GRACE COLLEGE OF ENGINEERING

(Approved by AICTE, New Delhi & Affiliated to ANNA University, Chennai)

MULLAKKADU, THOOTHUKUDI - 628 005

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BE- Computer Science and Engineering

Anna University Regulation: 2021

CS3501- COMPILER DESIGN

III Year/V Semester

LAB MANUAL

Prepared By,

Mrs.J.SWEETLINE ARPUTHAM, AP/CSE

Ex.No:1	Develop a lexical analyzer to recognize a few patterns in C. (Ex.identifiers, constants, comments, operators etc.).

AIM:

To Write a LEX program to recognize few patterns in C.

ALGORITHM:

Step 1: Take the C code as input.

Step 2: Initialize an empty list to store tokens.

Read the input code character by character.

Step 3: Lexical Analysis:

Identifiers:

Start reading characters. If the first character is a letter (a to z or A to Z) or an underscore (_), continue reading characters until a non-letter, non-digit, or non-underscore character is encountered. Add the identified word as an identifier token to the list.

Constants (Integers and Floating-Point Numbers):

If the current character is a digit, start reading characters until a non-digit or a dot (.) is encountered. If a dot is encountered, continue reading digits as it might be a floating-point number. Add the identified number as a constant token to the list.

Operators and Special Characters:

Check for operators and special characters such as +, -, *, /, =, ==, !=, <, >, etc. Identify these characters individually and add them as operator tokens to the list.

Comments:

Check for comment patterns such as // for single-line comments and /* */ for multi-line comments. Ignore the characters inside comments as they are not part of the tokens.

Whitespace and Newlines:

Ignore whitespace characters, tabs, and newline characters as they are typically not part of the tokens. They are used for code formatting.

Step 4: Output the list of tokens identified during the lexical analysis. Each token should be labelled with its type (identifier, constant, operator, etc.) and the actual value (if applicable).

PROGRAM:

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

```
    #include<stdlib.h>
    #include<string.h>
    #define MAX_IDENTIFIER_LENGTH 50
%}
%option noyywrap
%option yylineno
void addtosymboltable(const char*)
%%
[\t] /*ignore whitespace*/
\n /*ignore newline*/
\/\*([^*]|\*+[^*/])*\*+\/ /*ignore comments*/
[0-9]+ {printf("Constant:%s \n",yytext);}
= {printf("%s is an Assignment Operator\n",yytext);}
\* |
\+ |
\- {printf("%s is a Operator\n",yytext);}
[a-zA-Z][a-zA-Z0-9]* {printf("identifier:%s\n",yytext);addtosymboltable(yytext);}
. {printf("Invalid token:%s \n",yytext);}
%%
typedef struct
{
    char name[MAX_IDENTIFIER_LENGTH];
} Symbol;
Symbol symbol_table[100];
int symbol_count=0;
void addtosymboltable(const char * identifier){
    if(symbol_count < 100)
    {
    strncpy(symbol_table[symbol_count].name,identifier,MAX_IDENTIFIER_LENGTH-1);
    symbol_table[symbol_count].name[MAX_IDENTIFIER_LENGTH-1]='\0';
```

```
    symbol_count++;
    printf("Identifier %s is entered in the symbol table\n",identifier);
    }
    else
    {
        printf("Symbol table is full.Cannot add more identifier.\n");
        exit(0);
    }
}
int main()
{
    yylex();
    return 1;
}
```

OUTPUT:

RESULT:

Thus the program for developing a lexical analyzer to recognize a few patterns in C has been executed successfully.

Ex.No:2	Implement a Lexical Analyzer using Lex Tool

AIM

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

ALGORITHM

Step 1: Declare and Initialize the required variable.

Step 2: If the statement starts with #.* print it as preprocessor directive.

Step 3: Check for the given list of keywords and print them as keyword if it is encountered.

Step 4: If the given string is '/*' or '*/' print it as comment line.

Step 5: For a function, print the beginning and ending of the function block.

Step 6: Similarly print the corresponding statements for numbers, identifiers and assignment operators.

Step 7: In the main function get the input file as argument and open the file in read mode.

Step 8: Then read the file and print the corresponding lex statement given above.

