypointer = malloc(sizeof(char)); // allocate 1 byte

            symbol\_address[x] = mypointer;

            symbol\_list[x] = c;

            printf("%c\t%p\tidentifier\n", c, mypointer);

            x++;

        }

        // If it's an operator

        else if (c == '+' || c == '-' || c == '\*' || c == '=') {

            mypointer = malloc(sizeof(char)); // allocate 1 byte

            symbol\_address[x] = mypointer;

            symbol\_list[x] = c;

            printf("%c\t%p\toperator\n", c, mypointer);

            x++;

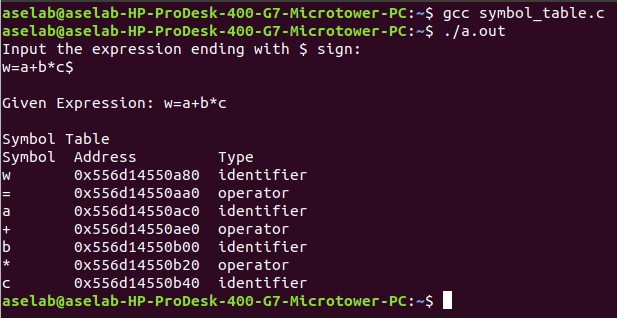
        }

        j++;

    }

}

**Output :**



**Result :** Thus the program to implement symbol table has been executed successfully.

**Implementation of Intermediate Code Generation**

**Experiment No. :** 6

**Aim :** To implementation of intermediate code generation.

**Algorithm :**

1. Take the parse tree tokens from the syntax analyser.
2. Generate intermediate code using temp variable
3. Assign the final temp variable to initial variable

**Code :**

**icg.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int i = 1, j = 0, no = 0, tmpch = 'Z';

char str[100], left[15], right[15];

struct exp {

    int pos;

    char op;

} k[15];

void findopr();

void explore();

void fleft(int);

void fright(int);

void main() {

    printf("\t\tINTERMEDIATE CODE GENERATION\n\n");

    printf("Enter the Expression (no spaces, like a=b+c\*d): ");

    scanf("%s", str);

    printf("\nIntermediate Code:\n");

    findopr();

    explore();

}

void findopr() {

    for (i = 0; str[i] != '\0'; i++) {

        if (str[i] == '=') {

            k[j].pos = i;

            k[j++].op = '=';

        }

    }

    for (i = 0; str[i] != '\0'; i++) {

        if (str[i] == '/' || str[i] == '\*') {

            k[j].pos = i;

            k[j++].op = str[i];

        }

    }

    for (i = 0; str[i] != '\0'; i++) {

        if (str[i] == '+' || str[i] == '-') {

            k[j].pos = i;

            k[j++].op = str[i];

        }

    }

}

void explore() {

    i = 1;

    while (k[i].op != '\0') {

        fleft(k[i].pos);

        fright(k[i].pos);

        str[k[i].pos] = tmpch;

        printf("\t%c := %s %c %s\n", tmpch, left, k[i].op, right);

        tmpch--;

        i++;

    }

    fright(-1);

    if (no == 0) {

        fleft(strlen(str));

        printf("\t%s := %s\n", right, left);

        exit(0);

    }

    printf("\t%s := %c\n", right, str[k[--i].pos]);

}

void fleft(int x) {

    int w = 0, flag = 0;

    x--;

    while (x != -1 && str[x] != '+' && str[x] != '-' && str[x] != '\*' &&

           str[x] != '/' && str[x] != '=' && str[x] != '\0') {

        if (str[x] != '$' && flag == 0) {

            left[w++] = str[x];

            left[w] = '\0';

            str[x] = '$';

            flag = 1;

        }

        x--;

