(a) To recognize whether the given input string is a keyword, identifier, number (integer or real) and operators.

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

%%

int|float|if|else|while|return      { printf("Keyword: %s\n", yytext); }

[A-Za-z_][A-Za-z0-9_]*            { printf("Identifier: %s\n", yytext); }

[0-9]+                          { printf("Integer: %s\n", yytext); }

[0-9]+\.[0-9]+                  { printf("Real Number: %s\n", yytext); }

[+\-*/=<>!&|]                     { printf("Operator: %s\n", yytext); }

.                               { printf("Invalid Token: %s\n", yytext); }

%%
int yywrap(void){}
int main() {
   yylex();
   return 0;
}
```

(b) To count the number of vowels and consonants in a given input.

```
%{
int vow_count = 0;
int const_count = 0;
%}
%%
[aeiouAEIOU] { vow_count++; }
[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { const_count++; }
.|\n {}
%%
int yywrap() {
   return 1;  // End of input
}
int main() {
   printf("Enter the string of vowels and consonants: ");
   yylex();  // Start Lexical analysis
   printf("Number of vowels: %d\n", vow_count);
   printf("Number of consonants: %d\n", const_count);
   return 0;
}
```

2(a) C Program to construct the recursive descent parser for the following grammar

S->aABb

A->c| ε

B->d| ε

```
#include <stdio.h>
#include <stdlib.h>
/*
Grammar:
S -> aABb
A -> c | ε
B -> d | ε
*/
char l;
int match(char c) {
   if (l == c) {
      l = getchar();
      return 1;
   } else {
      return 0;
   }
}
void B() {
   if (l == 'd') {
      match('d');
   }
}
void A() {
   if (l == 'c') {
      match('c');
```

```
    }
}

void S() {
    if (match('a')) {
        A();
        B();
        if (!match('b')) {
            printf("Error: Expected 'b'\n");
            exit(1);
        }
    } else {
        printf("Error: Expected 'a'\n");
        exit(1);
    }
}
int main() {
    printf("Enter a string ending with $:\n");
    l = getchar();
    S();
    if (l == '$') {
        printf("\nParsing Successful\n");
    } else {
        printf("\nError: Unexpected input '%c' after parsing\n", l);
        exit(1);
    }
    return 0;
```

```
}
```

Output : Enter a string ending with $:

acdb$

Parsing Successful

2 (b) C Program to construct the recursive descent parser for the following grammar

S -> iEtS'|a

S'-> eS|ε

E->b

```
#include <stdio.h>
#include <stdlib.h>
char lookahead;
void S();
void S_prime();
void E();
void getNextChar() {
   lookahead = getchar();
}
void match(char expected) {
   if (lookahead == expected) {
      getNextChar();
   } else {
      printf("Syntax error: Expected '%c', but found '%c'\n", expected, lookahead);
```

```
        exit(1);  // Exit on error
    }
}
void E() {
    if (lookahead == 'b') {
        match('b');
    } else {
        printf("Syntax error in E: Expected 'b', but found '%c'\n", lookahead);
        exit(1);
    }
}
void S_prime() {
    if (lookahead == 'e') {
        match('e');
        S();
    }
}
void S() {
    if (lookahead == 'i') {
        match('i');
        E();
        match('t');
        S();
        S_prime();
    } else if (lookahead == 'a') {
        match('a');
    } else {
```

```
        printf("Syntax error in S: Expected 'i' or 'a', but found '%c'\n", lookahead);
        exit(1);
    }
}
int main() {
    printf("Enter the input string (end with '$'):\n");

    getNextChar();
    S();
    if (lookahead == '$') {
        printf("Parsing successful!\n");
    } else {
        printf("Invalid input: Expected end of string '$', but found '%c'\n", lookahead);
        exit(1);
    }
    return 0;
}
```

output:

```
Enter the input string (end with '$'):
ibta$
Parsing Successful
```

3 AIM: Develop and design a program to find out FIRST ( ) and FOLLOW ( ) of all the Non Terminals of the given context free grammar.

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
// Check for FIRST/FIRST conflicts
bool hasFirstFirstConflict(char firstA[][10], int n) {
   for (int i = 0; i < n; i++)
      for (int j = i + 1; j < n; j++)
         if (strcmp(firstA[i], firstA[j]) == 0)
            return true;
   return false;
}

// Check for FIRST/FOLLOW conflicts
bool hasFirstFollowConflict(char firstA[][10], int n, char followA[]) {
   for (int i = 0; i < n; i++)
      if (strchr(followA, firstA[i][0]))
         return true;
   return false;
}

int main() {
   int n;
   printf("Enter the number of productions for A: ");
   scanf("%d", &n);
```

```
    char firstA[n][10], followS[10], followA[10];
    printf("Enter the FIRST set elements for A:\n");
    for (int i = 0; i < n; i++) {
        printf("FIRST(A%d): ", i + 1);
        scanf("%s", firstA[i]);
    }

    printf("Enter the FOLLOW set for S: ");
    scanf("%s", followS);
    printf("Enter the FOLLOW set for A: ");
    scanf("%s", followA);

    printf("\nFIRST(A): { ");
    for (int i = 0; i < n; i++)
        printf("%s%s", firstA[i], (i == n - 1) ? " " : ", ");
    printf("}\nFOLLOW(S): { %s }\nFOLLOW(A): { %s }\n", followS, followA);

    if (hasFirstFirstConflict(firstA, n))
        printf("FIRST/FIRST conflict detected! Grammar is not LL(1).\n");
    else if (hasFirstFollowConflict(firstA, n, followA))
        printf("FIRST/FOLLOW conflict detected! Grammar is not LL(1).\n");
    else
        printf("The grammar is LL(1).\n");

    return 0;
}
```

output:

cc Practical3.c

./a.out

Enter the number of productions for A: 2

Enter the FIRST set elements for A:

FIRST(A1): c

FIRST(A2): ε

Enter the FOLLOW set for S: $

Enter the FOLLOW set for A: b

FIRST(A): { c, ε }

FOLLOW(S): { $ }

FOLLOW(A): { b }

The grammar is LL(1).

4 AIM: Implement a Program to check whether the given context grammar is LL (1).

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

// Check for FIRST/FIRST conflicts
bool hasFirstFirstConflict(char firstA[][10], int n) {
   for (int i = 0; i < n; i++)
      for (int j = i + 1; j < n; j++)
```

```
            if (strcmp(firstA[i], firstA[j]) == 0)
                return true;
    return false;
}

// Check for FIRST/FOLLOW conflicts
bool hasFirstFollowConflict(char firstA[][10], int n, char followA[]) {
    for (int i = 0; i < n; i++)
        if (strchr(followA, firstA[i][0]))
            return true;
    return false;
}

int main() {
    int n;
    printf("Enter the number of productions for A: ");
    scanf("%d", &n);

    char firstA[n][10], followS[10], followA[10];
    printf("Enter the FIRST set elements for A:\n");
    for (int i = 0; i < n; i++) {
        printf("FIRST(A%d): ", i + 1);
        scanf("%s", firstA[i]);
    }

    printf("Enter the FOLLOW set for S: ");
    scanf("%s", followS);
```

```
    printf("Enter the FOLLOW set for A: ");
    scanf("%s", followA);

    printf("\nFIRST(A): { ");
    for (int i = 0; i < n; i++)
        printf("%s%s", firstA[i], (i == n - 1) ? " " : ", ");
    printf("}\nFOLLOW(S): { %s }\nFOLLOW(A): { %s }\n", followS, followA);

    if (hasFirstFirstConflict(firstA, n))
        printf("FIRST/FIRST conflict detected! Grammar is not LL(1).\n");
    else if (hasFirstFollowConflict(firstA, n, followA))
        printf("FIRST/FOLLOW conflict detected! Grammar is not LL(1).\n");
    else
        printf("The grammar is LL(1).\n");

    return 0;
}
```

output:

```
vi Practical4.c
cc Practical4.c
 ./a.out
Enter the number of productions for A: 2
Enter the FIRST set elements for A:
FIRST(A1): c
FIRST(A2): ε
```

Enter the FOLLOW set for S: $

Enter the FOLLOW set for A: b

FIRST(A): { c, ε }

FOLLOW(S): { $ }

FOLLOW(A): { b }

The grammar is LL(1).

cc Practical4.c

./a.out

Enter the number of productions for A: 2

Enter the FIRST set elements for A:

FIRST(A1): a

FIRST(A2): a

Enter the FOLLOW set for S: $

Enter the FOLLOW set for A: $

FIRST(A): { a, a }

FOLLOW(S): { $ }

FOLLOW(A): { $ }

FIRST/FIRST conflict detected! Grammar is not LL(1).

vi Practical4.c

5 AIM: Construct a program to convert an Infix expression into Postfix expression using Lex and Yacc.

Lex Code:

```
%{
#include"y.tab.h"
extern int yylval;
%}
%%
[0-9]+  {yylval=atoi(yytext); return NUM;}
\n      return 0;
.       return *yytext;
%%

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

Yacc Code:

```
%{
#include <stdio.h>
int yylex();  // Declaration of yylex()
int yyerror(char *msg);  // Declaration of yyerror()
%}

%token NUM
%left '+' '-'
%left '*' '/'
%right NEGATIVE

%%
S:  E { printf("\n"); }
   ;

E:  E '+' E { printf("+"); }
   | E '*' E { printf("*"); }
```

```
   | E '-' E { printf("-"); }
   | E '/' E { printf("/"); }
   | '(' E ')'
   | '-' E %prec NEGATIVE { printf("-"); }
   | NUM { printf("%d", yylval); }
   ;

%%

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

int yyerror(char *msg) {
   printf("error YACC: %s\n", msg);
   return 1;
}
```

output:

vi Practical5.l

vi Practical5.y

yacc -d Practical5.y

flex Practical5.l

cc lex.yy. c y.tab.c

./a.out

2+6*2-5/3

262*+53/-

./a.out

2+6/7*6

267/6*+

6 AIM: Construct a Program to generate Intermediate code using three address statements for logical expression.

Lex Code:

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

%option noyywrap

%%

[ \t\n]      { /* Ignore whitespace */ }
[0-9]+       { yylval.str = strdup(yytext); return NUM; }
[a-zA-Z][a-zA-Z0-9]* { yylval.str = strdup(yytext); return ID; }
"="          { return ASSIGN; }
"+"          { return PLUS; }
"-"          { return MINUS; }
"*"          { return MUL; }
"/"          { return DIV; }
";"          { return SEMI; }
"("          { return LPAREN; }
")"          { return RPAREN; }
.            { printf("Unexpected character: %s\n", yytext); exit(1); }

%%
```

Yacc Code:

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

int temp_var_count = 1; // Counter for temporary variables

void generate_code(char* result, char* op1, char* op, char* op2) {
   printf("%s = %s %s %s\n", result, op1, op, op2);
}

void generate_assignment(char* id, char* value) {
   printf("%s = %s\n", id, value);
}

// Explicit declaration of yyerror to avoid implicit declaration warning
void yyerror(const char* msg);

%}

%union {
   char* str;
}

%token <str> ID NUM
%token PLUS MINUS MUL DIV ASSIGN SEMI LPAREN RPAREN
%left PLUS MINUS
%left MUL DIV
%type <str> expr stmt

%%

stmt : ID ASSIGN expr SEMI {
        generate_assignment($1, $3);
```

```
        free($1);
        free($3);
      }
    ;

expr : expr PLUS expr {
        $$ = (char*) malloc(10);
        sprintf($$, "t%d", temp_var_count++);
        generate_code($$, $1, "+", $3);
        free($1);
        free($3);
      }
    | expr MINUS expr {
        $$ = (char*) malloc(10);
        sprintf($$, "t%d", temp_var_count++);
        generate_code($$, $1, "-", $3);
        free($1);
        free($3);
      }
    | expr MUL expr {
        $$ = (char*) malloc(10);
        sprintf($$, "t%d", temp_var_count++);
        generate_code($$, $1, "*", $3);
        free($1);
        free($3);
      }
    | expr DIV expr {
        $$ = (char*) malloc(10);
        sprintf($$, "t%d", temp_var_count++);
        generate_code($$, $1, "/", $3);
        free($1);
        free($3);
      }
    | ID { $$ = strdup($1); }
    | NUM { $$ = strdup($1); }
    | LPAREN expr RPAREN { $$ = $2; }
    ;

%%
```

```
int main() {
   printf("Enter an expression (e.g., a = b + c * d;):\n");
   yyparse();
   return 0;
}

void yyerror(const char* msg) {
   fprintf(stderr, "Syntax Error: %s\n", msg);
}
```

