Ex. No : 6 d) **Odd or Even**

Date :

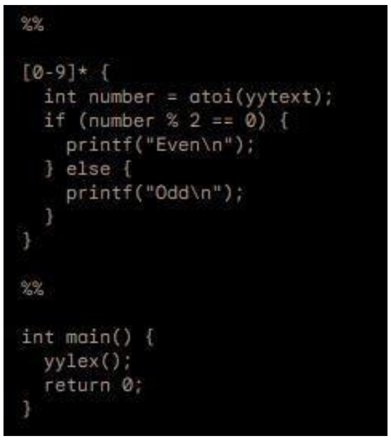
AIM:

To write a Lex program to decide whether number is odd or even.

Procedure:

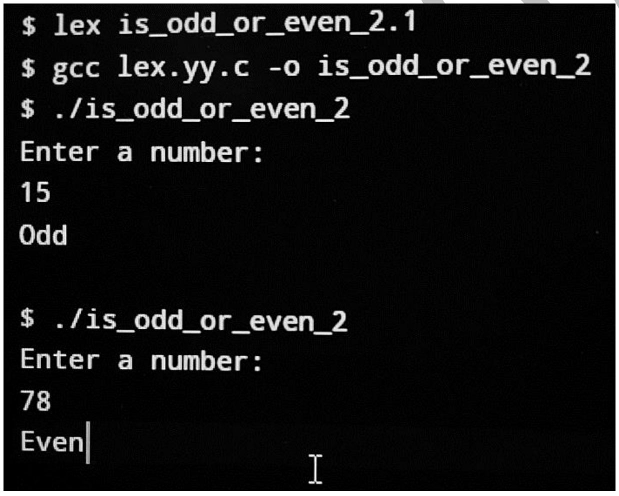
- Program uses the atoi() function to convert the string representation of the number to an integer before checking whether it is odd or even.
- This is necessary because the % operator in Lex only works on integers.

Code:



```
%%

[0-9]* {
  int number = atoi(yytext);
  if (number % 2 == 0) {
    printf("Even\n");
  } else {
    printf("Odd\n");
  }
}

%%

int main() {
  yylex();
  return 0;
}
```

Output:



```
$ lex is_odd_or_even_2.1
$ gcc lex.yy.c -o is_odd_or_even_2
$ ./is_odd_or_even_2
Enter a number:
15
Odd

$ ./is_odd_or_even_2
Enter a number:
78
Even
```

Result:

Thus, we executed a Lex program to decide whether number is odd or even.

Ex. No : 6 e) **Email Validation**

Date :

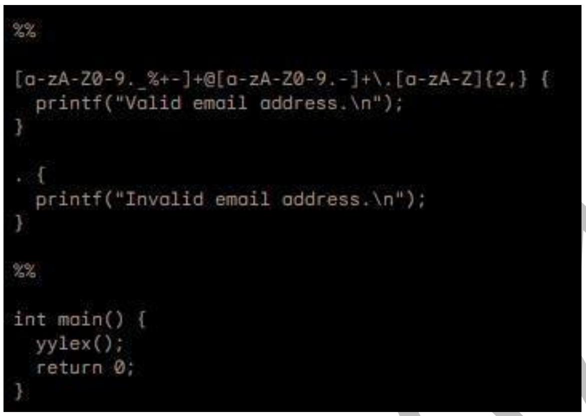
AIM:

To write a Lex program for email validation.

Procedure:

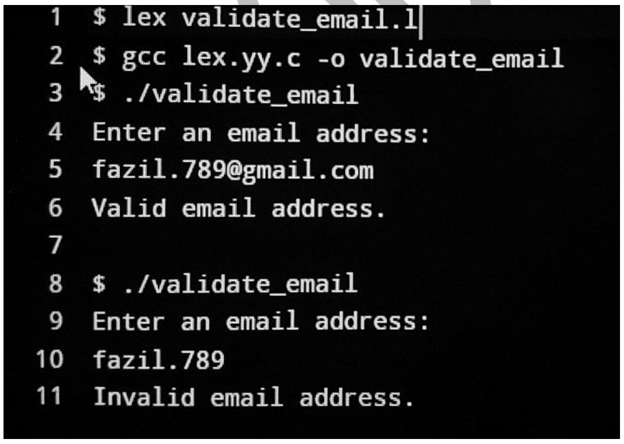
- This program will recognize any string that consists of alphanumeric characters, underscores, periods, percent signs, plus signs, and hyphens, followed by an @ symbol, followed by another string that consists of alphanumeric characters, periods, and hyphens, followed by a two-letter domain extension.
- If the string matches this pattern, the program will print "Valid email address."
- Otherwise, it will print "Invalid email address."

Code:



```
%%

[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,} {
  printf("Valid email address.\n");
}

. {
  printf("Invalid email address.\n");
}

%%

int main() {
  yylex();
  return 0;
}
```

Output:



```
$ lex validate_email.l
$ gcc lex.yy.c -o validate_email
$ ./validate_email
Enter an email address:
fazil.789@gmail.com
Valid email address.

$ ./validate_email
Enter an email address:
fazil.789
Invalid email address.
```

Result:

Thus, we executed a Lex program for email validation.