    }

}

void fright(int x) {

    int w = 0, flag = 0;

    x++;

    while (x != -1 && str[x] != '+' && str[x] != '-' && str[x] != '\*' &&

           str[x] != '/' && str[x] != '=' && str[x] != '\0') {

        if (str[x] != '$' && flag == 0) {

            right[w++] = str[x];

            right[w] = '\0';

            str[x] = '$';

            flag = 1;

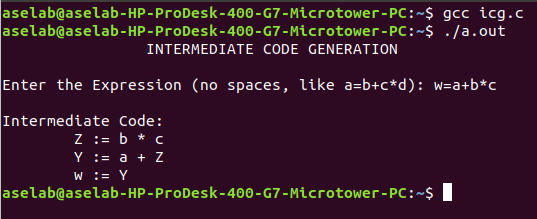
        }

        x++;

    }

}

**Output :**



**Result :** Thus, the program to implement intermediate code generation has been executed successfully.

**Implementation of Code Optimization Techniques**

**Experiment No. :** 7

**Aim :** To implementation of Code Optimization Techniques

**Algorithm :**

1. Start the program

2. Declare the variables and functions.

3. Enter the expression and state it in the variable a, b, c. 24

4. Calculate the variables b & c with ‘temp’ and store it in f1 and f2.

5. If(f1=null && f2=null) then expression could not be optimized.

6. Print the results.

7. Stop the program.

**Code :**

**code\_optimization.c**

#include <stdio.h>

#include <string.h>

struct op {

    char l;       // Left side of assignment

    char r[20];   // Right side expression

} op[10], pr[10];

void main() {

    int i, j, k, m, n, z = 0, a, q;

    char temp, t;

    char \*p, \*l, \*tem;

    printf("Enter the Number of Expressions: ");

    scanf("%d", &n);

    for (i = 0; i < n; i++) {

        printf("Left: ");

        scanf(" %c", &op[i].l);

        printf("Right: ");

        scanf(" %s", op[i].r);

    }

    printf("\nIntermediate Code:\n");

    for (i = 0; i < n; i++) {

        printf("%c = %s\n", op[i].l, op[i].r);

    }

    // Dead Code Elimination

    for (i = 0; i < n - 1; i++) {

        temp = op[i].l;

        for (j = 0; j < n; j++) {

            p = strchr(op[j].r, temp);

            if (p) {

                pr[z].l = op[i].l;

                strcpy(pr[z].r, op[i].r);

                z++;

                break;

            }

        }

    }

    // Always keep the last instruction

    pr[z].l = op[n - 1].l;

    strcpy(pr[z].r, op[n - 1].r);

    z++;

    printf("\nAfter Dead Code Elimination:\n");

    for (k = 0; k < z; k++) {

        printf("%c = %s\n", pr[k].l, pr[k].r);

    }

    // Common Subexpression Elimination

    for (m = 0; m < z; m++) {

        tem = pr[m].r;

        for (j = m + 1; j < z; j++) {

            p = strstr(tem, pr[j].r);

            if (p) {

                t = pr[j].l;

                pr[j].l = pr[m].l;

                for (i = 0; i < z; i++) {

                    l = strchr(pr[i].r, t);

                    if (l) {

                        a = l - pr[i].r;

                        pr[i].r[a] = pr[m].l;

                    }

                }

            }

        }

    }

    printf("\nAfter Common Subexpression Elimination:\n");

    for (i = 0; i < z; i++) {

        printf("%c = %s\n", pr[i].l, pr[i].r);