output:

vi Practical6.l

vi Practical6.y

yacc -d Practical6.y

flex Practical6.l

cc lex.yy.c y.tab.c

./a.out

Enter an expression (e.g., a = b + c * d;):

t1 = c * d

t2 = d + t1

a = t2

7 Aim: Develop a program to detect common subexpression in three address code using data structure DAG.

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

```
#include <string.h>

#define MAX_INSTRUCTIONS 100
#define MAX_OPERANDS 3
#define MAX_OPERAND_SIZE 20

typedef struct DAGNode {
   char operation[MAX_OPERAND_SIZE];
   char operands[MAX_OPERANDS][MAX_OPERAND_SIZE];
   struct DAGNode *left;
   struct DAGNode *right;
   int id;
} DAGNode;

DAGNode* dagNodes[MAX_INSTRUCTIONS];
int dagNodeCount = 0;

typedef struct Instruction {
   char result[MAX_OPERAND_SIZE];
   char operand1[MAX_OPERAND_SIZE];
   char operand2[MAX_OPERAND_SIZE];
   char operation[MAX_OPERAND_SIZE];
} Instruction;

int findInDAG(char* op, char* operand1, char* operand2) {
   for (int i = 0; i < dagNodeCount; i++) {
      if (strcmp(dagNodes[i]->operation, op) == 0 &&
         strcmp(dagNodes[i]->operands[0], operand1) == 0 &&
         strcmp(dagNodes[i]->operands[1], operand2) == 0) {
         return dagNodes[i]->id;
      }
   }
   return -1;
}

int createDAGNode(char* op, char* operand1, char* operand2) {
   DAGNode* newNode = (DAGNode*)malloc(sizeof(DAGNode));
   strcpy(newNode->operation, op);
   strcpy(newNode->operands[0], operand1);
   strcpy(newNode->operands[1], operand2);
```

```
    newNode->id = dagNodeCount++;
    dagNodes[dagNodeCount - 1] = newNode;
    return newNode->id;
}

void processTACInstruction(Instruction* tacInstr) {
    int existingNodeId = findInDAG(tacInstr->operation, tacInstr->operand1, tacInstr->operand2);

    if (existingNodeId != -1) {
        printf("Common subexpression found: %s = %s %s %s (Reuse node ID: %d)\n",
            tacInstr->result, tacInstr->operand1, tacInstr->operation, tacInstr->operand2,
existingNodeId);
    } else {
        int newNodeId = createDAGNode(tacInstr->operation, tacInstr->operand1, tacInstr-
>operand2);
        printf("No common subexpression: %s = %s %s %s (New node ID: %d)\n",
            tacInstr->result, tacInstr->operand1, tacInstr->operation, tacInstr->operand2,
newNodeId);
    }
}

int main() {
    int numInstructions;

    printf("Enter number of TAC instructions: ");
    scanf("%d", &numInstructions);

    Instruction tacInstructions[numInstructions];

    for (int i = 0; i < numInstructions; i++) {
        printf("Enter instruction %d (format: result operand1 operation operand2):\n", i + 1);
        scanf("%s %s %s %s", tacInstructions[i].result, tacInstructions[i].operand1,
tacInstructions[i].operation, tacInstructions[i].operand2);
    }

    for (int i = 0; i < numInstructions; i++) {
        processTACInstruction(&tacInstructions[i]);
    }
```

```
    return 0;
}
```

