Varuvan Vadivelan Institute of Technology

Dharmapuri – 636 703

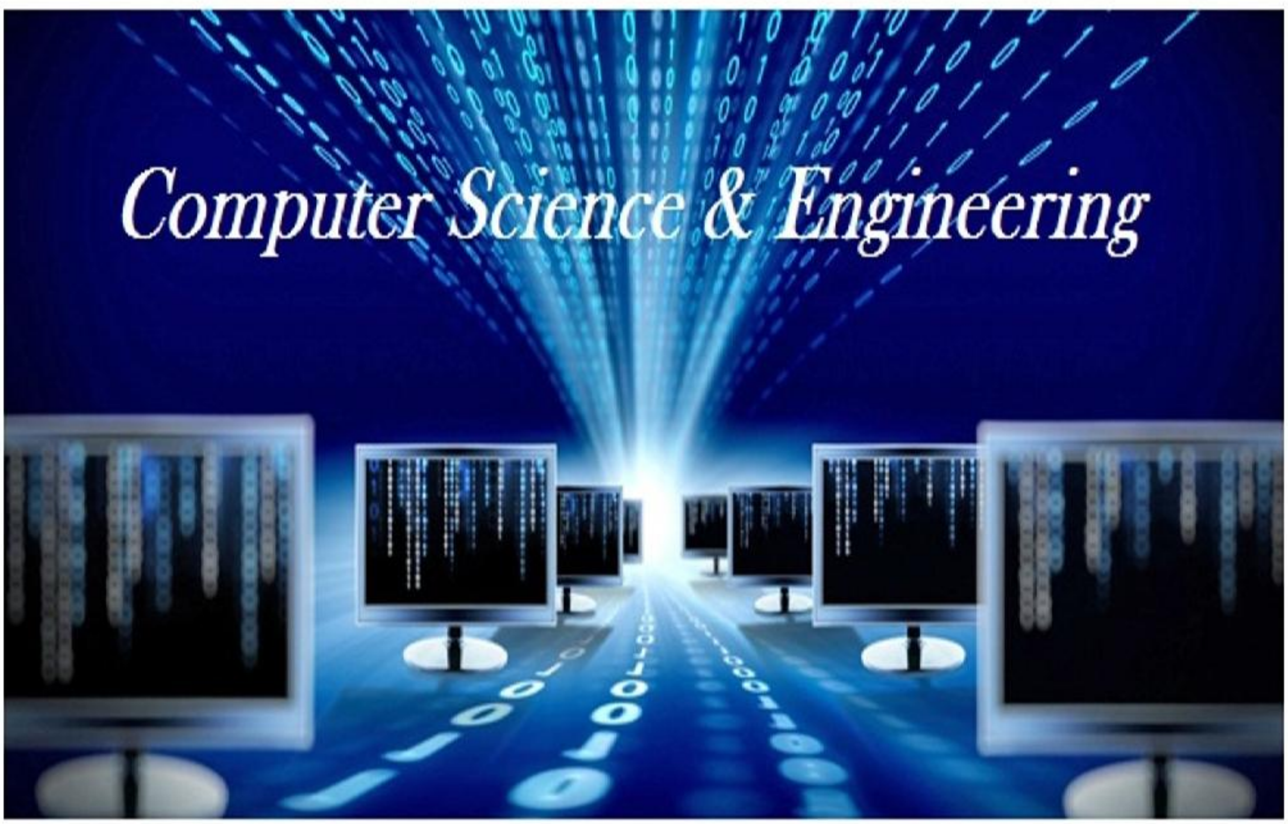
LAB MANUAL

Regulation : 2013

Branch : *B.E.* – CSE

Year & Semester : III Year / VI Semester

CS6612 COMPILER LABORATORY



ANNA UNIVERSITY, CHENNAI
REGULATION – 2013
CS6612 – COMPILER LABORATORY

LIST OF EXPERIMENTS:

1. Implementation of symbol table.
2. Develop a lexical analyzer to recognize a few patterns in c (ex. Identifers, constants, comments, operators etc.)
3. Implementation of lexical analyzer using lex tool.
4. Generate yacc specification for a few syntatic categories.
   a) Program to recognize a valid arithmetic expression that uses operator +, -, * and /.
   b) Program to recognize a valid variable which starts with a letter followed by any number of letter or digits.
   c) Implementation of calculator using lex and yacc.
5. Convert the bnf rules into yacc form and write code to generate abstract syntax tree.
6. Implement type checking
7. Implement control flow analysis and data flow analysis.
8. Implement any one storage allocation strategies(heap, stack, static)
9. Construction of DAG
10. Implement the back end of the compiler which takes the three address code and produces the 8086nassembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move , add, sub, jump. Also simple addressing modes are used.
11. Implementation of simple code optimization techniques (constant folding. etc.)

TOTAL: 45 PERIODS

INDEX

EX. NO: 1
DATE:

IMPLEMENTATION OF SYMBOL TABLE

AIM:

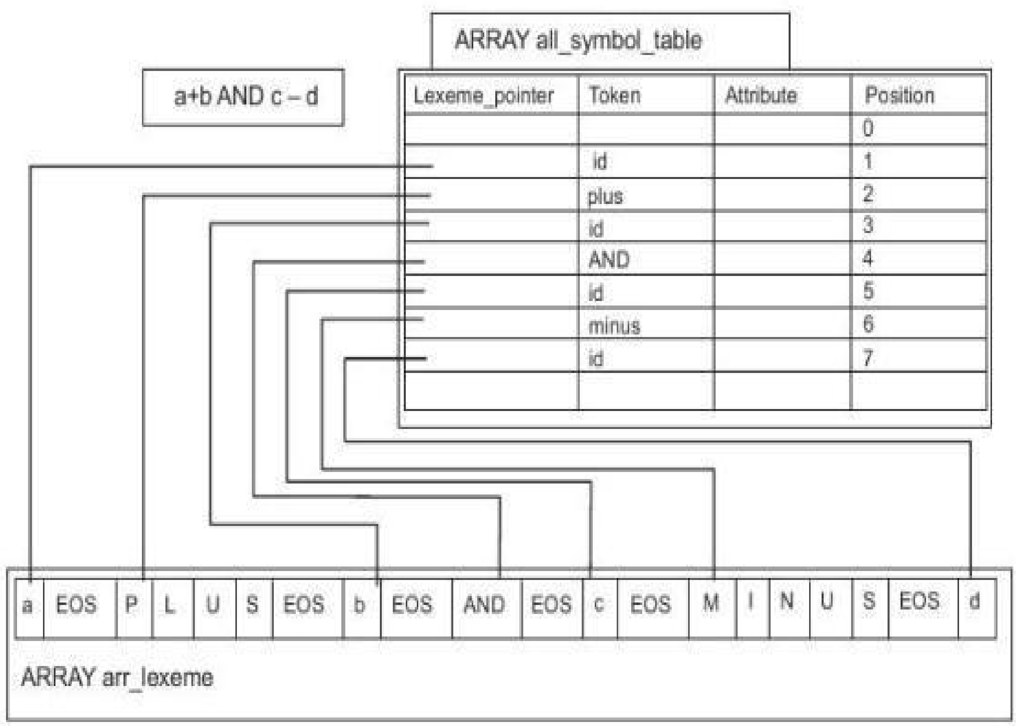
To write a C program to implement a symbol table.