Ex. No : 6 f) **Validation of Date**

Date :

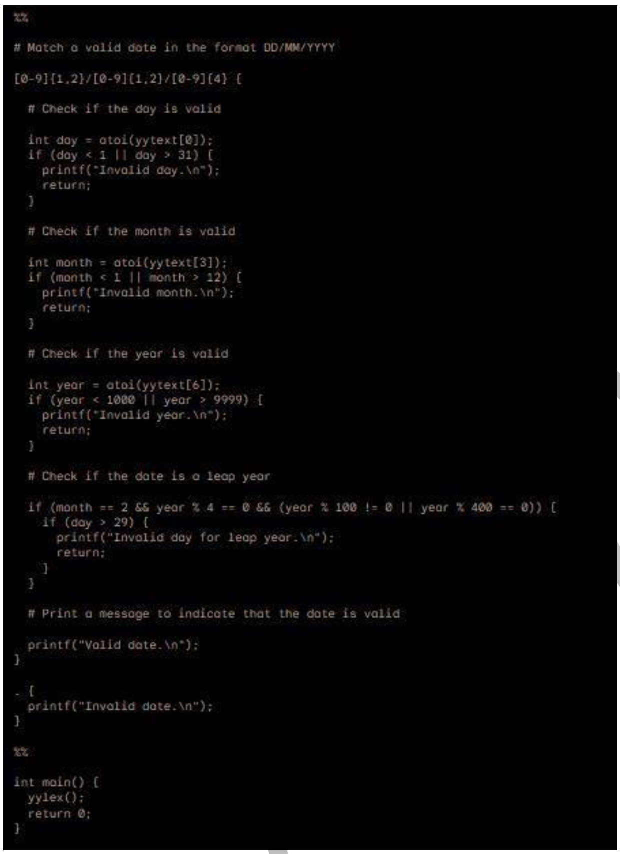
AIM:

To write a Lex program for validation of date.

Procedure:

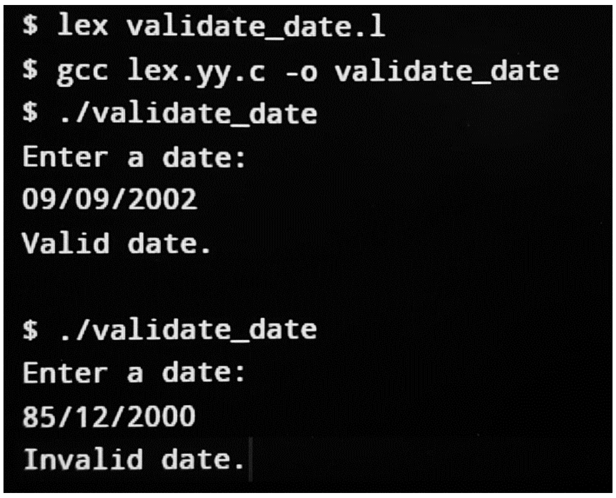
- This program will match a valid date in the format DD/MM/YYYY.
- It will also check if the day, month, and year are valid, and if the date is a leap year.
- If the date is valid, the program will print "Valid date."
- Otherwise, it will print "Invalid date."

Code:



```
%%

# Match a valid date in the format DD/MM/YYYY

[0-9]{1,2}/[0-9]{1,2}/[0-9]{4} {

  # Check if the day is valid

  int day = atoi(yytext[0]);
  if (day < 1 || day > 31) {
    printf("Invalid day.\n");
    return;
  }

  # Check if the month is valid

  int month = atoi(yytext[3]);
  if (month < 1 || month > 12) {
    printf("Invalid month.\n");
    return;
  }

  # Check if the year is valid

  int year = atoi(yytext[6]);
  if (year < 1000 || year > 9999) {
    printf("Invalid year.\n");
    return;
  }

  # Check if the date is a leap year

  if (month == 2 && year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) {
    if (day > 29) {
      printf("Invalid day for leap year.\n");
      return;
    }
  }

  # Print a message to indicate that the date is valid

  printf("Valid date.\n");
}

. {
  printf("Invalid date.\n");
}

%%

int main() {
  yylex();
  return 0;
}
```

Output:



```
$ lex validate_date.l
$ gcc lex.yy.c -o validate_date
$ ./validate_date
Enter a date:
09/09/2002
Valid date.

$ ./validate_date
Enter a date:
85/12/2000
Invalid date.
```

Result:

Thus, we executed a Lex program for validation of date.

Ex. No : 6 g) **String Starts with 'a'**

Date :

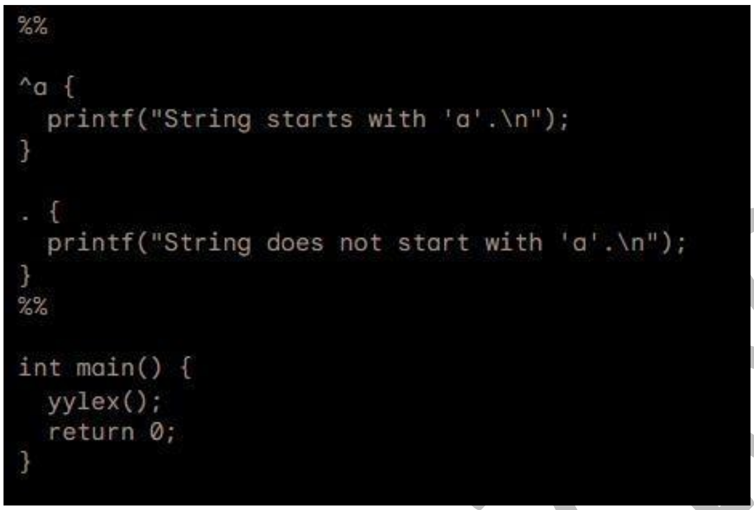
AIM:

To write a Lex program to determine if string starts with letter 'a' or not.

