

SRM Valliammai Engineering College

Kattankulathur

Department of Computer Science and Engineering



1904611 - COMPILER DESIGN LABORATORY

LAB MANUAL

ACADEMIC YEAR 2024-2025

YEAR : III Yr B.E CSE

SEMESTER : 6

Prepared by

Ms.V.Vijaypriya,Assistant Professor/CSE
Ms.Christina Sweetline, Assistant Professor/CSE
Dr.K.Shanmugam, Assistant Professor/CSE

1904611

COMPILER DESIGN LABORATORY

L T P C

0 0 4 2

COURSE OBJECTIVES :

- Deepen the understanding of compiler design.
- To implement Lexical Analyzer using Lex tool & Syntax Analyzer or parser using YACC Tool.
- To implement NFA and DFA from a given regular expression.
- To implement front end of the compiler by means of generating Intermediate codes
- To implement code optimization techniques

List of experiments

1. Implementation of Symbol Table
2. Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)
3. Implementation of Lexical Analyzer using Lex Tool
4. Generate YACC specification for a few syntactic 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 letters 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)
9. Construction of DAG
10. Implementation of Simple Code Optimization Techniques.

TOTAL : 60 PERIODS

TABLE OF CONTENT

EX. NO	List Of Experiments	Page NO
1	IMPLEMENTATION OF SYMBOL TABLE	5
2	DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C	9
3	IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL	13
4A	RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, -, * AND / USING YACC	18
4B	RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS USING YACC	23
4C	IMPLEMENTATION OF CALCULATOR USING LEX AND YACC	25
5	CONVERT THE BNF RULES INTO YACC FORM AND WRITE CODE TO GENERATE ABSTRACT SYNTAX TREE.	30
6	IMPLEMENT TYPE CHECKING	39
7	STORAGE ALLOCATION STRATEGIES : STACK IMPLEMENT	42
8	CONSTRUCTION OF DAG	49
9	IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUE	52
10A	IMPLEMENTATION OF CONTROL FLOW ANALYSIS	56
10B	IMPLEMENTATION OF DATA FLOW ANALYSIS	63



EX. NO:1

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

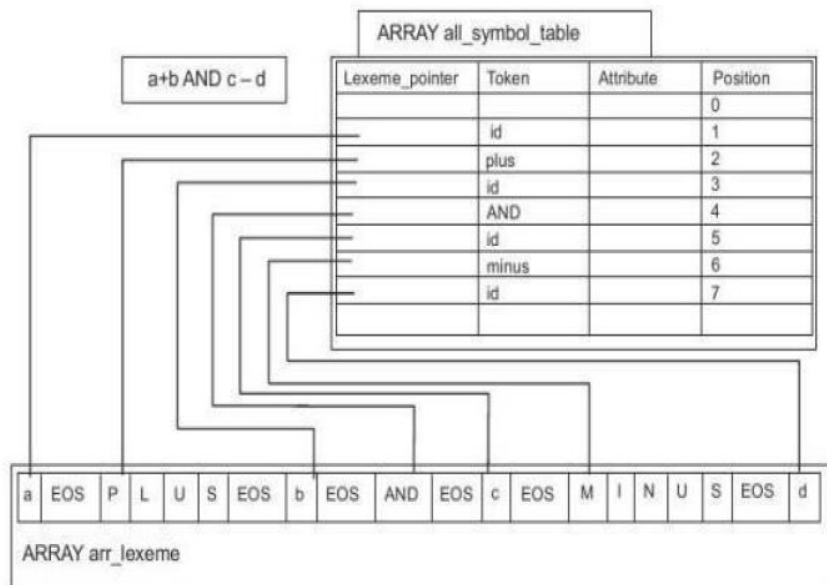
Attribute:

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



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 search 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:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.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("\ngiven expression:");
    i=0;
    while(i<=n)
    {
        printf("%c",b[i]);
        i++;
    }
}
```



```
printf("\nsymbol table\n");
printf("\nsymbol\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=='*'||ch=='=')
            {
                p=malloc(c);
                add[x]=p;
                d[x]=c;
                printf("%c\t%d\tidentifier\n",c,p);
                x++;
            }
        }
    }
    j++;
}
printf("Enter the identifier to be searched\n");
srch=getch();
for(i=0;i<=x;i++)
{
    if(srch==d[i])
    {
        printf("symbol found\n");
        printf("%c @Address %d \n",srch,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=d$
symbol 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

**DEVELOP A LEXICAL ANALYZER TO RECOGNIZE
A FEW PATTERNS IN C**

AIM:

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

INTRODUCTION:

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

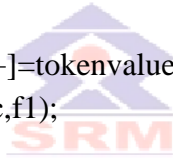
ALGORITHM:

1. Read the C program. Press Ctrl+z for end of the program
2. Check whether input is alphabet or digits then store it as identifier
3. If the input is an operator store it as symbol
4. Check the input matches with the keywords provided, display them as keywords
5. Display the symbols, identifiers, keywords and the number of lines in a program

PROGRAM:

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
#include<conio.h>
void keyword(char str[10])
{
    if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strc
mp("static",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0||strcmp("void",str)==0||strem
p("printf",str)==0 )
        printf("\n%s is a keyword",str);
    else
        printf("\n%s is an identifier",str);
}
void main()
{
    FILE *f1,*f2,*f3;
    char c,str[10],st1[10];
    int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
    clrscr();
```

```
printf("\nEnter the c program");
f1=fopen("input","w");
while((c=getchar())!=EOF)
    putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF)
{
    if(isdigit(c))
    {
        tokenvalue=c-'0';
        c=getc(f1);
        while(isdigit(c))
        {
            tokenvalue*=10+c-'0';
            c=getc(f1);
        }
        num[i++]=tokenvalue;
        ungetc(c,f1);
    }
    else if(isalpha(c))
    {
        putc(c,f2);
        c=getc(f1);
        while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
        {
            putc(c,f2);
            c=getc(f1);
        }
        putc(' ',f2);
        ungetc(c,f1);
    }
    else if(c==' '||c=='\t')
        printf(" ");
    else if(c=="\n")
        lineno++;
    else
        putc(c,f3);
}
```