INTRODUCTION:

A Symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source
Possible entries in a symbol table:

- Name : a string
- Attribute:
    1. Reserved word
    2. Variable name
    3. Type Name
    4. Procedure name
    5. Constant name
- Data type
- Scope information: where it can be used.
- Storage allocation

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 checked for corresponding variable, the variable along its address is displayed as result.
8. Stop the program.

PROGRAM: (IMPLEMENTATION OF SYMBOL TABLE)

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<string.h>
#include<math.h>
#include<ctype.h>
void main()
{
int i=0,j=0,x=0,n,flag=0; void *p,*add[15];
char ch,srch,b[15],d[15],c;
//clrscr();
printf("expression terminated by $:");
while((c=getchar())!='$')
{
b[i]=c; i++;
}
n=i-1;
printf("given expression:");
i=0;
```

```
while(i<=n)
{
printf("%c",b[i]); i++;
}
printf("symbol table\n");
 printf("symbol\taddr\ttype\n");
while(j<=n)
{
c=b[j]; if(isalpha(toascii(c)))
{
if(j==n)
{
p=malloc(c); add[x]=p;
d[x]=c;
printf("%c\t%d\tidentifier\n",c,p);
}
else
{
ch=b[j+1];
if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
{
p=malloc(c);
add[x]=p;
d[x]=c;
 printf("%c\t%d\tidentifier\n",c,p);
 x++;
}
}
} j++;
}
```

```
printf("the symbol is to be searched\n");
srch=getch();
for(i=0;i<=x;i++)
{
if(srch==d[i])
{
printf("symbol found\n");
printf("%c%s%d\n",srch,"@address",add[i]);
flag=1;
}
}
if(flag==0)
printf("symbol not found\n");
//getch();
```

OUTPUT:

```
expression terminated by $:a+b+c=d$
given expression:a+b+c=dsymbol table
symbol  addr    type
a       1892    identifier
b       1994    identifier
c       2096    identifier
d       2200    identifier
the symbol is to be searched
_
```

RESULT:

Thus the C program to implement the symbol table was executed and the output is verified.

EX. NO:2
DATE:

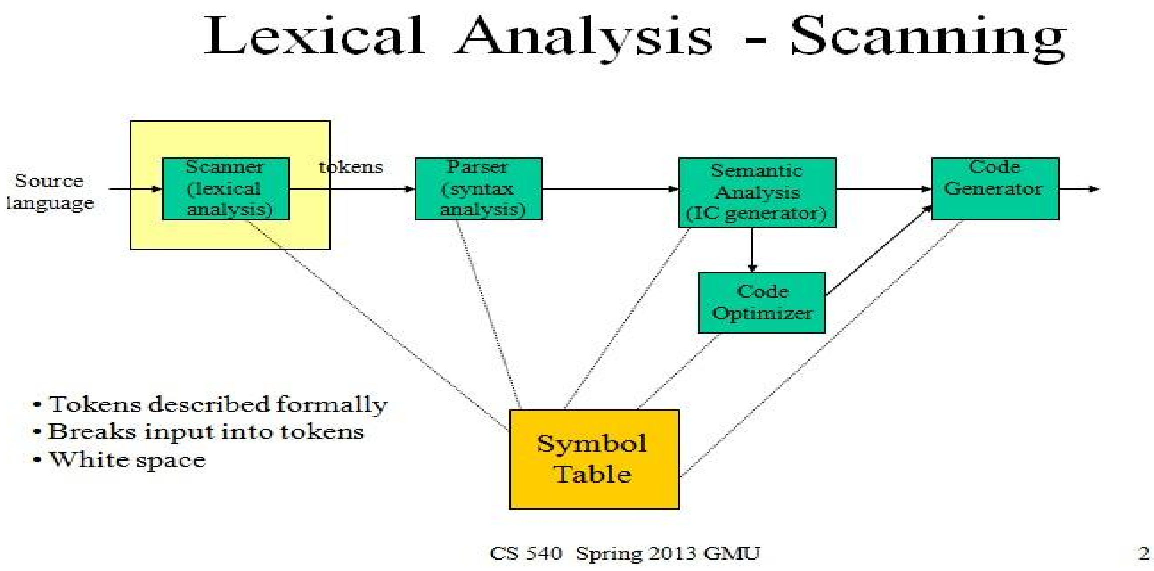
DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C

AIM:

To Write a C program to develop a lexical analyzer to recognize a few patterns in C.

INTRODUCTION:

Lexical analysis is the process of converting a sequence of characters (such as in a computer program of web page) into a sequence of tokens (strings with an identified "meaning"). A program that perform lexical analysis may be called a lexer, tokenize or scanner.



TOKEN

A token is a structure representing a lexeme that explicitly indicates its categorization for the Purpose of parsing. A category of token is what in linguistics might be called a part-of-speech. Examples of token categories may include "identifier" and "integer literal", although the set of Token differ in different programming languages.

The process of forming tokens from an input stream of characters is called tokenization.

Consider this expression in the C programming language:

Sum=3 + 2;

Tokenized and represented by the following table:

Lexeme	Token category
Sum	“identifier”
=	“assignment operator”
3	“integer literal”
+	“addition operator”
2	“integer literal”
;	“end of the statement”

ALGORITHM:

1. Start the program
2. Include the header files.
3. Allocate memory for the variable by dynamic memory allocation function.
4. Use the file accessing functions to read the file.
5. Get the input file from the user.
6. Separate all the file contents as tokens and match it with the functions.
7. Define all the keywords in a separate file and name it as key.c
8. Define all the operators in a separate file and name it as open.c
9. Give the input program in a file and name it as input.c
10. Finally print the output after recognizing all the tokens.
11. Stop the program.

PROGRAM: (DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C)

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
 void main()
{
FILE *fi,*fo,*fop,*fk;
int flag=0,i=1;
char c,t,a[15],ch[15],file[20];
clrscr();
printf("\n Enter the File Name:");
 scanf("%s",&file);
fi=fopen(file,"r");
```

```
fo=fopen("inter.c","w");

fop=fopen("oper.c","r");
fk=fopen("key.c","r");
c=getc(fi);
while(!feof(fi))
{
if(isalpha(c)||isdigit(c)||(c=='['||c==']'||c=='.'==1))
 fputc(c,fo);
else
{
if(c=='\n')
fprintf(fo,"\t$\t");
 else fprintf(fo,"\t%c\t",c);
}
c=getc(fi);
}
fclose(fi);
 fclose(fo);
fi=fopen("inter.c","r");
printf("\n Lexical Analysis");
fscanf(fi,"%s",a);
printf("\n Line: %d\n",i++);
while(!feof(fi))
{
if(strcmp(a,"$")==0)
{
printf("\n Line: %d \n",i++);
fscanf(fi,"%s",a);
}
fscanf(fop,"%s",ch);
 while(!feof(fop))
{
if(strcmp(ch,a)==0)
{
fscanf(fop,"%s",ch);
printf("\t\t%s\t:\t%s\n",a,ch);
flag=1;
```

```
} fscanf(fop,"%s",ch);
}
rewind(fop);
 fscanf(fk,"%s",ch);
while(!feof(fk))
{
if(strcmp(ch,a)==0)
{
fscanf(fk,"%k",ch);
printf("\t\t%s\t:\tKeyword\n",a);
flag=1;
}
fscanf(fk,"%s",ch);
}
rewind(fk);
 if(flag==0)
{
 if(isdigit(a[0]))
 printf("\t\t%s\t:\tConstant\n",a);
 else
printf("\t\t%s\t:\tIdentifier\n",a);
}
flag=0;
fscanf(fi,"%s",a); }
getch();
}
Key.C:
int
void
main
 char
 if
 for
while
else
 printf
scanf
 FILE
```

```
Include
 stdio.h
 conio.h
iostream.h
```

```
Oper.C:
( open para
) closepara
{ openbrace
} closebrace
< lesser
> greater
" doublequote ' singlequote
: colon
; semicolon
# preprocessor
= equal
== asign
% percentage
^ bitwise
& reference
* star
+ add
- sub
\ backslash
/ slash
```

```
Input.C:
#include "stdio.h"
#include "conio.h"
void main()
{
int a=10,b,c;
 a=b*c;
getch();
}
```

