SITAM

SATYA INSTITUTE OF TECHNOLOGY AND MANAGEMENT

EAMCET CODE: SGVP ACCREDITED BY NAAC, RECOGNISED UNDER 2(F) AND 12 (B) OF UGC JNTU - GV CODE: B6

APPROVED BY AICTE, NEW DELHI,

(PERMANENTLY AFFILIATED TO JNTU-GV, RECOGNISED BY SBTET, GOVERNMENT OF A.P.)

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

COMPILER DESIGN (R20)

COMPILER DESIGN LAB SYLLABUS

Sl. No.	List of Experiments
1	Write a C program to identify different types of Tokens in a given Program.
2	Write a Lex Program to implement a Lexical Analyzer using Lex tool.
3	Write a C program to Simulate Lexical Analyzer to validating a given input String.
4	Write a C program to implement the Brute force technique of Top down Parsing.
5	Write a C program to implement a Recursive Descent Parser.
6	Write C program to compute the First and Follow Sets for the given Grammar
7	Write a C program for eliminating the left recursion and left factoring of a given grammar
8	Write a C program to check the validity of input string using Predictive Parser.
9	Write a C program for implementation of LR parsing algorithm to accept a given input string..
10	Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar
11	Simulate the calculator using LEX and YACC tool.
12	Generate YACC specifications for a few syntactic categories.
13	Write a C program for generating the three address code of a given expression / statement.
14	Write a C program for implementation of a Code Generation Algorithm of a given expression / statement.

EXPERIMENT- 1

OBJECTIVE:

Write a C program to identify different types of Tokens in a given Program

PROGRAM:

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

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch) {
return (ch == ' ' || ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
ch == ',' || ch == ';' || ch == '>' || ch == '<' || ch == '=' ||
ch == '(' || ch == ')' || ch == '[' || ch == ']' ||
ch == '{' || ch == '}');
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch) {
return (ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
ch == '>' || ch == '<' || ch == '=');
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str) {
if (str[0] >= '0' && str[0] <= '9' || isDelimiter(str[0]))
return false;
return true;
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str) {
const char* keywords[] = {"if", "else", "while", "do", "break", "continue", "int", "double", "float",
"return", "char", "case", "sizeof", "long", "short", "typedef", "switch",
"unsigned", "void", "static", "struct", "goto"};

for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
if (!strcmp(str, keywords[i]))
return true;
}
return false;
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str) {
int len = strlen(str);
if (len == 0)
return false;

for (int i = 0; i < len; i++) {
if ((str[i] < '0' || str[i] > '9') || (str[i] == '-' && i > 0))
return false;
}
return true;
}

// Returns 'true' if the string is a REAL NUMBER.
```

```
bool isRealNumber(char* str) {
int len = strlen(str);
bool hasDecimal = false;

if (len == 0)
return false;

for (int i = 0; i < len; i++) {
if ((str[i] < '0' || str[i] > '9') && str[i] != '.' || (str[i] == '-' && i > 0))
return false;
if (str[i] == '.')
hasDecimal = true;
}
return hasDecimal;
}

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right) {
int len = right - left + 1;
char* subStr = (char*)malloc(sizeof(char) * (len + 1));
strncpy(subStr, &str[left], len);
subStr[len] = '\0';
return subStr;
}

// Parsing the input STRING.
void parse(char* str) {
int left = 0, right = 0;
int len = strlen(str);

while (right <= len && left <= right) {
if (!isDelimiter(str[right]))
right++;

if (isDelimiter(str[right]) && left == right) {
if (isOperator(str[right]))
printf("'%c' IS AN OPERATOR\n", str[right]);
right++;
left = right;
} else if (isDelimiter(str[right]) && left != right || (right == len && left != right)) {
char* subStr = subString(str, left, right - 1);

if (isKeyword(subStr))
printf("'%s' IS A KEYWORD\n", subStr);
else if (isInteger(subStr))
printf("'%s' IS AN INTEGER\n", subStr);
else if (isRealNumber(subStr))
printf("'%s' IS A REAL NUMBER\n", subStr);
else if (validIdentifier(subStr) && !isDelimiter(str[right - 1]))
printf("'%s' IS A VALID IDENTIFIER\n", subStr);
else if (!validIdentifier(subStr) && !isDelimiter(str[right - 1]))
printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

left = right;
free(subStr);
}
}
}

// DRIVER FUNCTION
int main() {
char str[100] = "int a = b + 1c;";
```

```
parse(str);
return 0;
}
```

Output:

```
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'1c' IS NOT A VALID IDENTIFIER
```

EXPERIMENT- 2

OBJECTIVE:

Write a Lex Program to implement a Lexical Analyzer using Lex tool.

STEPS:

1.INSTALL FLEX.
2.OPEN NEW FILE LEX AND WRITE LEX PROGRAM and save it as lexp.l
3. Then Go to Tools and click Compile Lex
4.yylex.c is created. go to Tools and click Lex Build.
5.Then lexp.exe is created.
6.Then open cmd and go to editor plus in FLEX directory and write a.exe

PROGRAM:

```
/* program name is lexp.l */
%{
   /* 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;
}
```

Input:

```
#include<stdio.h>
main()
{
int a,b;
}
```

Output:

```
#include<stdio.h> is a PREPROCESSOR DIRECTIVE
FUNCTION
main (
)
BLOCK BEGINS
int is a KEYWORD
a IDENTIFIER
b IDENTIFIER
BLOCK ENDS
```

EXPERIMENT- 3

OBJECTIVE:

Write a C program to Simulate Lexical Analyzer to validating a given input String

PROGRAM:

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

int main() {
    char string[50];
    int count = 0, i;

    printf("Enter the String: ");
    gets(string);

    if ((string[0] >= 'a' && string[0] <= 'z') ||
        (string[0] >= 'A' && string[0] <= 'Z') ||
        (string[0] == '_') ||
        (string[0] == '$')) {

        for (i = 1; i < strlen(string); i++) {
            if ((string[i] >= 'a' && string[i] <= 'z') ||
                (string[i] >= 'A' && string[i] <= 'Z') ||
                (string[i] >= '0' && string[i] <= '9') ||
                (string[i] == '_')) {
                count++;
            }
        }
    }

    if (count == (strlen(string) - 1)) {
        printf("Input string is a valid identifier");
    } else {
        printf("Input string is not a valid identifier");
    }

    return 0;
}
```

OUTPUT:

```
Enter the String: Welcome123
Input String is valid identifier

Enter the String:123ygfjy
Input String is not a valid identifier
```

EXPERIMENT- 4

OBJECTIVE:

Write a C program to implement the Brute force technique of Top down Parsing

PROGRAM:

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

void check(void);
void set_value_backtracking(void);
void get_value_backtracking(void);
void display_output_string(void);

int iptr = 0, optr = 0, current_optr = 0;
char output_string[20], current_output_string[20], input_string[20], temp_string[20];

int main() {
    printf("\nEnter the string to check: ");
    scanf("%s", input_string);
    check();
    return 0;
}

void check(void) {
    int flag = 1, rule2_index = 1;
    strcpy(output_string, "S");

    printf("\nThe output string in different stages are:\n");

    while (iptr <= strlen(input_string)) {
        if (strcmp(output_string, temp_string) != 0) {
            display_output_string();
        }

        if ((iptr != strlen(input_string)) || (optr != strlen(output_string))) {
            if (input_string[iptr] == output_string[optr]) {
                iptr = iptr + 1;
                optr = optr + 1;
            } else {
                if (output_string[optr] == 'S') {
                    memset(output_string, 0, strlen(output_string));
                    strcpy(output_string, "cAd");
                } else if (output_string[optr] == 'A') {
                    set_value_backtracking();
                    if (rule2_index == 1) {
```

```
            memset(output_string, 0, strlen(output_string));
            strcpy(output_string, "cabd");
          } else {
            memset(output_string, 0, strlen(output_string));
            strcpy(output_string, "cad");
          }
        } else if (output_string[optr] == 'b' && input_string[iptr] == 'd') {
          rule2_index = 2;
          get_value_backtracking();
          iptr = iptr - 1;
        } else {
          printf("\nThe given string, '%s' is invalid.\n\n", input_string);
          break;
        }
      }
    } else {
      printf("\nThe given string, '%s' is valid.\n\n", input_string);
      break;
    }
  }
}

void set_value_backtracking(void) {
  // Setting values for backtracking
  current_optr = optr;
  strcpy(current_output_string, output_string);
  return;
}

void get_value_backtracking(void) {
  // Backtracking and obtaining previous values
  optr = current_optr;
  memset(output_string, 0, strlen(output_string));
  strcpy(output_string, current_output_string);
  return;
}

void display_output_string(void) {
  printf("%s\n", output_string);
  memset(temp_string, 0, strlen(temp_string));
  strcpy(temp_string, output_string);
  return;
}
```

OUTPUT:

```
Enter the string to check: cad
The output string in different stages are:
S
cAd
cabd
cAd
cad

The given string, 'cad' is valid.
```

EXPERIMENT- 5

OBJECTIVE:

Write a C program to implement a Recursive Descent Parser.

PROGRAM:

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

char input[10];
int i, error;
void E();
void T();
void Eprime();
void Tprime();
void F();

int main() {
   i = 0;
   error = 0;
   printf("Enter an arithmetic expression   :  "); // Eg: a+a*a
   gets(input);
   E();
   if (strlen(input) == i && error == 0)
      printf("\nAccepted..!!!\n");
   else
      printf("\nRejected..!!!\n");
   return 0;
}

void E() {
   T();
   Eprime();
}

void Eprime() {
   if (input[i] == '+') {
      i++;
      T();
      Eprime();
   }
}

void T() {
   F();
   Tprime();
}

void Tprime() {
   if (input[i] == '*') {
      i++;
      F();
      Tprime();
   }
```

```
}

void F() {
    if (isalnum(input[i]))
        i++;
    else if (input[i] == '(') {
        i++;
        E();
        if (input[i] == ')')
            i++;
        else
            error = 1;
    } else
        error = 1;
}
```

OUTPUT:

```
Enter an algebraic expression  a+b*c
Accepted !!!

Enter an algebraic expression  a-b
Rejected !!!
```

EXPERIMENT- 6

OBJECTIVE:

Write C program to compute the First and Follow Sets for the given Grammar

PROGRAM:

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

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;
char calc_first[10][100]; // Stores First Sets
char calc_follow[10][100]; // Stores Follow Sets
char production[10][10]; // Stores production rules
char f[10], first[10];
int k, m = 0, e;
char ck;

int main() {
    int jm = 0, km = 0, i, kay, ptr = -1;
    char c, done[10], donee[10];

    count = 8;
    // Input grammar
    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");

    // Initializing calc_first array
    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
        }
    }

    int point1 = 0, point2, xxx;
    for (k = 0; k < count; k++) {
        c = production[k][0];
        point2 = 0;
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++) {
```

```
        if (c == done[kay]) xxx = 1;
    }
    if (xxx == 1) continue;
    findfirst(c, 0, 0);
    done[++ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;
    for (i = jm; i < n; i++) {
        int lark, chk = 0;
        for (lark = 0; lark < point2; lark++) {
            if (first[i] == calc_first[point1][lark]) {
                chk = 1;
                break;
            }
        }
        if (chk == 0) {
            printf("%c, ", first[i]);
            calc_first[point1][point2++] = first[i];
        }
    }
    printf("}\n");
    jm = n;
    point1++;
}

printf("\n-----------------------------------------------\n\n");
ptr = -1;
// Initializing calc_follow array
for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}

point1 = 0;
for (e = 0; e < count; e++) {
    ck = production[e][0];
    point2 = 0;
    xxx = 0;
    for (kay = 0; kay <= ptr; kay++) {
        if (ck == donee[kay]) xxx = 1;
    }
    if (xxx == 1) continue;
    follow(ck);
    donee[++ptr] = ck;
    printf(" Follow(%c) = { ", ck);
    calc_follow[point1][point2++] = ck;
    for (i = km; i < m; i++) {
        int lark, chk = 0;
        for (lark = 0; lark < point2; lark++) {
            if (f[i] == calc_follow[point1][lark]) {
                chk = 1;
                break;
            }
        }
        if (chk == 0) {
            printf("%c, ", f[i]);
```

```
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c) {
    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (int i = 0; i < 10; i++) {
        for (int j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    followfirst(production[i][j + 1], i, j + 2);
                }
                if (production[i][j + 1] == '\0' && c != production[i][0]) {
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2) {
    if (!isupper(c)) {
        first[n++] = c;
    }
    for (int j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else
                    findfirst(production[q1][q2], q1, q2 + 1);
            } else if (!isupper(production[j][2])) {
                first[n++] = production[j][2];
            } else {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2) {
    if (!isupper(c)) {
        f[m++] = c;
    } else {
        int i, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c) break;
        }
        while (calc_first[i][j] != '!') {
            if (calc_first[i][j] != '#') {
```

```
                f[m++] = calc_first[i][j];
            } else {
                if (production[c1][c2] == '\0') {
                    follow(production[c1][0]);
                } else {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

OUTPUT:

```
First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { *, #, }

First(F) = { (, i, }

-----------------------------------------------

Follow(E) = { $, ),  }

Follow(R) = { $, ),  }

Follow(T) = { +, $, ),  }

Follow(Y) = { +, $, ),  }

Follow(F) = { *, +, $, ),  }
```