```
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\nThe no's in the program are-->");
for(j=0;j<i;j++)
    printf(" %d",num[j]);
printf("\n");
f2=fopen("identifier","r");
k=0;
printf("The keywords and identifiers are:");
while((c=getc(f2))!=EOF)
{
    if(c!=' ')
        str[k++]=c;
    else
    {
        str[k]='\0';
        keyword(str);
        k=0;
    }
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\nSpecial characters are:");
while((c=getc(f3))!=EOF)
    printf("%c\t",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:-->%d",lineno);
getch();
}
```



OUTPUT:

```
Enter the c program
void main()
{
    int num1,num2;
    float cal;
    cal=num1+num2/13;
}→

The no's in the program are--> 13
The keywords and identifiers are:
void is a keyword
main is an identifier
int is a keyword
num1 is an identifier
num2 is an identifier
float is a keyword
cal is an identifier
cal is an identifier
num1 is an identifier
num2 is an identifier
Special characters are:(      )      {      ,      ;      =
+      /      ;      }
Total no. of lines are:-->6
```

Activate Windows
Go to Settings to activate Windows.



RESULT:

Thus the C program to develop a lexical analyser for recognizing a few patterns in c was executed and the output is verified.

EX NO: 3

IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

AIM:

To write a program for implementing a Lexical analyser using LEX tool.

STRUCTURE OF LEX PROGRAM:

1. Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

The format is as follows:

definitions

%%

rules

%%

user_subroutines

2. In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.
3. In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.
4. Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.
5. When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.
6. In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.
7. The lex command uses the rules and actions contained in file to generate a program,lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

RESOURCE:

Install : flex-2.5.4a-1, tdm-gcc-webdl. After installation, copy the following paths in environment variable “ Path”

→ C:\TDM-GCC-64\bin

→ C:\Program Files (x86)\GnuWin32\bin

ALGORITHM:

1. Read the input string.
2. Check whether the string is identifier/ keyword /symbol by using the rules of identifier and keywords using LEX Tool

PROGRAM:

```
% {
  /* program to recognize a c program */
  int COMMENT=0;
% }

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

%%

#.* { printf("\n %s is a PREPROCESSOR DIRECTIVE", yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
```



```
else |
goto    { printf("\n \t %s is a KEYWORD", yytext);}

"/*" {COMMENT = 1;}

"*/" {COMMENT = 0;}

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

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

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

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

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

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

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

\(\    ECHO;

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

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

%%

int main(int argc,char **argv)
{
    if (argc > 1)
    {
        FILE *file;
        file = fopen(argv[1],"r");
        if(!file)
        {
```



```
        printf("could not open %s \n",argv[1]);
        exit(0);
    }
    yyin = file;
}
yylex();
printf("\n\n");
return 0;
}
int yywrap()
{
    return 0;
}
```

input.txt file

```
/*comment line*/
#include<stdio.h>
main()
{
int a,b;
a=20;
printf("%d",a);
}
```

Compilation Commands:

```
D:/lab/>flex lexprogram.l
D:/lab/>gcc lex.yy.c
D:/lab/>a.exe input.txt
```



OUTPUT:

```
#include<stdio.h> is a PREPROCESSOR DIRECTIVE

FUNCTION
  main(
  )

  BLOCK BEGINS

    int is a KEYWORD
  a IDENTIFIER,
  b IDENTIFIER;

    float is a KEYWORD
  c IDENTIFIER
  = is an ASSIGNMENT OPERATOR
  20 is a NUMBER.
  5 is a NUMBER;

FUNCTION
  printf(
  "%d" is a STRING,
  c IDENTIFIER
  );

  BLOCK ENDS
^C
```

Press ctrl+c to stop execution



RESULT:

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

EX NO : 4A

RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, -, *
AND / USING YACC

AIM

To write a program for recognizing a valid arithmetic expression that uses operator +, -, * and / using YACC (Yet Another Compiler-Compiler).

STRUCTURE AND EXPLANATION ABOUT YACC PROGRAM:

A Yacc source program has three parts as follows:

Declarations
%%
translation rules
%%
supporting C routines

→Declarations Part

This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which is delimited by %{ and %}. Any temporary variable used by the second and third sections will be kept in this

→Translation Rule Part

After the first %% pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action.

A set of productions:

$$\langle \text{head} \rangle \Rightarrow \langle \text{body} \rangle 1 \mid \langle \text{body} \rangle 2 \mid \dots \mid \langle \text{body} \rangle n$$

would be written in YACC as