Procedure:

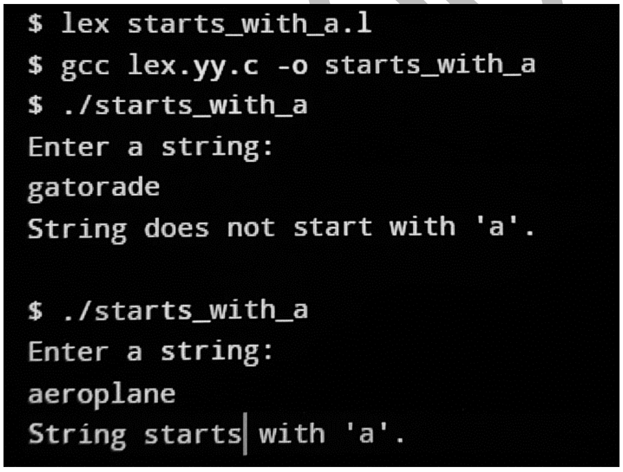
- This program will match any string that starts with the letter 'a'.
- If the string matches this pattern, the program will print "String starts with 'a'."
- Otherwise, it will print "String does not start with 'a'."

Code:



```
%%

^a {
  printf("String starts with 'a'.\n");
}

. {
  printf("String does not start with 'a'.\n");
}
%%

int main() {
  yylex();
  return 0;
}
```

Output:



```
$ lex starts_with_a.l
$ gcc lex.yy.c -o starts_with_a
$ ./starts_with_a
Enter a string:
gatorade
String does not start with 'a'.

$ ./starts_with_a
Enter a string:
aeroplane
String starts with 'a'.
```

Result:

Thus, we executed a Lex program to determine if string starts with 'a' or not.

Ex. No : 6 h) **Mobile Number Validation**

Date :

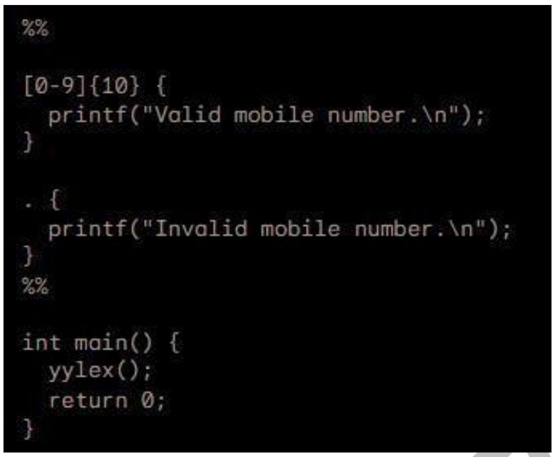
AIM:

To write a Lex program for mobile number validation.

Procedure:

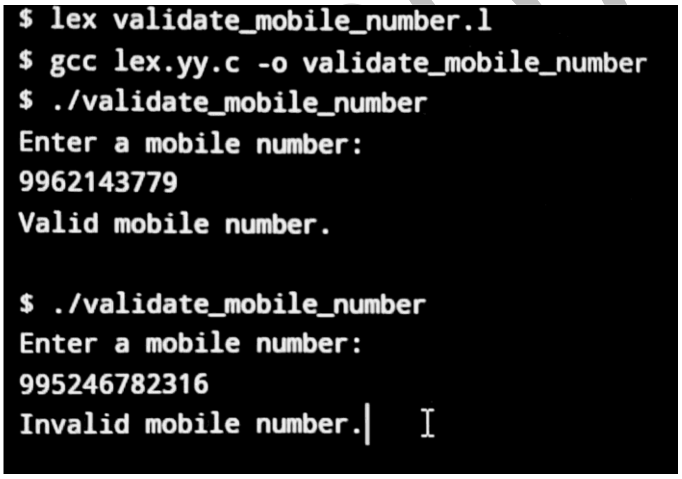
- This program will match any string that consists of 10 digits.
- If the string matches this pattern, the program will print "Valid mobile number."
- Otherwise, it will print "Invalid mobile number."

Code:



```
%%

[0-9]{10} {
  printf("Valid mobile number.\n");
}

. {
  printf("Invalid mobile number.\n");
}
%%

int main() {
  yylex();
  return 0;
}
```

Output:



```
$ lex validate_mobile_number.l
$ gcc lex.yy.c -o validate_mobile_number
$ ./validate_mobile_number
Enter a mobile number:
9962143779
Valid mobile number.

$ ./validate_mobile_number
Enter a mobile number:
995246782316
Invalid mobile number.
```

Result:

Thus, we executed a Lex program for mobile number validation.

Ex. No : 6 i) Lexical analyser (Token Separation) using LEX

Date :

AIM:

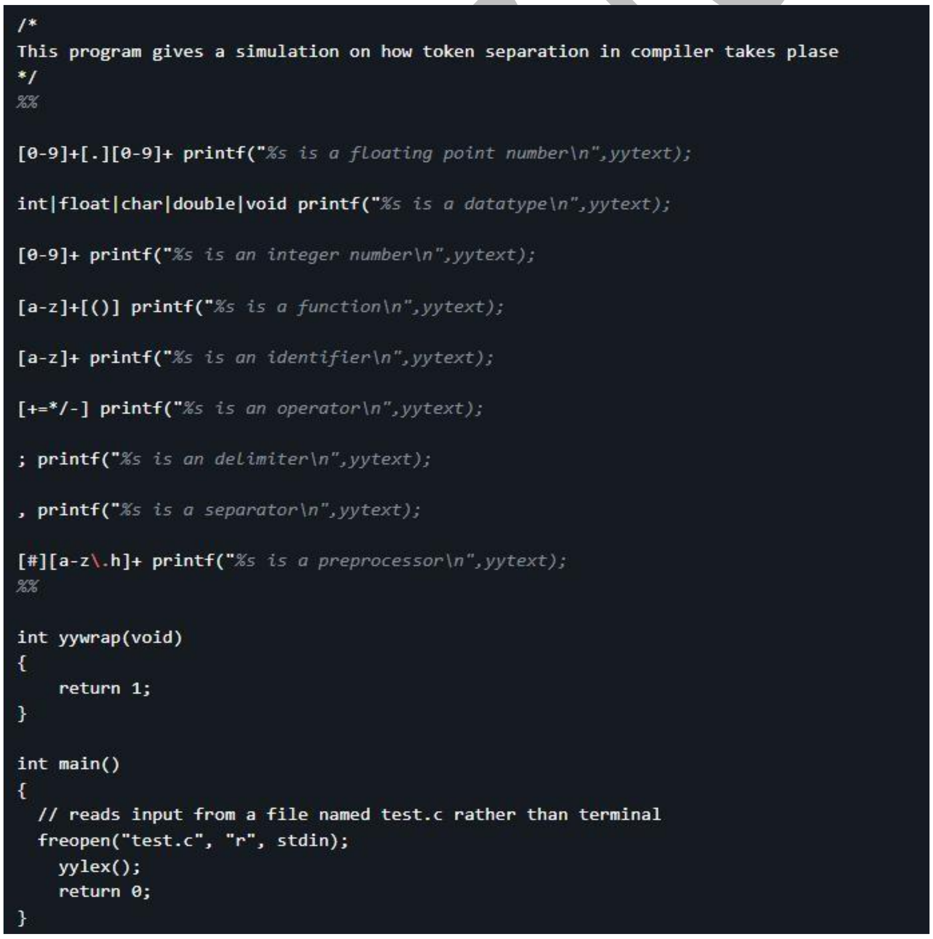
To write a Lex program for token separation.

Procedure:

- The Lex program for tokenizing C code will recognize the following tokens:
- Floating point numbers: Strings of digits followed by a period followed by more digits.
- Datatypes: The keywords int, float, char, double, and void.
- Integer numbers: Strings of digits.
- Functions: Strings of alphanumeric characters followed by parentheses.
- Identifiers: Strings of alphanumeric characters.
- Operators: The symbols +, -, *, and /.
- Delimiters: The semicolon ;.
- Separators: The comma ,.
- Preprocessor directives: Strings that start with the hash symbol # followed by alphanumeric characters and periods.

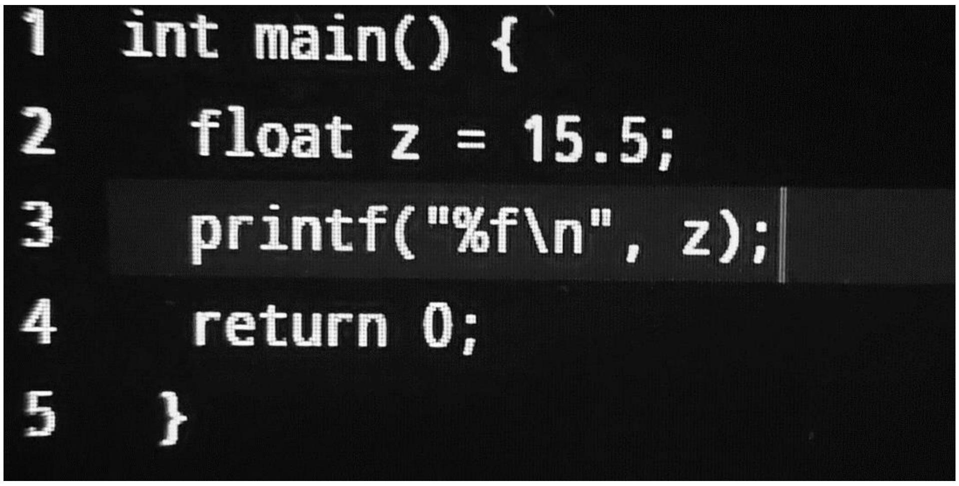
The program will read the input from the file test.c and tokenize it. It will then print the tokens to the standard output, along with their type.

Code:



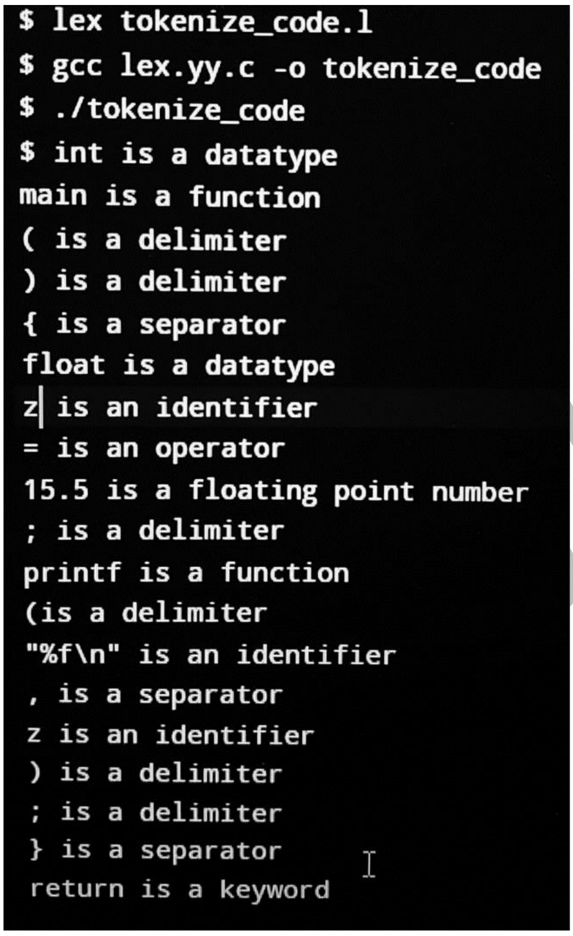
```
/*
This program gives a simulation on how token separation in compiler takes plase
*/
%%

[0-9]+[.][0-9]+ printf("%s is a floating point number\n",yytext);

int|float|char|double|void printf("%s is a datatype\n",yytext);

[0-9]+ printf("%s is an integer number\n",yytext);

[a-z]+[()] printf("%s is a function\n",yytext);

[a-z]+ printf("%s is an identifier\n",yytext);

[+=*/-] printf("%s is an operator\n",yytext);

; printf("%s is an delimiter\n",yytext);

, printf("%s is a separator\n",yytext);

[#][a-z\.h]+ printf("%s is a preprocessor\n",yytext);
%%

int yywrap(void)
{
    return 1;
}

int main()
{
  // reads input from a file named test.c rather than terminal
  freopen("test.c", "r", stdin);
    yylex();
    return 0;
}
```

