

DEVELOP A LEXICAL ANALYZER TO RECOGNIZE FEW PATTERNS IN C (EX. IDENTIFIERS, CONSTANTS, COMMENTS, OPERATORS ETC.) AND IMPLEMENTATION OF A SYMBOL TABLE

Ex.No:1

Date:

AIM:

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

ALGORITHM:

Step1:Start the program

Step 2:Read the input string.

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

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

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

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

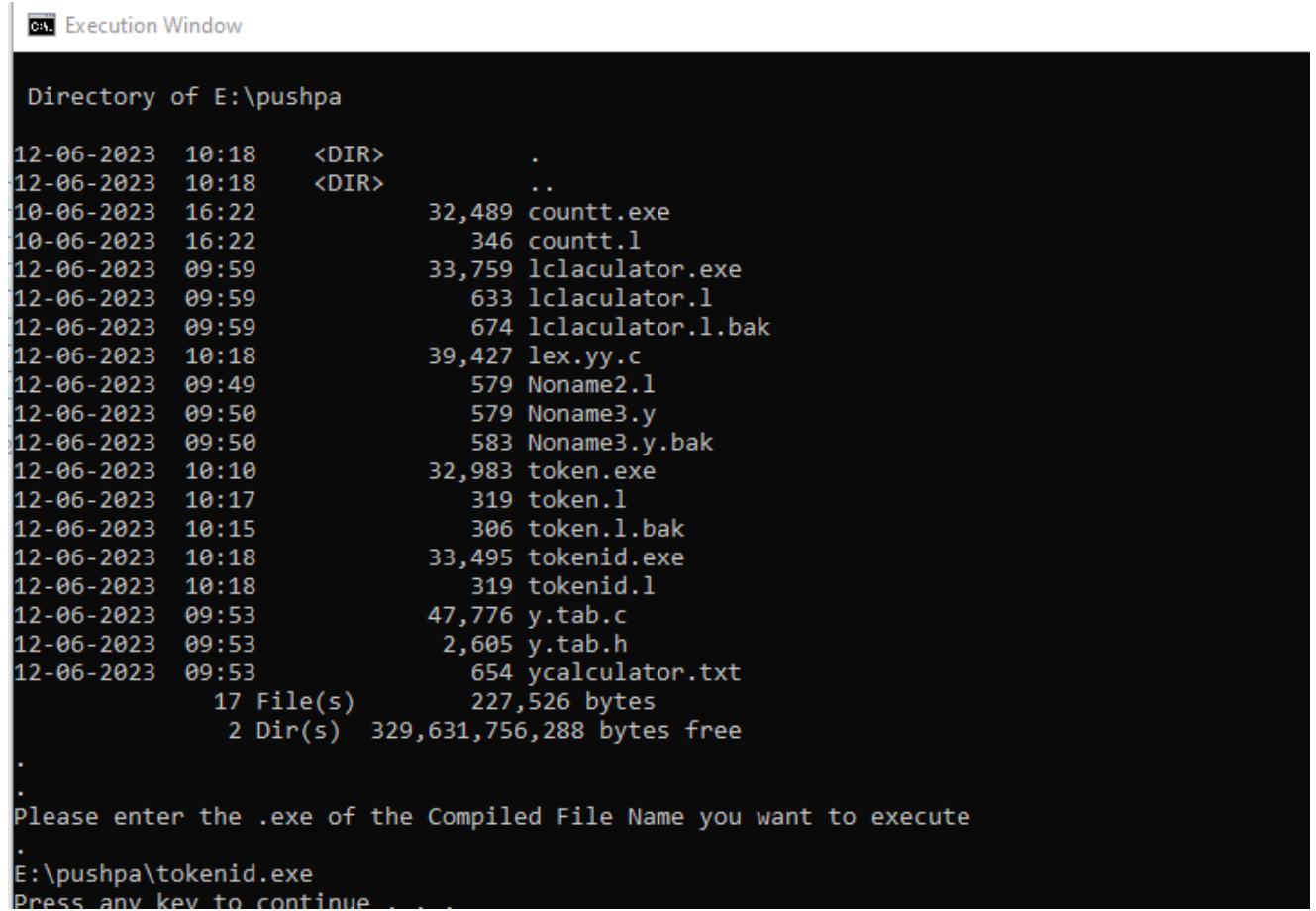
Step 7:Stop the program.