```

$$\begin{aligned} \langle \text{head} \rangle & : \langle \text{body} \rangle 1 && \{ \langle \text{semantic action} \rangle 1 \} \\ & | \langle \text{body} \rangle 2 && \{ \langle \text{semantic action} \rangle 2 \} \\ & && \dots \\ & | \langle \text{body} \rangle n && \{ \langle \text{semantic action} \rangle n \} \\ & ; \end{aligned}$$

```

→Supporting C–Rules

It is the last part of the YACC specification and should provide a lexical analyzer named **yylex()**. These produced tokens have the token's name and are associated with its

attribute value. Whenever any token like DIGIT is returned, the returned token name should have been declared in the first part of the YACC specification.

The attribute value which is associated with a token will communicate to the parser through a variable called **yyval**. This variable is defined by a YACC.

Whenever YACC reports that there is a conflict in parsing-action, we should have to create and consult the file **y.output** to see why this conflict in the parsing-action has arisen and to see whether the conflict has been resolved smoothly or not.

Instructed YACC can reduce all parsing action conflict with the help of two rules that are mentioned below:

- A reduce/reduce conflict can be removed by choosing the production which has conflict mentioned earlier in the YACC specification.
- A shift/reduce conflict is reduced in favor of shift. A shift/reduce conflict that arises from the dangling-else ambiguity can be solved correctly using this rule.

RESOURCES:

Install : bison-2.4.1-setup

Set the following environmental variable in path

C:\ProgramFiles\GnuWin32\bin; (Note: folder names should not contain any space while installing bison software)

STEPS TO CREATE AND RUN YACC PROGRAM:

1. Open any editor, type the needed yacc program and save it with extension **.y** (example as **arithexp.y**).
2. Process the **yacc** grammar file using the **-d** optional flag (which informs the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

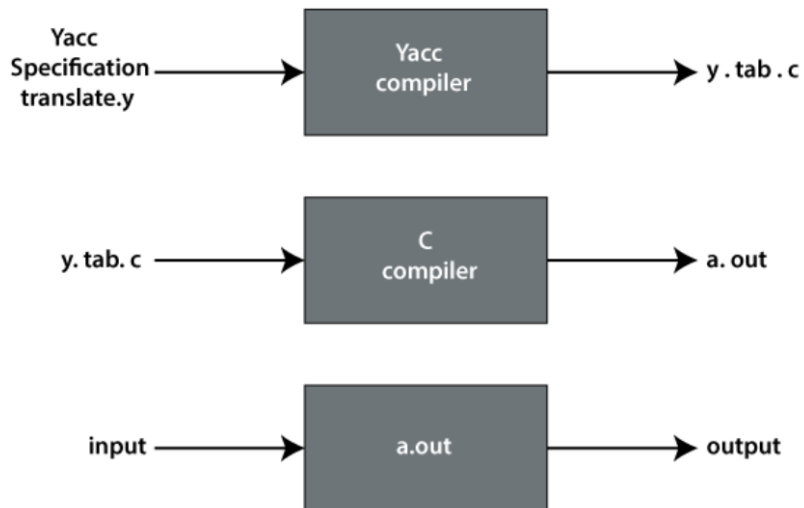
```
bison -d arithexp.y
```

Following two files will be get generated.

- a. **arithexp.tab.c** -- The C language source file that the yacc command created for the parser

- b. **arithexp.tab.h** -- A header file containing define statements for the tokens used by the parser
3. Compile the c program **arithexp.tab.c**
gcc **arithexp.tab.c** (Note: Warning will be generated, Please ignore it)
Following file will be generated : **a.exe**
4. Run the executable file : a.exe

The construction of translation using YACC is illustrated in the figure below:



Program: (arithexp.y)

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

%token num let
%left '+' '-'
%left '*' '/'

%%

Stmt : Stmt '\n'

{ printf ("\n.. Valid Expression.. \n");
  exit(0);
```

```
}
|  expr
|  error '\n'
{ printf ("\n..Invalid ..\n");
  exit(0);
}
;
```

```
expr : num
|  let
|  expr '+' expr
|  expr '-' expr
|  expr '*' expr
|  expr '/' expr
|  '(expr )'
;
```

```
%%
```

```
main ( )
```

```
{
    printf ("Enter an expression to validate :");
    yyparse();
```

```
}
```

```
yylex()
```

```
{
```

```
    int ch;
```

```
    while ( ( ch = getchar() ) == ' ');
```

```
    if ( isdigit(ch) )
```

```
        return num; // return token num
```

```
    if ( isalpha(ch) )
```

```
        return let; // return token let
```

```
    return ch;
```

```
}
```

```
yyerror (char *s)
```

```
{
```

```
    printf ( "%s", s );
```

```
}
```



OUTPUT

```
D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>bison -d arithexp.y
D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>gcc arithexp.tab.c
arithexp.tab.c: In function 'yyparse':
arithexp.tab.c:592:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
  592 | # define YYLEX yylex (
      |                ^~~~~
arithexp.tab.c:1237:16: note: in expansion of macro 'YYLEX'
 1237 |     yychar = YYLEX;
      |                ^~~~~
arithexp.tab.c:1367:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
 1367 |     yyerror (YY_("syntax error"));
      |     ^~~~~~
      |     yyerrok
arithexp.y: At top level:
arithexp.y:28:1: warning: return type defaults to 'int' [-Wimplicit-int]
   28 | main ( )
      |     ^~~
arithexp.y:34:1: warning: return type defaults to 'int' [-Wimplicit-int]
   34 | yylex()
      |     ^~~~~
arithexp.y:45:1: warning: return type defaults to 'int' [-Wimplicit-int]
   45 | yyerror (char *s)
      |     ^~~~~~

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>a.exe
Enter an expression to validate :a+5

.. Valid Expression..

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>a.exe
Enter an expression to validate :(1+v*r)/6

.. Valid Expression..

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>a.exe
Enter an expression to validate :a+4
syntax error
..Invalid ..

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>_
```

RESULT:

Thus the program for the exercise to recognize a valid arithmetic expression that uses operator +, -, * and / using YACC has been successfully executed and output is verified.

EX.NO.4 (B)

**RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED
BY ANY NUMBER OF LETTERS OR DIGITS USING YACC**

AIM

To write a program for recognizing a valid variable which starts with a letter followed by any number of letters or digits using Yacc

PROGRAM: (VARIABLEVALID.Y)

```
% {  
    #include <stdio.h>  
    #include <ctype.h>  
% }
```

```
%token let dig
```

```
%%
```

```
TERM : XTERM '\n'  
    {  
        printf (" \nAccepted\n" );  
        exit(0);  
    }  
| error  
    {  
        printf (" \nRejected\n" );  
        exit(0);  
    }
```



```
XTERM : XTERM let  
    | XTERM dig  
    | let  
;
```

```
%%
```

```
yylex()  
{  
    char ch;  
    while ((ch = getchar())!=' ');
```

