GOVERNMENT COLLEGE OF ENGINEERING

TIRUNELVELI – 627 007.

20 - 20

Register No.

CERTIFICATE

This is a Bonafide record of work done by ……...

…… Government College of Engineering, Tirunelveli during the year 20- 20

STATION: TIRUNELVELI – 7.

DATE:

Staff-in-Charge **Head of the Department**

Submitted for the Anna University Practical Examination held at Government College of Engineering, Tirunelveli on ……

……

Internal Examiner **External Examiner**

TABLE OF CONTENTS

EX NO: 1 DATE:	IMPLEMENTATION OF SYMBOL TABLE

AIM:

To write a c program for implementation of symbol table

ALGORITHM:

Step 1: Start and Display Menu: Show options to insert, print, search, delete, or exit.

Step 2: Take User Input: Read the user's choice and validate it.

Step 3: Insert Symbols:

3.1: Read an expression ending with $.

3.2: Tokenize it into identifiers.

3.3: Add valid identifiers to the symbol table using insertSymbol.

Step 4: Print Symbol Table: Display all symbols, their memory addresses, and types.

Step 5: Search or Delete Symbol:

5.1: Search: Find and display the symbol's address if it exists.

5.2: Delete: Free memory, remove the symbol, and shift the table entries.

Step 6: Exit Program: Free all allocated memory and terminate.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX_SYMBOLS 10 // Adjust size as needed
```

```
#define MAX_LENGTH 100 // Max length of identifiers
typedef struct {
    char symbol[MAX_LENGTH];
    void *addr;
} Symbol;
Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;
void insertSymbol(const char *sym) {
    if (symbolCount>= MAX_SYMBOLS) {
        printf("Symbol table is full.\n");
        return;
    }
    // Allocate memory for the symbol
    void *p = malloc(sizeof(char));
    if (p == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    // Insert symbol and address
    strncpy(symbolTable[symbolCount].symbol, sym, MAX_LENGTH - 1);
    symbolTable[symbolCount].symbol[MAX_LENGTH - 1] = '\0'; // Ensure null-termination
    symbolTable[symbolCount].addr = p;
    symbolCount++;
```

```
    printf("Symbol '%s' inserted.\n", sym);
}
void searchSymbol(const char *sym) {
    int found = 0;
    for (int i = 0; i<symbolCount; i++) {
        if (strcmp(symbolTable[i].symbol, sym) == 0) {
            printf("Symbol Found\n");
            printf("%s @address %p\n", sym, symbolTable[i].addr);
            found = 1;
            break;
        }
    }
    if (!found) {
        printf("Symbol Not Found\n");
    }
}
void deleteSymbol(const char *sym) {
    int i;
    for (i = 0; i<symbolCount; i++) {
        if (strcmp(symbolTable[i].symbol, sym) == 0) {
            free(symbolTable[i].addr); // Free allocated memory
            // Shift symbols down
            for (int j = i; j <symbolCount - 1; j++) {
```

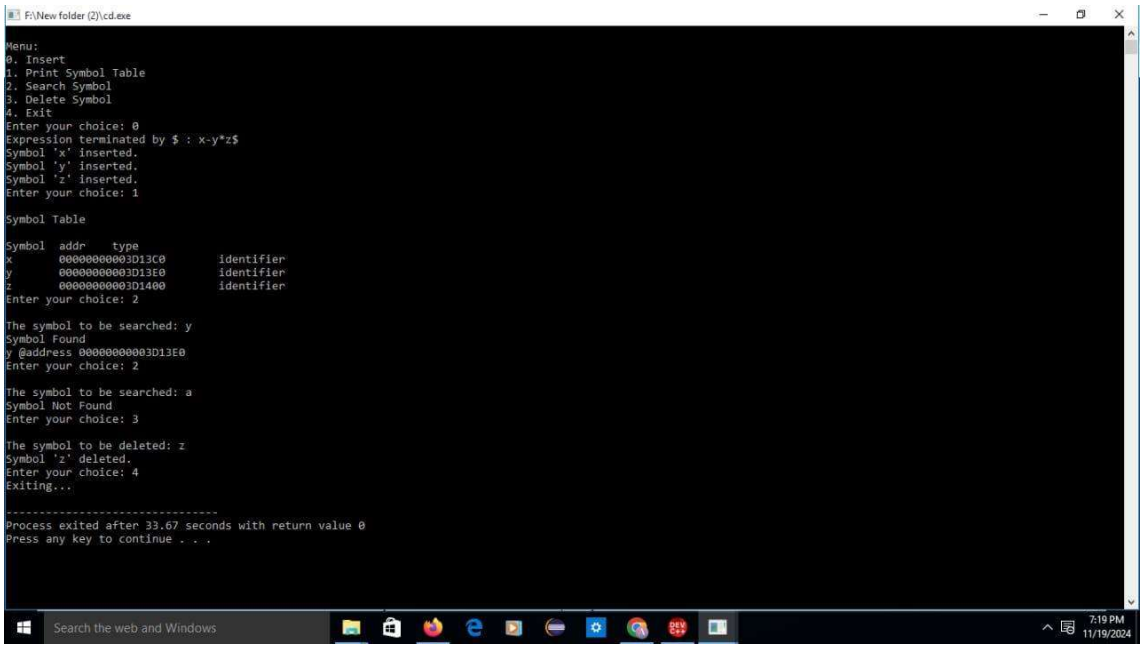
```
            symbolTable[j] = symbolTable[j + 1];
        }
        symbolCount--;
        printf("Symbol '%s' deleted.\n", sym);
        return;
      }
    }
    printf("Symbol '%s' not found for deletion.\n", sym);
}
void printSymbolTable() {
    printf("\nSymbol Table\n");
    printf("\nSymbol\taddr\ttype\n");
    for (int i = 0; i<symbolCount; i++) {
       printf("%s\t%p\tidentifier\n", symbolTable[i].symbol, symbolTable[i].addr);
    }
}
int main() {
    char input[200]; // Increased buffer size for longer expressions
    char word[MAX_LENGTH];
    int choice;
      printf("\nMenu:\n");
      printf("0. Insert\n1. Print Symbol Table\n");
      printf("2. Search Symbol\n");
```