    }

    // Remove duplicates

    for (i = 0; i < z; i++) {

        for (j = i + 1; j < z; j++) {

            q = strcmp(pr[i].r, pr[j].r);

            if ((pr[i].l == pr[j].l) && !q) {

                pr[i].l = '\0'; // Mark for deletion

            }

        }

    }

    printf("\nOptimized Code:\n");

    for (i = 0; i < z; i++) {

        if (pr[i].l != '\0') {

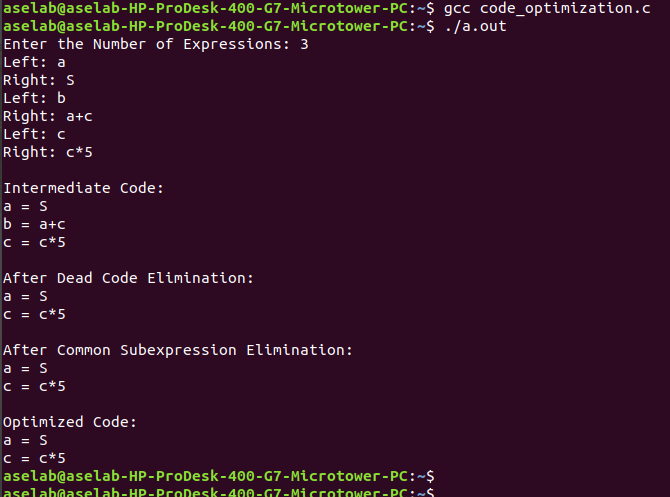
            printf("%c = %s\n", pr[i].l, pr[i].r);

        }

    }

}

**Output :**



**Result :** Thus, the program to implement code optimization has been executed successfully.

**Implementation of Symbol Table**

**Experiment No. :** 8

**Aim :** To write a program that implements the target code generation

**Algorithm :**

1. Read input string

2. Consider each input string and convert it to machine code instructions using switch case

3. Load the input variables into new variables as operands and display them using “load”

4. With the help of arithmetic operation, we will display arithmetic operations like add, sub, div, mul for the respective operations in switch case

5. Generate 3 address code for each input variable.

6. If ‘=‘ is seen as arithmetic operation, then store the result in a variable and display it with “store”.

7. Repeat this for each line in the input string.

8. Display the output which gives a transformed input string of assembly language code.

**Code :**

1. **target.c**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int label[20];

int no=0;

int check\_label(int k)

{

    int i;

    for(i=0;i<no;i++)

    {

        if(k==label[i]) return 1;

    }

    return 0;

}

int main()

{

    FILE \*fp1,\*fp2;

    char fname[10],op[10],ch;

    char operand1[8],operand2[8],result[8];

    int i=0,j=0;

    printf("\n Enter filename of the intermediate code: ");

    scanf("%s",fname);

    fp1=fopen(fname,"r");

    fp2=fopen("target.txt","w");

    if(fp1==NULL || fp2==NULL)

    {

        printf("\n Error opening the file");

        exit(0);

    }

    while(!feof(fp1))

    {

        fprintf(fp2,"\n");

        fscanf(fp1,"%s",op);

        i++;

        if(check\_label(i))

            fprintf(fp2,"\nlabel#%d",i);

        if(strcmp(op,"print")==0)

        {

            fscanf(fp1,"%s",result);

            fprintf(fp2,"\n\t OUT %s",result);

        }

        if(strcmp(op,"goto")==0)

        {

            fscanf(fp1,"%s %s",operand1,operand2);

            fprintf(fp2,"\n\t JMP %s,label#%s",operand1,operand2);

            label[no++]=atoi(operand2);

        }

        if(strcmp(op,"[]=")==0)

        {

            fscanf(fp1,"%s %s %s",operand1,operand2,result);

            fprintf(fp2,"\n\t STORE %s[%s],%s",operand1,operand2,result);

        }

        if(strcmp(op,"uminus")==0)

        {

            fscanf(fp1,"%s %s",operand1,result);

            fprintf(fp2,"\n\t LOAD -%s,R1",operand1);

            fprintf(fp2,"\n\t STORE R1,%s",result);

        }

        switch(op[0])