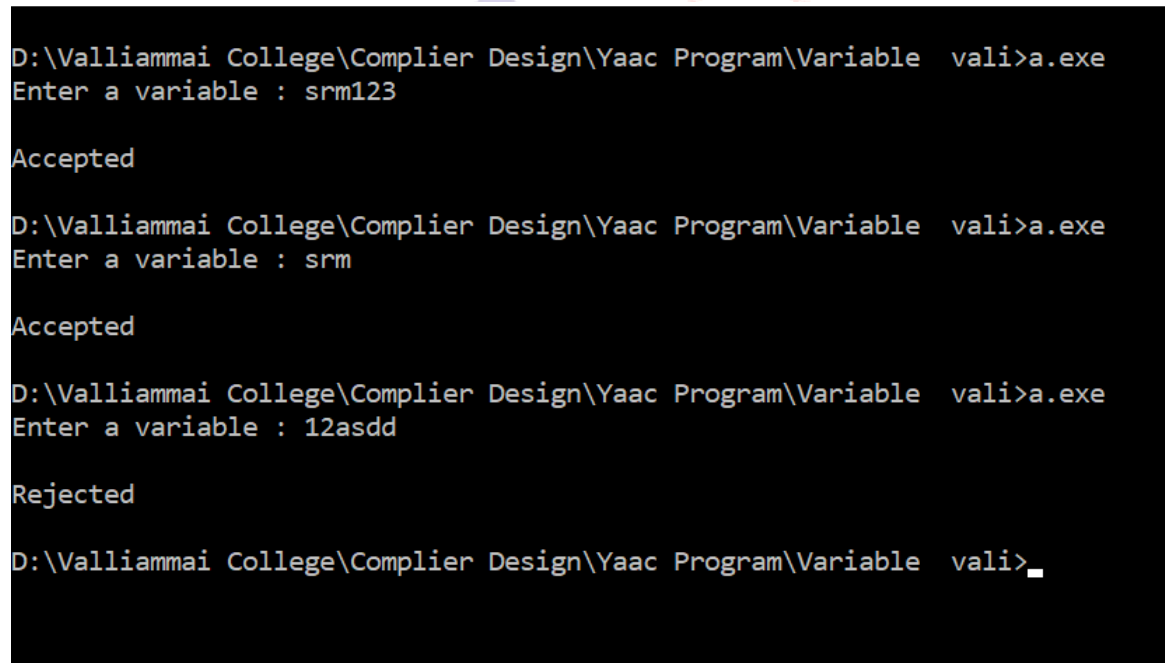


```
        if(isalpha(ch))
            return let;
        if(isdigit(ch))
            return dig;
    return ch;
}

main()
{
    printf ("Enter a variable : ");
    yyparse ();
}

int yyerror()
{
    return 0;
}
```

OUTPUT:



```
D:\Valliammai College\Complier Design\Yaac Program\Variable vali>a.exe
Enter a variable : srm123

Accepted

D:\Valliammai College\Complier Design\Yaac Program\Variable vali>a.exe
Enter a variable : srm

Accepted

D:\Valliammai College\Complier Design\Yaac Program\Variable vali>a.exe
Enter a variable : 12asdd

Rejected

D:\Valliammai College\Complier Design\Yaac Program\Variable vali>_
```

RESULT:

Thus the program for the exercise to recognize a valid variable which starts with a letter followed by any number of letters or digits using Yacc has been successfully executed and output is verified.

EX.NO.4 (C)

IMPLEMENTATION OF CALCULATOR USING LEX AND YACC

AIM

To write a program for implementing a calculator using Lex and Yacc

STEPS TO CREATE , RUN - LEX (CALC.L) AND YACC(CALC.Y) PROGRAM:

1. Open any editor, type the needed yacc and lex programs and save it with extension .y (example as **calc.y**) and .l (**calc.l**) respectively
2. Process the **yacc** grammar file using the **-d** optional flag (which informs the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

`bison -d calc.y`

Following two files will be get generated.

c. **calc.tab.c** -- The C language source file that the yacc command created for the parser

d. **calc.tab.h** -- A header file containing define statements for the tokens used by the parser

3. Process the **lex** specification

`flex calc.l`



Following file will be generated

`lex.yy.c` -- The C language source file that the lex command created for the lexical analyzer

4. Compile and link the two C language source

`gcc lex.yy.c calc.tab.c` (Note: Warning will be generated, Please ignore it)

Following file will be generated : **a.exe**

5. Run the executable file : a.exe

PROGRAM: Calc.y

```
% {  
    #include <stdio.h>  
    int regs[26];  
    int base;  
% }  
%start list  
%token DIGIT LETTER
```

```
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /*supplies precedence for unary minus */
```

```
%% /* beginning of rules section */
```

```
list: /*empty */
```

```
|
list stat '\n'
|
list error '\n'
{
yyerrok;
}
```

```
;
```

```
stat: expr
```

```
{
printf("%d\n",$1);
}
|
LETTER '=' expr
{
regs[$1] = $3;
}
```

```
;
```

```
expr: '(' expr ')'
```

```
{
$$ = $2;
}
```

```
|
expr '*' expr
```

```
{
$$ = $1 * $3;
}
```

```
|
expr '/' expr
```

```
{
```



```
    $$ = $1 / $3;
}
|
expr '%' expr
{
    $$ = $1 % $3;
}
|
expr '+' expr
{
    $$ = $1 + $3;
}
|
expr '-' expr
{
    $$ = $1 - $3;
}
|
expr '&' expr
{
    $$ = $1 & $3;
}
|
expr '|' expr
{
    $$ = $1 | $3;
}
|
'-' expr %prec UMINUS
{
    $$ = -$2;
}
|
LETTER
{
    $$ = regs[$1];
}
|
number
;

```



```
number: DIGIT
{
    $$ = $1;
    base = ($1==0) ? 8 : 10;
}
|
number DIGIT
{
    $$ = base * $1 + $2;
}
;
```

%%

```
main()
{
    return(yyparse());
}
```

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

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

calc.l

```
%{
    #include <stdio.h>
    #include "calc.tab.h"
    int c;
    extern int yylval;
%}
%%
" " ;
[a-z] {
```



```
        c = yytext[0];
        yylval = c - 'a';
        return(LETTER);
    }
[0-9]  {
        c = yytext[0];
        yylval = c - '0';
        return(DIGIT);
    }
[^a-z0-9\b] {
        c = yytext[0];
        return(c);
    }
}
```

