DEVELOPALEXICALANALYZERTORECOGNIZEFEWPATTERNSINC(EX. IDENTIFIERS, CONSTANTS, COMMENTS,OPERATORSETC.)AND IMPLEMENTATIONOFASYMBOLTABLE

Ex.No:1

Date:

AIM:

To develop a lexical analyzer to recognize few patterns in C (Ex. Identifiers, Constants, Comments, Operators etc.) and Implementation of a symbol table.

ALGORITHM:

Step1:Start the program

Step 2:.Read the input string.

Step 3: Check whether the string is identifier,operator,symbol by using the rules of identifier and keywords using lex tool using the following steps.

Step 4:.If the string starts with letter followed by any number of letter or digit then display it as a identifier.

Step 5.:If it is operator print it as a operator

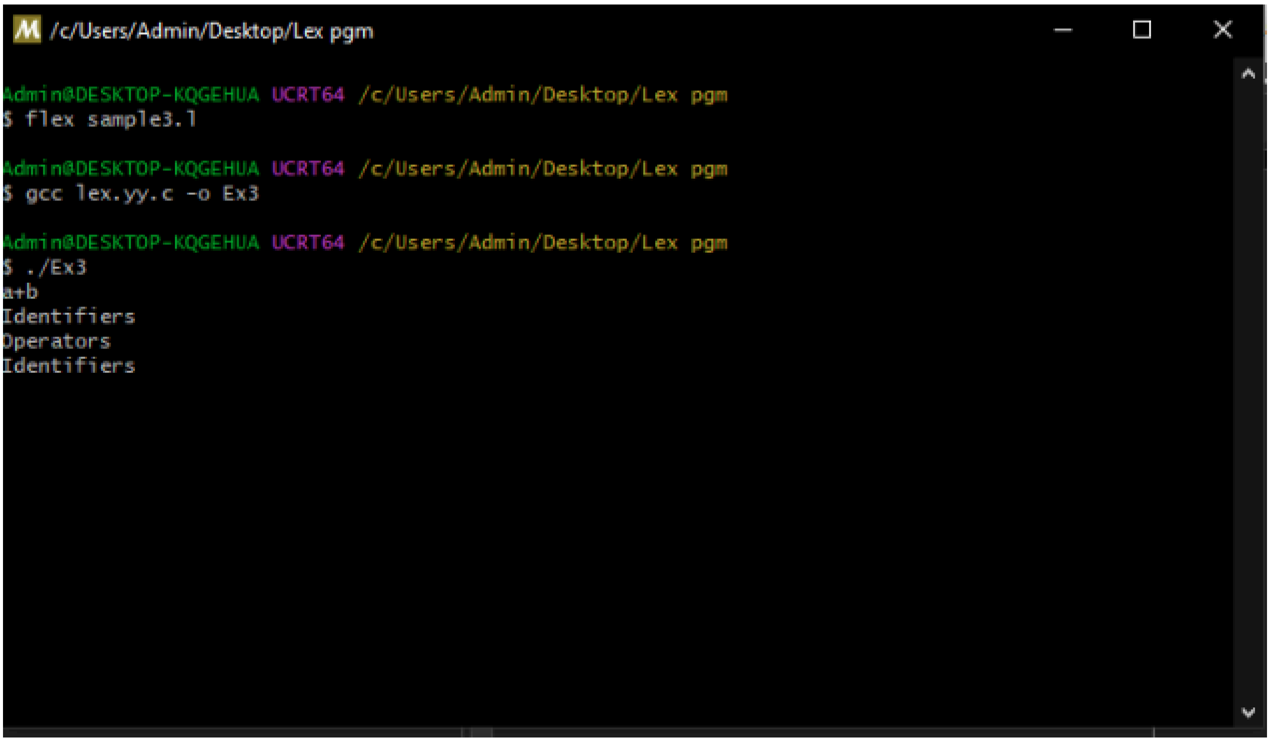
Step 6: If it is number print it as a number