output:

```
Enter number of TAC instructions: 5
Enter instruction 1 (format: result operand1 operation operand2):
t1 a + b
Enter instruction 2 (format: result operand1 operation operand2):
t2 c + d
Enter instruction 3 (format: result operand1 operation operand2):
t3 t1 + c
Enter instruction 4 (format: result operand1 operation operand2):
t4 a + b
Enter instruction 5 (format: result operand1 operation operand2):
t5 t2 * t3

No common subexpression: t1 = x + y (New node ID: 0)
No common subexpression: t2 = p + q (New node ID: 1)
No common subexpression: t3 = t1 + p (New node ID: 2)
Common subexpression found: t4 = x + y (Reuse node ID: 0)
No common subexpression: t5 = t2 * t3 (New node ID: 3)
```

8 . AIM: Construct a program to generate an assembly language instructions from the given three address code using simple code generation algorithm.

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

// Maximum number of 3-address code instructions and assembly instructions
#define MAX_INSTRUCTIONS 50
#define MAX_REGISTERS 4

// Data structure for 3-address code instructions
typedef struct {
   char result[10];   // Result variable
   char operand1[10]; // First operand
   char operand2[10]; // Second operand
   char op[2];        // Operator (+, -, *, /)
} ThreeAddressCode;

// Function to generate assembly code
void generateAssembly(ThreeAddressCode tac[], int numInstructions) {
   // Register names
   char* registers[MAX_REGISTERS] = {"R0", "R1", "R2", "R3"};
   int regIndex = 0; // Register index for the next free register

   // Output assembly code
   for (int i = 0; i < numInstructions; i++) {
      ThreeAddressCode ins = tac[i];

      // For the first instruction, move the first operand into a register
      if (regIndex < MAX_REGISTERS) {
         printf("MOV %s, %s\n", registers[regIndex], ins.operand1);
         regIndex++;
      }

      // Generate the corresponding operation
      if (strcmp(ins.op, "+") == 0) {
         printf("ADD %s, %s\n", registers[regIndex - 1], ins.operand2);
      } else if (strcmp(ins.op, "-") == 0) {
         printf("SUB %s, %s\n", registers[regIndex - 1], ins.operand2);
      } else if (strcmp(ins.op, "*") == 0) {
```

```
        printf("MUL %s, %s\n", registers[regIndex - 1], ins.operand2);
    } else if (strcmp(ins.op, "/") == 0) {
        printf("DIV %s, %s\n", registers[regIndex - 1], ins.operand2);
    }
```

output:

```
 vi Practical8.c
 cc Practical8.c
 ./a.out
Enter the number of 3-address code instructions: 4

Enter instruction 1 (result operand1 operand2 operator):
Result: t1
Operand1: a
Operand2: b
Operator (+, -, *, /): +

Enter instruction 2 (result operand1 operand2 operator):
Result: t2
Operand1: t1
Operand2: c
Operator (+, -, *, /): -

Enter instruction 3 (result operand1 operand2 operator):
Result: t3
Operand1: t2
```

Operand2: d

Operator (+, -, *, /): *

Enter instruction 4 (result operand1 operand2 operator):

Result: t4

Operand1: t3

Operand2: e

Operator (+, -, *, /): /

Operator (+, -, *, /): /

MOV R0, a

ADD R0, b

MOV t1, R0

MOV R1, t1

SUB R1, c

MOV t2, R1

MOV R2, t2

MUL R2, d

MOV t3, R2

MOV R3, t3

DIV R3, e

MOV t4, R3

Post lab

AIM: Develop a C++ (small subset implementing using class keyword) to C preprocessor using LEX and YACC tools.

Lex code :