OUTPUT

```
D:\Valliammai College\Complier Design\Yaac Program\Calculator>bison -d calc.y
D:\Valliammai College\Complier Design\Yaac Program\Calculator>flex calc.l
D:\Valliammai College\Complier Design\Yaac Program\Calculator>gcc lex.yy.c calc.tab.c
calc.tab.c: In function 'yyparse':
calc.tab.c:613:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
  613 | # define YYLEX yylex ()
      |                ^~~~~
calc.tab.c:1258:16: note: in expansion of macro 'YYLEX'
 1258 |     yychar = YYLEX;
      |                ^~~~~
calc.tab.c:1508:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
 1508 |     yyerror (YY_("syntax error"));
      |     ^~~~~~
      |     yyerrok
calc.y: At top level:
calc.y:104:1: warning: return type defaults to 'int' [-Wimplicit-int]
  104 | main()
      |     ^~~~~
calc.y:109:1: warning: return type defaults to 'int' [-Wimplicit-int]
  109 | yyerror(s)
      |     ^~~~~
calc.y:115:1: warning: return type defaults to 'int' [-Wimplicit-int]
  115 | yywrap()
      |     ^~~~~

D:\Valliammai College\Complier Design\Yaac Program\Calculator>a.exe
10+20*-10
-190
(10+10)*-10
-200
1+2*3/4-2
0
1*10
10
```

RESULT:

Thus the program for the exercise to implement a calculator using Lex and Yacc has been successfully executed and output is verified.

EX.NO.5

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

AIM:

To write YACC program to generate abstract syntax tree by converting the BNF (Backus Naur Form) rules into YACC form.

STEPS TO CREATE , RUN - LEX (LEXBNF.L) AND YACC(YACCBNF.Y) PROGRAM:

1. Open any editor, type the needed yacc and lex programs and save it with extension .y (example as **yaccbnf.y**) and .l (**lexbnf.l**) respectively
2. Process the yacc grammar file using the -d optional flag (which informs the yacc command to create a file that defines the tokens used in addition to the C language source code):

bison -d yaccbnf.y

Following two files will be get generated.

e. **yaccbnf.tab.c** -- The C language source file that the yacc command created for the parser

f. **yaccbnf.tab.h** -- A header file containing define statements for the tokens used by the parser

3. Process the lex specification

flex lexbnf.l

Following file will be generated

lex.yy.c -- The C language source file that the lex command created for the lexical analyzer

4. Compile and link the two C language source

gcc lex.yy.c yaccbnf.tab.c (Note: Warning will be generated, Please ignore it)

Following file will be generated : **a.exe**

5. Run the executable file : **a.exe input.c**

PROGRAM: lexbnf.l

```
%{
#include"yaccbnf.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];

%%

Yaccbnf.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 -----");
printf("\n\t\t Pos Operator\t Arg1\t Arg2\t Result");
printf("\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,"%d",tIndex++);
    strcpy(result,QUAD[Index++].result);
}

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

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



input.c

```
main()
{
    int a,b,c;
    a=10;
    b=20;
    if(a<b)
    {
        a=a+b;
    }
    while(a<b)
    {
```

```
a=a+b;
}
if(a<=b)
{
  c=a-b;
}
else
{
  c=a+b;
}
}
```

OUTPUT:

```
D:\Valliammai College\Complier Design\Yaac Program\BNF_Syntax tree>a.exe input.c
```

Pos	Operator	Arg1	Arg2	Result
0	=	10		a
1	=	20		b
2	<	a	b	t0
3	==	t0	FALSE	7
4	+	a	b	t1
5	=	t1		a
6	GOTO			7
7	<	a	b	t2
8	==	t2	FALSE	12
9	+	a	b	t3
10	=	t3		a
11	GOTO			7
12	<=	a	b	t4
13	==	t4	FALSE	17
14	-	a	b	t5
15	=	t5		c
16	GOTO			19
17	+	a	b	t6
18	=	t6		c

RESULT:

Thus the C program to convert the BNF rules into YACC form and write code to generate abstract syntax tree was executed and output is verified

EX.NO.6

IMPLEMENT TYPE CHECKING

AIM:

To write a C program to check whether the type of all variables in an expression are valid or not.

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

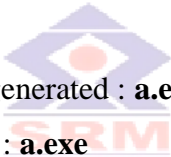
STEPS TO RUN C PROGRAM USING GCC COMPLIER:

1. Open any editor, type the needed C program and save as **typecheck.c**
2. Compile C language source

gcc typecheck.c

Following file will be generated : **a.exe**

3. Run the executable file : **a.exe**



PROGRAM:

```
#include<stdio.h>
void 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;
}
}
}
}
```

OUTPUT



```
D:\Valliammai College\Compiler Design\Yaac Program\Type Check\C program>gcc type.c
D:\Valliammai College\Compiler Design\Yaac Program\Type Check\C program>a.exe
Enter the number of variables:4
Enter the variable[0]:a
Enter the variable-type[0](float-f,int-i):f
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):i
Enter the variable[3]:d
Enter the variable-type[3](float-f,int-i):i
Enter the Expression(end with $):d=a/b+c$
Identifier d must be a float type..!

D:\Valliammai College\Compiler Design\Yaac Program\Type Check\C program>a.exe
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):f
Enter the variable[2]:c
Enter the variable-type[2](float-f,int-i):i
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..!
```

RESULT:

Thus the C program to implement type checking was executed and the output is verified.

EX. NO. 7

STORAGE ALLOCATION STRATEGIES : STACK IMPLEMENT

AIM:

To write a c program for implementing Stack storage allocation strategy.

TYPES OF MEMORY

MEMORY IN C – THE STACK, THE HEAP, AND STATIC

The great thing about C is that it is so intertwined with memory – and by that I mean that the programmer has quite a good understanding of “what goes where”. C has three different pools of memory.

- static: global variable storage, permanent for the entire run of the program.
- stack: local variable storage (automatic, continuous memory).
- heap: dynamic storage (large pool of memory, not allocated in contiguous order).

STATIC MEMORY

Static memory persists throughout the entire life of the program, and is usually used to store things like global variables, or variables created with the static clause. For example:

```
int theforce;
```

On many systems this variable uses 4 bytes of memory. This memory can come from one of two places. If a variable is declared outside of a function, it is considered global, meaning it is accessible anywhere in the program. Global variables are static, and there is only one copy for the entire program. Inside a function the variable is allocated on the stack. It is also possible to force a variable to be static using the static clause. For example, the same variable created inside a function using the static clause would allow it to be stored in static memory.