Step 7:Stop the program

PROGRAM:

```
%{
#include <stdio.h>
%}
%%
bool|int|float|char printf("Keyword\n");
[-+]+ printf("Operators\n");
[0-9]+ printf("Numbers\n");
[,.'"]+ printf("Punctuation Chars\n");
[&%*$@!]+ printf("Special Characters\n");
[a-zA-Z]+ printf("Identifiers\n");
%%
int main() {
 yylex();
 return 0;
}
int yywrap() {
 return 1;
}
```

OUTPUT:



CONCLUSION:

Thus to develop a lexical analyzer to recognize few patterns in C (Ex. Identifiers, Constants, Comments, Operators etc.) was executed successfully

PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION

Ex.No:2.a

Date:

AIM:

To write a c program to recognize a valid arithmetic expression.

ALGORITHM:

Step1: Start the program.

Step2: Reading an expression .

Step3: Checking the validating of the given expression according to the rule using yacc.

Step4: Using expression rule print the result of the given values

Step5: Stop the program.

PROGRAM:

LEX Program(Validarith.l)

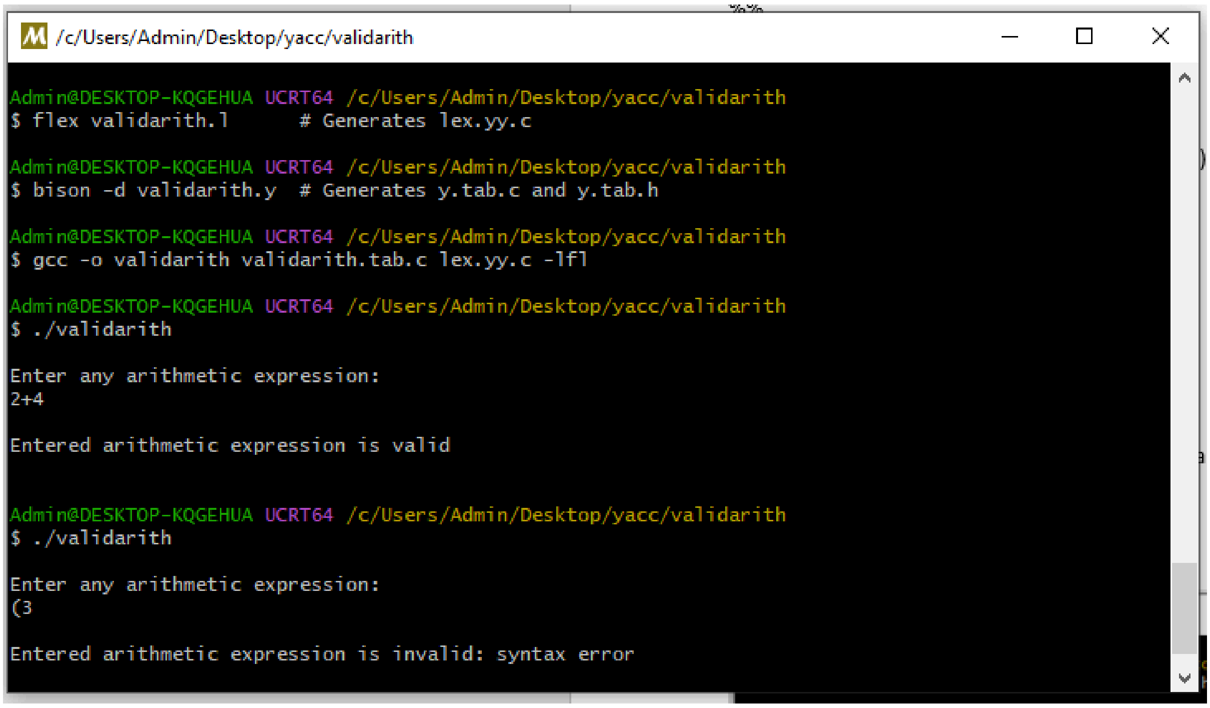
```
%{
#include&lt;stdio.h&gt;
#include &quot;Validarith.tab.h&quot;
%}
%%
[a-zA-Z]+ return VARIABLE;
[0-9]+ return NUMBER;
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

YACC Program (Validarith.y)