PROGRAM :

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

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

OUTPUT:



```
C:\ Execution Window  
  
Directory of E:\pushpa  
12-06-2023 10:18 <DIR> .  
12-06-2023 10:18 <DIR> ..  
10-06-2023 16:22 32,489 countt.exe  
10-06-2023 16:22 346 countt.l  
12-06-2023 09:59 33,759 lclaculator.exe  
12-06-2023 09:59 633 lclaculator.l  
12-06-2023 09:59 674 lclaculator.l.bak  
12-06-2023 10:18 39,427 lex.yy.c  
12-06-2023 09:49 579 Noname2.l  
12-06-2023 09:50 579 Noname3.y  
12-06-2023 09:50 583 Noname3.y.bak  
12-06-2023 10:10 32,983 token.exe  
12-06-2023 10:17 319 token.l  
12-06-2023 10:15 306 token.l.bak  
12-06-2023 10:18 33,495 tokenid.exe  
12-06-2023 10:18 319 tokenid.l  
12-06-2023 09:53 47,776 y.tab.c  
12-06-2023 09:53 2,605 y.tab.h  
12-06-2023 09:53 654 ycalculator.txt  
17 File(s) 227,526 bytes  
2 Dir(s) 329,631,756,288 bytes free  
.  
.  
Please enter the .exe of the Compiled File Name you want to execute  
.  
E:\pushpa\tokenid.exe  
Press any key to continue . . .
```

E:\pushpa\tokenid.exe

```
int
Keyword
a
Identifiers
D
Identifiers
-
Operators
8
Numbers
"
Punctuation Chars
```

CONCLUSION:

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

IMPLEMENT A LEXICAL ANALYZER USING LEX TOOL

Ex.No:2

Date:

AIM:

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

ALGORITHM:

Step 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

Step 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

Step 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.

Step 4: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

Step 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.

Step 6: In user subroutine section, main routine calls `yylex()`. `yywrap()` is used to get more input.

Step 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.

PROGRAM:

```
%{
/* program to recognize a c program */
int COMMENT=0;
int cnt=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; cnt++;}
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\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 Total No.Of comments are %d",cnt);
return 0;
}
int yywrap()
{
return 1;
}
```

}

OUTPUT:

```
#include<stdio.h>
#include<stdio.h> is a PREPROCESSOR DIRECTIVE
void main
    void is a KEYWORD
main IDENTIFIER
int a,b;
    int is a KEYWORD
a IDENTIFIER,
b IDENTIFIER;
```

CONCLUSION:

Thus a program for implementing a Lexical analyser using LEX tool in Linux platform was executed successfully.

PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION

Ex.No:3.a

Date:

AIM:

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

ALGORITHM:

Step1: Start the program.

Step2: Reading an expression .

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

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

Step5: Stop the program.

PROGRAM:

LEX Program(Validarith.l)

```
%{  
#include<stdio.h>  
#include "y.tab.h"  
%}  
  
%%  
[a-zA-Z]+ return VARIABLE;  
[0-9]+ return NUMBER;  
[\t] ;  
[\n] return 0;  
. return yytext[0];  
%%  
int yywrap()  
{  
return 1;  
}
```


YACC Program (Validarith.y)

