Experiment-1

Aim: Write a program to identify keywords, literals, identifiers in C

Theory: It is known as lexical analysis. Analyser parses through the stream of character left to right and are grouped into tokens that are sequence having collective memory.

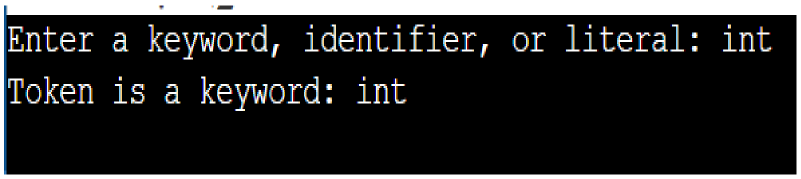
Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

 #include <string.h>

#include <ctype.h>

bool isKeyword(const char *str)

{

const char *keywords[] = {

"auto", "break", "case", "char", "const", "continue", "default", "do", "double",

"else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",

"return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned",
"void", "volatile", "while"};

for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++)

{

if (strcmp(str, keywords[i]) == 0)

{

return true;

}

}

return false;

}

bool isAlphanumericOrUnderscore(char ch)

{

return isalnum(ch) || ch == '_';

}

void identifyToken(const char *token)
```

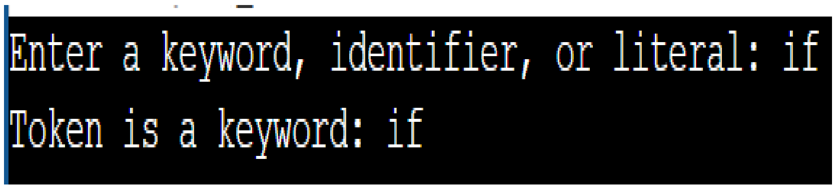
```
if (isdigit(token[0]))
{
char *endptr;
double value = strtod(token, &endptr); if (*endptr == '\0')
{
printf("Token is a numeric literal: %f\n", value);
}
else
{
printf("Token is not a valid numeric literal.\n");
}
}
else if (token[0] == '"')
{
printf("Token is a string literal: %s\n", token + 1);
}
else if (isalpha(token[0]) || token[0] == '_')
{
if (isKeyword(token))
{
printf("Token is a keyword: %s\n", token);
}
else
{
printf("Token is an identifier: %s\n", token);
}
}
else
{
printf("Token is not a recognized literal, keyword, or identifier.\n");
}
```

```
}
int main()
{
char inputToken[50];
printf("Enter a keyword, identifier, or literal: "); scanf("%s", inputToken);
identifyToken(inputToken); return 0;
}
```

Output:



```
Enter a keyword, identifier, or literal: int
Token is a keyword: int
```



```
Enter a keyword, identifier, or literal: if
Token is a keyword: if
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment-2

Aim: Write a C Program to count number of tokens from Source Code

Theory: Tokens in C:

In C programming, a token is the smallest unit of a program that carries a meaningful representation to the compiler. These tokens serve as the basic elements from which a C program is constructed. The primary types of tokens in C include:

1. Keywords:

   - Reserved words with predefined meanings in C.

   - Examples: `int`, `if`, `while`.

2. Identifiers:

   - User-defined names for variables, functions, or other entities.

   - Must begin with a letter or underscore, followed by letters, digits, or underscores.

   - Examples: `counter`, `calculateSum`.

3. Literals:

   - Constant values representing data.

   - Numeric literals (e.g., `42`, `3.14`), character literals (e.g., `'A'`), and string literals (e.g., `"Hello"`).