```
%{
#include &lt;stdio.h&gt;

#include &lt;stdlib.h&gt;

// Function prototype
int yylex();
void yyerror(const char *s);
```

```
%}

%token NUMBER
%token VARIABLE
%left &#39;+&#39; &#39;-&#39;
%left &#39;*&#39; &#39;/&#39; &#39;%&#39;
%left &#39;(&#39; &#39;)&#39;

%%

// Grammar rules
S: E {
printf(&quot;\nEntered arithmetic expression is valid\n\n&quot;);
return 0;
}
;

E: E &#39;+&#39; E
| E &#39;-&#39; E
| E &#39;*&#39; E
| E &#39;/&#39; E
| E &#39;%&#39; E
| &#39;(&#39; E &#39;)&#39;
| NUMBER
| VARIABLE
;

%%

int main() {
printf(&quot;\nEnter any arithmetic expression: \n&quot;);
yyparse();
return 0;
}

void yyerror(const char *s) {
printf(&quot;\nEntered arithmetic expression is invalid: %s\n\n&quot;, s);
}
```

Output:



CONCLUSION:

Thus a program to recognize a valid arithmetic expression was executed successfully.

PROGRAM TO RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS

Ex.No:2.b

Date:

AIM :

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

ALGORITHM:

Step1: Start the program

Step2: Reading an expression

Step3: Checking the validating of the given expression according to the rule using yacc.

Step4: Using expression rule print the result of the given values

Step5: Stop the program

PROGRAM CODE:

LEX Program(Valid_identifier.l)

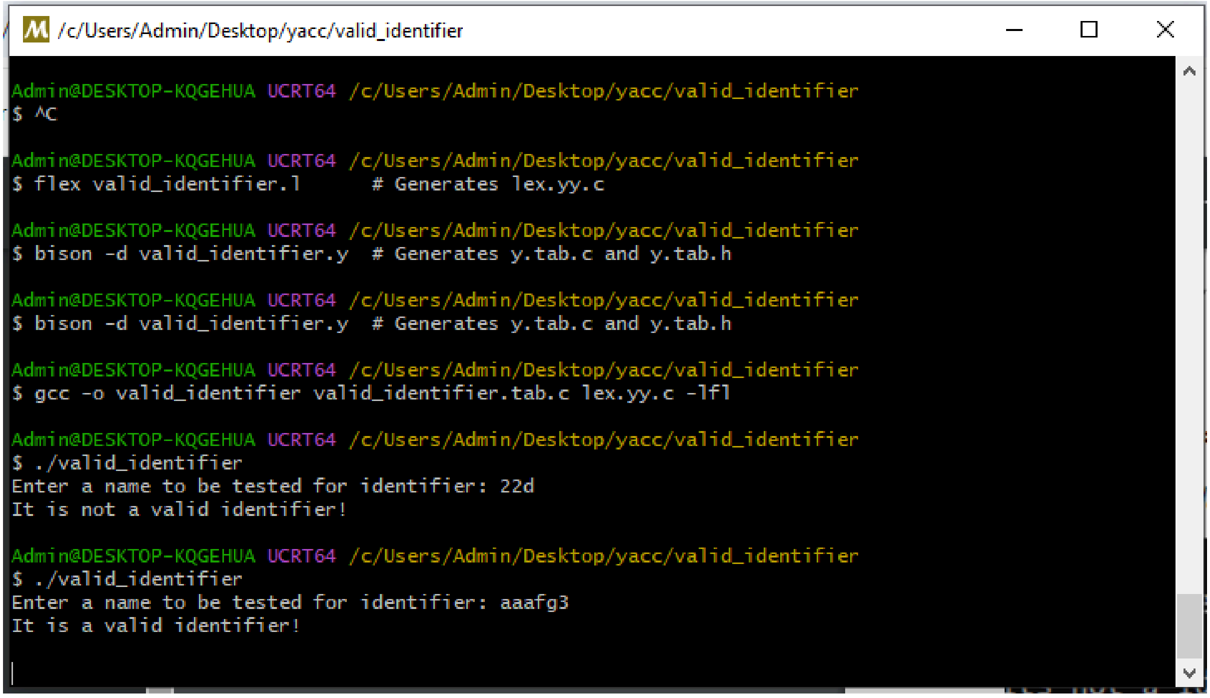
```
%{
#include "valid_identifier.tab.h"
%}