EXPERIMENT- 7

OBJECTIVE:

Write a C program for eliminating the left recursion and left factoring of a given grammar

PROGRAM FOR LEFT RECURSION:

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

void main() {
    char input[100], l[50], r[50], temp[10], tempprod[20], productions[25][50];
    int i = 0, j = 0, flag = 0, consumed = 0;

    printf("Enter the productions: ");
    scanf("%1s->%s", l, r);
    printf("%s", r);

    while (sscanf(r + consumed, "%[^|]s", temp) == 1 && consumed <= strlen(r)) {
        if (temp[0] == l[0]) {
            flag = 1;
            sprintf(productions[i++], "%s->%s%s'", l, temp + 1, l);
        } else {
            sprintf(productions[i++], "%s'->%s%s'", l, temp, l);
        }
        consumed += strlen(temp) + 1;
    }

    if (flag == 1) {
        sprintf(productions[i++], "%s->ε", l);
        printf("The productions after eliminating Left Recursion are:\n");
        for (j = 0; j < i; j++)
            printf("%s\n", productions[j]);
    } else {
        printf("The Given Grammar has no Left Recursion");
    }
}
```

OUTPUT:

```
Enter the productions: E->E+E|T
The productions after eliminating Left Recursion are:
E->+EE'
E'->TE'
E->ε
```

PROGRAM FOR LEFT FACTORING:

```
#include <stdio.h>
#include <string.h>
int main() {
    char gram[50], part1[50], part2[50], modifiedGram[50], newGram[50];
    int i, j = 0, k = 0, pos;

    printf("Enter Production: A->");
    fgets(gram, sizeof(gram), stdin);
    gram[strcspn(gram, "\n")] = '\0'; // Remove newline character if present

    // Splitting the production into two parts at '|'
    for (i = 0; gram[i] != '|'; i++, j++) {
        part1[j] = gram[i];
    }
    part1[j] = '\0';

    for (j = ++i, i = 0; gram[j] != '\0'; j++, i++) {
        part2[i] = gram[j];
    }
    part2[i] = '\0';

    // Finding common prefix
    for (i = 0; i < strlen(part1) && i < strlen(part2); i++) {
        if (part1[i] == part2[i]) {
            modifiedGram[k++] = part1[i];
            pos = i + 1;
        }
    }

    // If common prefix exists, modify the productions
    if (k > 0) {
        modifiedGram[k] = 'X'; // Add new non-terminal X
        modifiedGram[++k] = '\0'; // Null-terminate the modified production

        // Storing the remaining part of the first production in newGram
        for (i = pos, j = 0; part1[i] != '\0'; i++, j++) {
            newGram[j] = part1[i];
        }
        newGram[j++] = '|'; // Add '|' to separate alternate productions

        // Storing the remaining part of the second production in newGram
        for (i = pos; part2[i] != '\0'; i++, j++) {
            newGram[j] = part2[i];
        }
        newGram[j] = '\0'; // Null-terminate the new production

        // Printing the left-factored grammar
        printf("\nGrammar Without Left Factoring:\n");
        printf("A->%s\n", modifiedGram);
        printf("X->%s\n", newGram);
    } else {
        // If no common prefix found, print original grammar
        printf("\nNo left factoring needed. Grammar remains the same:\n");
        printf("A->%s|%s\n", part1, part2);
    }
```

```
    return 0;
}
```

OUTPUT:

```
Enter the productions: A->Ba | Bb
Grammar Without Left Factoring:

A->BX
X->a | b
```

EXPERIMENT- 8

OBJECTIVE:

Write a C program to check the validity of input string using Predictive Parser**.**

PROGRAM:

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

#define id 0
#define CONST 1
#define mulop 2
#define addop 3
#define op 4
#define cp 5
#define err 6
#define col 7
#define size 50

int token;
char lexbuff[size];
int lookahead = 0;

int lexer();
int E();
int T();
int EPRIME();
int TPRIME();
int F();
void parser();

int main() {
    // Replace clrscr with a clearer alternative for UNIX-like systems
    printf("\033[H\033[J");  // ANSI escape sequence to clear screen

    printf("Enter the string: ");
    fgets(lexbuff, size, stdin);  // Use fgets instead of gets for safer input
    parser();
    return 0;
}

void parser() {
    if (E())
        printf("Valid string\n");
    else
        printf("Invalid string\n");
}

int E() {
    if (T()) {
        if (EPRIME())
            return 1;
        else
            return 0;
```

```
    } else {
        return 0;
    }
}

int T() {
    if (F()) {
        if (TPRIME())
            return 1;
        else
            return 0;
    } else {
        return 0;
    }
}

int EPRIME() {
    token = lexer();
    if (token == addop) {
        lookahead++;
        if (T()) {
            if (EPRIME())
                return 1;
            else
                return 0;
        } else {
            return 0;
        }
    } else {
        return 1;
    }
}

int TPRIME() {
    token = lexer();
    if (token == mulop) {
        lookahead++;
        if (F()) {
            if (TPRIME())
                return 1;
            else
                return 0;
        } else {
            return 0;
        }
    } else {
        return 1;
    }
}

int F() {
    token = lexer();
    if (token == id)
        return 1;
    else {
        if (token == op) {
            if (E()) {
```

```
            if (token == cp)
                return 1;
            else
                return 0;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
  }
}

int lexer() {
    if (lexbuff[lookahead] != '\n') {
        while (lexbuff[lookahead] == ' ' || lexbuff[lookahead] == '\t')
            lookahead++;

        if (isalpha(lexbuff[lookahead])) {
            while (isalnum(lexbuff[lookahead]))
                lookahead++;
            return id;
        } else {
            if (isdigit(lexbuff[lookahead])) {
                while (isdigit(lexbuff[lookahead]))
                    lookahead++;
                return CONST;
            } else {
                if (lexbuff[lookahead] == '+')
                    return addop;
                else if (lexbuff[lookahead] == '*')
                    return mulop;
                else if (lexbuff[lookahead] == '(') {
                    lookahead++;
                    return op;
                } else if (lexbuff[lookahead] == ')') {
                    return cp;
                } else {
                    return err;
                }
            }
        }
    } else {
        return col;
    }
}
```

OUTPUT:

```
Enter the string: id*id+id
valid string

Enter the string: id-id
invalid string
```

EXPERIMENT- 9

OBJECTIVE:

Write a C program for implementation of LR parsing algorithm to accept a given input string

PROGRAM:

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

#define MAX_STACK_SIZE 100

typedef struct {
    char stack[MAX_STACK_SIZE];
    int top;
} Parser;

void initParser(Parser* parser) {
    parser->top = -1;
}

void push(Parser* parser, char symbol) {
    if (parser->top < MAX_STACK_SIZE - 1) {
        parser->stack[++(parser->top)] = symbol;
    }
}

char pop(Parser* parser) {
    if (parser->top >= 0) {
        return parser->stack[(parser->top)--];
    }
    return '\0';
}

bool isTerminal(char c) {
    return c == 'a' || c == 'b' || c == '+' || c == '*' || c == '(' || c == ')';
}

bool parse(char* input) {
    Parser parser;
    initParser(&parser);
    int i = 0;

    while (input[i] != '\0') {
        // Simple recursive descent parsing
        if (input[i] == 'a' || input[i] == 'b') {
            push(&parser, 'F');  // Factor
        } else if (input[i] == '+') {
            if (parser.stack[parser.top] != 'F') return false;
            pop(&parser);
            push(&parser, 'E');  // Expression
        } else if (input[i] == '*') {
            if (parser.stack[parser.top] != 'F') return false;
```

```
            pop(&parser);
            push(&parser, 'T');  // Term
        } else if (input[i] == '(') {
            push(&parser, input[i]);
        } else if (input[i] == ')') {
            // Reduce parenthesized expression
            while (parser.top >= 0 && parser.stack[parser.top] != '(') {
                if (parser.stack[parser.top] == 'F' || parser.stack[parser.top] == 'E' || parser.stack[parser.top] ==
'T') {
                    pop(&parser);
                } else {
                    return false;
                }
            }
            if (parser.top < 0) return false;
            pop(&parser);  // Remove '('
            push(&parser, 'F');
        }
        i++;
    }

    // Final validation
    while (parser.top >= 0) {
        char top = parser.stack[parser.top];
        if (top != 'F' && top != 'E' && top != 'T') return false;
        pop(&parser);
    }

    return true;
}

int main() {
    // Test input strings
    char* accepted_strings[] = {
        "a",
        "b",
        "a+b",
        "a*b",
        "(a+b)",
        "a+a*b",
        "(a)*b",
        "a+(b*a)"
    };

    char* rejected_strings[] = {
        "+",
        "*",
        "a++b",
        "(a",
        "a)",
        "a+*b"
    };

    printf("Accepted Strings:\n");
    for (int i = 0; i < sizeof(accepted_strings) / sizeof(char*); i++) {
        printf("%s - %s\n",
               accepted_strings[i],
```