Program Code

```
%{
int COMMENT=0;
%}
identifier[a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n %s is a preprocessor directive",yytext);}
int |
float |
void |
main |
if |
else |
printf |
scanf |
for |
```

```
char |
getch |
while {printf("\n %s is a keyword",yytext);}
"/*" {COMMENT=1;}
"*/" {COMMENT=0;}
{identifier}\( {if(!COMMENT)printf("\n Function:\t %s",yytext);}
\{ {if(!COMMENT)printf("\n Block begins");}
\} {if(!COMMENT)printf("\n Block ends");}
{identifier}(\[[0-9]*\])? {if(!COMMENT)printf("\n %s is an Identifier",yytext);}
\".*\" {if(!COMMENT)printf("\n %s is a string",yytext);}
[0-9]+ {if(!COMMENT)printf("\n %s is a number",yytext);}
\)(\;)? {if(!COMMENT)printf("\t");ECHO;printf("\n");}
\(ECHO;
= {if(!COMMENT)printf("\n %s is an Assignment operator",yytext);}
\<= |
\>= |
\< |
== {if(!COMMENT)printf("\n %s is a relational operator",yytext);}
.|\n
%%
int main(int argc, char **argv)
{
    if(argc>1)
    {
      FILE*file;
      file=fopen(argv[1],"r");
      if(!file)
      {
       printf("\n Could not open the file:%s ",argv[1]);
       exit(0);
      }
      yyin=file;
    }
    yylex();
    printf("\n\n");
```

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

Output:

Result:

Thus the program for implementing a Lexical analyser using LEX tool was executed successfully.

Ex.No:3a	Program to recognize a valid arithmetic expression that uses operator + , -, * and /.

AIM

To write a program for recognizing a valid arithmetic expression that uses the operator +,-* and / using semantic rules of the YACC tool and LEX.

ALGORITHM

Step 1: A Yacc source program has three parts as follows: Declarations %% translation rules %% supporting C routines

Step 2: Declarations Section: This section contains entries that:

Include standard I/O header file.

Define global variables.

Define the list rule as the place to start processing.

Define the tokens used by the parser.

Step 3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step 4: 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.

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

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

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.

Step 5:calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM CODE

LEX PART: ex3a.l

```
%{
   #include "ex3a.tab.h"
%}
%%
"=" {printf("\n Operator is EQUAL");}
```

```
"+" {printf("\n Operator is PLUS");}
"-" {printf("\n Operator is MINUS");}
"/" {printf("\n Operator is DIVISION");}
"*" {printf("\n Operator is MULTIPLICATION");}
[a-zA-Z]*[0-9]* {printf("\n Identifier is %s",yytext);return ID;}
. return yytext[0];
\n return 0;
%%
int yywrap()
{
   return 1;
}
```

YACC PART: ex3a.y

```
%{
   #include<stdio.h>
%}
%token A
%token ID

%%
statement:A'='E
| E{
   printf("\n Valid arithmetic expression");
   $$=$1;
};
E:E'+'ID
|E'-'ID
|E'*'ID
|E'/'ID
|ID
;
%%
```

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

Output:

```
[root@localhost]# lex ex3a.1
[root@localhost]# yacc –d ex3a.y
[root@localhost]# gcc lex.yy.c ex3a.tab.c
[root@localhost]# ./a.out
x=a+b;

Identifier is x
Operator is EQUAL
Identifier is a
Operator is PLUS
Identifier is b
```


RESULT:

Thus the program was executed successfully.

Ex.No:3b	**Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.**

AIM

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

ALGORITHM

Step 1: A Yacc source program has three parts as follows: Declarations %% translation rules %% supporting C routines

Step 2: Declarations Section: This section contains entries that:

Include standard I/O header file.

Define global variables.

Define the list rule as the place to start processing.

Define the tokens used by the parser.

Step 3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step 4: 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.

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

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

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.