%%

[a-zA-Z][a-zA-Z0-9]* { return IDENTIFIER; }

. { return 0; } /* For any other invalid characters */

%%

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

YACC Program(Valid_identifier.y):

```
%{

#include <stdio.h>
#include <ctype.h>

// Declaration of yylex function
int yylex(void);

// Declaration of yyerror function
```

```
int yyerror(const char *s);

%}

%token IDENTIFIER

%%

start:
identifier_check
;

identifier_check:
IDENTIFIER { printf("It is a valid identifier!\n"); }
| /* error handling for invalid input */
{ printf("It is not a valid identifier!\n"); }
;

%%

int yyerror(const char *s) {
printf("It is not a valid identifier!\n");
return 0;
}

int main() {
printf("Enter a name to be tested for identifier: ");
yyparse();
return 0;
}
```

Output:



CONCLUSION:

Thus a program to check valid variable followed by letter or digits was executed successfully.

PROGRAM TO RECOGNIZE WHILE LOOP

Ex.No:3.a

Date:

AIM:

To write a yacc program to recognize while loop.

ALGORITHM:

Step1: Start the program.

Step2: Reading an expression .

Step3: Checking the validating of the given while loop according to the rule using yacc.

Step4: Print the result of the given while loop

Step5: Stop the program.

LEX program(wh.l):

```
%{
#include "wh.tab.h"
%}

%option noyywrap
%option nounput

%%

"while"        { return WHILE; }
"("            { return LPAREN; }
")"            { return RPAREN; }
"{"            { return LBRACE; }
"}"            { return RBRACE; }
"="            { return ASSIGN; }
"<"            { return LT; }
">"            { return GT; }
"=="           { return EQ; }
"+"            { return PLUS; }
"-"            { return MINUS; }
"*"            { return MULT; }
"/"            { return DIV; }
";"            { return ';'; }  // Allow semicolon without error
[0-9]+         { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* { return IDENTIFIER; }
[ \t\n]+       { /* skip whitespace */ }
.              { printf("Unexpected character: %s\n", yytext); return yytext[0]; }


<<EOF>>        { return 0; }
%%
```

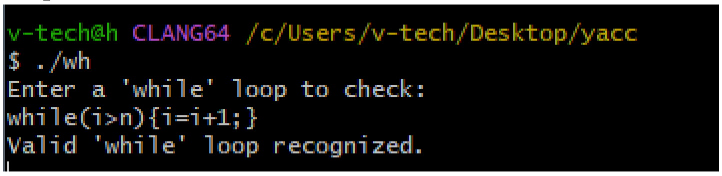
YACC Program(wh.y):

```
%{
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

void yyerror(const char *s);
int yylex(void);
%}

/* Define tokens */
%token WHILE LPAREN RPAREN LBRACE RBRACE SEMICOLON IDENTIFIER NUMBER
%token ASSIGN LT GT EQ PLUS MINUS MULT DIV
%left PLUS MINUS
%left MULT DIV
%left LT GT EQ
%right ASSIGN
%%

program:
    statement
    ;

statement:
    WHILE LPAREN condition RPAREN LBRACE statements RBRACE
    {
        printf("Valid 'while' loop recognized.\n");
    }
    ;

statements:
    statement
    | assignment ';'
    | statements statement
    | /* empty */
    ;

assignment:
    IDENTIFIER ASSIGN expression
    ;

condition:
    expression LT expression
    | expression GT expression
    | expression EQ expression
    ;

expression:
     IDENTIFIER    | NUMBER
    | expression PLUS expression
    | expression MINUS expression
    | expression MULT expression
    | expression DIV expression
    ;

statement:
    IDENTIFIER '=' expression
    | IDENTIFIER
    | NUMBER
    ;

%%
```