OUTPUT:

```
Enter the file name : input.c
                LEXICAL ANALYSIS

line : 1
                #        :       preprocessor
                include  :       keyword
                "        :       doublequote
                stdio.h  :       keyword
                "        :       doublequote

line : 2
                #        :       preprocessor
                include  :       keyword
                "        :       doublequote
                conio.h  :       keyword
                "        :       doublequote

line : 3
                void     :       keyword
                main     :       keyword
                (        :       openpara
                )        :       closepara

line : 4
                {        :       openbrace

line : 5
                int      :       keyword
                a        :       identifier
                =        :       equal
                10       :       constant
                ,        :       identifier
                b        :       identifier
                ,        :       identifier
                c        :       identifier
                ;        :       semicolon

line : 6
                a        :       identifier
                =        :       equal
                b        :       identifier
                *        :       star
                c        :       identifier
                ;        :       semicolon

line : 7
                getch    :       identifier
                (        :       openpara
                )        :       closepara
                ;        :       semicolon

line : 8
                }        :       closebrace

line : 9
                $        :       identifier
```

RESULT:

Thus the above program for developing the lexical the lexical analyzer and recognizing the few pattern s in C is executed successfully and the output is verified.

EX.NO:3
DATE:

IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

AIM:

To write a program to implement the Lexical Analyzer using lex tool.

INTRODUCTION:

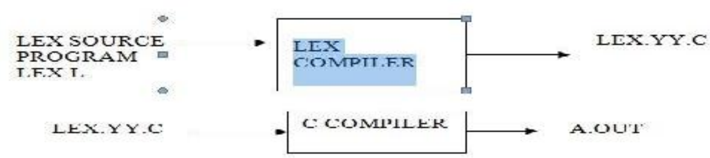
THEORY:

- A language for specifying lexical analyzer.
- There is a wide range of tools for construction of lexical analyzer. The majority of these tools are based on regular expressions.
- The one of the traditional tools of that kind is lex.

LEX:

- The lex is used in the manner depicted. A specification of the lexical analyzer is preferred by creating a program lex.1 in the lex language.
- Then lex.1 is run through the lex compiler to produce a 'c' program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.1 together with a standard routine that uses table of recognize leximes.
- Lex.yy.c is run through the 'C' compiler to produce as object program a.out, which is the lexical analyzer that transform as input stream into sequence of tokens.

LEX SOURCE:



ALGORITHM:

1. Start the program
2. Lex program consists of three parts.
3. Declaration %%
4. Translation rules %%
5. Auxiliary procedure.
6. The declaration section includes declaration of variables, main test, constants and regular
7. Definitions.
8. Translation rule of lex program are statements of the form
9. P1{action}
10. P2{action}
11. …..
12. …..
13. Pn{action}
14. Write program in the vi editor and save it with .1 extension.
15. Compile the lex program with lex compiler to produce output file as lex.yy.c.
16. Eg. $ lex filename.1
17. $gcc lex.yy.c-11
18. Compile that file with C compiler and verify the output.

PROGRAM: (LEXICAL ANALYZER USING LEX TOOL)

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
#include<string.h>
char vars[100][100];
int vcnt;
char input[1000],c;
char token[50],tlen;
int state=0,pos=0,i=0,id;
char *getAddress(char str[])
{
for(i=0;i<vcnt;i++)
if(strcmp(str,vars[i])==0)
```

```
return vars[i];
strcpy(vars[vcnt],str);
return vars[vcnt++];
}
int isrelop(char c)
{
if(c=='+'||c=='-'||c=='*'||c=='/'||c=='%'||c=='^')
return 1;
else
return 0;
}
int main(void)
{
clrscr();
printf("Enter the Input String:");
gets(input);
do
{
c=input[pos];
putchar(c);
switch(state)
{
case 0:
if(isspace(c))
printf("\b");
if(isalpha(c))
{
token[0]=c;
tlen=1;
state=1;
}
if(isdigit(c))
state=2;
if(isrelop(c))
state=3;
if(c==';')
printf("\t<3,3>\n");
if(c=='=')
```

```
printf("\t<4,4>\n");
break;
case 1:
if(!isalnum(c))
{
token[tlen]='\o';
printf("\b\t<1,%p>\n",getAddress(token));
state=0;
pos--;
}
else
token[tlen++]=c;
break;
case 2:
if(!isdigit(c))
{
printf("\b\t<2,%p>\n",&input[pos]);
state=0;
pos--;
}
break;
case 3:
id=input[pos-1];
if(c=='=')
printf("\t<%d,%d>\n",id*10,id*10);
else{
printf("\b\t<%d,%d>\n",id,id);
pos--;
}state=0;
break;
}
pos++;
}
while(c!=0);
getch();
return 0;
}
```

OUTPUT

RESULT:

Thus the program for the exercise on lexical analysis using lex has been successfully executed and output is verified.

EX.NO:4
DATE:

ENERATE YACC SPECIFICATION FOR A FEW SYNTACTIC CATEGORIES.

AIM :

To write a c program to do exercise on syntax analysis using YACC.

INTRODUCTION :

YACC (yet another compiler) is a program designed to produce designed to compile a LALR (1) grammar and to produce the source code of the synthetically analyses of the language produced by the grammar.

ALGORITHM :

1. Start the program.
2. Write the code for parser. l in the declaration port.
3. Write the code for the ‘y’ parser.
4. Also write the code for different arithmetical operations.
5. Write additional code to print the result of computation.
6. Execute and verify it.
7. Stop the program.

PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, - , * AND /.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{   char s[5];
   clrscr();
    printf("\n Enter any operator:");
    gets(s);
    switch(s[0])
    {
    case'>': if(s[1]=='=')
            printf("\n Greater than or equal");
             else
             printf("\n Greater than");
             break;
```

```
case'<': if(s[1]=='=')
         printf("\n Less than or equal");
         else
         printf("\nLess than");
         break;
case'=':  if(s[1]=='=')
         printf("\nEqual to");
         else
         printf("\nAssignment");
          break;
case'!':  if(s[1]=='=')
        printf("\nNot Equal");
         else
         printf("\n Bit Not");
         break;
case'&':  if(s[1]=='&')
         printf("\nLogical AND");
         else
         printf("\n Bitwise AND");
         break;
case'|':  if(s[1]=='|')
         printf("\nLogical OR");
          else
         printf("\nBitwise OR");
         break;
case'+':  printf("\n Addition");
         break;
case'-':  printf("\nSubstraction");
         break;
case'*':  printf("\nMultiplication");
         break;
case'/':  printf("\nDivision");
         break;
case'%': printf("Modulus");
         break;
default:  printf("\n Not a operator");    }    getch();  }
```