```
  printf("3. Delete Symbol\n");
  printf("4. Exit\n");
while (1) {
    printf("Enter your choice: ");
    scanf("%d", &choice);
    getchar(); // Clear newline from buffer
    switch (choice) {
        case 0:
            printf("Expression terminated by $ : ");
            fgets(input, sizeof(input), stdin);
            char *ptr = strchr(input, '$');
            if (ptr) *ptr = '\0'; // Remove the $ character
        // Tokenize the input into words
            char *token = strtok(input, " \t\n+=-*/(){}[],");
            while (token != NULL) {
                if (isalpha(token[0])) {
                    insertSymbol(token);
                }
                token = strtok(NULL, " \t\n+=-*/(){}[],");
            }
            break;
```

```
case 1:
            printSymbolTable();
            break;
        case 2:
            printf("\nThe symbol to be searched: ");
            fgets(word, sizeof(word), stdin);
            word[strcspn(word, "\n")] = '\0'; // Remove newline character
            searchSymbol(word);
            break;
        case 3:
            printf("\nThe symbol to be deleted: ");
            fgets(word, sizeof(word), stdin);
            word[strcspn(word, "\n")] = '\0'; // Remove newline character
            deleteSymbol(word);
            break;
        case 4:
            printf("Exiting...\n");
            // Free all remaining allocated memory
            for (int i = 0; i<symbolCount; i++) {
                free(symbolTable[i].addr);
            }
            exit(0);
```

```
default:
            printf("Invalid choice. Please enter a number between 0 and 4.\n");
        }
    }
    return 0;
}
```

OUTPUT:



RESULT:

Thus the c program for implementation of symbol table was written,executed and the output was verified successfully.

EXNO:2 DATE:	IMPLEMENTATION OF LEXICAL ANALYSER USING LEXTOOL

AIM:

To implement the lexical analyser using lex tool.

ALGORITHM:

Step 1: Start

Step 2: Read C file as input.

Step 3: Generate tokens using yylex() function.

Step 4: Recognize the patterns of token using grammar rules.

4.1: Identify “Constants” using the following rule: [0-9]+ | [0-9]+”.”[0-9]+

4.2: Identify “Identifiers” using the following rule: [a-zA-Z_][a-zA-Z0-9_]*

4.3: Identify “Commands” using the following rule:

“//” [a-zA-Z0-9_\t]+ or “/*”[a-zA-Z0-9_\t\n]+”*/”

4.4: Identify “Operators” using the following rule: [-+*/=%&|^]

Step 5: Display the results and number of tokens.

Step 6: Stop.

PROGRAM:

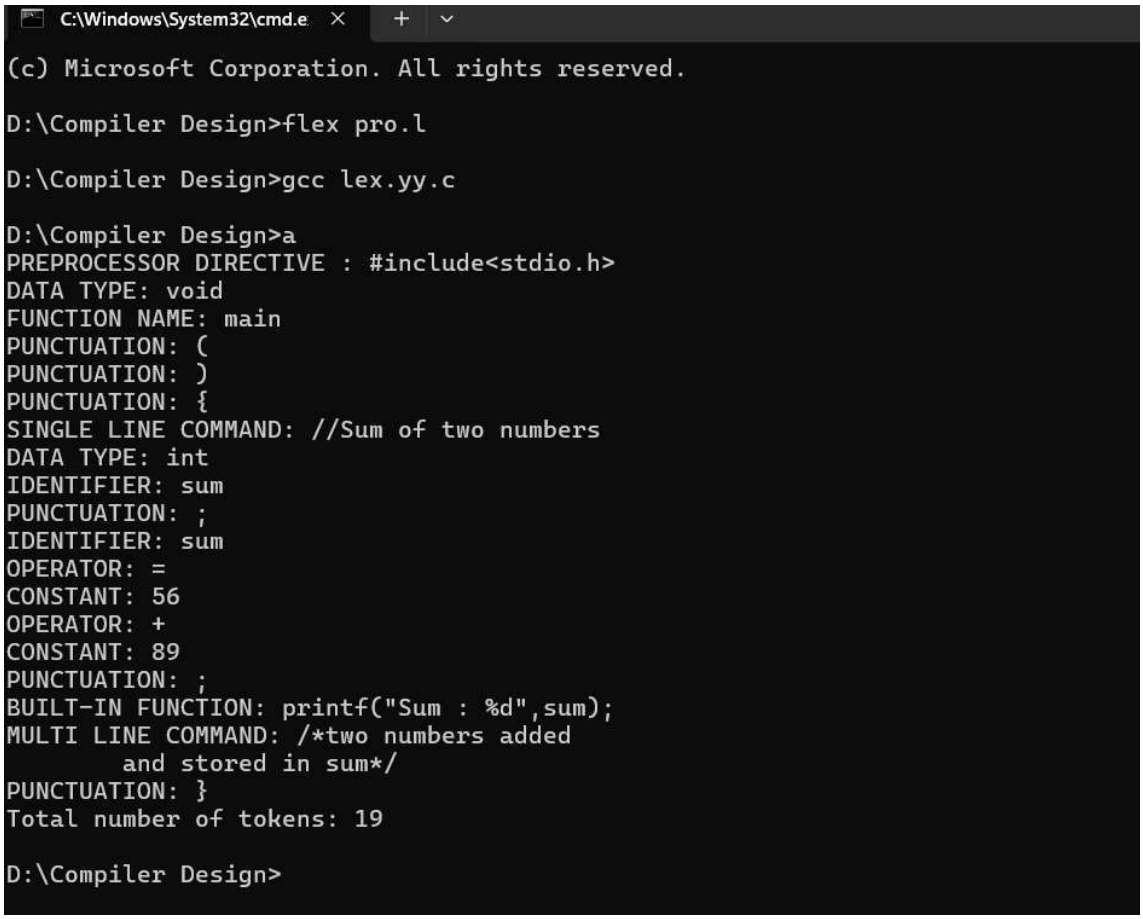
```
%{
#include <stdio.h>
int token_count = 0;
%}
%%
"#"[a-zA-Z_ ]+"<"[a-zA-Z_.]+">" {
printf("PREPROCESSOR DIRECTIVE : %s\n",yytext);
```

```
token_count++;