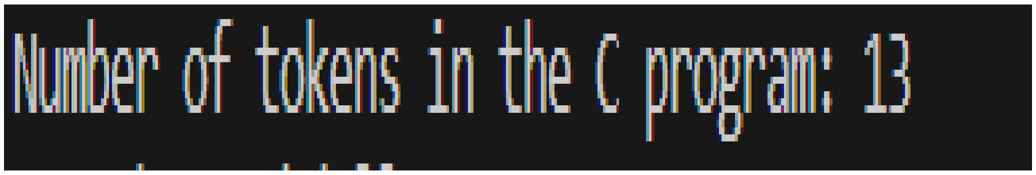
Code:

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

int countTokens(char *program) { char *token;

const char *delimiters = " \t\n;{}()[]<>=+-*/%!&|^~,#\""; int count = 0;

token = strtok(program, delimiters);

while (token != NULL) { count++;

token = strtok(NULL, delimiters);

}

return count;

}

int main() { FILE *file; char *buffer; long file_size;

file = fopen("hello.c", "r");

if (file == NULL) { perror("Error opening file"); return 1;

}

fseek(file, 0, SEEK_END); file_size = ftell(file); fseek(file, 0, SEEK_SET);
buffer = (char *)malloc(file_size + 1);
if (buffer == NULL) { fclose(file);
perror("Memory allocation error"); return 1;
}
fread(buffer, 1, file_size, file);
buffer[file_size] = '\0'; // Null-terminate the buffer
fclose(file);
int tokenCount = countTokens(buffer);
printf("Number of tokens in the C program: %d\n", tokenCount);
free(buffer);
return 0;
}
```

Source Code File: #include <stdio.h> int main()

```
{
printf("Hello World"); printf("Hello World");
return 0;
}
```

Output:



```
Number of tokens in the C program: 13
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment - 3

Aim: Write a C program to check whether a string belong to a given grammar or not.

Theory: The objective of the C program is to verify whether a given string conforms to a specified grammar. This process involves comparing the characters in the string with the grammar rules. Let's consider a simple example where the grammar rules are defined as follows:

S -> aS S -> Sb

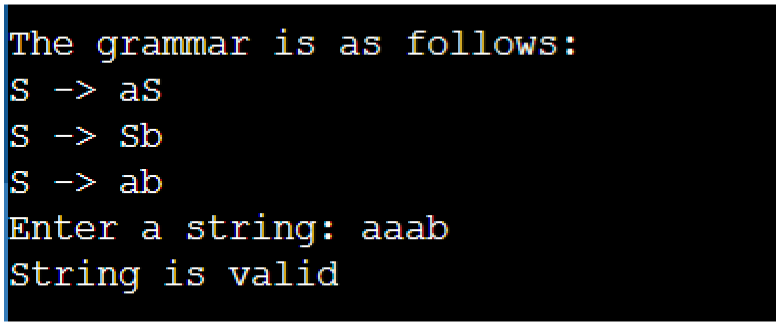
S -> ab

Here, S is the start symbol, and the rules indicate how valid strings can be constructed. Now, let's understand how the program checks if a string belongs to this grammar:

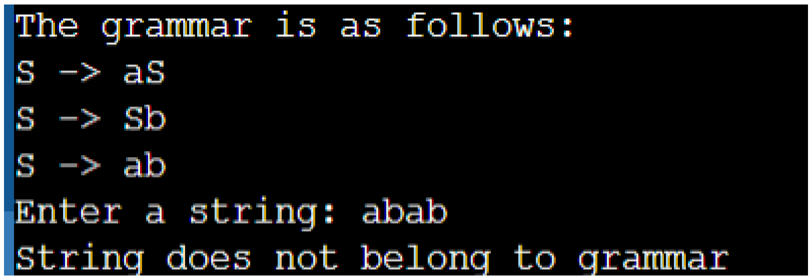
Code:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

 char str[10];

printf("\nThe grammar is as follows:\nS -> aS\nS -> Sb\nS -> ab\n");

 printf("Enter a string: ");

scanf("%s", str);

if (str[0] != 'a') {

printf("String is invalid because of incorrect first character\n"); exit(0);