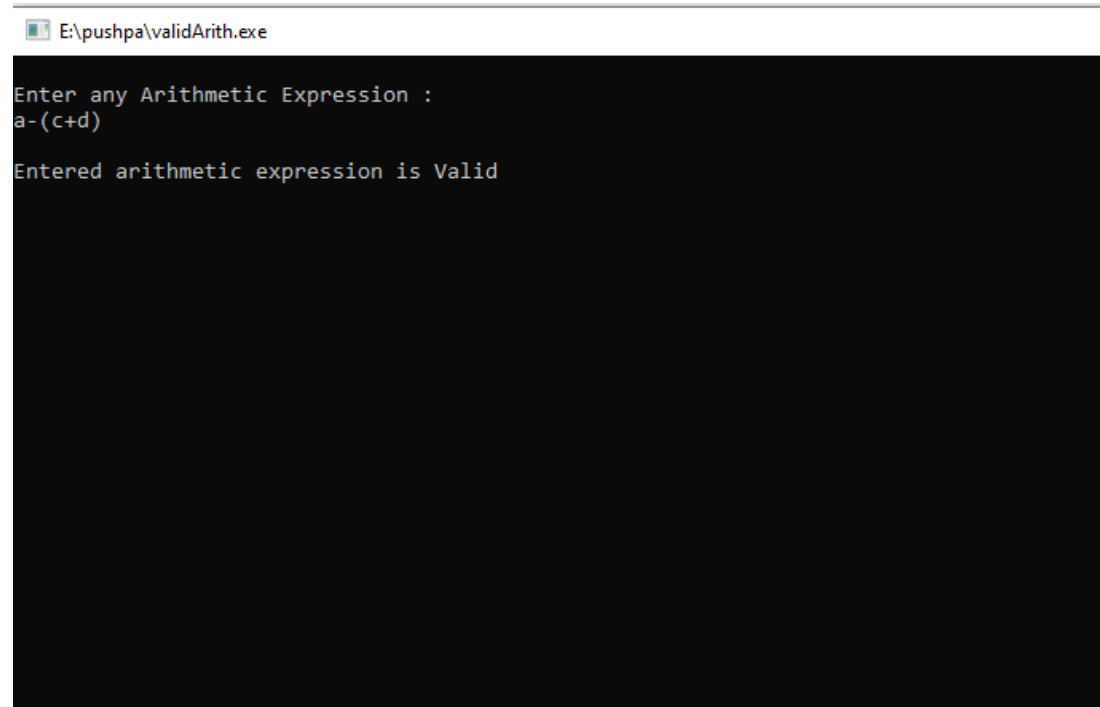
```
%{
    #include<stdio.h>
    #include<conio.h>
}%
%token NUMBER
%token VARIABLE

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%
S: E {
    printf("\nEntered arithmetic expression is Valid\n\n");
    return 0;
}
E: E '+' E
  | E '-' E
  | E '*' E
  | E '/' E
  | E '%' E
  | '(' E ')'
  | NUMBER
  | VARIABLE
;
%%
void main()
{
    printf("\nEnter any Arithmetic Expression : \n");
    yyparse();
    getch();
}
int yyerror(void)
```

```
{  
    printf("\n Entered arithmetic expression is Invalid\n\n");  
}
```

OUTPUT(for valid Expression)



The screenshot shows a Windows command prompt window titled "E:\pushpa\validArith.exe". The prompt displays the text "Enter any Arithmetic Expression :" followed by the input "a-(c+d)". The program's output is "Entered arithmetic expression is Valid".

OUTPUT(for invalid Expression)



The screenshot shows a Windows command prompt window titled "E:\pushpa\validArith.exe". The prompt displays the text "Enter any Arithmetic Expression :" followed by the input "(a+b-c". The program's output is "Entered arithmetic expression is Invalid".

CONCLUSION:

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

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

Ex.No:3.b

Date:

AIM :

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

ALGORITHM:

Step1: Start the program

Step2: Reading an expression

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

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

Step5: Stop the program

PROGRAM CODE:

LEX Program(Valididentifier.l)

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

```
return 1;
```

```
}
```

YACC Program(Valididentifier.y)

```
%{
```

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

```
    int valid=1;
```

```
%}
```

```
%token digit letter
```

```
%%
```

```
start : letter s
```

```
s :   letter s
```

```
      | digit s
```

```
      |
```

```
      ;
```

```
%%
```

```
int yyerror()
```

```
{
```

```
    printf("\nIts not a identifier!\n");
```

```
    valid=0;
```

```
    return 0;
```

```
}
```

```
int main()
```

```
{
```

```
    printf("\nEnter a name to tested for identifier ");
```

```
    yyparse();
```

```
    if(valid)
```

```
    {
```


```
        printf("\nIt is a identifier!\n");
```

```
    }
```

```
    getch();
```

```
}
```

OUTPUT (for invalid identifier):

 E:\pushpa\ValidIdentifier.exe

```
Enter a name to tested for identifier 22d  
Its not a identifier!
```

OUTPUT (for valid identifier):

```
Enter a name to tested for identifier aaafg3  
It is a identifier!
```

CONCLUSION:

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

PROGRAM TO RECOGNIZE WHILE LOOP

Ex.No:3.c

Date:

AIM:

To write a lex program to recognize while loop.