Test.c :



```
int main() {
  float z = 15.5;
  printf("%f\n", z);
  return 0;
 }
```

Output:



```
$ lex tokenize_code.l
$ gcc lex.yy.c -o tokenize_code
$ ./tokenize_code
$ int is a datatype
main is a function
( is a delimiter
) is a delimiter
{ is a separator
float is a datatype
z is an identifier
= is an operator
15.5 is a floating point number
; is a delimiter
printf is a function
(is a delimiter
"%f\n" is an identifier
, is a separator
z is an identifier
) is a delimiter
; is a delimiter
} is a separator
return is a keyword
```

Result:

Thus, we executed a Lex program for token separation.

Ex. No : 8 **Parser Program Using YACC**

Date :

AIM:

To write a YACC program to parse the input.

Procedure:

- Open telnet 172.20.105.110 .
- Type ‘vim ppl.l’ to create lex file and ‘vim ppl.y’ to create YACC file.
- Press ‘i’ to insert the Lex code as well as YACC code in their respective file.
- In Lex file, rule section / definition section type all header file and required variables.
- IN YACC file, definition section, type all the header file and variables.
- In Lex as well as YACC file type the rule section with parsing of input in Lex code and validating the parsed input in the YACC code.
- After insertion of code in Lex and YACC press ‘ESC’ button and type ‘wq’.
- Type ‘Lex ppl.l’ followed by ‘yacc – d ppl.y’ to compile both Lex as well as YACC file.
- Type ‘cc lex.yy.c’ and then ‘y.tab.c -ll -ly’ along with ‘./a.out’ command for output.
- Close the telnet.

Lex code :

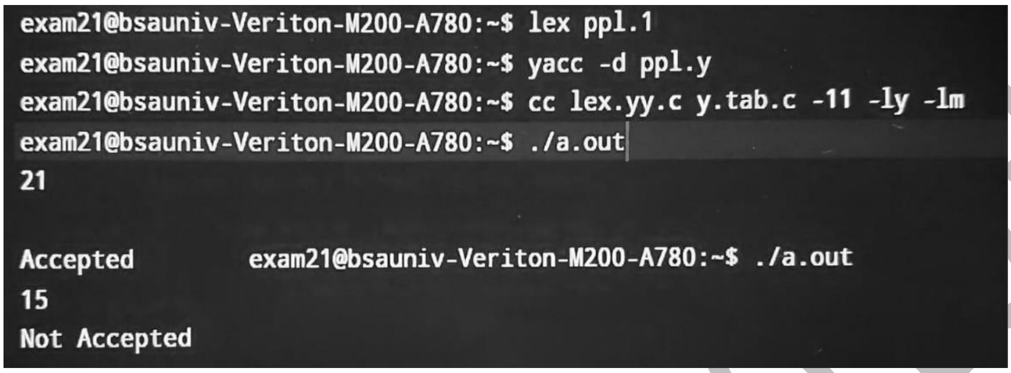
```
%{ #include<stdio.h>
 #include"y.tab.h"
 extern int yylval;
%}
%%
0           {yylval=0;return Zero;}
1           {yylval=1;return One;}
[\t] return 0;
.return yytext[0];
%%
```

YACC Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
void yyerror(char*s);
 int yylex();
%}
%token Zero One
%%
stmt:S;
S:S A | A;
A:Zero Zero | One One;
```

```
%%
int main()
{yyparse();
printf("Accepted\t");
exit(0);}
void yyerror(char *s)
{printf("Not Accepted\n");
exit(0);}
```

Output:



```
exam21@bsauniv-Veriton-M200-A780:~$ lex ppl.1
exam21@bsauniv-Veriton-M200-A780:~$ yacc -d ppl.y
exam21@bsauniv-Veriton-M200-A780:~$ cc lex.yy.c y.tab.c -11 -ly -lm
exam21@bsauniv-Veriton-M200-A780:~$ ./a.out
21