```
%{
#include "y.tab.h"
#include <stdio.h>
#include <string.h>
%}
%%
"class" return CLASS;
"public" return PUBLIC;
"private" return PRIVATE;
"protected" return PROTECTED;
"int" return INT;
"float" return FLOAT;
"char" return CHAR;
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return IDENTIFIER; }
"{" return LBRACE;
"}" return RBRACE;
";" return SEMICOLON;
[ \t\n] ; // Ignore whitespaces and new lines
. return yytext[0];
```

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

Yacc Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void yyerror(const char *s);
extern int yylex();
%}
%union {
char* str;
}
%token <str> CLASS PUBLIC PRIVATE PROTECTED IDENTIFIER INT FLOAT CHAR LBRACE RBRACE
SEMICOLON
%type <str> class_decl members member type
%%
program:
class_decl
;
class_decl:
CLASS IDENTIFIER LBRACE members RBRACE SEMICOLON {
printf("/* Converted C struct */\n");
```

```
printf("typedef struct %s {\n", $2);
printf("%s} %s;\n", $4, $2);
}
;
members:
/* empty */ { $$ = strdup(""); }
| members member {
char *temp = malloc(strlen($1) + strlen($2) + 10);
sprintf(temp, "%s%s", $1, $2);
$$ = temp;
}
;
member:
type IDENTIFIER SEMICOLON {
char *temp = malloc(strlen($1) + strlen($2) + 10);
sprintf(temp, " %s %s;\n", $1, $2);
$$ = temp;
}

;
type:
INT { $$ = "int"; }
| FLOAT { $$ = "float"; }
| CHAR { $$ = "char"; }
;
%%
void yyerror(const char *s) {
```

```
fprintf(stderr, "Error: %s\n", s);
}
int main() {
printf("Enter C++ class definition:\n");
yyparse();
return 0;
}
```

output:

```
vi PostLab.l
vi PostLab.y
yacc -d PostLab.y
lex PostLab.l
cc lex.yy.c y.tab.c
./a.out
Enter C++ class definition:
class Car {
   int speed;
   float mileage;
   char model;
};
/* Converted C struct */
typedef struct Car {
   int speed;
   float mileage;
   char model;
} Car;
```

Develop a Program to convert the postfix into prefix using LEX and YACC tools.

Lex Code:

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z]     { yylval.str = strdup(yytext); return OPERAND; }
[+\-*/]      { yylval.str = strdup(yytext); return OPERATOR; }
[\n]         { return '\n'; }
[ \t]        ; // ignore whitespace
.            { printf("Invalid character: %s\n", yytext); }
%%
int yywrap() {
   return 1;
}
```

yacc code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* concat(char* op, char* left, char* right) {
   char* result = (char*)malloc(strlen(op) + strlen(left) + strlen(right) + 3);
```

```
    sprintf(result, "%s %s %s", op, left, right);
    return result;
}
void yyerror(const char* s);
int yylex(void);
%}

%union {
    char* str;
}
%token <str> OPERAND OPERATOR
%type <str> expr
%%
input:
    expr '\n' {
        printf("Prefix: %s\n", $1);
        free($1);
    }
    ;
expr:
    OPERAND {
        $$ = strdup($1);
    }
    | expr expr OPERATOR {
        $$ = concat($3, $1, $2);
        free($1);
        free($2);
```

```
        free($3);
    }
    ;
%%
void yyerror(const char* s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

run:

```
lex postfix.l
yacc -d postfix.y
gcc lex.yy.c y.tab.c -o postfix_converter
```

```
$ ./postfix_converter
ab+c*
Prefix: * + a b c
```

Develop a program for if the expression is give identify that the expression is prefix, infix, postfix.

Lex code:

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

%%
[a-zA-Z]    { yylval.str = strdup(yytext); return OPERAND; }
[+\-*/]     { yylval.str = strdup(yytext); return OPERATOR; }
[ \t\n]     ; // skip whitespace
.           { return INVALID; }
%%
int yywrap() {
   return 1;
}
```

Yacc Code:

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

extern int yylex(void);
void yyerror(const char* s) {}

int is_prefix = 0, is_infix = 0, is_postfix = 0;
```

```
%}

%union {
   char* str;
}

%token <str> OPERAND OPERATOR INVALID
%type <str> expr_prefix expr_infix expr_postfix