OUTPUT:

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

EX.NO:5
DATE:

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

PROGRAM :

variable_test.l

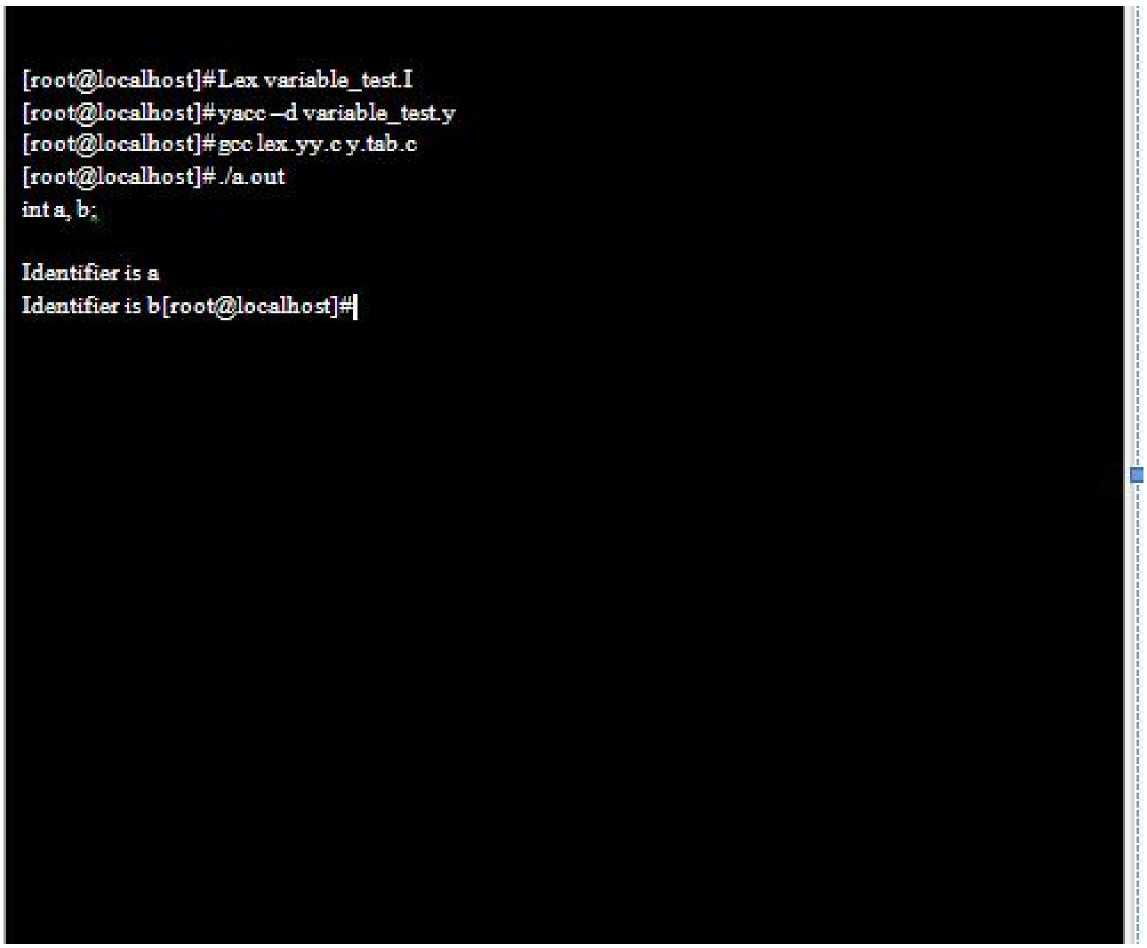
```
%{
/* This LEX program returns the tokens for the Expression */
#include "y.tab.h"
%}
%%
"int " {return INT;}
"float" {return FLOAT;}
"double" {return DOUBLE;}
[a-zA-Z]*[0-9]*{
printf("\nIdentifier is %s",yytext);
return ID;
}
return yytext[0];
\n return 0;
int yywrap()
{
return 1;
}
```

variable_test.y

```
%{
#include
/* This YACC program is for recognising the Expression*/
%}
%token ID INT FLOAT DOUBLE
%%
D;T L
;
L:L,ID
|ID
;
T:INT
|FLOAT
|DOUBLE
;
%%
extern FILE *yyin;
main()
```

```
{
do
{
yyparse();
}while(!feof(yyin));
}
yyerror(char*s)
{
}
```

OUTPUT:



RESULT:

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

EX.NO.6
DATE:

IMPLEMENTATION OF CALCULATOR USING LEX AND YACC

PROGRAM:

```
%{
#include<stdio.h>
int op=0,i;
float a,b;
%}
dig[0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul"*"
div "/"
pow "^"
ln \n
%%
{dig}{digi();}
{add}{op=1;}
{sub}{op=2;}
{mul}{op=3;}
{div}{op=4;}
{pow}{op=5;}
{ln}{printf("\n the result:%f\n\n",a);}
%%
digi()
{
if(op==0)
a=atof(yytext);
```

```
else
{
b=atof(yytext);
switch(op)
{
case 1:a=a+b;
break;
case 2:a=a-b;
break;
case 3:a=a*b;
break;
case 4:a=a/b;
break;
case 5:for(i=a;b>1;b--)
a=a*i;
break;
}
op=0;
}
}
main(int argv,char *argc[])
{
yylex();
}
yywrap()
{
return 1;
}
```

OUTPUT:

Lex cal.l

Cc lex.yy.c-ll

a.out

4*8

The result=32

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed Successfully and Output is verified.