ALGORITHM:

Step1: Start the program.

Step2: Reading an expression .

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

Step4: Print the result of the given while loop

Step5: Stop the program.

LEX program(while.l):

alpha [A-Za-z]

digit [0-9]

%%

[\t\n]

while return WHILE;

{digit}+ return NUM;

{alpha}({alpha}|{digit})* return ID;

"<=" return LE;

">=" return GE;

"==" return EQ;

"!=" return NE;

"||" return OR;

"&&" return AND;

. return yytext[0];

%%

YACC Program(while.y)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
}%
%token ID NUM WHILE LE GE EQ NE OR AND
%right '='
%left AND OR
%left '<' '>' LE GE EQ NE
%left '+' '-'
%left '*' '/'
%right UMINUS
%left '!'
%%
S      : ST1 {printf("Input accepted.\n");exit(0);};
ST1    :  WHILE '(' E2 ')' '{' ST '}'
ST      :  ST ST
        | E ';'
        ;
E       : ID '=' E
        | E '+' E
        | E '-' E
        | E '*' E
        | E '/' E
        | E '<' E
        | E '>' E
        | E LE E
        | E GE E
        | E EQ E
        | E NE E
        | E OR E
        | E AND E
        | ID
```

```
    | NUM
    ;
E2  : E'<'E
    | E'>'E
    | E LE E
    | E GE E
    | E EQ E
    | E NE E
    | E OR E
    | E AND E
    | ID
    | NUM
    ;
```

```
%%
#include "lex.yy.c"
void main()
{
    printf("Enter the exp: ");
    yyparse();
    getch();
}
```

OUTPUT:

While(i<n)Input Accepted

CONCLUSION:

Thus a program to recognize while loop was executed successfully.

IMPLEMENTATION OF CALCULATOR USING LEX & YACC

Ex.No:3.d

Date:

AIM:

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

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Use a function for printing the error message.

Step 4: Get the input from the user and parse it.

Step 5: Check the input is a valid expression or not.

Step 6: Write separate operations for addition, subtraction, multiplication and division using the expr and matching it with the operators in the input.

Step 7: Print the error messages for the invalid operators.

Step 8: Print the output of the expression. **Step 9:** Terminate the program.

LEX program(Calculator.l):

```
%{  
#include <stdlib.h>  
#include <stdio.h>  
#include "y.tab.h"  
void yyerror(char*);  
extern int yylval;  
%}  
%%  
[ \t]+ ;  
[0-9]+ {yylval = atoi(yytext);
```

```
return INTEGER;}
[-+*/] {return *yytext;}
"(" {return *yytext;}
")" {return *yytext;}
\n {return *yytext;}
. {char msg[25];
printf(msg,"%s <%s>","invalid character",yytext);
yyerror(msg);
}
```

YACC program(Calculator.y):

```
%{
#include <stdlib.h>
#include <stdio.h>
int yylex(void);
#include "y.tab.h"
%}