```
static int theforce;
```

STACK MEMORY

The stack is used to store variables used on the inside of a function (including the main()function). It's a LIFO, “Last-In,-First-Out”, structure. Every time a function declares a new variable it is “pushed” onto the stack. Then when a function finishes running, all the variables associated with that function on the stack are deleted, and the memory they use is freed up. This leads to the “local” scope of function variables. The stack is a special region of memory, and automatically managed by the CPU – so you don't have to allocate or deallocate memory. Stack memory is divided into successive frames where each time a function is called, it allocates itself a fresh stack frame.

Note that there is generally a limit on the size of the stack – which can vary with the operating system (for example OSX currently has a default stack size of 8MB). If a program tries to put too much information on the stack, stack overflow will occur. Stack overflow happens when all the memory in the stack has been allocated, and further allocations begin overflowing into other sections of memory. Stack overflow also occurs in situations where recursion is incorrectly used.

A summary of the stack:

- the stack is managed by the CPU, there is no ability to modify it
- variables are allocated and freed automatically
- the stack is not limitless – most have an upper bound
- the stack grows and shrinks as variables are created and destroyed
- stack variables only exist whilst the function that created them exists

HEAP MEMORY

The heap is the diametrical opposite of the stack. The heap is a large pool of memory that can be used dynamically – it is also known as the “free store”. This is memory that is not automatically managed – you have to explicitly allocate (using functions such as malloc), and deallocate (e.g. free) the memory. Failure to free the memory when you are finished with it will result in what is known as a memory leak – memory that is still “being used”, and not available to other processes. Unlike the stack, there are generally no restrictions on the size of the heap (or the variables it creates), other than the physical size of memory in the machine. Variables created on the heap are accessible anywhere in the program.

A summary of the heap:

- the heap is managed by the programmer, the ability to modify it is somewhat boundless
- in C, variables are allocated and freed using functions like malloc() and free()
- the heap is large, and is usually limited by the physical memory available
- the heap requires pointers to access it

AN EXAMPLE OF MEMORY USE

Consider the following example of a program containing all three forms of memory:

```
#include <stdio.h>
#include <stdlib.h>
int x;

int main(void)
{
    int y;
    char *str;

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```

The variable x is static storage, because of its global nature. Both y and str are dynamic stack storage which is deallocated when the program ends. The function malloc() is used to allocate 100 pieces of dynamic heap storage, each the size of char, to str. Conversely, the function free(), deallocates the memory associated with str.

ALGORITHM:

- Step1:** Initially check whether the stack is empty
- Step2:** Insert an element into the stack using push operation
- Step3:** Insert more elements onto the stack until stack becomes full
- Step4:** Delete an element from the stack using pop operation
- Step5:** Display the elements in the stack
- Step6:** Top the stack element will be displayed

STEPS TO RUN C PROGRAM USING GCC COMPLIER:

1. Open any editor, type the needed C program and save as **Memory.c**
2. Compile C language source

gcc Memory.c

Following file will be generated : **a.exe**

3. Run the executable file : **a.exe**

PROGRAM:

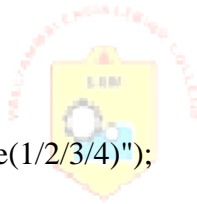
```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
void main()
{

    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t-----");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
```



```
    {
        push();
        break;
    }
    case 2:
    {
        pop();
        break;
    }
    case 3:
    {
        display();
        break;
    }
    case 4:
    {
        printf("\n\t EXIT POINT ");
        break;
    }
    default:
    {
        printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
    }

}
}
while(choice!=4);
}
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");
    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
}
```



```
    }
}
void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d",stack[top]);
        top--;
    }
}
void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }
}
```



OUTPUT

```
D:\Valliammai College\Complier Design\Yaac Program\Stack>gcc Memory.c
D:\Valliammai College\Complier Design\Yaac Program\Stack>a.exe

Enter the size of STACK[MAX=100]:4

    STACK OPERATIONS USING ARRAY
-----
    1.PUSH
    2.POP
    3.DISPLAY
    4.EXIT
Enter the Choice:1
Enter a value to be pushed:10

Enter the Choice:1
Enter a value to be pushed:20

Enter the Choice:1
Enter a value to be pushed:30

Enter the Choice:1
Enter a value to be pushed:40

Enter the Choice:1

    STACK is over flow
Enter the Choice:3

The elements in STACK

40
30
20
10
Press Next Choice
```



```
Enter the Choice:1
Enter a value to be pushed:10

Enter the Choice:1
Enter a value to be pushed:20

Enter the Choice:1
Enter a value to be pushed:30

Enter the Choice:1
Enter a value to be pushed:40

Enter the Choice:1

        STACK is over flow
Enter the Choice:3

The elements in STACK

40
30
20
10
Press Next Choice
Enter the Choice:2

        The popped elements is 40
Enter the Choice:2

        The popped elements is 30
Enter the Choice:3

The elements in STACK

20
10
Press Next Choice
Enter the Choice:4

        EXIT POINT
D:\Valliammai College\Complier Design\Yaac Program\Stack>_
```

Result:

Thus the C program to implement Stack storage allocation strategy was executed and the output is verified.

EX NO :8

CONSTRUCTION OF DAG

AIM:

To write a C program to construct a Direct Acyclic Graph(DAG).

INTRODUCTION:

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

They are:

- The semantic equivalence 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. Get the expression to draw the DAG with proper parenthesis.
3. Check for the postfix expression and construct in order DAG representation.
4. Print the output with the left pointer, right pointer values ,the node and leaf values in a table format
5. Stop the program

Program:

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
void main()
{
    struct da
    {
        int ptr, left , right;
        char label;
    }dag[25];
    int ptr,j,change,n=0, i=0,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("Hint: Provide the expression inside paranthesis stating the priority option");
    printf("\nFor Example:a+b*c is given as (a+(b*c)) ");
    printf("\nENTER THE EXPRESSION");
    scanf("%s",input1);

    /*EX: ((a*b-c))+((b-c)*d) like this give with      parenthesis limit is 25 char u can
change      that*/

    for(i=0;i<25;i++)
        input[i]=NULL;