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

int main(void) {
    printf("Enter a 'while' loop to check:\n");
    yyparse();
    return 0;
}
```

Output:



CONCLUSION:

Thus a program to recognize while loop was executed successfully.

IMPLEMENTATION OF CALCULATOR USING LEX & YACC

Ex.No:3.b
Date:

AIM:

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

ALGORITHM:

Step 1: Start the program.
Step 2: Include the necessary header files.
Step 3: Use a function for printing the error message.
Step 4: Get the input from the user and parse it.
Step 5: Check the input is a valid expression or not.
Step 6: Write separate operations for addition, subtraction, multiplication and division using the expr and matching it with the operators in the in the input.
Step 7: Print the error messages for the invalid operators.
Step 8: Print the output of the expression. Step 9: Terminate the program.

LEX program(Calc.l):

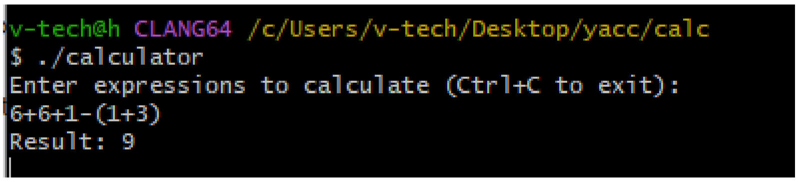
```
%{
#include "calc.tab.h"  // Include the parser's header for token definitions
%}

%%

[0-9]+          { yylval = atoi(yytext); return NUMBER; }
[ \t]+          { /* Ignore whitespace */ }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MULT; }
"/"             { return DIV; }
"("             { return LPAREN; }
")"             { return RPAREN; }
\n              { return '\n'; }  // Recognize newline as a token
.               { printf("Unexpected character: %s\n", yytext); }

%%
```

YACC program(Calc.y):

```
%{
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
int yylex();
%}

%token NUMBER
%token PLUS MINUS MULT DIV LPAREN RPAREN

%left PLUS MINUS
%left MULT DIV

%%

input:
   /* empty */
```

```
    | input line
    ;

line:
    '\n'
    | expression '\n'   { printf("Result: %d\n", $1); }
    ;

expression:
    NUMBER              { $$ = $1; }
    | expression PLUS expression    { $$ = $1 + $3; }
    | expression MINUS expression   { $$ = $1 - $3; }
    | expression MULT expression    { $$ = $1 * $3; }
    | expression DIV expression     { $$ = $1 / $3; }
    | LPAREN expression RPAREN      { $$ = $2; }
    ;

%%

int main(void) {
    printf("Enter expressions to calculate (Ctrl+C to exit):\n");
    yyparse();
    return 0;
}

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

Output:



CONCLUSION:

Thus a program to implement the calculator using lex & yacc was executed successfully

Implementation of Three Address Code using LEX and YACC

Ex.No:4
Date:

AIM:

To write a program for implementing Three Address Code using LEX and YACC.