%token INTEGER

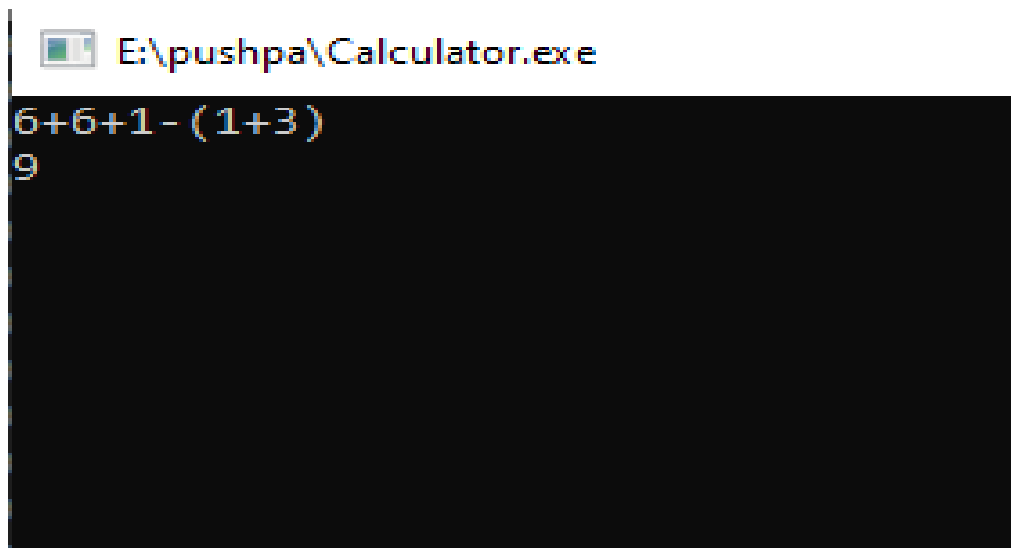
%%

expr:
expr '\n' { printf("%d\n",$1); }
| '\n'
expr:
expr '+' mulex { $$ = $1 + $3; }
| expr '-' mulex { $$ = $1 - $3; }
| mulex { $$ = $1; }
mulex:
mulex '*' term { $$ = $1 * $3; }
| mulex '/' term { $$ = $1 / $3; }
| term { $$ = $1; }
term:
 '(' expr ')' { $$ = $2; }
| INTEGER { $$ = $1; }
%%

void yyerror(char *s)
```

```
{  
fprintf(stderr,"%s\n",s);  
return;  
}  
yywrap()  
{  
return(1);  
}  
int main(void)  
{  
yyparse();  
return 0;  
getch();  
}
```

OUTPUT:



CONCLUSION:

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

Implementation of Three Address Code using LEX and YACC

Ex.No:4

Date:

AIM:

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

ALGORITHM:

Step1: A Yacc source program has three parts as follows

Declarations %% translation rules %% supporting C routines

Step2: Declarations Section: This section contains entries that:

- i. Include standard I/O header file.
- ii. Define global variables.
- iii. Define the list rule as the place to start processing.
- iv. Define the tokens used by the parser. v. Define the operators and their precedence.

Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

Step5: Main- The required main program that calls the yyparse subroutine to start the program.

Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.

Step7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmer file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.

Step8: calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM:

LEX part (three.l):

```
%{  
  
#include"y.tab.h"  
  
extern char yyval;  
  
%}  
  
%%  
  
[0-9]+ {yyval.symbol=(char)(yytext[0]);return NUMBER;}  
  
[a-z] {yyval.symbol= (char)(yytext[0]);return LETTER;}  
  
. {return yytext[0];}  
  
\n {return 0;}  
  
%%
```

YACC Part (three.y):

```
%{  
  
#include"y.tab.h"  
  
#include<stdio.h>  
  
char addtotable(char,char,char);
```

```
int index1=0;

char temp = 'A'-1;

struct expr{

char operand1;

char operand2;

char operator;

char result;

};

%}

%union{

char symbol;

}

%left '+' '-'

%left '/' '*'

%token <symbol> LETTER NUMBER

%type <symbol> exp

%%

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

exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}

|exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}

|exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}
```

```
|exp '*' exp {$$ = addtotable((char)$1,(char)$3,'*');}
```

```
|(' exp ') {$$= (char)$2;}
```

```
|NUMBER {$$ = (char)$1;}
```

```
|LETTER {(char)$1};;
```

```
%%
```

```
struct expr arr[20];
```

```
void yyerror(char *s){
```

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

```
}
```

```
char addtotable(char a, char b, char o){
```

```
    temp++;
```

```
    arr[index1].operand1 =a;
```

```
    arr[index1].operand2 = b;
```

```
    arr[index1].operator = o;
```

```
    arr[index1].result=temp;
```

```
    index1++;
```

```
    return temp;
```

```
}
```

```
void threeAdd()
```

```
{
```

```
    int i=0;
```

```
char temp='A';

while(i<index1){

    printf("%c:=\t",arr[i].result);

    printf("%c\t",arr[i].operand1);

    printf("%c\t",arr[i].operator);

    printf("%c\t",arr[i].operand2);

    i++;

    temp++;

    printf("\n");