```
        parse(accepted_strings[i]) ? "Accepted" : "Rejected");
    }

    printf("\nRejected Strings:\n");
    for (int i = 0; i < sizeof(rejected_strings) / sizeof(char*); i++) {
        printf("%s - %s\n",
            rejected_strings[i],
            parse(rejected_strings[i]) ? "Accepted" : "Rejected");
    }

    return 0;
}
```

OUTPUT:

```
Accepted Strings:
a - Accepted
b - Accepted
a+b - Accepted
a*b - Accepted
(a+b) - Accepted
a+a*b - Accepted
(a)*b - Accepted
a+(b*a) - Accepted

Rejected Strings:
+ - Rejected
* - Rejected
a++b - Rejected
(a - Rejected
a) - Rejected
a+*b - Rejected
```

EXPERIMENT- 10

OBJECTIVE:

Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar

PROGRAM:

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

int k = 0, z = 0, i = 0, j = 0, c = 0;
char a[16], ac[20], stk[15], act[10];

void check();

int main() {
    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("Enter input string: ");
    gets(a);

    c = strlen(a);
    strcpy(act, "SHIFT->");

    puts("Stack \t Input \t Action");

    for (k = 0, i = 0; j < c; k++, i++, j++) {
        if (a[j] == 'i' && a[j + 1] == 'd') {
            stk[i] = a[j];
            stk[i + 1] = a[j + 1];
            stk[i + 2] = '\0';
            a[j] = ' ';
            a[j + 1] = ' ';
            printf("\n$%s\t%s$\t%sid", stk, a, act);
            check();
        } else {
            stk[i] = a[j];
            stk[i + 1] = '\0';
            a[j] = ' ';
            printf("\n$%s\t%s$\t%ssymbols", stk, a, act);
            check();
        }
    }
}

void check() {
    strcpy(ac, "REDUCE TO E");

    for (z = 0; z < c; z++) {
        if (stk[z] == 'i' && stk[z + 1] == 'd') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            j++;
```

```
        }
    }

    for (z = 0; z < c; z++) {
        if (stk[z] == 'E' && stk[z + 1] == '+' && stk[z + 2] == 'E') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i - 2;
        }
    }

    for (z = 0; z < c; z++) {
        if (stk[z] == 'E' && stk[z + 1] == '*' && stk[z + 2] == 'E') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i - 2;
        }
    }

    for (z = 0; z < c; z++) {
        if (stk[z] == '(' && stk[z + 1] == 'E' && stk[z + 2] == ')') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i - 2;
        }
    }
}
```

OUTPUT:

```
GRAMMAR is E->E+E
E->E*E
E->(E)
E->id enter input string
id+id*id+id

stack         input          action
$id           +id*id+id$     SHIFT->id
$E            +id*id+id$     REDUCE TO E
$E+            id*id+id$     SHIFT->symbols
$E+id            *id+id$     SHIFT->id
$E+E             *id+id$     REDUCE TO E
$E               *id+id$     REDUCE TO E
$E*               id+id$     SHIFT->symbols
$E*id               +id$     SHIFT->id
$E*E                +id$     REDUCE TO E
$E                  +id$     REDUCE TO E
$E+                  id$     SHIFT->symbols
$E+id                  $     SHIFT->id
$E+E                   $     REDUCE TO E
$E                     $     REDUCE TO E
```

EXPERIMENT- 11

OBJECTIVE:

Simulate the calculator using LEX and YACC tool.

PROGRAM:

```
%{
/* Definition section */
#include <stdio.h>
int flag = 0;
%}

%token NUMBER

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

/* Rule Section */
%%

ArithmeticExpression: E {
    printf("\nResult = %d\n", $$);
    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; }
;

%%

// Driver code
void main() {
    printf("\nEnter any arithmetic expression which can have operations Addition, Subtraction,
Multiplication, Division, Modulus, and Round brackets:\n");
    yyparse();

    if (flag == 0)
        printf("\nEntered arithmetic expression is valid\n\n");
}

void yyerror() {
    printf("\nEntered arithmetic expression is invalid\n\n");
    flag = 1;
}
```

OUTPUT:

Input: 4+5
Output: Result=9
Entered arithmetic expression is Valid

Input: 10-5
Output: Result=5
Entered arithmetic expression is Valid

Input: 10+5-
Output:
Entered arithmetic expression is Invalid

EXPERIMENT- 12

OBJECTIVE:

Generate YACC specifications for a few syntactic categories.