Step 5:calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM CODE

LEX PART: ex3b.l

```
%{
   #include "ex3b.tab.h"
%}
```

```
%%
"int" {return INT;}
"float" {return FLOAT;}
"double" {return DOUBLE;}
[a-zA-Z]*[0-9]* {printf("\n Identifier is %s",yytext);return ID;}
. return yytext[0];
\n return 0;

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

YACC PART: ex3b.y

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

```
[root@localhost]# lex ex3b.l
[root@localhost]# yacc –d ex3b.y
[root@localhost]# gcc lex.yy.c ex3b.tab.c
[root@localhost]# ./a.out
int a,b;

Identifier is a
Identifier is b
```


RESULT:

Thus the program was executed successfully.

Ex.No:3c	Program to recognize a valid control structures syntax of c language(For loop, While loop, if-else, if-else-if, switch case, etc.,

AIM

To write a program to recognize a valid control structures syntax of c language (For loop, While loop, if-else, if-else-if, switch case, etc.

ALGORITHM

Step 1: Take the C code as input.

Step 2: Tokenize the input code. Split the code into individual tokens (keywords, identifiers, operators, etc.) for easier processing.

Step 3: Syntax Checking:

For Loop:

Check for tokens that match the pattern: for (initialization; condition; increment) {/* code */}

Ensure proper semicolons separating initialization, condition, and increment parts.

Recursively check the code inside the loop using the same steps.

While Loop:

Check for tokens that match the pattern: while (condition) {/* code */}

Ensure proper parentheses and braces.

Recursively check the code inside the loop.

If-Else Statements:

Check for tokens that match the pattern: if (condition) {/* code */} else {/* code */}

Ensure proper parentheses and braces for both if and else parts.

Recursively check the code inside both if and else blocks.

Else-If Ladder:

Check for tokens that match the pattern:

if (condition) { /* code */ } else if (condition) { /* code */ } ... else { /* code */ }

Ensure proper parentheses and braces for each if and else block.

Recursively check the code inside each block.

Switch-Case Statements:

Check for tokens that match the pattern:

switch (variable) { case constant1: /* code */ break; case constant2: /* code */ break; ... default: /* code */ break; }

Ensure proper parentheses, colons after cases, and break statements.

Recursively check the code inside each case and default block.

Step 4: If the code passes all syntax checks, output a message indicating that the control structures are valid. Otherwise, indicate the specific error encountered during the parsing process.

PROGRAM:

LEX PART: ex3c.l

```
%{
#include<stdio.h>
#include "ex3c.tab.h"
%}
%%
"if"            { return IF; }
"else"          { return ELSE; }
"while"         { return WHILE; }
"for"           { return FOR; }
"switch"        { return SWITCH; }
"case"          { return CASE; }
"default"       { return DEFAULT; }
"break"         { return BREAK; }
"("             { return OPEN_BRACKET; }
")"             { return CLOSE_BRACKET; }
"{"             { return OPEN_BRACE; }
"}"             { return CLOSE_BRACE; }
";"             { return SEMICOLON; }
[\t\n]          ;
.               ;
%%
int yywrap()
{
```

```
    return 1;
}
```

YACC PART: ex3c.y

```
%{
#include<stdio.h>
int yylex();
%}

%token IF ELSE WHILE FOR SWITCH CASE DEFAULT OPEN_BRACE CLOSE_BRACE
SEMICOLON COLON OPEN_BRACKET CLOSE_BRACKET BREAK

%%
program: statement
    | program statement
    ;

statement:if_statement
    |while_loop
    |switch_case_statement
    |for_loop
    ;

if_statement:IF OPEN_BRACKET expression_opt CLOSE_BRACKET OPEN_BRACE
expression_opt CLOSE_BRACE ELSE OPEN_BRACE expression_opt CLOSE_BRACE
        {
          printf("Recognized IF Else statement\n");
        }
        ;
while_loop:WHILE OPEN_BRACKET expression_opt CLOSE_BRACKET OPEN_BRACE
expression_opt CLOSE_BRACE
        {
          printf("Recognized WHILE loop\n");
        }
```