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

        for(j=i;input1[j]!='\0';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 DAG FOR GIVEN EXPRESSION");
printf("\n\n PTR \t LEFT PTR \t RIGHT PTR \t LABEL");
for(i=0;i<n;i++)
/*draw DAG for the following output with pointer value*/
    printf("\n %d\t%d\t%d\t%d\t%c",dag[i].ptr,dag[i].left,dag[i].right,dag[i].label);
getch();
}

```

Output:

```

Hint: Provide the expression inside paranthesis stating the priority option
For Example:a+b*c is given as (a+(b*c))

ENTER THE EXPRESSION(((a+b)*(c-d))/(a+b))

DAG FOR GIVEN EXPRESSION

PTR      LEFT PTR      RIGHT PTR      LABEL
0        0              0              a
1        0              0              b
2        0              1              +
3        0              0              c
4        0              0              d
5        3              4              -
6        2              5              *
7        0              0              a
8        0              0              b
9        7              8              +
10       6              9              /

```

RESULT:

Thus the C program to construct the DAG was executed and the output is verified.

EX NO:9

IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUE

AIM:

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

ALGORITHM:

1. Start the program.
2. Create an array which contains three address code.
3. Scan the array values from left to right.

Common sub expression elimination:

4. Store the first expression in String.
5. Compare the String with the other expression in the array
6. If there is a match, remove the expression from the array
7. Perform 3 to 6 for all values in the array.

Dead code Elimination

8. Get the operand before the operator from the three address code
9. Check whether the operand is used in any other expression in three address code.
10. If the operand is not used, then eliminate the complete expression from the three address code
11. Else perform step 8 to 10 for all operands
12. Stop the program

STEPS TO RUN C PROGRAM USING GCC COMPLIER:

6. Open any editor, type the needed C program and save as codeoptimize.c
7. Compile C language source

```
gcc codeoptimize.c
```

Following file will be generated : a.exe

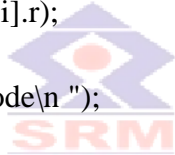
8. Run the executable file : a.exe

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
```

```
struct op
{
```

```
char l;  
char r[20];  
}  
op[10], pr[10];  
  
int 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);  
    for (i = 0;i < n; i++)  
    {  
        printf("\tleft\t");  
        op[i].l = getche();  
        printf("\tright :t");  
        scanf("%s", op[i].r);  
    }  
    printf("Intermediate Code\n ");  
    for (i = 0;i < n; i++)  
    {  
        printf("%c\t=", 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);
}
//sub expression elimination
for (m = 0; m < z; m++)
{
    tem = pr[m].r;
    for (j = m + 1; j < z; j++)
    {
        p = strstr(tem, pr[j].r);
        if (p)
        {
            t = pr[j].l;
            pr[j].l = pr[m].l;
            for (i = 0; i < z; i++)
            {
                l = strchr(pr[i].r, t);
                if (l)
                {
                    a = l - pr[i].r;
                    pr[i].r[a] = pr[m].l;
                }
            }
        }
    }
}
printf("eliminate common expression\n");
for (i = 0; i < z; i++)
{
    printf("%c\t=", pr[i].l);
    printf("%s\n", pr[i].r);
}
//duplicate production elimination
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:

```

C:\Users\Subha\Desktop\DAG\Code optim>a.exe
enter no of values4
    left  a      right : 5
    left  b      right : c+d
    left  e      right : c+d
    left  q      right : b+e
Intermediate Code
a      =5
b      =c+d
e      =c+d
q      =b+e
after dead code elimination
b      =c+d
e      =c+d
q      =b+e
eliminate common expression
b      =c+d
b      =c+d
q      =b+b
optimized code
b=c+d
q=b+b

```

```

C:\Users\Subha\Desktop\DAG\Code optim>gcc "CODE_O~1.C"
C:\Users\Subha\Desktop\DAG\Code optim>a.exe
enter no of values5
    left  a      right : 9
    left  b      right : c+d
    left  e      right : c+d
    left  r      right : f
    left  g      right : a+b
Intermediate Code
a      =9
b      =c+d
e      =c+d
r      =f
g      =a+b
after dead code elimination
a      =9
b      =c+d
g      =a+b
eliminate common expression
a      =9
b      =c+d
g      =a+b
optimized code
a=9
b=c+d
g=a+b

```

RESULT:

Thus the C program to implement code optimization technique was executed and the output is verified

EX NO :10a

IMPLEMENTATION OF CONTROL FLOW ANALYSIS

AIM:

To write a C program to implement the control flow analysis technique.

INTRODUCTION:

Control flow analysis can be represented by basic blocks. It depicts how the program control is being passed among the blocks.

ALGORITHM:

1. Start the program execution
2. Declare the necessary variables for accessing statements from cdp.txt
3. Find out the Leaders, Conditional statements and blocks from the file
4. Display the blocks with Block No
5. Display the control flow movement with block number
6. Stop the program execution



PROGRAM:

```
# include<stdio.h>
# include<conio.h>
#include<malloc.h>
#include<string.h>
struct Listnode
{
char data[50];
int leader,block,u_goto,c_goto;
struct Listnode *next;
char label[10],target[10];
}*temp,*cur,*first=NULL,*last=NULL,*cur1;
FILE *fpr;
void createnode(char code[50])
{
temp=(struct Listnode *)malloc(sizeof(struct Listnode));
strcpy(temp->data,code);
strcpy(temp->label,'\0');
strcpy(temp->target,'\0');
temp->leader=0;
```

```
temp->block=0;
temp->u_goto=0;
temp->c_goto=0;
temp->next=NULL;
if(first==NULL)
{
first=temp;
last=temp;
}
else
{
last->next=temp;
last=temp;
}
}
void main()
{
char codeline[50];
char c,dup[50],target[10];
char *substring,*token;
int i=0,block,block1;
int j=0;
fpr= fopen("cdp.txt","r");
clrscr();
while((c=getc(fpr))!=EOF)
{
if(c!='\n')
{
codeline[i]=c;
i++;
}
else
{
codeline[i]='\0';
createnode(codeline);
i=0;
}
}
//create last node
codeline[i]='\0';
createnode(codeline);
```