}

printf"("[^)]*");"|scanf"("[^)]*");" {

printf("BUILT-IN FUNCTION: %s \n",yytext);

token_count++;

}

main {

printf("FUNCTION NAME: %s\n",yytext);

token_count++;

}

int|float|char|double|long|short|void   {

printf("DATA TYPE: %s\n", yytext);

token_count++; }

[a-zA-Z_][a-zA-Z0-9_]*          {

printf("IDENTIFIER: %s\n", yytext);

token_count++; }

"//"[a-zA-Z_ ]+ {

printf("SINGLE LINE COMMAND: %s\n",yytext);

token_count++;

}

"/*"[a-zA-Z_0-9\t\n ]+"*/" {

printf("MULTI LINE COMMAND: %s\n",yytext);

token_count++;

}
```

```
[0-9]+|[0-9]+"."[0-9]+                    {
printf("CONSTANT: %s\n", yytext);
token_count++;
}
[ \t\n]+                    ;
[-+*/=]                     {
printf("OPERATOR: %s\n", yytext);
token_count++;
}
[(){};,.]                   {
printf("PUNCTUATION: %s\n", yytext);
token_count++; }
%%
int main() {
  FILE *file=fopen("sum.c","r");
if(!file){
fprintf(stderr,"couldn't load file");
}
yyin=file;
yylex();
printf("Total number of tokens: %d\n", token_count);
   return 0;
}
```

```
int yywrap() {
    return 1;
}
```

OUTPUT:



RESULT:

Thus, the lexical analyser using lex tool was implemented successfully.

EX NO:3A DATE:	PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION

AIM:

To write a yacc program to check valid arithmetic expression using YACC.

ALGORITHM:

Step1: Start.

Step 2: Read an arithmetic expression as input from the user.

Step 3: Perform lexical analysis using yylex() function:

Step 3.1: Identify identifiers using the rule: [a-zA-Z_][a-zA-Z_0-9]*

Step 3.2: Identify constants using the rule: [0-9]+(\.[0-9]*)?

Step 3.3: Identify operators using the rule: [+/*]

Step 3.4: Identify other characters (e.g., ;, (,)).

Step 4: Pass the recognized tokens to the YACC parser to validate syntax:

Step 4.1: Check the starting structure: start → id '=' s ';'

Step 4.2: Validate sub-expressions:

i. Identifiers, constants, or parenthesized expressions.

ii. Operators and extended sub-expressions.

Step 4.3: Ensure that the expression follows the grammar rules.

Step 5: If the syntax matches the grammar:

i. Print "Valid expression."

ii. Otherwise, print "Invalid expression."

Step 6: Display the results of validation.

Step 7: Stop.

PROGRAM:

LEX PART:

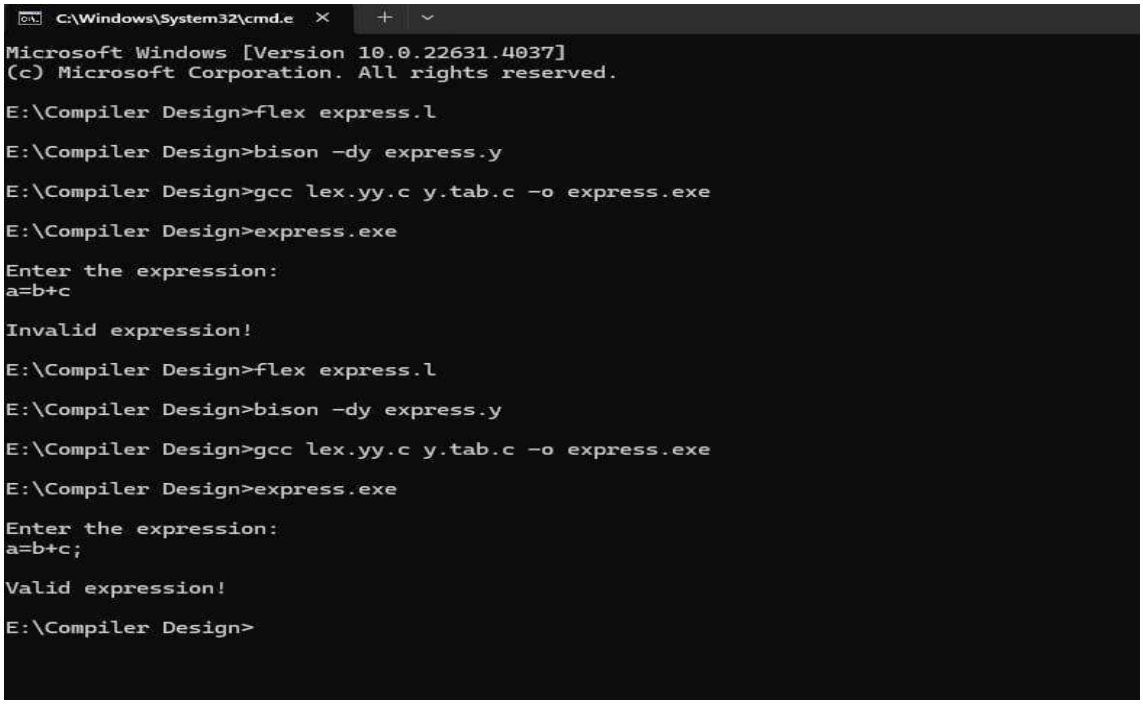
```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)?  return num;
[+/*]              return op;
.                  return yytext[0];
\n                 return 0;
%%
int yywrap()
{
return 1;
}
```

YACC PART

```
%{
#include<stdio.h>
int valid=1;
%}
%token num id op
%%
start : id '=' s ';'
```