```
        ;
switch_case_statement:SWITCH OPEN_BRACKET expression_opt CLOSE_BRACKET
OPEN_BRACE case_list CLOSE_BRACE
        {
          printf("Recognized SWITCH_CASE statement with DEFAULT\n");
        }
        ;
for_loop:FOR OPEN_BRACKET expression_opt SEMICOLON expression_opt
CLOSE_BRACKET OPEN_BRACE expression_opt CLOSE_BRACE
        {
          printf("Recognized FOR loop\n");
        }
        ;
case_list:CASE expression COLON expression BREAK
       SEMICOLON DEFAULT COLON expression_opt
       ;
expression_opt:/*empty*/
          |expression
          |expression SEMICOLON
          ;
expression:
       ;
%%
int yyerror(const char *s)
{
   fprintf(stderr,"Error=%s\n", s);
   return 1;
}

int main() {
   if(yyparse()==0){
      printf("Parsing completed successfully\n");
```

```
    }
    else{
        fprintf(stderr,"Parsing encountered errors\n");
    }
    return 0;
}
```

OUTPUT:

```
for(i=0;i<10;i++)
{
a=c+b
}
Recognized FOR loop
if(a<b)
{
a=a+b
}
else
{
a=a-b
}
Recognized IF ELSE statement
```


RESULT:

Thus the program was executed successfully.

Ex.No:3d	Implement an Arithmetic Calculator using LEX and YACC

AIM

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

ALGORITHM

1.A Yacc source program has three parts as follows: Declarations %% translation rules %% supporting C routines

2.Declarations Section: This section contains entries that:

Include standard I/O header file.

Define global variables.

Define the list rule as the place to start processing.

Define the tokens used by the parser.

Define the operators and their precedence.

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

4.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.

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

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

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

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

PROGRAM CODE

LEX PART: ex3d.l

```
%{
#include <stdio.h>
#include "ex3d.tab.h"
extern int yylval;
%}
%%
```

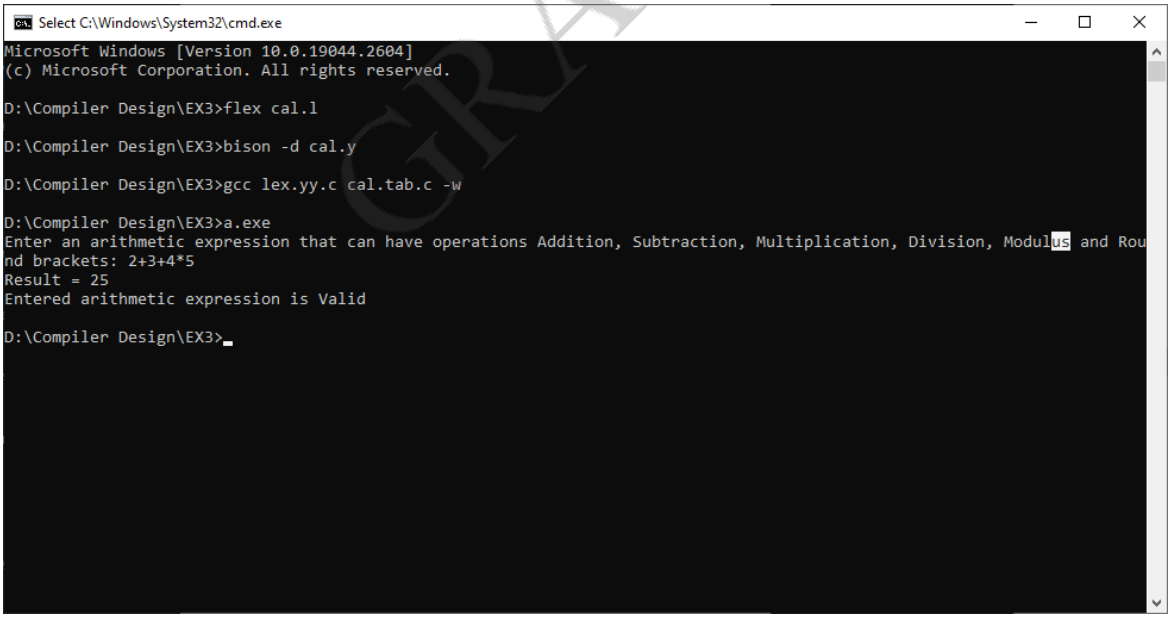
```
[0-9]+ {
yylval = atoi(yytext);
return NUMBER;
}
[ \t] ;
[\n] return 0;
. {return yytext[0];}
%%
int yywrap()
{
return 1;
}
```

YACC PART: ex3d.y