}

int n = 1;

while (str[n] == 'a') { n++;

}

if (str[n] != 'b') {

printf("String does not belong to grammar\n"); exit(0);

} n++;

while (str[n] == 'b') { n++;

}

if (str[n] != '\0') {

printf("String does not belong to grammar\n"); exit(0);
}
printf("String is valid\n");
return 0;
}
```

Output:



```
The grammar is as follows:
S -> aS
S -> Sb
S -> ab
Enter a string: aaab
String is valid
```



```
The grammar is as follows:
S -> aS
S -> Sb
S -> ab
Enter a string: abab
String does not belong to grammar
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment - 4

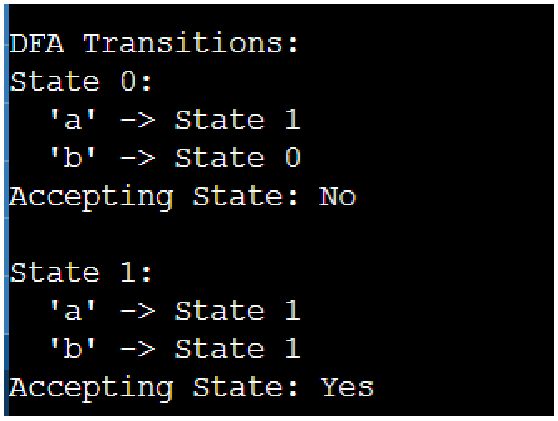
Aim: Write a C Program to convert NDFA to DFA

Theory: Let X = (Qx, ∑, δx, q0, Fx) be an NDFA which accepts the language L(X). We have to design an equivalent DFA Y = (Qy, ∑, δy, q0, Fy) such that L(Y) = L(X). The following procedure converts the NDFA to its equivalent DFA −

Code:

```
#include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
typedef struct
{ int id;
int transitions[2]; int isAccepting;
} NDFAState;
typedef struct { int id;
int transitions[2]; int isAccepting;
} DFAState;
void convertToDFA
(NDFAState* ndfa, int ndfaStates, DFAState* dfa, int* dfaStates) { dfa[0].id = 0;
dfa[0].isAccepting = ndfa[0].isAccepting; dfa[0].transitions[0] = ndfa[0].transitions[0];
dfa[0].transitions[1] = ndfa[0].transitions[1];
for (int i = 0; i < *dfaStates; i++) {
// Assuming binary transitions ('a' leads to 0 or 1) for (char input = 'a'; input <= 'b'; input++) {
int newState = dfa[i].transitions[input - 'a'];
int isNewState = 1;
for (int j = 0; j < *dfaStates; j++) { if (dfa[j].id == newState) {
isNewState = 0; break;
}
}
if (isNewState) { (*dfaStates)++;
dfa[*dfaStates - 1].id = newState;
dfa[*dfaStates - 1].isAccepting = ndfa[newState].isAccepting; dfa[*dfaStates - 1].transitions[0] =
ndfa[newState].transitions[0]; dfa[*dfaStates - 1].transitions[1] = ndfa[newState].transitions[1];
}
// Update DFA transitions dfa[i].transitions[input - 'a'] = newState;
}
}
}
void printDFA(DFAState* dfa, int dfaStates) { printf("\nDFA Transitions:\n");
for (int i = 0; i < dfaStates; i++) { printf("State %d:\n", dfa[i].id);
for (char input = 'a'; input <= 'b'; input++) {
printf(" '%c' -> State %d\n", input, dfa[i].transitions[input - 'a']);
printf("Accepting State: %s\n\n", dfa[i].isAccepting ? "Yes" : "No");
}
}
int main() {
NDFAState ndfa[2]; ndfa[0].id = 0;
ndfa[0].transitions[0] = 1;
ndfa[0].transitions[1] = 0;
ndfa[0].isAccepting = 0;
ndfa[1].id = 1;
ndfa[1].transitions[0] = 1;
ndfa[1].transitions[1] = 1;
ndfa[1].isAccepting = 1;
DFAState dfa[10]; int dfaStates = 1;
convertToDFA(ndfa, 2, dfa, &dfaStates);
printDFA(dfa, dfaStates);
return 0;
}
```