ALGORITHM:

Step1: A Yacc source program has three parts as follows
Declarations %% translation rules %% supporting C routines
Step2: Declarations Section: This section contains entries that:
i. Include standard I/O header file.
ii. Define global variables.
iii. Define the list rule as the place to start processing.
iv. Define the tokens used by the parser. v. Define the operators and their precedence.
Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.
Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.
Step5: Main- The required main program that calls the yyparse subroutine to start the program.
Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.
20
Step7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.
Step8: calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM:
LEX part (tac.l):

```
%{
#include "tac.tab.h"
#include <stdio.h>
#include <stdlib.h>
%}

%%


[0-9]+       { yylval.symbol = atoi(yytext); return NUMBER; }
[a-zA-Z]     { yylval.symbol = yytext[0]; return LETTER; }
"+"          { return '+'; }
"-"          { return '-'; }
"*"          { return '*'; }
"/"          { return '/'; }
"="          { return '='; }
";"          { return ';'; }
"("          { return '('; }
")"          { return ')'; }
[ \t\n]      { /* skip whitespace */ }
.            { printf("Unexpected character: %s\n", yytext); }

%%
int yywrap() {
   return 1; // Indicate no more input
```

```
}
```

Yacc program(tac.y):

```
%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void); // Declaration of yylex
struct expr {
    char operand1;
    char operand2;
    char operator;
    char result;
};

char addtotable(char a, char b, char o);
void threeAdd();
void yyerror(const char *s);

// Global variables
struct expr arr[20];
int index1 = 0;
char temp = 'A' - 1;
%}

%union {
    char symbol;
}

%left '+' '-'
%left '*' '/'
%token <symbol> LETTER NUMBER
%type <symbol> exp

%%

// Grammar rules
statement:
    LETTER '=' exp ';' { addtotable($1, $3, '='); }
    ;

exp:
    exp '+' exp { $$ = addtotable($1, $3, '+'); }
    | exp '-' exp { $$ = addtotable($1, $3, '-'); }
    | exp '*' exp { $$ = addtotable($1, $3, '*'); }
    | exp '/' exp { $$ = addtotable($1, $3, '/'); }
    | '(' exp ')' { $$ = $2; }
    | NUMBER { $$ = $1; }
    | LETTER { $$ = $1; }
    ;

%%

// Error handling function
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

// Function to add a record to the table
```

```
char addtotable(char a, char b, char o) {
    temp++;
    arr[index1].operand1 = a;
    arr[index1].operand2 = b;
    arr[index1].operator = o;
    arr[index1].result = temp;
    index1++;
    return temp;
}

// Function to display the three-address code
void threeAdd() {
    for (int i = 0; i < index1; i++) {
        printf("%c := %c %c %c\n", arr[i].result, arr[i].operand1, arr[i].operator, arr[i].operand2);
    }
}

// Main function
int main() {
    printf("Enter the expression: ");
    yyparse();
    threeAdd();
    return 0;
}
```

Output:

```
v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ flex tac.l

v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ bison -d tac.y

v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ gcc -o tac tac.tab.c lex.yy.c -lfl

v-tech@h CLANG32 /c/Users/v-tech/Desktop/yacc/tac
$ ./tac
Enter the expression:
a=b+c*d-e
Error: syntax error
A := c * d
B := b + A
C := B - e
```

Conclusion:

Thus the program for implementing Three Address Code using LEX and YACC was executed successfully.

IMPLEMENTATION OF TYPE CHECKING

Ex.No:5

Date:

AIM:

To write a C program to implement type checking.

ALGORITHM:

Step1: Track the global scope type information (e.g. classes and their members)

Step2: Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.