Accepted        exam21@bsauniv-Veriton-M200-A780:~$ ./a.out
15
Not Accepted
```

Result:

Thus, Parsing program using YACC has been executed successfully.

Ex. No : 9 **Calculator Program using Lex and YACC**

Date :

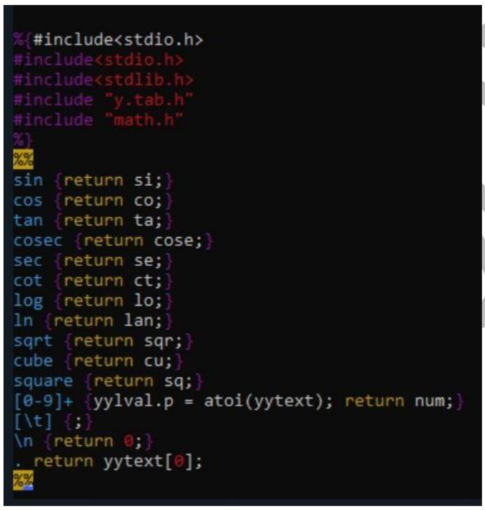
AIM:

To write a calculator program using Lex and YACC.

Procedure:

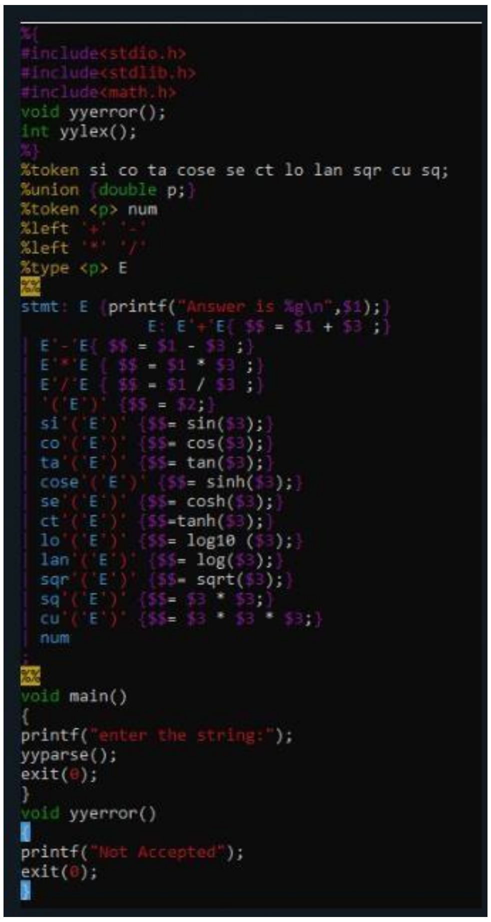
- Open telnet 172.20.105.110 .
- Type 'vim calc.l' to create lex file and 'vim calc.y' to create YACC file.
- Open 'vim calc.l' file and press 'i' to insert the code.
- In definition section, type all required header.
- In rule section, type all required various mathematical function using variables.
- [0-9]+ {yylval.p = atoi(yy.text); return num;}
- Open YACC file and insert the following by pressing 'i' and then type the code.
- In definition section, type all required header file and variable declaration.
- In rule section, type all formula for every mathematical functions.
- In main method, get user input for performing math calculation.
- Type the following command to compile both the files : lex calc.l and yacc -d calc.y
- Type the commands : cc lex.yy.c ytab.c -ll -ly -lm
- Type './a.out' for output.
- Close the telnet.

Lex code :



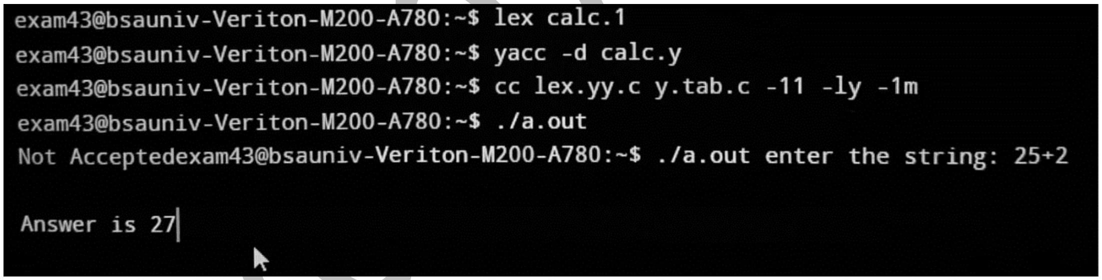
```
%{#include<stdio.h>
#include<stdio.h>
#include<stdlib.h>
#include "y.tab.h"
#include "math.h"
%}
%%
sin {return si;}
cos {return co;}
tan {return ta;}
cosec {return cose;}
sec {return se;}
cot {return ct;}
log {return lo;}
ln {return lan;}
sqrt {return sqr;}
cube {return cu;}
square {return sq;}
[0-9]+ {yylval.p = atoi(yytext); return num;}
[\t] {;}
\n {return 0;}
. return yytext[0];
%%
```

YACC Code:



```
%{
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
void yyerror();
int yylex();
%}
%token si co ta cose se ct lo lan sqr cu sq;
%union {double p;}
%token <p> num
%left '+' '-'
%left '*' '/'
%type <p> E
%%
stmt: E {printf("Answer is %g\n",$1);}
                E: E'+'E{ $$ = $1 + $3 ;}
 | E'-'E{ $$ = $1 - $3 ;}
 | E'*'E { $$ = $1 * $3 ;}
 | E'/'E { $$ = $1 / $3 ;}
 | '('E')' {$$ = $2;}
 | si'('E')' {$$= sin($3);}
 | co'('E')' {$$= cos($3);}
 | ta'('E')' {$$= tan($3);}
 | cose'('E')' {$$= sinh($3);}
 | se'('E')' {$$= cosh($3);}
 | ct'('E')' {$$=tanh($3);}
 | lo'('E')' {$$= log10 ($3);}
 | lan'('E')' {$$= log($3);}
 | sqr'('E')' {$$= sqrt($3);}
 | sq'('E')' {$$= $3 * $3;}
 | cu'('E')' {$$= $3 * $3 * $3;}
 | num
;
%%
void main()
{
printf("enter the string:");
yyparse();
exit(0);
}
void yyerror()
{
printf("Not Accepted");
exit(0);
}
```

Output:



```
exam43@bsauniv-Veriton-M200-A780:~$ lex calc.1
exam43@bsauniv-Veriton-M200-A780:~$ yacc -d calc.y
exam43@bsauniv-Veriton-M200-A780:~$ cc lex.yy.c y.tab.c -11 -ly -1m
exam43@bsauniv-Veriton-M200-A780:~$ ./a.out
Not Acceptedexam43@bsauniv-Veriton-M200-A780:~$ ./a.out enter the string: 25+2

Answer is 27
```

Result:

Thus, Calculator program using YACC has been executed successfully.

Ex. No : 10 **Three Address Code Generation using Lex and YACC**

Date :

AIM:

To write a program to generate three address code using Lex and YACC.

Procedure:

- Open telnet 172.20.105.110 .
- Type ‘vim 3ac.l’ to create lex file and ‘vim 3ac.y’ to create YACC file.
- Press ‘i’ to insert code into Lex and YACC file.
- In Lex file, definition section type the required header file and variable declaration.
- In rule section, type function to create three address code.
- In YACC file, definition section type header file and return the three address code.
- After the insertion of code, type the command : lex 3ac.l and yacc -d 2ac,l to compile the files.
- Type the following : cc lex.yy.c ytab.c -ll -ly and type the ‘./a.out’ command for output.
- Type the following command to compile both the files : lex calc.l and yacc -d calc.y
- Close the telnet.

Code:

vi 3ac.l

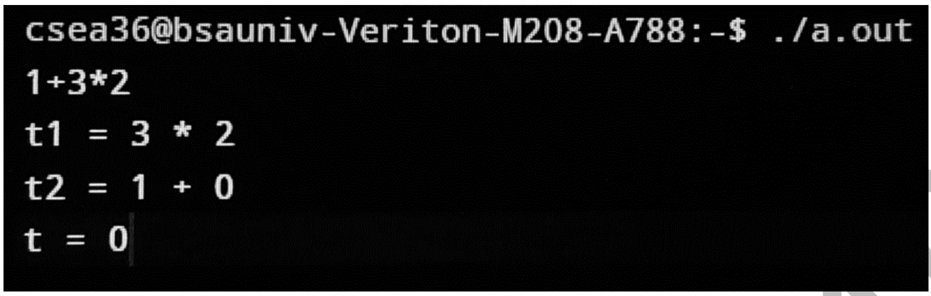
```
%{
#include<stdio.h>
 #include"y. tab.h"
int k=1;
void yyerror(char* str);
char *gencode(char word[],char first, char op, char second);
%}
%%
[0-9]+ { yylval.dval=yytext|0]; return NUM; }
\n {return 0;
}
. {return yytext|0];}
%%
void yyerror (char* str)
{ printf("n%s", str);}
char *gencode(char word[],char first, char op, char second)
{ char temp[10];
sprintf(temp,"%d", k); strcat(word, temp);
```

```
k++;

printf("%s = %c%c %c\n", word, first, op,second); return word; } int yywrap()

  { return 1 ; }

int main(){ yyparse();

return 0;}
```

vi 3ac.y

```
%{
#include <stdio.h> int aaa:
void yyerror (char* str);
char *gencode(char word[],char first,char op,char second); int yylex();
%}
%union{
char dval;}
%token <dval> NUM
%type <dval> E
%left '+' '-'
%left '*' '/' '%'
%%
statement: E {printf("\nt = %c \n",$1):}: E : E'+'E
{char word[]="t"; char *test=gencode(word,$1, '+', $3);
$$=test: }
| E'-'E {
char word[]="t"; char *test=gencode(word,$1,'-',$3);
$$=test;}
| E'%'E {
char word[]="t"; char *test=gencode(word,$1,'%',$3);
$$=test;}
| E'*'E {
char word[]="t"; char *test=gencode(word,$1,'*',$3);
$$=test;}
| E'/'E {
char word[]="t",
char *test=gencode(word,$1,’/‘,$3);
$$=test;}
| ‘(‘E’)’ { $$=$2;}
| NUM {$$=$1;};
%%
```

Output:



```
csea36@bsauniv-Veriton-M208-A788:-$ ./a.out
1+3*2
t1 = 3 * 2
t2 = 1 + 0
t = 0
```

Result:

Thus, A three address code generation using Lex and YACC has been executed successfully.

Ex. No : 11 **Code Optimization**

Date :

AIM:

To optimize the given snip of code and get the optimized code as output.

Procedure:

- Open any C compiler to write the code and get the optimized output.
- Write all the required header file.
- Define a structure with all the required variables to be used.
- Write the main method and start to write all the required logics.
- Get the user input in form of left and right where left indicates left operand and right indicate right operand.
- Eliminate the common expression and perform Dead code elimination.
- Print the optimized code.
- Stop the program.

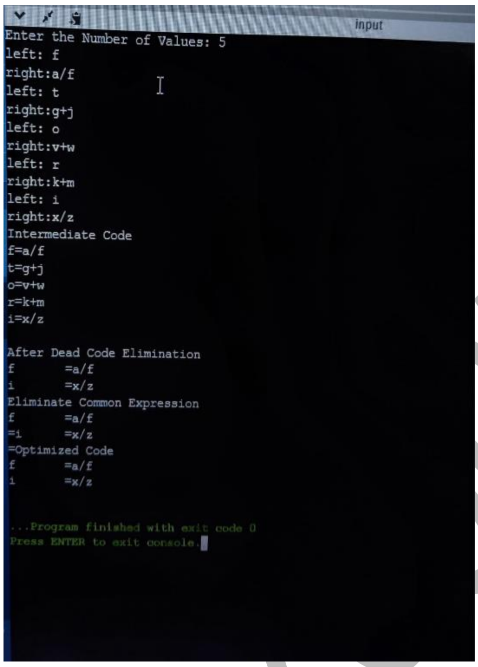
Code:

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

struct op {
   char l;
   char r[20];
};

struct op op[10], pr[10];

int main() {
   int i, 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\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 (int 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;

            // Add a declaration for 'a' here
            int a;

            pr[j].l = pr[m].l;

            for (i = 0; i < z; i++) {
                l = strchr(pr[i].r, t);
                if (l) {
                    // Use the correct variable 'a' here
                    printf("pos: %d\n", 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';
            }
        }
    }

    printf("Optimized Code\n");
    for (i = 0; i < z; i++) {
        if (pr[i].l != '\0')
            printf("%c\t=", pr[i].l);
            printf("%s\n", pr[i].r);
    }

    return 0;
}
```