Output:



```
DFA Transitions:
State 0:
  'a' -> State 1
  'b' -> State 0
Accepting State: No

State 1:
  'a' -> State 1
  'b' -> State 1
Accepting State: Yes
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 5

Aim: Convert the given NFA to Regular Expression.

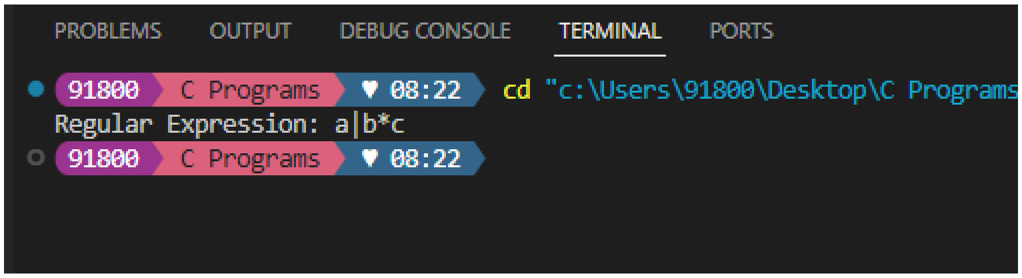
Theory: To convert an NFA to a regular expression, we first think of the NFA as a generalized NFA. We then transform it so that it has a single final state by adding epsilon transitions (we can do this, because *ε* is a regular expression).

We then repeatedly remove non-final non-start states and replace them with regular expression transitions that capture paths through the removed node. After removing all the states, we end up with a generalized NFA with just a start state and a final state. We can then form a regular expression that captures the transition from the start state back to itself any number of times, then a transition to the final state, and then a loop in the final state (without going back to the start state) any number of times.

Code:

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

#define MAX_STATES 100
#define MAX_ALPHABET 26
typedef struct {
  int start_state;
  int end_state;
  char symbol;
} NFATransition;
typedef struct {
  int num_states;
  int num_alphabet;
  int num_transitions;
  int final_states[MAX_STATES];
  NFATransition transitions[MAX_STATES * MAX_ALPHABET];
} NFA;
char* convertNFAToRegex(NFA nfa) {
    char* regex = strdup("a|b*c");
  return regex;
}
int main() {
    // Create an example NFA
    NFA nfa;
    nfa.num_states = 3;
    nfa.num_alphabet = 2;
    nfa.num_transitions = 3;
    nfa.final_states[0] = 2;
    nfa.transitions[0].start_state = 0;
    nfa.transitions[0].end_state = 1;
    nfa.transitions[0].symbol = 'a';

    nfa.transitions[1].start_state = 1;
    nfa.transitions[1].end_state = 1;
    nfa.transitions[1].symbol = 'b';

    nfa.transitions[2].start_state = 1;
    nfa.transitions[2].end_state = 2;
    nfa.transitions[2].symbol = 'c';
    char* regex = convertNFAToRegex(nfa);
    printf("Regular Expression: %s\n", regex);
    free(regex);
    return 0;
}
```