```
s : id x
  | num x
  | '-' num x
  | '(' s ')' x
  ;
x : op s
  | '-' s
  |
```

OUTPUT:



RESULT

Thus, the program to recognize a valid arithmetic expression has been executed successfully.

EX NO: 3B DATE:	PROGRAM TO RECOGNIZE A VALID VARIABLE

AIM:

To write a YACC program to check valid variable followed by letter or digits.

ALGORITHM:

Step 1: The program reads the input and identifies whether each character is a letter or a digit.

Step 2: It checks if the input starts with a letter. If it doesn't, it is invalid.

Step 3: After the first letter, the input can have more letters or digits.

Step 4: If the input follows these rules, the program says, "It is an identifier."

Step 5: If the input breaks any rules, the program says, "It is not an identifier."

PROGRAM:

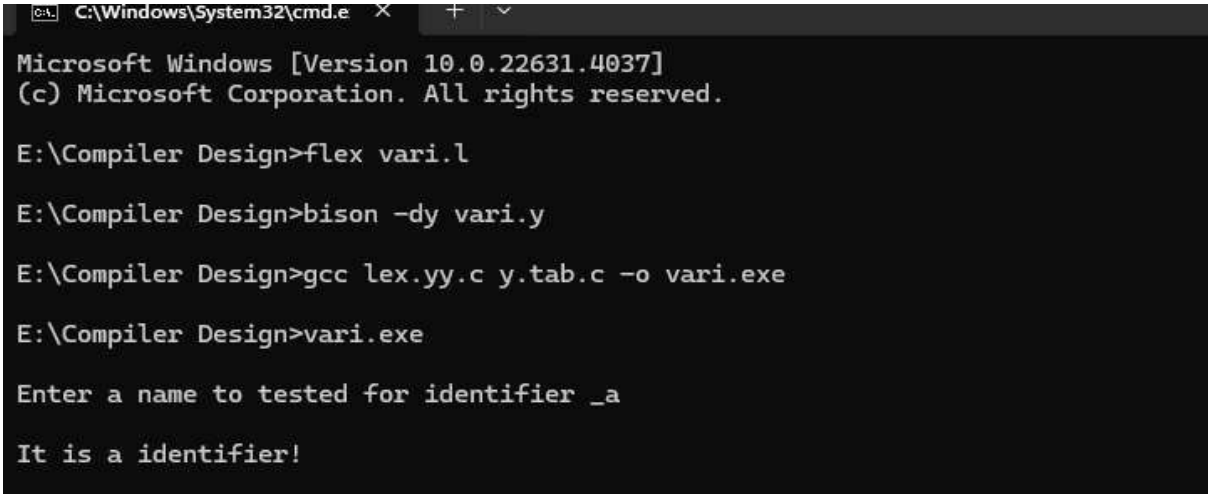
LEX PART:

```
%{
   #include "y.tab.h"
%}
[a-zA-Z_] [a-zA-Z_0-9] * return letter;
[0-9] return digit;
. return yytext[0];
\n return 0;
%%
int yywrap() {
   return 1;
}
```

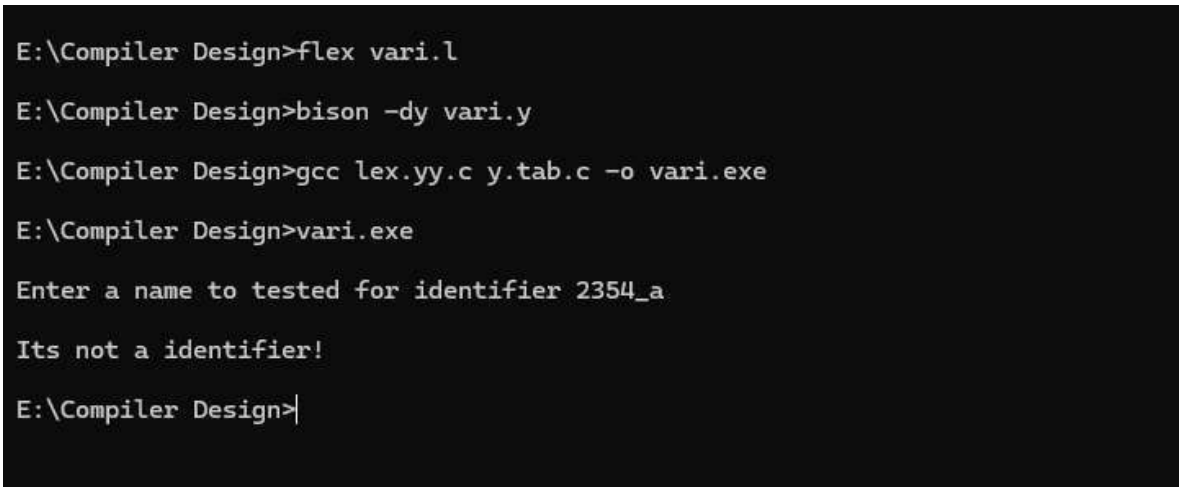
YACC PART:

```
%{
   #include<stdio.h>
   int valid=1;
%}
%token digit letter
%%
start : letter s
s : letter s
     | digit s
     |
     ;
%%
int yyerror() {
   printf("\nIt is not an identifier!\n");
   valid = 0;
   return 0;
}
int main() {
   printf("\nEnter a name to test for identifier: ");
   yyparse();
   if (valid) {
      printf("\nIt is an identifier!\n");
   }
}
```

OUTPUT 1:



OUTPUT 2:



RESULT:

Thus, the program to recognize a valid variable has been executed successfully.

EX NO: 3C DATE:	PROGRAM TO RECOGNIZE THE SYNTAX OF A VALID CONTROL STRUCTURES

AIM:

To write a yacc program to check valid structure syntax of c language.

ALGORITHM:

Step1: Identify Keywords, Symbols, and Identifiers using lex.

Step2: Use the Parser(yacc) to process tokens and apply grammar rules.

Step3: Check if-else Syntax: Validate if and if-else structures.

Step4: Validate loops: Recognize and Validate for, while, and do-while loops.

Step5: Handle switch-case: Parse switch statement along with case, default, and break statements.

Step6: Output: Print “Valid” for matching structures; otherwise, display an error message.

PROGRAM:

LEX PART

```
%{
#include "parser.tab.h"
#include <stdio.h>
#include <string.h>
%}
%%
"if" { return IF; }
"else" { return ELSE; }
"for" { return FOR; }
"while" { return WHILE; }
"do" { return DO; }
"switch" { return SWITCH; }
```