Output:



```
Enter the Number of Values: 5
left: f
right:a/f
left: t
right:g+j
left: o
right:v+w
left: r
right:k+m
left: i
right:x/z
Intermediate Code
f=a/f
t=g+j
o=v+w
r=k+m
i=x/z

After Dead Code Elimination
f       =a/f
i       =x/z
Eliminate Common Expression
f       =a/f
=i      =x/z
=Optimized Code
f       =a/f
i       =x/z


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

Thus, a C program for code optimization has been executed successfully.

Ex. No : 12 **Code Generation**

Date :

AIM:

To optimize the code generation of three address code using c programming language.

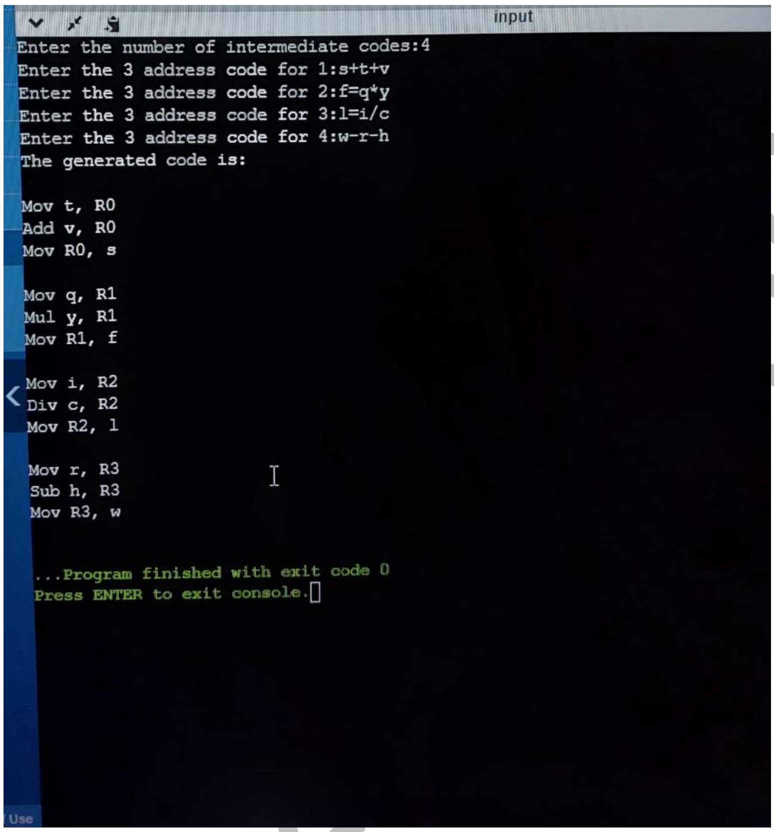
Procedure:

- Open any C compiler to write the code for code generation.
- Write all the required header file ‘stdio.h’ , ‘conio.h’, ‘string.h’.
- Create a structure for storing required variables.
- Create required function for obtaining assignment operators, uminus variable, code generator, tuple, etc.
- Write a function define a logic to read through the code and identify every characters and its categories.
- Write the main method to get the input from the user and generate code, assemble them orderly.
- Print the final three address code generated.
- Compile the entire code in C compiler and then obtain the output.
- Stop the program.

Code:

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

void main() {
   int n, i, j;
   char a[50][50];

   printf("Enter the number of intermediate codes:");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      printf("Enter the 3 address code for %d:", i + 1);
      for (j = 0; j < 6; j++) {
         scanf("%c", &a[i][j]);
      }
   }

   printf("The generated code is:\n");
   for (i = 0; i < n; i++) {
      printf("\nMov %c, R%d", a[i][3], i);
      if (a[i][4] == '-') {
         printf("\nSub %c, R%d", a[i][5], i);
      }
      if (a[i][4] == '+') {
            printf("\nAdd %c, R%d", a[i][5], i);
        }
        if (a[i][4] == '*') {
            printf("\nMul %c, R%d", a[i][5], i);
        }
        if (a[i][4] == '/') {
            printf("\nDiv %c, R%d", a[i][5], i);
        }
        printf("\nMov R%d, %c", i, a[i][1]);
        printf("\n");
    }

    getch();
}
```

Output:



```
Enter the number of intermediate codes:4
Enter the 3 address code for 1:s=t+v
Enter the 3 address code for 2:f=q*y
Enter the 3 address code for 3:l=i/c
Enter the 3 address code for 4:w=r-h
The generated code is:

Mov t, R0
Add v, R0
Mov R0, s

Mov q, R1
Mul y, R1
Mov R1, f

Mov i, R2
Div c, R2
Mov R2, l

Mov r, R3
Sub h, R3
Mov R3, w


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

Thus, a C program for code generation has been executed successfully.