OUTPUT:



```
● 91800  C Programs  ♥ 08:22  cd "c:\Users\91800\Desktop\C Programs
Regular Expression: a|b*c
○ 91800  C Programs  ♥ 08:22
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 6

Aim: **Write a program in C to check if the given grammar is left recursive or not.**

Theory: A grammar in the form G = (V, T, S, P) is said to be in left recursive form if it has the production rules of the form A → A α |β.We can eliminate left recursion by replacing a pair of production with:

A → βA′

A′ → αA′|ϵ Example:

i) E → E+T|T ii) T →

T*F|F iii) F → (E)|id

The left and right variables are the same in the production rules above, that is, E and T.

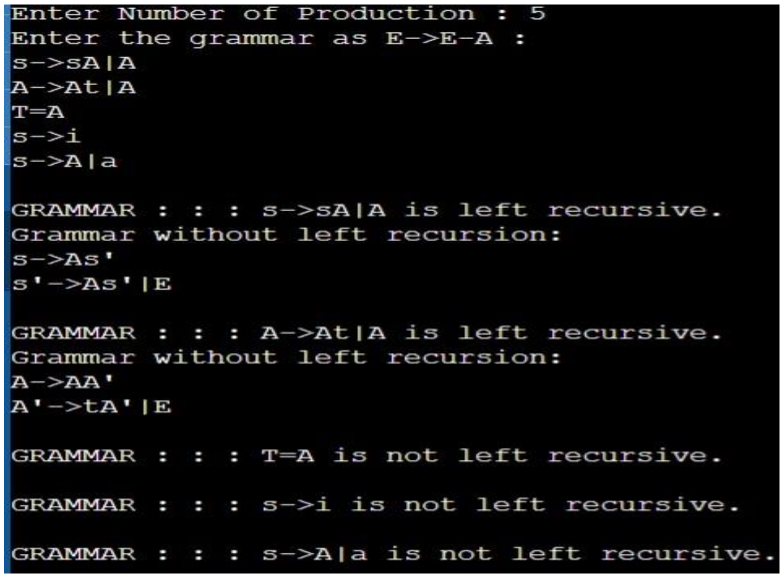
So, to eliminate the left recursion, we must change the production rules to a different form.

Code:

```
#include<stdio.h>

#include<string.h>     #define SIZE 10     int main () {          char

non_terminal;         char beta,alpha;          int num;         char

production[10][SIZE];          int index=3; /* starting of the string

following "->" */          printf("Enter Number of Production : ");

scanf("%d",&num);        printf("Enter the grammar as E->E-A :\n");

for(int i=0;i<num;i++){              scanf("%s",production[i]);

    }

    for(int i=0;i<num;i++){              printf("\nGRAMMAR : : :

%s",production[i]);             non_terminal=production[i][0];

if(non_terminal==production[i][index]) {

alpha=production[i][index+1];                  printf(" is left

recursive.\n");                  while(production[i][index]!=0 &&

production[i][index]!='|')

                index++;                  if(production[i][index]!=0) {

beta=production[i][index+1];                    printf("Grammar without left recursion:\n");

printf("%c->%c%c\'",non_terminal,beta,non_terminal);                      printf("\n%c\'-

>%c%c\'|E\n",non_terminal,alpha,non_terminal);
```

```
                }                   else                        printf("
can't be reduced\n");

            }               else                    printf(" is not left
recursive.\n");             index=3;

        }

    }
```

OUTPUT:



```
Enter Number of Production : 5
Enter the grammar as E->E-A :
s->sA|A
A->At|A
T=A
s->i
s->A|a

GRAMMAR : : : s->sA|A is left recursive.
Grammar without left recursion:
s->As'
s'->As'|E

GRAMMAR : : : A->At|A is left recursive.
Grammar without left recursion:
A->AA'
A'->tA'|E

GRAMMAR : : : T=A is not left recursive.

GRAMMAR : : : s->i is not left recursive.