```
"case" { return CASE; }
"default" { return DEFAULT; }
"break" { return BREAK; }
"(" { return OPEN_PAREN; }
")" { return CLOSE_PAREN; }
"{" { return OPEN_BRACE; }
"}" { return CLOSE_BRACE; }
";" { return SEMICOLON; }
"++" { return INC; }
":" { return COLON; }
"<"|">"|"=="|"<="|">="|"!="|"=" { return SYM; }
[0-9]+ { return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
[ \t\n] ;
. { return yytext[0];
}
%%
int yywrap() {
   return 0;
}
```

YACC PART

```
%{
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 int yylex();
 int yyerror(const char *msg);
%}
%token IF ELSE FOR WHILE DO SWITCH CASE DEFAULT BREAK
```

```
OPEN_PAREN CLOSE_PAREN OPEN_BRACE CLOSE_BRACE
SEMICOLON IDENTIFIER SYM   NUMBER INC COLON
%%
program:
     loop_statement { printf("valid for loop statement\n"); exit(0); }
   | if_statement { printf("valid for if statement\n"); exit(0); }
   | if_else { printf("valid for if-else statement\n"); exit(0); }
   | switch_statement { printf("valid switch statement\n"); exit(0); }
   ;
if_else:
   if_statement ELSE OPEN_BRACE statement CLOSE_BRACE
   ;
if_statement:
   IF OPEN_PAREN condition CLOSE_PAREN OPEN_BRACE
statement CLOSE_BRACE
   ;
loop_statement:
     FOR OPEN_PAREN condition SEMICOLON condition
SEMICOLON expression INC CLOSE_PAREN OPEN_BRACE
statement CLOSE_BRACE
   | WHILE OPEN_PAREN expression CLOSE_PAREN
OPEN_BRACE statement CLOSE_BRACE
   | DO OPEN_BRACE statement CLOSE_BRACE WHILE
OPEN_PAREN expression CLOSE_PAREN SEMICOLON
   ;
switch_statement:
   SWITCH OPEN_PAREN expression CLOSE_PAREN
OPEN_BRACE case_list default_case CLOSE_BRACE
   ;
```

```
case_list:
     case_list CASE NUMBER COLON statement BREAK
SEMICOLON
   | CASE NUMBER COLON statement BREAK SEMICOLON
   ;
default_case:
     DEFAULT COLON statement BREAK SEMICOLON
   | /* optional */
   ;
statement:
     SEMICOLON
   | IDENTIFIER SEMICOLON
   ;
condition:
   IDENTIFIER SYM NUMBER
   ;
expression:
     IDENTIFIER
   | NUMBER
   ;
%%
int main() {
   yyparse();
   return 0;
}
```

```
int yyerror(const char* s) {
    fprintf(stderr, "Error: %s\n", s);
    return 0;
}
```

OUTPUT:

RESULT:

Thus, the program to recognize a valid control structure has been executed successfully.

EX NO: 3D DATE:	IMPLEMENTATION OF CALCULATOR USING LEX AND YACC.

AIM:

To write a program for implementation of calculator for computing the given expression using semantic rules of the YACC and LEX tool.

ALGORITHM:

Step 1: Read Expression: Get the expression from the user.

Step 2: Break into Parts: Split the expression into numbers, letters, and operators.

Step 3: Process Operations: For each operation (like +, *), create temporary variables (like t1, t2, etc.) to store the result.

Step 4: Store Results: Keep track of each operation in a table with the temporary variables.

Step 5: Generate Code: Print the steps of the expression as simple assignments (e.g., t1 := a + b).

Step 6: Show Three-Address Code: Display the final sequence of operations in the three-address code format.

PROGRAM:

LEX PART:

```
%{

#include "calc.tab.h" // Include the generated Bison header

%}

%%

[ \t]          ; // Ignore whitespace

[0-9]+          { yylval = atoi(yytext); return NUMBER;
} [-+*/()]     { return yytext[0];
```

```
\n              { return '\n'; }

.               { /* Ignore all other characters */ }
%%

int yywrap() {
   return 1;
}
```

YACC PART:

```
%{

#include <stdio.h>
#include <stdlib.h>
int yylex();

void yyerror(const char *msg);

%}

%token NUMBER

%left '+' '-'

%left '*' '/'

%%

calc : expr '\n' { printf("Valid Expression\nResult: %d\n", $1); }

    ;

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

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

    | term          { $$ = $1; }

    ;
```

```
term : term '*' factor { $$ = $1 * $3; }

| term '/' factor { $$ = $1 / $3; }

    | factor          { $$ = $1; }

    ;

factor : '(' expr ')' { $$ = $2; }

      | NUMBER        { $$ = $1; }

      ;

%%

int main() {

    printf("Enter an expression:\n");
    yyparse();
   return 0;

}

void yyerror(const char *msg) {
   fprintf(stderr, "Invalid expression");

}
```

OUTPUT 1

```
Microsoft Windows [Version 10.0.19045.5011]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Priya rajeswari\Downloads>flex cal.l

C:\Users\Priya rajeswari\Downloads>bison -d calc.y

C:\Users\Priya rajeswari\Downloads>gcc lex.yy.c calc.tab.c -o calc

C:\Users\Priya rajeswari\Downloads>calc
Enter an expression:
3 + 5 * (2 - 1)
Valid Expression
Result: 8
```

OUTPUT 2

```
Microsoft Windows [Version 10.0.19045.5011]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Priya rajeswari\Downloads>flex cal.l