EX.NO:7
DATE:

IMPLEMENTATION OF TYPE CHECKING

AIM:

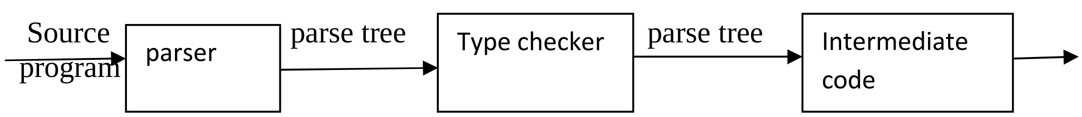
To write a C program for implementing type checking for given expression.

INTRODUCTION:

The type analysis and type checking is an important activity done in the semantic analysis phase. The need for type checking is

1. To detect the errors arising in the expression due to incompatible operand.
2. To generate intermediate code for expressions due to incompatible operand

Role of type checker



ALGORITHM:

1. Start a program.
2. Include all the header files.
3. Initialize all the functions and variables.
4. Get the expression from the user and separate into the tokens.
5. After separation, specify the identifiers, operators and number.
6. Print the output.
7. Stop the program.

PROGRAM: (TYPE CHECKING)

```
#include<stdio.h>
char str[50],opstr[75];
int f[2][9]={2,3,4,4,4,0,6,6,0,1,1,3,3,5,5,0,5,0};
int col,col1,col2;
char c;
swt()
{
   switch(c)
   {
```

```
        case'+':col=0;break;
        case'-':col=1;break;
        case'*':col=2;break;
        case'/':col=3;break;
        case'^':col=4;break;
        case'(':col=5;break;
        case')':col=6;break;
        case'd':col=7;break;
        case'$':col=8;break;
        default:printf("\nTERMINAL MISSMATCH\n");
         exit(1);
   }
  // return 0;
}
main()
{
   int i=0,j=0,col1,cn,k=0;
   int t1=0,foundg=0;
   char temp[20];
   clrscr();
   printf("\nEnter arithmetic expression:");
   scanf("%s",&str);
   while(str[i]!='\0')
     i++;
   str[i]='$';
   str[++i]='\0';
   printf("%s\n",str);
   come:
   i=0;
   opstr[0]='$';
   j=1;
   c='$';
   swt();
   col1=col;
   c=str[i];
   swt();
   col2=col;
```

```
        if(f[1][col1]>f[2][col2])
        {
          opstr[j]='>';
          j++;
        }
      else if(f[1][col1]<f[2][col2])
      {
           opstr[j]='<';
           j++;
       }
       else
       {
           opstr[j]='=';j++;
       }

       while(str[i]!='$')
       {
           c=str[i];
              swt();
            col1=col;
            c=str[++i];
            swt();
            col2=col;
            opstr[j]=str[--i];
            j++;
            if(f[0][col1]>f[1][col2])
            {
               opstr[j]='>';
               j++;
            }
            else if(f[0][col1]<f[1][col2])
            {
              opstr[j]='<';
               j++;
            }
            else
         {
            opstr[j]='=';j++;
```

```
        }
        i++;
    }
    opstr[j]='$';
    opstr[++j]='\0';
    printf("\nPrecedence Input:%s\n",opstr);
    i=0;
    j=0;
    while(opstr[i]!='\0')
    {
        foundg=0;
         while(foundg!=1)
         {
           if(opstr[i]=='\0')goto redone;
           if(opstr[i]=='>')foundg=1;
           t1=i;
           i++;
        }
         if(foundg==1)
         for(i=t1;i>0;i--)
          if(opstr[i]=='<')break;
          if(i==0){printf("\nERROR\n");exit(1);}
         cn=i;
          j=0;
          i=t1+1;
         while(opstr[i]!='\0')
         {
             temp[j]=opstr[i];
            j++;i++;
           }
          temp[j]='\0';
          opstr[cn]='E';
          opstr[++cn]='\0';
          strcat(opstr,temp);
          printf("\n%s",opstr);
           i=1;
       }
        redone:k=0;
```

```
while(opstr[k]!='\0')
     {
        k++;
        if(opstr[k]=='<')
       {
   Printf("\nError");
   exit(1);
        }
   }
   if((opstr[0]=='$')&&(opstr[2]=='$'))goto sue;
   i=1
   while(opstr[i]!='\0')
   {
      c=opstr[i];
      if(c=='+'||c=='*'||c=='/'||c=='$')
      {
 temp[j]=c;j++;}
 i++;
     }
      temp[j]='\0';
      strcpy(str,temp);
      goto come;
      sue:
      printf("\n success");
      return 0;
   }
```

OUTPUT:

```
Enter arithmetic expression:(d*d)+d$
(d*d)+d$$

Precedence Input:$<(<d>*<d>)>+<d>$

$<(E*<d>)>+<d>$
$<(E*E)>+<d>$
$E+<d>$
$E+E$
Precedence Input:$<$

Error
```

```
Enter arithmetic expression:(d*d)$
(d*d)$$

Precedence Input:$<(<d>*<d>)>$

$<(E*<d>)>$
$<(E*E)>$
$E$
 success_
```

RESULT:

Thus the program has been executed successfully and Output is verified.

EX.NO:8
DATE:

CONVERT THE BNF RULES INTO YACC FORM AND WRITE CODE TO GENERATE ABSTRACT SYNTAX TREE USING AND YACC.

AIM:

To write a program to convert the BNF rules into YACC

INTRODUCTION:

BNF-Backus Naur form is formal notationfor encoding grammars intended for human Consumption. Many programming languages, protocol or formats have BNF description in their Specification.

ALGORITHM:

1. Start the program.
2. Declare the declarations as a header file.
   {include<ctype.h>}
3. Token digit
4. Define the translations rule like line,expr,term,factor.
   Line:exp"\n"{print"\n%d\n",$1)}
   Expr:expr"+"term($$=$1=$3}
   Term:term"+"factor($$=$1*$3}
   Factor
   Factor☹"enter"),{$$=$2)
   %%
5. Define the supporting C routines.
6. Execute and verify it.
7. Stop the program.