}

}

int find(char l){

    int i;

    for(i=0;i<index1;i++)

        if(arr[i].result==l) break;

    return i;

}

int yywrap(){

    return 1;

}

void main()
```



```
{  
  
    printf("Enter the expression: ");  
  
    yyparse();  
  
    threeAdd();  
  
    printf("\n");  
  
    getch();  
  
}%{  
  
#include"y.tab.h"  
  
#include<stdio.h>  
  
char addtotable(char,char,char);  
  
int index1=0;  
  
char temp = 'A'-1;  
  
struct expr{  
  
    char operand1;  
  
    char operand2;  
  
    char operator;  
  
    char result;  
  
};  
  
%}  
  
%union{  
  
    char symbol;
```

```
}
```

```
%left '+' '-'
```

```
%left '/' '*'
```

```
%token <symbol> LETTER NUMBER
```

```
%type <symbol> exp
```

```
%%
```

```
statement: LETTER '=' exp ';' {addtotable((char)$1,(char)$3,'=');};
```

```
exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}
```

```
|exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}
```

```
|exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}
```

```
|exp '*' exp {$$ = addtotable((char)$1,(char)$3,'*');}
```

```
| '(' exp ')' {$$ = (char)$2;}
```

```
|NUMBER {$$ = (char)$1;}
```

```
|LETTER {(char)$1};
```

```
%%
```

```
struct expr arr[20];
```

```
void yyerror(char *s){
```

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

```
}
```

```
char addtotable(char a, char b, char o){
```

```
    temp++;
```

```
arr[index1].operand1 =a;

arr[index1].operand2 = b;

arr[index1].operator = o;

arr[index1].result=temp;

index1++;

return temp;

}

void threeAdd()

{

int i=0;

char temp='A';

while(i<index1){

printf("%c:=\t",arr[i].result);

printf("%c\t",arr[i].operand1);

printf("%c\t",arr[i].operator);

printf("%c\t",arr[i].operand2);

i++;

temp++;

printf("\n");

}

}
```

```
int find(char l){

    int i;

    for(i=0;i<index1;i++)

        if(arr[i].result==l) break;

    return i;

}

int yywrap(){

    return 1;

}

void main()

{

    printf("Enter the expression: ");

    yyparse();

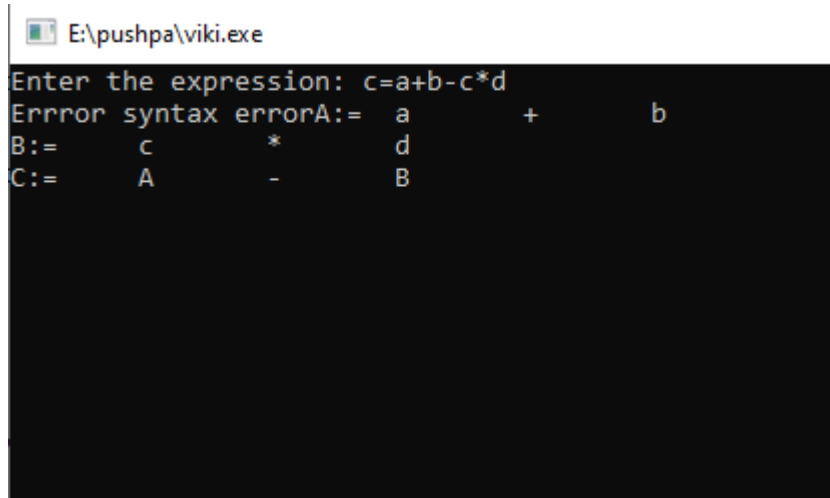
    threeAdd();

    printf("\n");

    getch();

}
```

OUTPUT:



```
E:\pushpa\wiki.exe
Enter the expression: c=a+b-c*d
Error syntax error
A:= a + b
B:= c * d
C:= A - B
```

CONCLUSION:

Thus a program to implement Three Address Code using LEX and YACC was executed successfully.

IMPLEMENTATION OF TYPE CHECKING

Ex.No:5

Date:

AIM:

To write a C program to implement type checking.

ALGORITHM:

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

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

Step3: If type found correct, do the operation

Step4: Type mismatches, semantic error will be notified

PROGRAM :

```
//To implement type checking
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n,i,k,flag=0;
    char vari[15],typ[15],b[15],c;
    printf("Enter the number of variables:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter the variable[%d]:",i);
        scanf("%c",&vari[i]);
        printf("Enter the variable-type[%d](float-f,int-i):",i);
        scanf("%c",&typ[i]);
        if(typ[i]=='f')
            flag=1;
    }
}
```

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

OUTPUT:

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

CONCLUSION:

Thus a program to implement type checking was executed successfully.

IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES

EX.NO:6

Date:

a)Dead Code Elimination

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

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

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

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

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

Step 6: Terminate the program

PROGRAM:

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

```
char temp,t;
char *tem;
clrscr();
printf("Enter the Number of Values:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("left: ");
op[i].l=getche();
printf("\tright: ");
scanf("%s",op[i].r);
}
printf("Intermediate Code\n") ;
for(i=0;i<n;i++)
{
printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
temp=op[i].l;
for(j=0;j<n;j++)
{
p=strchr(op[j].r,temp);
if(p)
{
pr[z].l=op[i].l;
strcpy(pr[z].r,op[i].r);
z++;
}
}
}
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
```

```

z++;
printf("\nAfter Dead Code Elimination\n");
for(k=0;k<z;k++)
{
printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++)
{
tem=pr[m].r;
for(j=m+1;j<z;j++)
{
p=strstr(tem,pr[j].r);
if(p)
{
t=pr[j].l;
pr[j].l=pr[m].l;
for(i=0;i<z;i++)
{
l=strchr(pr[i].r,t) ;
if(l)
{
a=l-pr[i].r;
printf("pos: %d",a);
pr[i].r[a]=pr[m].l;
}
}
}
}
printf("Eliminate Common Expression\n");
for(i=0;i<z;i++)
{
printf("%c\t=",pr[i].l);

```

```

printf("%s\n",pr[i].r);
}
for(i=0;i<z;i++)
{
for(j=i+1;j<z;j++)
{
q=strcmp(pr[i].r,pr[j].r);
if((pr[i].l==pr[j].l)&&!q)
{
pr[i].l='\0';
strcpy(pr[i].r,'\0');
}
}
}
printf("Optimized Code\n");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0')
{
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}
}
getch();
}

```

OUTPUT:

```

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

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

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

```

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

Aim:

To write a C program to implement Code Optimization Techniques.

loop-invariant code movement or code motion:

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

PROGRAM CODE:

```

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

```

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

OUTPUT:

Output without Code movement Technique :

Sum of First 5 Numbers:15

C)strength reduction :

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

PROGRAM CODE:

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

OUTPUT:

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

CONCLUSION:

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

IMPLEMENTING THE BACK END OF THE COMPILER

Ex.No:7

Date:

AIM:

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

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

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

Step 4: Get the Intermediate Code as the input.

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

Step 6: Terminate the program.

PROGRAM CODE:

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

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

```
scanf("%s",icode[i]);
} while(strcmp(icode[i++],"exit")!=0);
printf("\n target code generation");
printf("\n*****");
i=0;
do
{
strcpy(str,icode[i]);
switch(str[3])
{
case '+':
strcpy(opr,"ADD");
break;
case '-':
strcpy(opr,"SUB");
break;
case '*':
strcpy(opr,"MUL");
break;
case '/':
strcpy(opr,"DIV");
break;
}
printf("\n\tMov %c,R%d",str[2],i);
printf("\n\t%s%c,R%d",opr,str[4],i);
printf("\n\tMovR%d,%c",i,str[0]);
} while(strcmp(icode[++i],"exit")!=0);
getch();
}
```


OUTPUT:

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

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

CONCLUSION:

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