```
fclose(fpr);
// find out leaders,conditional stmts
cur=first;
cur->leader=1;
while(cur!=NULL)
{
substring=strstr((cur->data),"if");
if(substring==NULL)
{
if((strstr((cur->data),"goto"))!=NULL)
{
cur->u_goto=1;
(cur->next)->leader=1;
}
}
else
{
cur->c_goto=1;
(cur->next)->leader=1;
}
substring=strstr((cur->data),":");
if(substring!=NULL)
{
cur->leader=1;
}
substring=strstr((cur->data),"call");
if(substring!=NULL)
{
cur->leader=1;
}
if(strstr(cur->data,"return")!=NULL)
{
cur->leader=1;
(cur->next)->leader=1;
}
cur=cur->next;
}
//to find labels and targets
cur=first;
while(cur!=NULL)
{
```



```
if((cur->u_goto==1)||((cur->c_goto==1))
{
substring=strstr(cur->data,":");
if(substring!=NULL)
{
token=strstr(substring,"L" );
if(token!=NULL)
strcpy(cur->target,token);
}
else
{
substring=strstr(cur->data,"L");
if(substring!=NULL)
strcpy(cur->target,substring);
}
}
if(strstr(cur->data,":")!=NULL)
{
strcpy(dup,cur->data);
token=strtok(dup,":");
// printf("\ntoken:%s",token);
if(token!=NULL)
strcpy(cur->label,token);
}
cur=cur->next;
}
//to identify blocks
cur=first;
while(cur!= NULL)
{
cur=cur->next;
if((cur->leader)==1)
{
j++;
cur->block=j;
}
else
cur->block=j;
}
printf("\n\n.....Basic Blocks.....\n");
cur=first;
```



```
j=0;
printf("\nBlock %d:",j);
while(cur!=NULL)
{
if ((cur->block)==j)
{
printf("%s",cur->data);
printf("\n\t");
cur=cur->next;
}
else
{
j++;
printf("\nBlock %d:",j);
}
}
//to output the control flow from each block
printf ("\t\t\t.....Control Flow.....\n\n");
cur=first;
i=0;
while(cur!=NULL)
{
if((cur->block)!=((cur->next)->block))
{
block=cur->block;
if(cur->u_goto==1)
{
strcpy(target,cur->target);
cur1=first;
while(cur1!=NULL)
{
if(strcmp(cur1->label,target)==0)
{
block1=cur1->block;
printf("\t\t\tBlock%d----->Block%d\n",block,block1);
}
cur1=cur1->next;
}
}
else if(cur->c_goto==1)
{
```



```
strcpy(target,cur->target);
cur1=first;
while(cur1!=NULL)
{
if(strcmp(cur1->label,target)==0)
{
block1=cur1->block;
printf("\t\tBlock%d---TRUE--->Block%d---FALSE--->Block%d\n",block,block1,(block+1));
}
cur1=cur1->next;
}
}
else if(strstr(cur->data,"return")==NULL)
{
printf("\t\tBlock%d----->Block%d\n",block,(block+1));
}
else
printf("\t\tBlock%d----->NULL\n",block);
}
cur=cur->next;
}
cur=last;
block= cur->block;
printf("\t\tBlock%d----->NULL",block);
getch();
}
```



cdp.txt

```
m <-0
v <-0
L1:if v<n goto L2
r <-v
s <-0
return
L2:if r>=n goto L1
v <-v+1
```

OUTPUT

```
.....Basic Blocks.....  
  
Block 0:m <-0  
      v <-0  
  
Block 1:L1:if v<n goto L2  
  
Block 2:r <-v  
      s <-0  
  
Block 3:return  
  
Block 4:L2:if r>=n goto L1  
  
Block 5:v <-v+1  
  
.....Control Flow.....  
  
Block0----->Block1  
Block1---TRUE--->Block4---FALSE--->Block2  
Block2----->Block3  
Block3----->NULL  
Block4---TRUE--->Block1---FALSE--->Block5  
Block5----->NULL_
```

Activate Windows
Go to Settings to activate Windows.



RESULT:

Thus the C program to implement control flow analysis technique was executed and the output is verified

EX NO :10b

IMPLEMENTATION OF DATA FLOW ANALYSIS

AIM:

To write a C program to implement the data flow analysis technique.

INTRODUCTION:

Data flow analysis is a technique for gathering information about the possible set of value calculated at various points in a computer program.

ALGORITHM:

1. Start the program execution
2. Read the total Number of Expression
3. Read the Left and Right side of each Expressions
4. Display the Expression with Line No
5. Display the Data flow movement with particular expressions
6. Stop the program execution

PROGRAM:

```
#include <stdio.h>
#include <string.h >
struct op
{
char l[20];
char r[20];
}
op[10], pr[10];
void main()
{
int a, i, k, j, n, z = 0, m, q,lineno=1;
char * p, * l;
char temp, t;
char * tem;char *match;

printf("enter no of values");
scanf("%d", & n);
for (i = 0; i < n; i++)
{
```




```
printf("\tleft\t");
scanf("%s",op[i].l);
printf("\tright:\t");
scanf("%s", op[i].r);
}
printf("intermediate Code\n");

for (i = 0; i < n; i++)
{ printf("Line No=%d\n",lineno);
printf("\t\t\t%s=", op[i].l);
printf("%s\n", op[i].r);lineno++;
}
printf("***Data Flow Analysis for the Above Code ***\n");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
match=strstr(op[j].r,op[i].l);
if(match)
{
printf("\n %s is live at %s \n ", op[i].l,op[j].r);
}
}
}
}
```



OUTPUT:

```
enter no of values4
left a
right: a+b
left b
right: a+c
left c
right: a+b
left d
right: b+c+d
intermediate Code
Line No=1          a=a+b
Line No=2          b=a+c
Line No=3          c=a+b
Line No=4          d=b+c+d
***Data Flow Analysis for the Above Code ***
a is live at a+b
a is live at a+c
a is live at a+b
b is live at a+b
b is live at a+b
b is live at b+c+d
c is live at a+c
c is live at b+c+d
d is live at b+c+d
```

RESULT:

Thus the C program to implement data flow analysis technique was executed and the output is verified