Program:

YACC Specification (calc.y):

```
%{
/* Definition section */
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(char *s);
%}

/* Tokens */
%token NUMBER
%token PLUS MINUS MUL DIV

/* Precedence and associativity */
%left PLUS MINUS
%left MUL DIV

%%

/* Grammar Rules */

/* Start symbol: the arithmetic expression */
start:
   expression
   {
      printf("Result = %d\n", $1);
   }
   ;

expression:
   expression PLUS term   { $$ = $1 + $3; }
   | expression MINUS term { $$ = $1 - $3; }
   | term                 { $$ = $1; }
   ;

term:
   term MUL factor        { $$ = $1 * $3; }
   | term DIV factor      { $$ = $1 / $3; }
   | factor               { $$ = $1; }
   ;

factor:
   NUMBER                 { $$ = $1; }
   | '(' expression ')'   { $$ = $2; }
   ;

%%
```

```
/* Driver Code */

int main()
{
    printf("Enter an arithmetic expression: \n");
    yyparse();
    return 0;
}

/* Error handling function */
void yyerror(char *s)
{
    fprintf(stderr, "Error: %s\n", s);
}
```

Lex Specification (calc.l):

```
%{
#include "y.tab.h"  // Include the YACC header file generated by YACC
%}

digit   [0-9]
%%

{digit}+      { yylval = atoi(yytext); return NUMBER; }
"+"           { return PLUS; }
"-"           { return MINUS; }
"*"           { return MUL; }
"/"           { return DIV; }
"("           { return '('; }
")"           { return ')'; }
[ \t\n]       { /* skip whitespace */ }
.             { return yytext[0]; }

%%

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

Steps to Compile and Run:

1. Save the YACC file as calc.y and the Lex file as calc.l.
2. Run Lex to generate the lexer:
3. lex calc.l
4. Run YACC to generate the parser:
5. yacc -d calc.y
6. Compile the generated C files:
7. gcc lex.yy.c y.tab.c -o calc -lfl
8. Run the program:
9. ./calc

Output:

For input like:
Enter an arithmetic expression:
3 + 5 * (2 - 8)
Output will be:
Result = -13
In this case, the arithmetic expression 3 + 5 * (2 - 8) is parsed and evaluated correctly, considering operator precedence and parentheses, resulting in -13.

EXPERIMENT- 13

OBJECTIVE :

Write a C program for generating the three address code of a given expression / statement.

PROGRAM:

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

#define MAX 100

// Function to check if the character is an operator
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '%');
}

// Function to generate the next available temporary variable name
void generateTempVar(int tempCounter, char* tempVar) {
    sprintf(tempVar, "t%d", tempCounter);
}

// Function to generate the three-address code for the given expression
void generateThreeAddressCode(char* expr) {
    int tempCounter = 0;
    char tempVar[MAX];
    char stack[MAX][MAX];
    int top = -1;

    printf("Three Address Code for Expression: %s\n", expr);

    // Parsing the expression and generating the TAC
    for (int i = 0; i < strlen(expr); i++) {
        char current = expr[i];

        // If the character is a digit, push it to the stack
        if (isdigit(current)) {
            sprintf(stack[++top], "%c", current);
        }
        // If the character is an operator, pop two operands, apply the operator, and push the result
        else if (isOperator(current)) {
            char operand2[MAX], operand1[MAX];
            strcpy(operand2, stack[top--]);
            strcpy(operand1, stack[top--]);

            // Generate temporary variable
            generateTempVar(tempCounter++, tempVar);

            // Print the three-address code
            printf("%s = %s %c %s\n", tempVar, operand1, current, operand2);

            // Push the result (temporary variable) onto the stack
            strcpy(stack[++top], tempVar);
        }
    }
}
```

```
int main() {
    char expression[MAX];

    // Read the expression
    printf("Enter an arithmetic expression (e.g., 3+5*2): ");
    fgets(expression, MAX, stdin);

    // Remove the newline character from the input
    expression[strcspn(expression, "\n")] = '\0';

    // Generate and print the three-address code
    generateThreeAddressCode(expression);

    return 0;
}
```

OUTPUT:

```
Enter an arithmetic expression (e.g., 3+5*2): 3+5*2
Three Address Code for Expression: 3+5*2
t0 = 5 * 2
t1 = 3 + t0
```

EXPERIMENT- 14

OBJECTIVE:

Write a C program for implementation of a Code Generation Algorithm of a given expression / statement.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 100

int precedence(char op) {
    if(op == '+' || op == '-')
        return 1;
    if(op == '*' || op == '/')
        return 2;
    return 0;
}

// Convert infix expression to postfix
void infixToPostfix(char infix[], char postfix[]) {
    char stack[MAX];
    int top = -1, j = 0, len = strlen(infix);

    for (int i = 0; i < len; i++) {
        char token = infix[i];
        if(isalnum(token))
            postfix[j++] = token;
        else if(token == '(')
            stack[++top] = token;
        else if(token == ')') {
            while(top != -1 && stack[top] != '(')
                postfix[j++] = stack[top--];
            if(top != -1)
                top--; // Remove '('
        }
        else {
            while(top != -1 && precedence(stack[top]) >= precedence(token))
                postfix[j++] = stack[top--];
            stack[++top] = token;
        }
    }
    while(top != -1)
        postfix[j++] = stack[top--];

    postfix[j] = '\0';
}

typedef struct {
    char items[MAX][MAX];
    int top;
} Stack;

void push(Stack *s, char *str) {
    s->top++;
    strcpy(s->items[s->top], str);
```

```
}

void pop(Stack *s, char *str) {
    strcpy(str, s->items[s->top]);
    s->top--;
}

int main() {
    char input[MAX], infix[MAX], postfix[MAX], lhs[10] = "";

    printf("Enter an expression (e.g., a=b+c*d): ");
    if(fgets(input, sizeof(input), stdin) == NULL) {
        printf("Error reading input.\n");
        return 1;
    }
    size_t lenInput = strlen(input);
    if(lenInput > 0 && input[lenInput-1] == '\n')
        input[lenInput-1] = '\0';

    // Separate LHS and RHS if '=' is present
    char *eq_ptr = strchr(input, '=');
    if(eq_ptr != NULL) {
        int lhs_len = eq_ptr - input;
        strncpy(lhs, input, lhs_len);
        lhs[lhs_len] = '\0';
        strcpy(infix, eq_ptr + 1);
    } else {
        strcpy(infix, input);
    }

    // Remove spaces from the infix expression
    char temp[MAX];
    int k = 0;
    for(int i = 0; i < strlen(infix); i++) {
        if(infix[i] != ' ')
            temp[k++] = infix[i];
    }
    temp[k] = '\0';
    strcpy(infix, temp);

    infixToPostfix(infix, postfix);
    printf("\nPostfix expression: %s\n", postfix);

    // Generate assembly-like code from postfix expression
    Stack stack;
    stack.top = -1;
    int tempCount = 1;
    char op1[MAX], op2[MAX], tempVar[10];
    int lenPostfix = strlen(postfix);

    printf("\nGenerated Assembly Code:\n");
    for (int i = 0; i < lenPostfix; i++) {
        char token = postfix[i];
        if(isalnum(token)) {
            char operand[10];
            operand[0] = token;
            operand[1] = '\0';
```

```
            push(&stack, operand);
        } else {
            pop(&stack, op2); // Right operand
            pop(&stack, op1); // Left operand
            sprintf(tempVar, "T%d", tempCount++);
            switch(token) {
                case '+':
                    printf("MOV %s, %s\n", tempVar, op1);
                    printf("ADD %s, %s\n", tempVar, op2);
                    break;
                case '-':
                    printf("MOV %s, %s\n", tempVar, op1);
                    printf("SUB %s, %s\n", tempVar, op2);
                    break;
                case '*':
                    printf("MOV %s, %s\n", tempVar, op1);
                    printf("MUL %s, %s\n", tempVar, op2);
                    break;
                case '/':
                    printf("MOV %s, %s\n", tempVar, op1);
                    printf("DIV %s, %s\n", tempVar, op2);
                    break;
                default:
                    printf("Error: Unsupported operator '%c'\n", token);
                    break;
            }
            push(&stack, tempVar);
        }
    }

    char result[MAX];
    pop(&stack, result);

    if(strlen(lhs) > 0)
        printf("MOV %s, %s\n", lhs, result);
    else
        printf("Final result in %s\n", result);

    return 0;
}
```

OUTPUT:

```
Enter an expression (e.g., a=b+c*d): a+b+c+d*e-f
Postfix expression: ab+c+de*+f-
Generated Assembly Code:
MOV T1, a
ADD T1, b
MOV T2, T1
ADD T2, c
MOV T3, d
MUL T3, e
MOV T4, T2
ADD T4, T3
MOV T5, T4
SUB T5, f
Final result in T5
```