        {

            case '\*': fscanf(fp1,"%s %s %s",operand1,operand2,result);

                      fprintf(fp2,"\n \t LOAD %s,R0",operand1);

                      fprintf(fp2,"\n \t LOAD %s,R1",operand2);

                      fprintf(fp2,"\n \t MUL R1,R0");

                      fprintf(fp2,"\n \t STORE R0,%s",result);

                      break;

            case '+': fscanf(fp1,"%s %s %s",operand1,operand2,result);

                      fprintf(fp2,"\n \t LOAD %s,R0",operand1);

                      fprintf(fp2,"\n \t LOAD %s,R1",operand2);

                      fprintf(fp2,"\n \t ADD R1,R0");

                      fprintf(fp2,"\n \t STORE R0,%s",result);

                      break;

            case '-': fscanf(fp1,"%s %s %s",operand1,operand2,result);

                      fprintf(fp2,"\n\t LOAD %s,R0",operand1);

                      fprintf(fp2,"\n \t LOAD %s,R1",operand2);

                      fprintf(fp2,"\n \t SUB R1,R0");

                      fprintf(fp2,"\n \t STORE R0,%s",result);

                      break;

            case '/': fscanf(fp1,"%s %s %s",operand1,operand2,result);

                      fprintf(fp2,"\n \t LOAD %s,R0",operand1);

                      fprintf(fp2,"\n \t LOAD %s,R1",operand2);

                      fprintf(fp2,"\n \t DIV R1,R0");

                      fprintf(fp2,"\n \t STORE R0,%s",result);

                      break;

            case '%': fscanf(fp1,"%s %s %s",operand1,operand2,result);

                      fprintf(fp2,"\n \t LOAD %s,R0",operand1);

                      fprintf(fp2,"\n \t LOAD %s,R1",operand2);

                      fprintf(fp2,"\n \t DIV R1,R0");

                      fprintf(fp2,"\n \t STORE R0,%s",result);

                      break;

            case '=': fscanf(fp1,"%s %s",operand1,result);

                      fprintf(fp2,"\n\t STORE %s %s",operand1,result);

                      break;

            case '>': j++;

                      fscanf(fp1,"%s %s %s",operand1,operand2,result);

                      fprintf(fp2,"\n \t LOAD %s,R0",operand1);

                      fprintf(fp2,"\n\t JGT %s,label#%s",operand2,result);

                      label[no++]=atoi(result);

                      break;

            case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);

                      fprintf(fp2,"\n \t LOAD %s,R0",operand1);

                      fprintf(fp2,"\n\t JLT %s,label#%s",operand2,result);

                      label[no++]=atoi(result);

                      break;

        }

    }

    fclose(fp2);

    fclose(fp1);

    fp2=fopen("target.txt","r");

    if(fp2==NULL)

    {

        printf("Error opening the file\n");

        exit(0);

    }

    do

    {

        ch=fgetc(fp2);

        printf("%c",ch);

    }while(ch!=EOF);

    fclose(fp2);

    return 0;

}

1. **Input.txt**

/t3 t2 t2

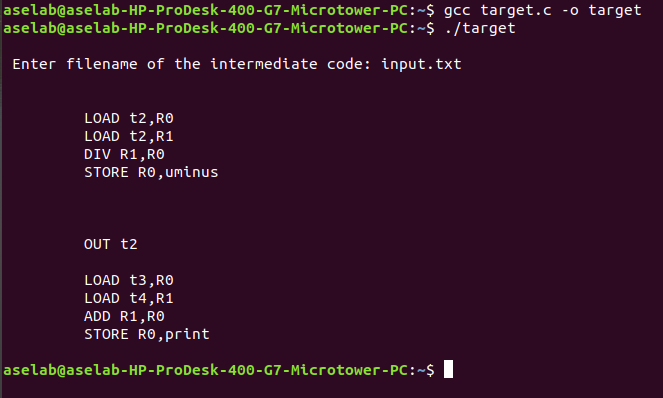
uminus t2 t2

print t2

+t1 t3 t4

print t4

**Output :**



**Result :** Thus, the program to implement target code generation has been successfully executed.