%%
input:
    check_prefix { if (is_prefix) printf("Prefix expression detected.\n"); }
  | check_infix  { if (is_infix) printf("Infix expression detected.\n"); }
  | check_postfix { if (is_postfix) printf("Postfix expression detected.\n"); }
  ;

check_prefix: expr_prefix { is_prefix = 1; };
check_infix:  expr_infix  { is_infix = 1; };
check_postfix: expr_postfix { is_postfix = 1; };

expr_prefix:
    OPERATOR expr_prefix expr_prefix
  | OPERAND
  ;

expr_infix:
    expr_infix OPERATOR expr_infix
```

```
  | '(' expr_infix ')'
  | OPERAND
  ;

expr_postfix:
    expr_postfix expr_postfix OPERATOR
  | OPERAND
  ;
%%
```

Run :

```
lex expr.l
yacc -d expr.y
gcc lex.yy.c y.tab.c -o expr_identifier
```

```
$ ./expr_identifier
+ a b
Prefix expression detected.

$ ./expr_identifier
a + b
Infix expression detected.

$ ./expr_identifier
a b +
```

Postfix expression detected.

Struct vala

Lex Code :

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

%%
"struct"          { return STRUCT; }
"{"               { return LBRACE; }
"}"               { return RBRACE; }
[a-zA-Z_][a-zA-Z0-9_]*    { yylval.str = strdup(yytext); return IDENTIFIER; }
[0-9]+            { yylval.str = strdup(yytext); return NUMBER; }
\[                { return LBRACKET; }
\]                { return RBRACKET; }
";"               { return SEMICOLON; }
[ \t\n]           ; // skip whitespace
.                 ; // skip other characters
%%
int yywrap() {
   return 1;
}
```

Yacc Code:

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

void yyerror(const char* s) {
   fprintf(stderr, "Error: %s\n", s);
}
int yylex(void);
%}

%union {
   char* str;
}

%token <str> IDENTIFIER NUMBER
%token STRUCT LBRACE RBRACE SEMICOLON LBRACKET RBRACKET

%%
input:
   STRUCT IDENTIFIER LBRACE members RBRACE SEMICOLON
   ;
```

```
members:
   members member
   | member
   ;

member:
   type IDENTIFIER SEMICOLON {
      printf("%s %s;\n", $1, $2);
   }
   | type IDENTIFIER LBRACKET NUMBER RBRACKET SEMICOLON {
      printf("%s %s[%s];\n", $1, $2, $4);
   }
   ;

type:
   IDENTIFIER
   ;
%%
```

Aim : Develop a Program to THE while loop into the for loop the code should be same using LEX and YACC tools.

📁 **Input Example:**

```
i = 0;
while (i < 10) {
   printf("%d\n", i);
   i++;
```

```
}
```

📁 **Expected Output:**

```
for (i = 0; i < 10; i++) {
   printf("%d\n", i);
}
```

🛠️ **LEX File: WhileToFor.l**

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

%%

"while"        { return WHILE; }
"("            { return LPAREN; }
")"            { return RPAREN; }
"{"            { return LBRACE; }
"}"            { return RBRACE; }
";"            { return SEMICOLON; }
"<"            { return LT; }
">"            { return GT; }
"="            { return EQUAL; }
"++"           { return INC; }
[0-9]+         { yylval.str = strdup(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { yylval.str = strdup(yytext); return ID; }
```

```
[ \t\n]        ;   // ignore whitespace
.              { return *yytext; }

%%
```

📜 **YACC File: WhileToFor.y**

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

void yyerror(const char* s);
int yylex(void);