```
%{
#include <stdio.h>
int flag = 0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E {
printf("Result = %d\n", $1);
return 0;
};
E: E '+' E { $$ = $1 + $3; }
| E '-' E { $$ = $1 - $3; }
| E '*' E { $$ = $1 * $3; }
| E '/' E { $$ = $1 / $3; }
| E '%' E { $$ = $1 % $3; }
| '(' E ')' { $$ = $2; }
| NUMBER { $$ = $1; }
```

```
;
%%
int main()
{
printf("\nEnter an arithmetic expression that can have operations Addition, Subtraction, Multiplication,Division, Modulus and Round brackets: ");
yyparse();
if (flag == 0) {
printf("\nEntered arithmetic expression is Valid\n");
}
return 0;
}
void yyerror(const char* s)
{
printf("\nEntered arithmetic expression is Invalid\n");
flag = 1;
}
```

OUTPUT



Result:

Thus the program for implementing a calculator for computing the given expression using semantic rules of the YACC tool and LEX was executed successfully.

Ex.No: 4	Generate three address code for a simple program using LEX and YACC

AIM:

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

ALGORITHM:

Step 1: Lexer (Lex):

Define lexical rules using Lex to tokenize the input code. Specify regular expressions for identifiers, constants, operators, and keywords. Lex generates tokens for recognized patterns.

Step 2: Parser (Yacc):

Define grammar rules using Yacc for expressions, statements, assignments, and control structures. Attach actions to productions for generating three-address code. Use semantic actions to handle grammar rules.

Step 3: Symbol Table:

Create a symbol table to store identifiers, their types, and memory locations. Update the symbol table during the parsing process. Assign memory addresses to variables during declarations.

Step 4: Intermediate Code Generation:

Integrate intermediate code generation logic within Yacc actions. When parsing expressions or statements, generate corresponding three-address code instructions. Store generated code in an intermediate code representation.

Step 5: After parsing the input code, the intermediate code will be generated based on the specified rules. Store or process the generated three-address code for further optimization or translation to machine code if necessary.

PROGRAM:

LEX PART: ex4.l

```
%{
   #include<stdio.h>
   #include "ex4.tab.h"
%}
%%
```

```
[0-9]+ {yylval=atoi(yytext);return NUM;}
[\t]   ;
\n   {return EOL;}
[-+*/()] {return yytext[0];}
. {fprintf(stderr,"Error:Invalid Character\n");}
%%
int yywrap(){
    return 1;
}
```

YACC PART: ex4.y

```
%{
    #include<stdio.h>
    #include<stdlib.h>
    int temp_count=0;
    void yyerror(const char*s){
        fprintf(stderr,"Error:%s\n",s);
    }
%}
%token NUM EOL
%left '+' '-'
%left '*' '/'
%%
program:lines
    ;
lines:lines line
    | line
    ;
line:expr EOL
```

```
{
    printf("Result:t%d\n",$1);
}
;
expr:NUM{
    $$=$1;
}
| '(' expr ')'
{
    $$=$2;
}
| expr '+' expr
{
    printf("t%d=%d+%d\n",++temp_count,$1,$3);
    $$=temp_count;
}
| expr '-' expr
{
    printf("t%d=%d-%d\n",++temp_count,$1,$3);
    $$=temp_count;
}
| expr '*' expr
{
    printf("t%d=%d*%d\n",++temp_count,$1,$3);
    $$=temp_count;
}
| expr '/' expr
{
    if($3==0)
    {yyerror("Division by zero");
```

```
    $$=0;}
    else{
    printf("t%d=%d/%d\n",++temp_count,$1,$3);
    $$=temp_count;
    }
}
;
%%
int main()
{
    yyparse();
    return 0;
}
```

OUTPUT:

```
2*10/2+5-1
t1=2*10
t2=1/2
t3=2+5
t4=3-1
Result:t4
```

RESULT:

Thus the program for Generate three address code for a simple program using LEX and YACC was executed successfully.

Ex.No:5	Implement type checking using Lex and Yacc.

AIM:

To write a program for implement type checking using Lex and Yacc.

ALGORITHM:

Step 1: Define Grammar Rules:

Specify the grammar rules for the programming language, including expressions, statements, declarations, and data types. Incorporate type information where applicable.

Step 2: Lexer (Lex):

Implement lexical analysis using Lex to tokenize the input code. Define regular expressions for identifiers, keywords, operators, constants, and other language constructs. Lex generates tokens for recognized patterns.

Step 3: Symbol Table:

Create a symbol table data structure to store identifiers, their declared types, and other relevant information during the parsing process. Initialize the symbol table.

Step 4: Parser (Yacc):

Define Yacc grammar rules based on the language's syntax. Include actions within Yacc rules to handle type checking logic.

During parsing, populate the symbol table with identifier names and their declared types. Handle type conversions and promotions within Yacc actions.

Step 5: Type Checking Logic:

Implement type checking logic within Yacc actions. Perform type comparisons and validations according to the language's rules. For example:

- Check compatibility of operand types in expressions.
- Verify function return types match declared types.
- Ensure compatibility in assignments and comparisons.
- Handle type coercion and promotion.

Step 6: Error Handling:

Implement error handling mechanisms within Yacc actions for type mismatch errors. Generate meaningful error messages when type mismatches are detected, indicating the source of the error.

Program:

lex part:

```
%{

#include "type.tab.h"
```

```
%}
%%
[0-9]+ {
    yylval = atoi(yytext);
    return INTEGER;
}
[0-9]+"."[0-9]* {
    yylval = atof(yytext);
    return FLOAT;
}
[a-zA-Z]+ {
    yylval= yytext;
    return CHAR;
}
[ \t] ;  // Ignore whitespace and tabs
\n { return EOL; }  // Newline character
. { return yytext[0]; }  // Return other characters as is
%%
int yywrap() {
    return 1;
}
```

Yacc part:

```
%{
#include <stdio.h>
void yyerror(const char* s) {
    fprintf(stderr, "Parse error: %s\n", s);
}
int yylex();  // Declare the lexer function
%}
%token INTEGER FLOAT  CHAR EOL
%%
```

```
program:
    /* empty */
    | program line
    ;
line:
    statement EOL {
        if ($1 == INTEGER) {
            printf("Type: INTEGER\n");
        } else if ($1 == FLOAT) {
            printf("Type: FLOAT\n");
        }
        else if ($1 == CHAR) {
            printf("Type: CHAR/STRING\n");
        } else {
            printf("Invalid type\n");
        }
    }
    ;
statement:
    expression {
        $$ = $1;
    }
    ;
expression:
    INTEGER {
        $$ = INTEGER;
    }
    | FLOAT {
        $$ = FLOAT;
    }
    | CHAR {
        $$ = CHAR;
```