PROGRAM: (CONVERT THE BNF RULES INTO YACC)

```
<int.l>
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
% }
identifier [a- zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0- 9]*\.[0-9]+)
%%
main\(\) return MAIN;
if  return  IF;
else return  ELSE;
while return  WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return  VAR;}
{number} {strcpy(yylval.var,yytext);
return  NUM;}
\< |
\> |
\>=  |
\<=  |
==  {strcpy(yylval.var,yytext);
return  RELOP;}
[ \t] ;
```

```
\n  LineNo++;

 return yytext[0];
%%
<int.y>
%{
#include<string.h>
#include<stdio.h>
struct  quad
{
   char op[5];
   char arg1[10];
   char arg2[10];
   char result[10];
   }QUAD[30];
   struct  stack
  {
   int  items[100];
   int  top;
   }stk;
   int Index=0,tIndex=0,StNo,Ind,tInd;
   extern  int LineNo;
   %}
  %union
 {
    char var[10];
}
%token <var>  NUM VAR  RELOP
%token MAIN  IF ELSE  WHILE TYPE
%type <var>  EXPR ASSIGNMENT CONDITION IFST  ELSEST WHILELOOP
%left '-' '+'
```

```
%left '*' '/'

%%

PROGRAM : MAIN BLOCK

;

BLOCK: '{' CODE '}'

;

CODE:  BLOCK

| STATEMENT CODE

| STATEMENT

;

STATEMENT:  DESCT ';'

| ASSIGNMENT ';'

| CONDST

| WHILEST

;

DESCT:  TYPE  VARLIST

;

VARLIST: VAR ',' VARLIST

| VAR

;

ASSIGNMENT:  VAR '=' EXPR{

strcpy(QUAD[Index].op,"=");

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"");

strcpy(QUAD[Index].result,$1);

strcpy($$,QUAD[Index++].result);

}

;

EXPR: EXPR '+'  EXPR {AddQuadruple("+",$1,$3,$$);}

| EXPR '-' EXPR  {AddQuadruple("-",$1,$3,$$);}
```

```
| EXPR '*'  EXPR {  AddQuadruple("*",$1,$3,$$);}

| EXPR '/'  EXPR { AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {  AddQuadruple("UMIN",$2,"",$$);}

| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST  ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"- 1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"- 1");
push(Index);
```

```
Index++;
};
ELSEST:  ELSE{
tInd=pop();
Ind=pop();

push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index- 1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"- 1");
```

```
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");

strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"- 1");
push(Index);
Index++;
}
;
%%
extern  FILE *yyin;
int main(int  argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not  found");
exit(0);
}
yyin=fp;
}
yyparse();
```

```
printf("\n\n\t\t ----------------------------""\n\t\t
Pos Operator Arg1  Arg2  Result" "\n\t\t --------------------
");


for(i=0;i<Index;i++)

{

printf("\n\t\t %d\t %s\t  %s\t %s\t

%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);

}

printf("\n\t\t -----------------------");

printf("\n\n");

return  0;

}

void push(int data)

{

stk.top++;

if(stk.top==100)

{

printf("\n Stack  overflow\n");

exit(0);

}

stk.items[stk.top]=data;

}

int pop()

{

int data;

if(stk.top==- 1)

{

printf("\n Stack  underflow\n");

exit(0);

}
```

```
data=stk.items[stk.top--];

return  data;

}

void AddQuadruple(char op[5],char arg1[10],char  arg2[10],char
result[10])

{

strcpy(QUAD[Index].op,op);

strcpy(QUAD[Index].arg1,arg1);

strcpy(QUAD[Index].arg2,arg2);

sprintf(QUAD[Index].result,"t%d",tIndex++);

strcpy(result,QUAD[Index++].result);

}

yyerror()

{

printf("\n Error  on line no:%d",LineNo);

}
```

Input:

```
$vi  test.c

main()

{

int a,b,c;

if(a<b)

{

a=a+b;

}

while(a<b)

{ a=a+b;

}

if(a<=b)

{ c=a- b;

}
```

```
else
{ c=a+b;
}
}
```

OUTPUT:

$ lex int.l

$ yacc –d int.y

$ gcc lex.yy.c y.tab.c –ll –lm

$ ./a.out test.c

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	to
1	==	to	FALSE	5
2	+	a	b	t1
3	=	t1		a
4	GOTO			5
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t3
16	=	t6		c

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed successfully and output is verified.

EX.NO:9
DATE:

IMPLEMENT CONTROL FLOW ANALYSIS AND DATA FLOW ANALYSIS

AIM:

To Writs a C program to implement data flow and control flow analysis.

INTRODUCTION:

- Data flow analysis is a technique for gathering information about the possible set of value calculated at various points in a computer program.
- Control flow analysis can be represent by basic blocks. It depicts how th program control is being passed among the blocks.

ALGORITHM:

1. Start the program
2. Declare the necessary variables
3. Get the choice to insert, delete and display the values in stack
4. Perform PUSH() operation
   a. t = newnode()
   b. Enter info to be inserted
   c. Read n
   d. t ->info= n
   e. t ->next=top
   f. top = t
   g. Return
5. Perform POP() operation
   a. If (top=NULL)
   b. Print”underflow”
   c. Return
   d. X=top
   e. Top=top->next
   f. Delnode(x)
   g. Return
6. Display the values
7. Stop the program.

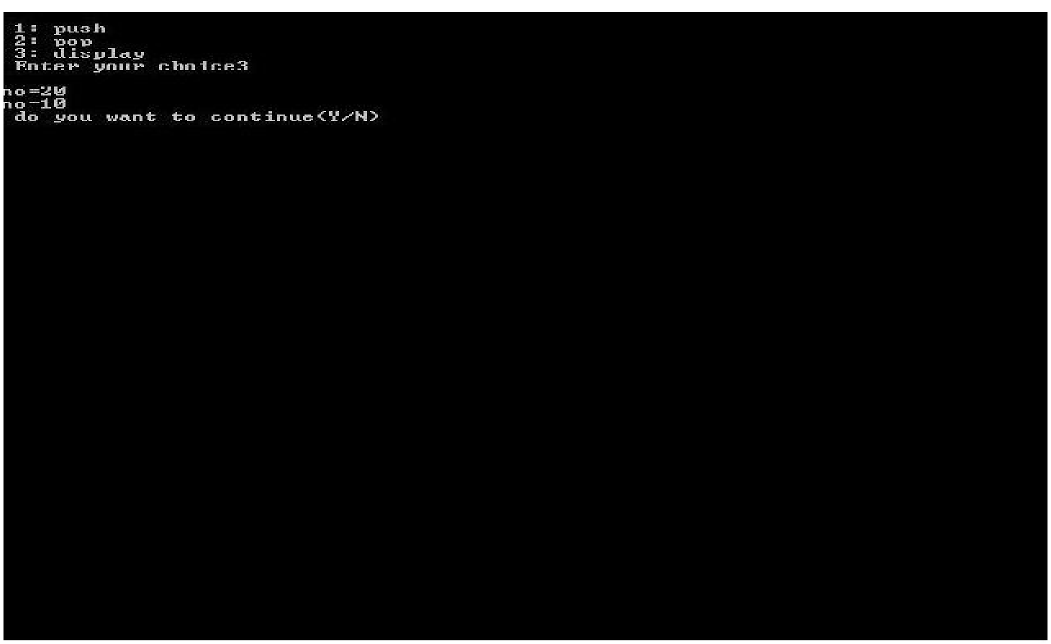
PROGRAM: (DATA FLOW AND CONTROL FLOW ANALYSIS)