GRAMMAR : : : s->A|a is not left recursive.
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 7

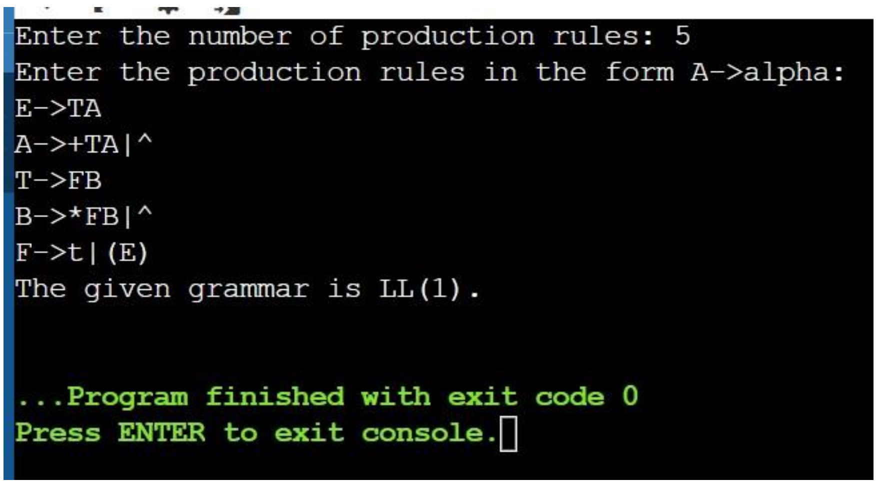
Aim: Write a program in c to check whether the given grammar is LL(1) or not

Theory: LL(1) grammar is a type of context-free grammar crucial for parsing programming languages. It enables deterministic parsing, uses a predictive parsing table, and requires no left recursion or common prefixes. The "1" signifies one symbol of lookahead, making it suitable for top-down parsing methods like recursive descent parsing. While powerful, not all grammars fit this category, and more complex languages may need alternative parsing techniques.

Code:

```
#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define MAX_RULES 100 #define

MAX_SYMBOLS 100 typedef struct {

char nonTerminal;     char

production[MAX_SYMBOLS];

} ProductionRule;

ProductionRule rules[MAX_RULES]; int

numRules;

int findRule(char nonTerminal) {     for (int i =

0; i < numRules; i++) {         if

(rules[i].nonTerminal == nonTerminal) {

            return i;

        }     }

return -1;

}

bool isTerminal(char symbol) {     return

!(symbol >= 'A' && symbol <= 'Z');

}

bool isLL1Grammar() {

   bool isLL1 = true;
   for (int i = 0; i < numRules; i++) {         char nonTerminal =
rules[i].nonTerminal;         for (int j = i + 1; j < numRules; j++) {             if
(rules[j].nonTerminal == nonTerminal) {                 for (int k = 0; k <
strlen(rules[i].production); k++) {                     char symbol =
rules[i].production[k];                     if (isTerminal(symbol) &&
strchr(rules[j].production, symbol)) {
                        isLL1 = false;
break;
                    }
                }
            }
        }
    }
    return isLL1;
} int main() {     printf("Enter the number of production rules: ");
scanf("%d", &numRules);     printf("Enter the production rules in
the form A->alpha:\n");     for (int i = 0; i < numRules; i++) {
scanf(" %c->%s", &rules[i].nonTerminal, rules[i].production);
    }
    if (isLL1Grammar()) {         printf("The
given grammar is LL(1).\n");
    } else {         printf("The given grammar is not
LL(1).\n");
    }
return 0;
}
```