Step3: If type found correct, do the operation

Step4: Type mismatches, semantic error will be notified

PROGRAM :

```
//To implement type checking
#include<stdio.h>
#include<stdlib.h>

int main()
{

int n,i,k,flag=0;
char vari[15],typ[15],b[15],c;
printf("Enter the number of variables:");
scanf(" %d",&n);
for(i=0;i<n;i++)
{

printf("Enter the variable[%d]:",i);
scanf(" %c",&vari[i]);
printf("Enter the variable-type[%d](float-f,int-i):",i);
scanf(" %c",&typ[i]);
if(typ[i]=='f')
flag=1;
}

printf("Enter the Expression(end with $):");
```

```
i=0;
getchar();
while((c=getchar())!='$')
{
b[i]=c
; i++;
}
k=i;
for(i=0;i<k;i++)
{
if(b[i]=='/')
{
flag=1;
break; } }
for(i=0;i<n;i++)
{
if(b[0]==vari[i])
{
if(flag==1)
{
if(typ[i]=='f')
{ printf("\nthe datatype is correctly defined..!\n");
break; }
else
{ printf("Identifier %c must be a float type..!\n",vari[i]);
break; } }
else
{ printf("\nthe datatype is correctly defined..!\n");
break; } }
}
return 0;
}
```

OUTPUT:

```
Enter the number of variables:4
Enter the variable[0]:A
Enter the variable-type[0](float-f,int-i):i
Enter the variable[1]:B
Enter the variable-type[1](float-f,int-i):i
Enter the variable[2]:C
Enter the variable-type[2](float-f,int-i):f
Enter the variable[3]:D
Enter the variable-type[3](float-f,int-i):i
Enter the Expression(end with $):A=B*C/D$
Identifier A must be a float type..!
```

CONCLUSION:

Thus a program to implement type checking was executed successfully.

IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES

EX.NO:6

Date:

a)Dead Code Elimination

AIM: To write a C program to implement Code Optimization Techniques.

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Declare necessary character arrays for input and output and also a structure toinclude it.

Step 4: Get the Input: Set of 'L' values with corresponding 'R' values and Step 5: Implement the principle source of optimization techniques.

Step 5: The Output should be of Intermediate code and Optimized code after eliminating common expressions. .

Step 6: Terminate the program

PROGRAM:

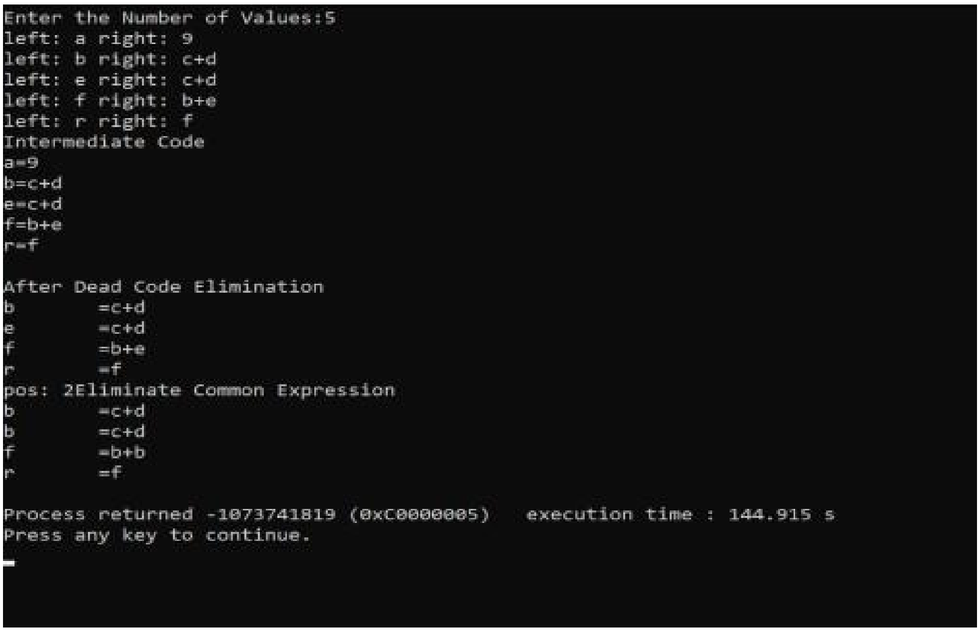
```
#include<stdio.h>
#include<conio.h>
#include<string.h
> struct op
{
char l;
char
r[20];
}
op[10],pr[10];
void main()
{
int
a,i,k,j,n,z=0,m,q;
char *p,*l;
```

```
char
temp,t;
char *tem;
clrscr();
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left: ");
op[i].l=getche();
printf("\tright: ");
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;
printf("pos: %d",a);
pr[i].r[a]=pr[m].l;
}
}
}
}
}
```

```
printf("Eliminate Common 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();
}
```