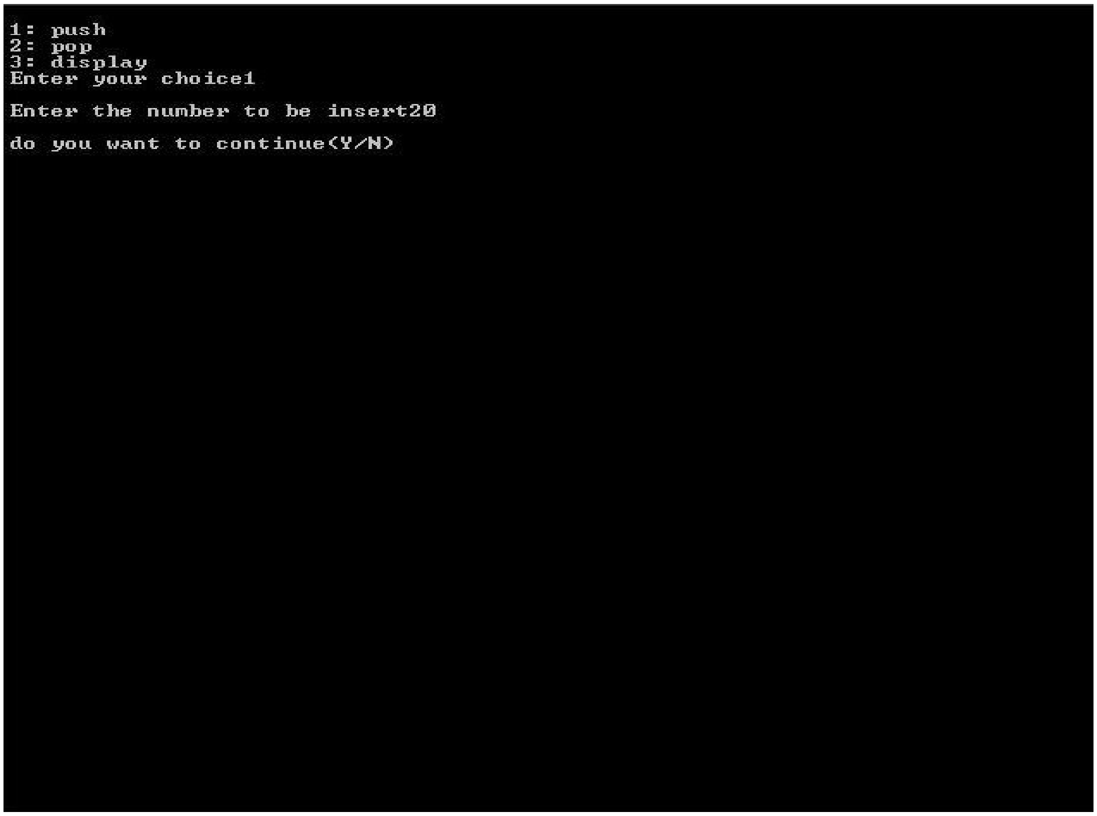
```
#include<conio.h>
struct stack
{
int no;
struct stack *next;
}
*start=null
typedef struct stack st;
voidpush();
int pop();
voiddisplay();
voidmain()
{
char ch;
int choice, item;
do
{
clrscr();
printf(“\n1:push”);
printf(“\n2:pop”);
printf(“\n3:display”);
printf(“\n enter your choice”);
scanf(“%d”,&choice);
switch(choice)
{
case1:push();
break;
case2:item=pop();
printf(“the delete element in %d”,item);
break;
case3:display();
break;
default:printf(“\nwrong choice”);
};
```

```
printf("\n do you want to continue(y/n”);
fflush(stdin);
scanf("%c",&ch);
}
while(ch=='y'||ch=='y');
}
voidpush()
{
st*node;
node=(st*)malloc(sizeof(st));
printf("\n enter the number to be insert");
scanf("%d",&node->no);
node->next=start;
start=node;
}
intpop();
{
st*temp;
temp=start;
if(start==null)
{
printf("stack is already empty");
getch();
exit();
}
else
{
start=start->next;
free(temp);
}
return(temp->no);
}
void display()
{
st*temp;
temp=start;
while(temp->next!=null)
{
```

```
printf("\nno=%d",temp->no);
temp=temp->next;
}
printf("\nno=%d",temp->no);
}
```

OUTPUT:





RESULT:

Thus the C program to implement data flow and control flow analysis was executed successfully.

EX.NO:10
DATE:

IMPLEMENT ANY ONE STORAGE ALLOCATION STRATEGIES (HEAP,STACK,STATIC)

AIM:

To write a C program for Stack to use dynamic storage allocation.

INTRODUCTION:

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- Code: It is known as the part of a program that does not change at runtime. Its memory requirements are at the compile time
- Procedures: Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- Variables: Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

ALGORITHM:

1. Start the program
2. Enter the expression for which intermediate code is to be generated
3. If the length of the string is greater than 3, than call the procedure to return the precedence
4. Among the operands.
5. Assign the operand to exp array and operators to the array.
6. Create the three address code using quadruples structure.
7. Reduce the no of temporary variables.
8. Continue this process until we get an output.
9. Stop the program.

PROGRAM: (STACK TO USE DYNAMIC STORAGE ALLOCATION)

```
#include <stdio.h>

#include <conio.h>

#include <process.h>

#include <alloc.h>

struct node

{

int label;
```

```
struct node *next;
};
void main()
{
int ch = 0;
int k;
struct node *h, *temp, *head;
head = (struct node*) malloc(sizeof(struct node));
head->next = NULL;
while(1)
{
printf("\n Stack using Linked List \n");
printf("1->Push ");
printf("2->Pop ");
printf("3->View");
printf("4->Exit \n");
printf("Enter your choice : ");
scanf("%d", &ch);
switch(ch)
{
case 1:
temp=(struct node *)(malloc(sizeof(struct node)));
printf("Enter label for new node : ");
scanf("%d", &temp->label);
h = head;
temp->next = h->next;
h->next = temp;
break;
case 2:
```

```
h = head->next;
head->next = h->next;
printf("Node %s deleted\n", h->label);
free(h);
break;
case 3:
printf("\n HEAD -> ");
h = head;
while(h->next != NULL)
{
h = h->next;
printf("%d -> ",h->label);
}
printf("NULL \n");
break;
case 4:
exit(0);
}
}}
```

OUTPUT:

```
 Stack using Linked List
1->Push 2->Pop 3->View4->Exit
Enter your choice : 1
Enter label for new node : 23

 Stack using Linked List
1->Push 2->Pop 3->View4->Exit
Enter your choice : 1
Enter label for new node : 45

 Stack using Linked List
1->Push 2->Pop 3->View4->Exit
Enter your choice : 2
Node . deleted

 Stack using Linked List
1->Push 2->Pop 3->View4->Exit
Enter your choice : 3

 HEAD -> 23 -> NULL

 Stack using Linked List
1->Push 2->Pop 3->View4->Exit
Enter your choice :
```

RESULT:

Thus the program for implement storage allocation to use dynamic process for stack has been successfully executed

EX.NO:11
DATE:

CONSTRUCTION OF DAG

AIM:

To write a C program to construct of DAG(**Directed Acyclic Graph**)

INTRODUCTION:

The code optimization is required to produce an efficient target code. These are two important issues that used to be considered while applying the techniques for code optimization.

They are:

- The semantics equivalences of the source program must not be changed.
- The improvement over the program efficiency must be achieved without changing the algorithm.