C:\Users\Priya rajeswari\Downloads>bison -d calc.y

C:\Users\Priya rajeswari\Downloads>gcc lex.yy.c calc.tab.c -o calc.y

C:\Users\Priya rajeswari\Downloads>calc
Enter an expression:
3 + *
Invalid expression
C:\Users\Priya rajeswari\Downloads>
```

RESULT:

Thus, the program to implement a calculator using lex and yacc tool has been written and executed successfully.

EX NO: 4 DATE:	IMPLEMENTATION OF THREE ADDRESS CODE

AIM:

To generate three address code for a simple program using LEX and YACC.

ALGORITHM:

Step 1: Get the expression from the user.

Step 2: Split the expression into numbers, letters, and operators.

Step 3: For each operation (like +, *), create temporary variables (like t1, t2, etc.) to store the result.

Step 4: Keep track of each operation in a table with the temporary variables.

Step 5: Print the steps of the expression as simple assignments (e.g., t1 := a + b).

Step 6: Display the final sequence of operations in the three-address code format.

PROGRAM:

YACC PART: //addresscode.y

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <ctype.h>
int yylex();
int yyerror(const char *msg);
void ThreeAddressCode();
char AddToTable(char ,char, char);
```

```
int ind=0;
char temp = '1'; // for t1, t2, t3, etc.
struct incod
{
char opd1;
char opd2;
char opr;
};
%}
%union
{
char sym;
}
%token <sym> LETTER NUMBER
%type <sym> expr term factor
%left '+' '-'
%left '*' '/'
%left '('
%%
statement: LETTER '=' expr ';' { AddToTable((char)$1, (char)$3, '='); }
| expr ';'
;
expr: expr '+' term { $$ = AddToTable((char)$1, (char)$3, '+'); }
| expr '-' term { $$ = AddToTable((char)$1, (char)$3, '-'); }
```

```
| term { $$ = (char)$1; }
;
term: term '*' factor { $$ = AddToTable((char)$1, (char)$3, '*'); }
| term '/' factor { $$ = AddToTable((char)$1, (char)$3, '/'); }
| factor { $$ = (char)$1; }
;
factor: '(' expr ')' { $$ = (char)$2; } /* Parentheses */
| NUMBER { $$ = (char)$1; }
| LETTER { $$ = (char)$1; }
;
%%
struct incodcode[20];
char AddToTable(char opd1, char opd2, char opr)
{
code[ind].opd1 = opd1;
code[ind].opd2 = opd2;
code[ind].opr = opr;
ind++;
return temp++;
}
void ThreeAddressCode()
{
int cnt = 0;
char temp = '1';
```

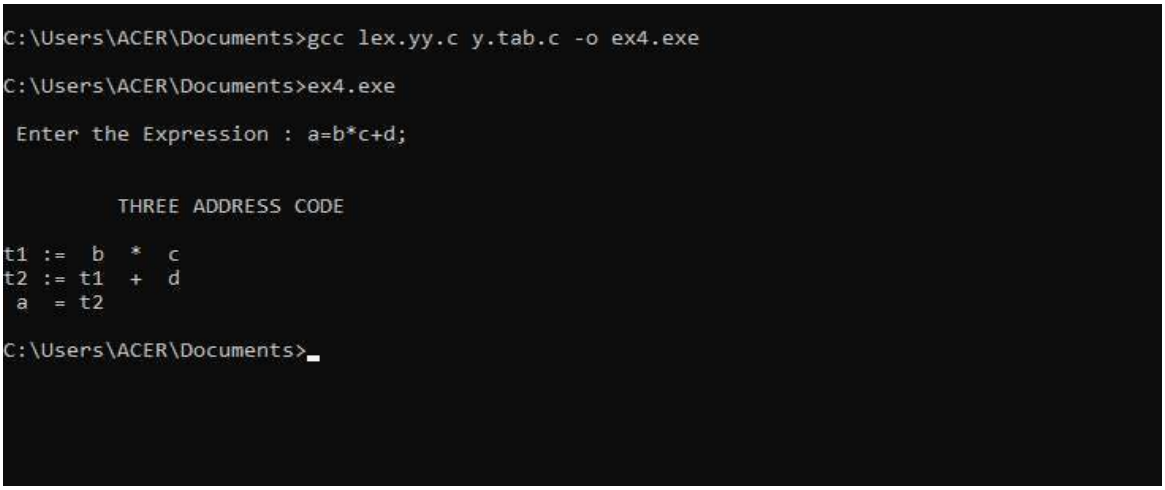
```
printf("\n\n\t THREE ADDRESS CODE\n\n");
while (cnt<ind)
{
if (code[cnt].opr != '=')
printf("t%c := ", temp++);
if (isalpha(code[cnt].opd1))
printf(" %c ", code[cnt].opd1);
else if (code[cnt].opd1 >= '1' && code[cnt].opd1 <= '9')
printf("t%c ", code[cnt].opd1);
printf(" %c ", code[cnt].opr);
if (isalpha(code[cnt].opd2))
printf(" %c\n", code[cnt].opd2);
else if (code[cnt].opd2 >= '1' && code[cnt].opd2 <= '9')
printf("t%c\n", code[cnt].opd2);
cnt++;
}
}
int main()
{
printf("\n Enter the Expression : ");
yyparse();
ThreeAddressCode();
}
int yyerror(const char* s)
```

```
{
fprintf(stderr, "Error: %s\n", s);
return 0;
}
```

LEX PART: //addresscode.l

```
%{
#include "y.tab.h"
%}
%%
[0-9]+ { yylval.sym = (char)yytext[0]; return NUMBER; }
[a-zA-Z]+ { yylval.sym = (char)yytext[0]; return LETTER; }
\n { return 0; }
. { returnyytext[0]; }
%%
int yywrap()
{
return 1;
}
```

OUTPUT:



RESULT:

Thus, the program to generate three address code was written, executed and output was verified successfully.

EX NO:5 DATE:	IMPLEMENTATION OF TYPE CHECKING USING LEX AND YACC

AIM:

To implement type checking using LEX and YACC using a program.

ALGORITHM:

Step 1 : Read the input expression (e.g., int x = 5;).

Step 2: Break the input into smaller pieces (tokens) like keywords (int, float), variable names(x), numbers (5), and symbols (=, ;).

Step 3: Check if the tokens match one of the valid assignment patterns: int variable = number;float variable = float number; char variable = 'char';

Step 4: Make sure the type matches:

4.1: An int can only be assigned a number.

4.2: A float can be assigned a float or a number.

4.3: A char can only be assigned a single character.

Step 5: Check the conditions.

5.1: If the assignment is correct, print valid.

5.2: If there's a mistake, print Error: TYPE MISMATCH.