%}

%union {
   char* str;
}

%token <str> ID NUMBER
%token WHILE LPAREN RPAREN LBRACE RBRACE SEMICOLON LT GT EQUAL INC

%%

program:
```

```
    statement
    ;

statement:
    assignment WHILE LPAREN condition RPAREN LBRACE body increment RBRACE {
        printf("for (%s; %s; %s) {\n%s}\n", $1, $4, $7, $6);
    }
    ;

assignment:
    ID EQUAL NUMBER SEMICOLON {
        char* result = malloc(100);
        sprintf(result, "%s = %s", $1, $3);
        $$ = result;
    }
    ;

condition:
    ID LT NUMBER {
        char* result = malloc(100);
        sprintf(result, "%s < %s", $1, $3);
        $$ = result;
    }
    ;

body:
    /* support body with single printf */
```

```
    ID LPAREN STRING RPAREN SEMICOLON {
        char* result = malloc(100);
        sprintf(result, "    %s(%s);\n", $1, $3);
        $$ = result;
    }
    ;

increment:
    ID INC SEMICOLON {
        char* result = malloc(100);
        sprintf(result, "%s++", $1);
        $$ = result;
    }
    ;

%%

void yyerror(const char* s) {
    fprintf(stderr, "Error: %s\n", s);
}

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

🖥️ **Commands to Compile and Run:**

lex WhileToFor.l

yacc -d WhileToFor.y

cc lex.yy.c y.tab.c -o whiletofor

./whiletofor

Aim : Develop a Program to THE for loop into the while loop the the code should be same using LEX and YACC tools.

📁 **Example Input (for loop):**

```
for (i = 0; i < 5; i++) {
    printf("%d\n", i);
}
```

📁 **Expected Output (while loop):**

```
i = 0;
while (i < 5) {
    printf("%d\n", i);
    i++;
}
```

LEX File: ForToWhile.l

```
%{
```

```
#include "y.tab.h"
%}

%%

"for"          { return FOR; }
"while"        { return WHILE; }
"("            { return LPAREN; }
")"            { return RPAREN; }
"{"            { return LBRACE; }
"}"            { return RBRACE; }
";"            { return SEMICOLON; }
"<"            { return LT; }
">"            { return GT; }
"="            { return EQUAL; }
"++"           { return INC; }
[0-9]+         { yylval.str = strdup(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { yylval.str = strdup(yytext); return ID; }
[ \t\n]        ;   // ignore whitespace
.              { return *yytext; }

%%
```

YACC File: ForToWhile.y

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

void yyerror(const char* s);
int yylex(void);

%}

%union {
   char* str;
}

%token <str> ID NUMBER
%token FOR LPAREN RPAREN LBRACE RBRACE SEMICOLON LT GT EQUAL INC

%%

program:
   loop
   ;

loop:
   FOR LPAREN init SEMICOLON condition SEMICOLON increment RPAREN LBRACE body RBRACE
{
```

```
        printf("%s;\n", $3);                    // Initialization
        printf("while (%s) {\n", $5);           // While condition
        printf("%s", $9);                       // Body
        printf("    %s;\n", $7);                // Increment
        printf("}\n");
    }
    ;

init:
    ID EQUAL NUMBER {
        char* res = malloc(100);
        sprintf(res, "%s = %s", $1, $3);
        $$ = res;
    }
    ;

condition:
    ID LT NUMBER {
        char* res = malloc(100);
        sprintf(res, "%s < %s", $1, $3);
        $$ = res;
    }
    ;

increment:
    ID INC {
        char* res = malloc(100);
```

```
        sprintf(res, "%s++", $1);
        $$ = res;
    }
    ;

body:
    ID LPAREN STRING RPAREN SEMICOLON {
        char* res = malloc(100);
        sprintf(res, "    %s(%s);\n", $1, $3);
        $$ = res;
    }
    ;

%%

void yyerror(const char* s) {
    fprintf(stderr, "Error: %s\n", s);
}

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

🧪 **How to Compile and Run:**

lex ForToWhile.l

```
yacc -d ForToWhile.y
cc lex.yy.c y.tab.c -o fortowhile
./fortowhile
```

🔎 **Sample Run (Input):**

```
for (i = 0; i < 5; i++) {
    printf("%d\n");
}
```

✅ **Output:**

```
i = 0;
while (i < 5) {
    printf("%d\n");
    i++;
}
```