ALGORITHM:

1. Start the program
2. Include all the header files
3. Check for postfix expression and construct the in order DAG representation
4. Print the output
5. Stop the program

PROGRAM: (TO CONSTRUCT OF DAG(DIRECTED ACYCLIC GRAPH))

```
#include<stdio.h>
main()
{
struct da
{
int ptr,left,right;
char label;
}dag[25];
int ptr,l,j,change,n=0,i=0,state=1,x,y,k;
char store,*input1,input[25],var;
clrscr();
for(i=0;i<25;i++)
{
```

```
dag[i].ptr=NULL;

dag[i].left=NULL;

dag[i].right=NULL;

dag[i].label=NULL;

}

printf("\n\nENTER THE EXPRESSION\n\n");

scanf("%s",input1);

/*EX:((a*b-c))+((b-c)*d)) like this give with paranthesis.limit
is 25 char ucan change that*/

for(i=0;i<25;i++)

input[i]=NULL;

l=strlen(input1);

a:

for(i=0;input1[i]!=')';i++);

for(j=i;input1[j]!='(';j--);

for(x=j+1;x<i;x++)

if(isalpha(input1[x]))

input[n++]=input1[x];

else

if(input1[x]!='0')

store=input1[x];

input[n++]=store;

for(x=j;x<=i;x++)

input1[x]='0';

if(input1[0]!='0')goto a;

for(i=0;i<n;i++)

{

dag[i].label=input[i];

dag[i].ptr=i;

if(!isalpha(input[i])&&!isdigit(input[i]))

{

dag[i].right=i-1;

ptr=i;
```

```
var=input[i-1];
if(isalpha(var))
ptr=ptr-2;
else
{
ptr=i-1;
b:
if(!isalpha(var)&&!isdigit(var))
{
ptr=dag[ptr].left;
var=input[ptr];
goto b;
}
else
ptr=ptr-1;
}
dag[i].left=ptr;
}
}
printf("\n SYNTAX TREE FOR GIVEN EXPRESSION\n\n");
printf("\n\n PTR \t\t LEFT PTR \t\t RIGHT PTR \t\t LABEL
\n\n");
for(i=0;i<n;i++)/* draw the syntax tree for the following
output with pointer value*/
printf("\n
%d\t%d\t%d\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].la
bel);
getch();
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if((dag[i].label==dag[j].label&&dag[i].left==dag[j].left)&&dag[
i].right==dag[j].right)
{
```

```
for(k=0;k<n;k++)
{
if(dag[k].left==dag[j].ptr)dag[k].left=dag[i].ptr;
if(dag[k].right==dag[j].ptr)dag[k].right=dag[i].ptr;
}
dag[j].ptr=dag[i].ptr;
}
}
}
printf("\n DAG FOR GIVEN EXPRESSION\n\n");
printf("\n\n PTR \t LEFT PTR \t RIGHT PTR \t LABEL \n\n");
for(i=0;i<n;i++)/*draw DAG for the following output with
pointer value*/
printf("\n %d
\t\t%d\t\t%d\t\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i
].label);
getch();
}
```

OUTPUT:

```
ENTER THE EXPRESSION
((a*(b-c))+((b-c)*d))
 SYNTAX TREE FOR GIVEN EXPRESSION

 PTR            LEFT PTR                 RIGHT PTR                LABEL

 0      0       0       b
 1      0       0       c
 2      0       1       -
 3      0       0       a
 4      2       3       *
 5      0       0       b
 6      0       0       c
 7      5       6       -
 8      0       0       d
 9      7       8       *
 10     4       9       +
```

RESULT:

Thus the program for implementation of DAG has been successfully executed and output is verified.

EX.NO.12
DATE:

IMPLEMENT THE BACK END OF THE COMPILER

AIM:

To implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.

INTRODUCTION:

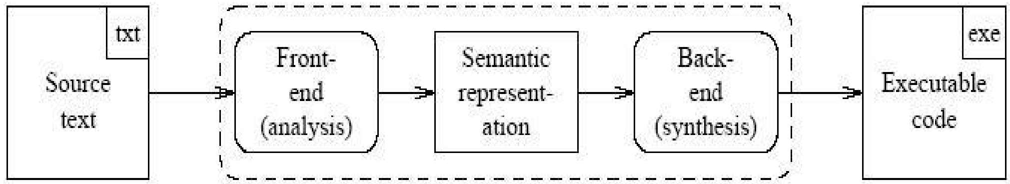
A compiler is a computer program that implements a programming language specification to “translate” programs, usually as a set of files which constitute the source code written in source language, into their equivalent machine readable instructions(the target language, often having a binary form known as object code). This translation process is called compilation.

BACK END:

- Some local optimization
- Register allocation
- Peep-hole optimization
- Code generation
- Instruction scheduling

The main phases of the back end include the following:

- Analysis: This is the gathering of program information from the intermediate representation derived from the input; data-flow analysis is used to build use-define chains, together with dependence analysis, alias analysis, pointer analysis, escape analysis etc.
- Optimization: The intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are expansion, dead, constant, propagation, loop transformation, register allocation and even automatic parallelization.
- Code generation: The transformed language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated modes. Debug data may also need to be generated to facilitate debugging.



ALGORITHM:

1. Start the program
2. Open the source file and store the contents as quadruples.
3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.
4. Write the generated code into output definition of the file in outp.c
5. Print the output.
6. Stop the program.

PROGRAM: (BACK END OF THE COMPILER)

```
#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\tMov R%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*e
exit

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

RESULT:

Thus the program was implemented to the TAC has been successfully executed.

EX.NO:13
DATE:

IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES

AIM:

To write a C program to implement simple code optimization technique.

INTRODUCTION:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (i.e. CPU, memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low level programming codes. A code optimizing process must follow the three rules given below:

The output code must not, in any way, change the meaning of the program.

- Optimization should increases the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and cpu registers.

Optimization can be categorized broadly into two types: Machine independent and Machine dependent.

Machine independent optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

For Example:

```
do
{
    item=10;
    value=value+item;
  }while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
item=10;
 do
   {
    value=value+item;
   }while(value<100);
```

Should not only save the cpu cycles, but can be used on any processor.

Machine dependent optimization

Machine dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

ALGORITHM:

1. Start the program
2. Declare the variables and functions.
3. Enter the expressionand state it in the variable a, b, c.
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.

PROGRAM: (SIMPLE CODE OPTIMIZATION TECHNIQUE)

Before:

Using for :

```
#include<iostream.h>
#include <conio.h>
int main()
{
int i, n;
int fact=1;
cout<<"\nEnter a number: ";
```

```
cin>>n;

for(i=n;i>=1;i--)

fact=fact *i;

cout<<"The factoral value is: "<<fact;

getch();

return 0;

}
```

OUTPUT:

After: (SIMPLE CODE OPTIMIZATION TECHNIQUE)

Using do-while:

```
#include<iostream.h>

#include<conio.h>
void main()

{

clrscr();
int n,f;

f=1;

cout<<"Enter the number:\n";

cin>>n;

do

{

f=f*n;

n--;

}while(n>0);

cout<<"The factorial value is:"<<f;

getch();

}
```

OUTPUT:

RESULT:

Thus the Simple Code optimization technique is successfully executed