OUTPUT:



```
Enter the Number of Values:5
left: a right: 9
left: b right: c+d
left: e right: c+d
left: f right: b+e
left: r right: f
Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f

After Dead Code Elimination
b       =c+d
e       =c+d
f       =b+e
r       =f
pos: 2Eliminate Common Expression
b       =c+d
b       =c+d
f       =b+b
r       =f

Process returned -1073741819 (0xC0000005)   execution time : 144.915 s
Press any key to continue.
```

b) Implementation of loop-invariant code movement or code motion Aim:

To write a C program to implement Code Optimization Techniques.

loop-invariant code movement or code motion:

Loop-invariant code consists of statements or expressions (in an imperative programming language) which can be moved outside the body of a loop without affecting the semantics of the program. Loop-invariant code motion (also called hoisting or scalar promotion) is a compiler optimization which performs this movement automatically.

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h
> #define max 6
void main()
{
int
n=1,s=0;
clrscr();
```

```
printf("Output without Code movement technique:\n");
while(n<=max-1)
{
s=s+n;
n++;
}
printf("Sum of First 5 Numbers:%d",s);
getch();
}
```

OUTPUT:

Output without Code movement Technique :

Sum of First 5 Numbers:15

C)strength reduction :

Strength reduction is a compiler optimization where expensive operations are replaced with equivalent but less expensive operations

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
void main()
{ int i,s;
clrscr();
printf("Output without strength reduction:\n");
for(i=1;i<=10;i++)
{ s=i*2;
printf("%d ",s);
}
getch();
}
```

OTPUT:

Output without strength reduction: 2 4 6 8 10 12 14 16 18 20

CONCLUSION:

Thus a program to implement simple code optimization techniques was executed Successfully.

IMPLEMENTING THE BACK END OF THE COMPILER

Ex.No:7

Date:

AIM:

To implement the back end of a compiler which takes the three address code and produces the 8086 assembly language instructions using a C program.

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Declare necessary character arrays for input and output and also a structure to include it.

Step 4: Get the Intermediate Code as the input.

Step 5: Display the options to do the various operations and use a switch case to implement that operation.

Step 6: Terminate the program.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{

char icode[10][30],str[20],opr[10];
int i=0;
clrscr();
printf("\n Enter the set of intermediate code (terminated by exit):\n"); do
{
```

```
scanf("%s",icode[i]);
} while(strcmp(icode[i++],"exit")!=0);
printf("\n target code generation");
printf("\n************************");
i=0;
do
{
strcpy(str,icode[i]);
switch(str[3])
{
case '+':
strcpy(opr,"ADD");
break;
case '-':
strcpy(opr,"SUB");
break;
case '*':
strcpy(opr,"MUL");
break;
case '/':
strcpy(opr,"DIV");
break;
}
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);
getch();
}
```

OUTPUT:

```
 Enter the set of intermediate code (terminated by exit):
d=2/3
c=4/5
a=2*3
exit

 target code generation
************************
        Mov 2,R0
        DIV3,R0
        Mov R0,d
        Mov 4,R1
        DIV5,R1
        Mov R1,c
        Mov 2,R2
        MUL3,R2
        Mov R2,a
```

CONCLUSION:

Thus a program to implement the back end of the compiler was executed successfully.