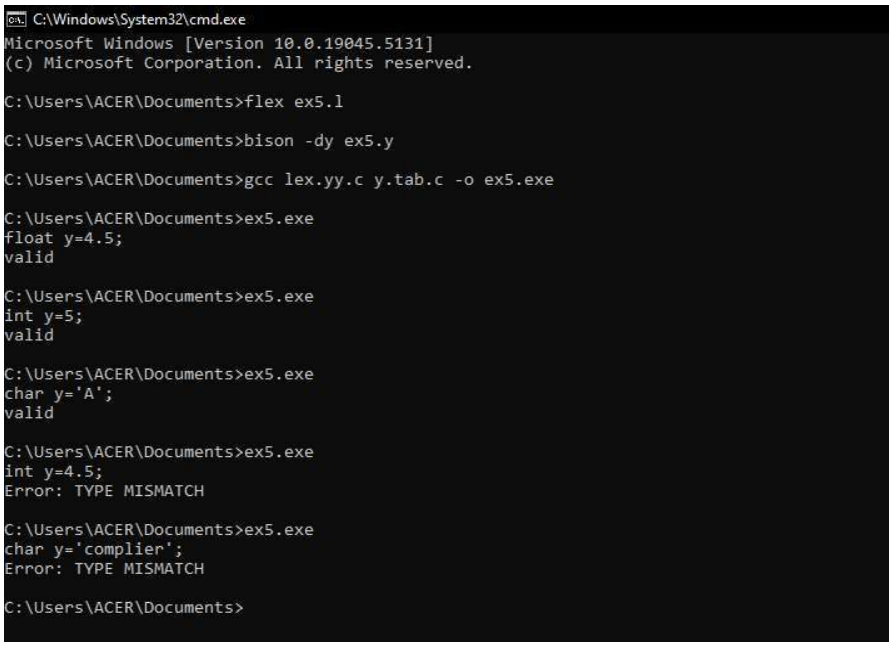
Step 6: Finish the program after checking the input.

PROGRAM:

```
%{
#include "typecheck.tab.h"
#include<string.h>
%}
%%
"int"    { return INT; }
"float" {return FLOAT; }
"char" {return CHAR; }
""       {return QUOTE;}
"=" {return EQL;}
[0-9]+ {return NUMBER;}
0\.[0-9]*|[1-9][0-9]*\.[0-9]* {yylval.float_value = atof(yytext);return FLOATVAL;}
[a-zA-Z_][a-zA-Z0-9_]* { yylval.strval = strdup(yytext);
If (strlen(yytext) == 1) {
Return CHARACTER;
} else { return IDENTIFIER;
}
}
";"      { return SEMICOLON; }
```

```
[ \t]      ; // Ignore whitespace
\n         {return NEXT;}
.          { return yytext[0]; }
%%
int yywrap() {
 return 0;
}
%{
#include <stdio.h>
#include <stdlib.h> #include <string.h> int yylex(); int yyerror(const char *msg);
%}
%union{ char char_value; char charval; float float_value; char *strval;
}
%token SEMICOLON IDENTIFIER EQL NUMBER NEXT INT FLOAT CHAR FLOATVAL
QUOTE CHARACTER
%%
Program: INT identifier EQL NUMBER SEMICOLON NEXT {printf("valid\n");exit(0);}
| FLOAT identifier EQL floatnum SEMICOLON NEXT {printf("valid\n");exit(0);}
| CHAR identifier EQL QUOTE charval QUOTE SEMICOLON NEXT

{printf("valid\n");exit(0);}
;
Charval:CHARACTER
;
Floatnum: FLOATVAL
| NUMBER
;
Identifier: IDENTIFIER
| CHARACTER
| /* Other expression types can be added here */
;
%%
Int main() { yyparse(); return 0;
}
Int yyerror(const char* s) {
Fprintf(stderr, "Error: TYPE MISMATCH\n");
}
```

OUTPUT:



RESULT:

Thus the program to implement type checking using LEX and YACC was written, executive and output is verified successfully.

EX. NO: 6 DATE:	IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES

AIM:

To Implement Simple Code Optimization Techniques (constant folding, strength reduction and algebraic transformation).

ALGORITHM:

Step1: Read the number of operations. For each operation, read the left and right operands (like a = b + c).

Step 2: Check if the result of an operation is used later. If not, remove it.

Step 3: If two operations have the same right side (e.g., a = b + c and x = b + c), use the result from the first operation.

Step 4: If two operations are exactly the same, remove the duplicate.

Step 5: Show the updated code after optimizations.

PROGRAM:

```
#include <stdio.h>

#include <conio.h>

#include <string.h>

#include<stdlib.h>

struct op

{

char l;
```

```
char r[20];
}
op[10], pr[10];
void main()
{
int a, i, k, j, n, z = 0, m, q;
char * p, * l;
char temp, t;
char * tem;
printf("enter no of values:");
scanf("%d", & n);
char input[1000];
fflush(stdin);
for (i = 0; i< n; i++){
printf("\tleft\t");
scanf(" %c", &op[i].l);
printf("\n\tright:\t");
scanf("%s", op[i].r);
}
printf("intermediate Code\n");
```

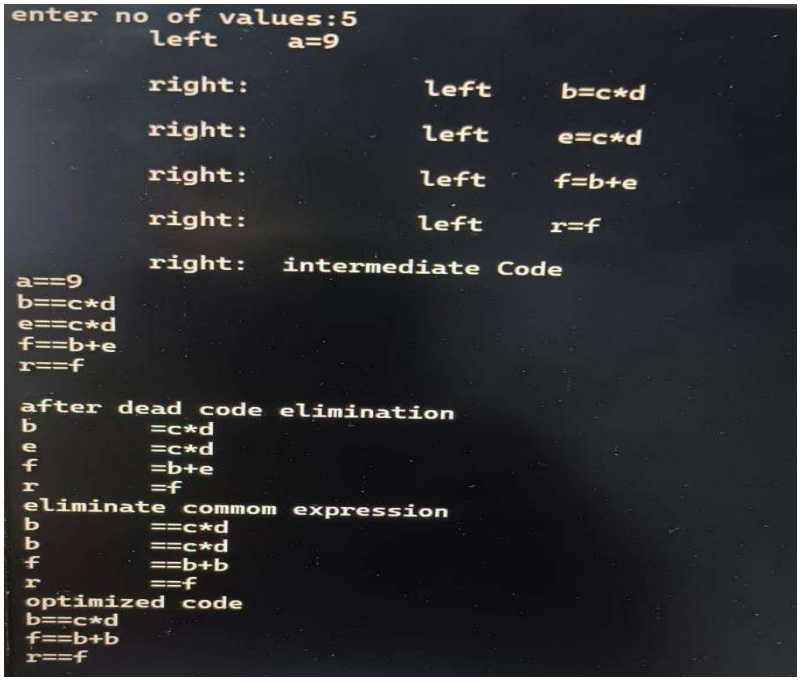
```
for (i = 0; i< n; i++)
{
printf("%c=", op[i].l);
printf("%s\n", op[i].r);
}
for (i=0; i<n-1; i++)
{
temp = op[i].l;
for (j = 0; j < n; j++)
{
p = strchr(op[j].r, temp);
if (p)
{
pr[z].l = op[i].l;
strcpy(pr[z].r, op[i].r);
z++;}
}
}
pr[z].l= op[n-1].l;
strcpy(pr[z].r, op[n - 1].r);
```

```
z++;
printf("\nafter dead code elimination\n");
for (k = 0; k < z; k++)
{
printf("%c\t", pr[k].l);
printf("%s\n", pr[k].r);
}
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("eliminate commom expression\n");
for (i=0; i<z; i++)
{
printf("%c\t=", pr[i].l);
printf("%s\n", pr[i].r);
}
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';
strcpy(pr[i].r,"\0");
}}}printf("optimized code\n");
for (i = 0; i< z; i++)
{
if (pr[i].l != '\0')
{
```

```
printf("%c=", pr[i].l);

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

}

}

getch();

return 0;

}
```