```
    }
    ;
%%
int main() {
    yyparse();
    return 0;
}
```

Output:

123

Type:INTEGER

123.897

Type:FLOAT

God

Type:CHAR/STRING

df24

Invalid type or Parse error

Result:

Thus the program for implement type checking using Lex and Yacc was written and executed successfully.

Ex.No:6	Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)

AIM

To write a program for implementation of Code Optimization Technique.

ALGORITHM

1. Generate the program for factorial program using for and do-while loop to specify optimization technique.
2. In for loop variable initialization is activated first and the condition is checked next. If the condition is true the corresponding statements are executed and specified increment / decrement operation is performed.
3. The for loop operation is activated till the condition failure.
4. In do-while loop the variable is initialized and the statements are executed then the condition checking and increment / decrement operation is performed.
5. When comparing both for and do-while loop for optimization dowhile is best because first the statement execution is done then only the condition is checked. So, during the statement execution itself we can find the inconvenience of the result and no need to wait for the specified condition result.
6. Finally when considering Code Optimization in loop do-while best with is respect to performance.

PROGRAM CODE

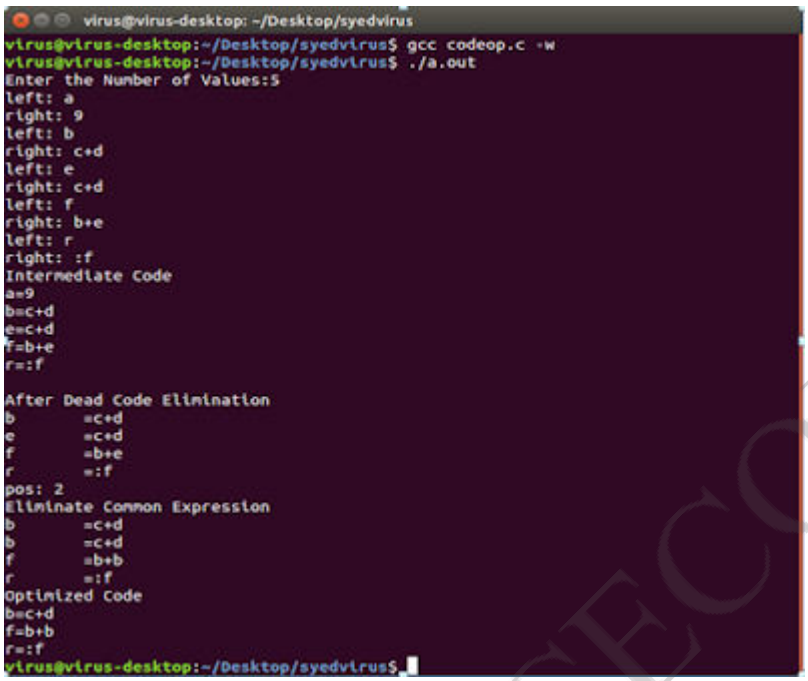
```
#include<stdio.h>
#include<string.h>
struct op
{
char l;
char r[20];
}
op[10],pr[10];
void main()
{
int a,i,k,j,n,z=0,m,q;
char *p,*l;
```

```
  char temp,t;
  char *tem;
printf("Enter the Number of Values:");
scanf("%d",&n);
  for(i=0;i<n;i++)
  {
printf("left: ");
scanf(" %c",&op[i].l);
printf("right: ");
scanf(" %s",&op[i].r);
  }
printf("Intermediate Code
") ;
  for(i=0;i<n;i++)
  {
printf("%c=",op[i].l);
printf("%s
",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("
After Dead Code Elimination
");
  for(k=0;k<z;k++)
  {
printf("%c      =",pr[k].l);
printf("%s
",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");
```

```
  for(i=0;i<z;i++)
  {
printf("%c      =",pr[i].l);
printf("%s",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='';
  }
  }
  }
printf("Optimized Code");
  for(i=0;i<z;i++)
  {
  if(pr[i].l!='')
  {
printf("%c=",pr[i].l);
printf("%s",pr[i].r);
  }
  }
  }
```

OUTPUT



Result:

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

Ex.No:7	**Implement back-end of the compiler for which the three address code is given as input and the 8086 assembly language code is produced as output.**

AIM

To write a C program to implement the Back end of the compiler.

ALGORITHM

1. Start the Program
2. Get the three variables from statements and stored in the text file k.txt.
3. Compile the program and give the path of the source file.
4. Execute the program.
5. Target code for the given statement was produced.
6. Stop 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(" Enter the set of intermediate code (terminated by exit):");
  do
  {
scanf("%s", icode[i]);
  } while (strcmp(icode[i++], "exit") != 0);
printf("
 target code generation");
printf("
************************");
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("
        Mov %c,R%d", str[2], i);
printf("
        %s%c,R%d", opr, str[4], i);
printf("
        Mov R%d,%c", i, str[0]);
} while (strcmp(icode[++i], "exit") != 0);
getch();
}
```

OUTPUT

```
Enter the set of intermediate code (terminated by exit):
a=a*b
c=f*h
g=a*h
f=Q+w
t=q-j
exit
target code generation
************************
Mov a,R0
MUL b,R0
Mov R0,a
Mov f,R1
MUL h,R1
Mov R1,c
Mov a,R2
MUL h,R2
Mov R2,g
Mov Q,R3
ADD w,R3
Mov R3,f
Mov q,R4
SUB j,R4
Mov R4,t
```

Result:

Thus the C program to implement the Back end of the compiler- for which the three address code is given as input and the 8086 assembly language code is produced as output was successfully executed.