OUTPUT:



RESULT:

Thus the C program for implementation of Code Optimization Technique such as Constant folding, Strength reduction and Algebraic Transformation was executed successfully.

EX NO: 7 DATE:	IMPLEMENTATION OF BACK-END OF THE COMPILER.

AIM:

To write a program for implementation of back-end of the compiler for which the three-address code is given as input and the 8086-assembly language code is produced as output.

ALGORITHM:

Step1: Initialize: Set up arrays to store code lines and operators.

Step2: Input Code: Enter each line of code, stopping with "exit".

Step 3: Identify Operator: For each line, detect the operator (e.g., +, -, *, /).

Step 4: Store Results: Keep track of each operation in a table with the temporary variables.

Step 5: Generate Assembly Code: Use the operator to create assembly instructions.

Step 6: Display Output: Print the generated assembly code for each line.

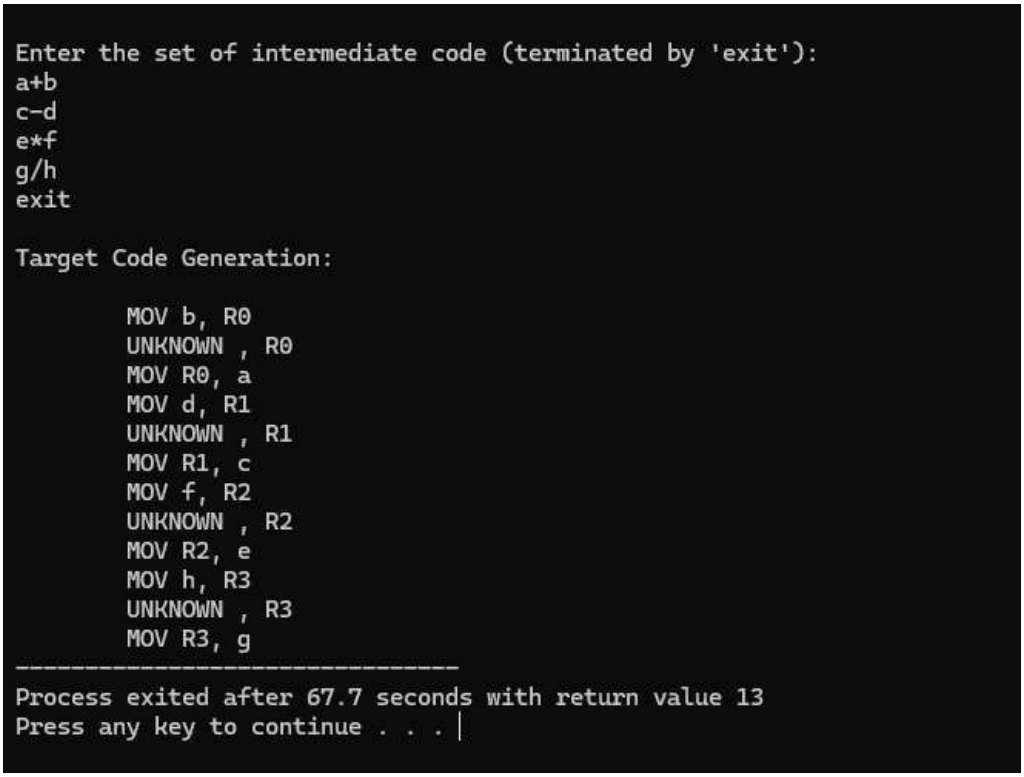
PROGRAM:

```
#include<stdio.h>

#include<string.h>

void main () {

char icode[10][30], str[20], opr[10];

int i = 0;

printf("\nEnter the set of intermediatecode (terminated by 'exit'):\n");

// Read intermediate code until "exit"is encountered
```

```
do {

scanf("%s", icode[i]);

} while (strcmp(icode[i++], "exit") != 0);

printf("\nTarget Code
    Generation:\n");

// Process each intermediate code line

i = 0;

do {

strcpy(str, icode[i]); // Copy the intermediate code to str

// Check the operator and assign corresponding operation

switch (str[3]) {

case '+':

strcpy(opr, "ADD");

break;

case '-':

strcpy(opr, "SUB");

break;

case '*':

strcpy(opr, "MUL");

break;
```

```
case '/':
strcpy(opr, "DIV");
break;
default:
strcpy(opr, "UNKNOWN");
break;
}
// Print target code generation instructions
printf("\n\tMOV %c, R%d", str[2], i);
printf("\n\t%s %c, R%d", opr, str[4], i);
printf("\n\tMOVR%d, %c", i, str[0]);
} while (strcmp(icode[++i], "exit") != 0);
// Wait for user input before exiting (not needed in most cases)
getch();
}
```

OUTPUT:



RESULT:

Thus, the program to implement the back-end of the compiler was written, executed and the output was verified successfully.