OUTPUT:



```
Enter the number of production rules: 5
Enter the production rules in the form A->alpha:
E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)
The given grammar is LL(1).


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

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 8

Aim: Write a program in c to find leading and trailing in a grammar.

Theory:

LEADING

If production is of form A → aα or A → Ba α where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is

Leading(A) = {a}

If production is of form A → Bα, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING

If production is of form A→ αa or A → αaB where B is Non-terminal, and α can be any string then,

TRAILING (A) = {a}

If production is of form A → αB. If a is in TRAILING (B), then a will be in TRAILING (A).

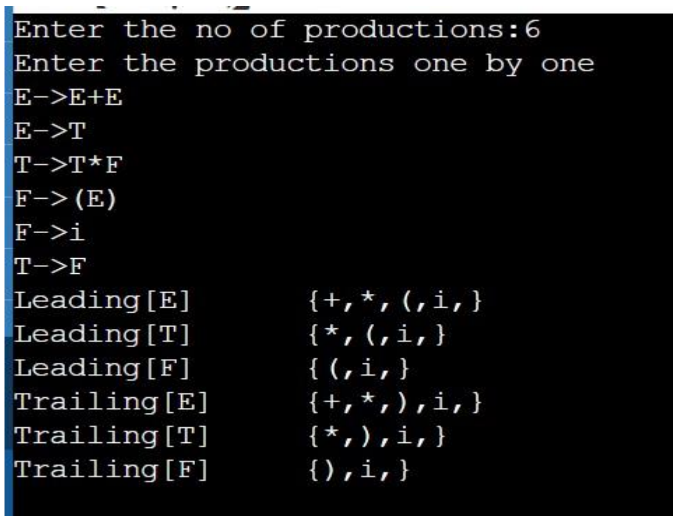
Code:

```
#include<stdio.h>

#include<string.h>

#include<conio.h> int

nt,t,top=0;

char s[50],NT[10],T[10],st[50],l[10][10],tr[50][50]; int

searchnt(char a)

{ int count=-1,i;

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

{

if(NT[i]==a)

return i; }

return count;

}
```

```
int searchter(char a) {
int count=-1,i;
for(i=0;i<t;i++) {
if(T[i]==a) return i; }
return count; } void
push(char a) {
s[top]=a; top++; } char
pop() { top--; return
s[top]; } void
installl(int a,int b) {
if(l[a][b]=='f') {
l[a][b]='t'; push(T[b]);
push(NT[a]);
} } void installt(int
a,int b) {
if(tr[a][b]=='f')
{
tr[a][b]='t'; push(T[b]);
push(NT[a]);
} } int main() { int i,s,k,j,n; char
pr[30][30],b,c; printf("Enter the no of
productions:"); scanf("%d",&n);
printf("Enter the productions one by one\n");
for(i=0;i<n;i++) scanf("%s",pr[i]); nt=0;
t=0; for(i=0;i<n;i++) {
if((searchnt(pr[i][0]))==-1)
NT[nt++]=pr[i][0]; }
for(i=0;i<n;i++) {
for(j=3;j<strlen(pr[i]);j++)
{ if(searchnt(pr[i][j])==-1)
{ if(searchter(pr[i][j])==-1)
T[t++]=pr[i][j];
}
}
}
for(i=0;i<nt;i++)
{
for(j=0;j<t;j++)
l[i][j]='f'; }
for(i=0;i<nt;i++)
{
for(j=0;j<t;j++)
tr[i][j]='f'; }
for(i=0;i<nt;i++)
{
for(j=0;j<n;j++)
{
if(NT[(searchnt(pr[j][0]))]==NT[i])
{ if(searchter(pr[j][3])!=-1)
installl(searchnt(pr[j][0]),searchter(pr[j][3]));
else { for(k=3;k<strlen(pr[j]);k++)
{ if(searchnt(pr[j][k])==-1)
{
installl(searchnt(pr[j][0]),searchter(pr[j][k]));
break; }
}
}
}
}
```

```
} while(top!=0)
{ b=pop();
c=pop();
for(s=0;s<n;s++)
{ if(pr[s][3]==b)
installl(searchnt(pr[s][0]),searchter(c));
} }
for(i=0;i<nt;i++)
{
printf("Leading[%c]\t{",NT[i]);
for(j=0;j<t;j++) { if(l[i][j]=='t')
printf("%c,",T[j]);
} printf("}\n"); }
top=0;
for(i=0;i<nt;i++)
{ for(j=0;j<n;j++)
{
if(NT[searchnt(pr[j][0])]==NT[i])
{ if(searchter(pr[j][strlen(pr[j])-1])!=-1)
installt(searchnt(pr[j][0]),searchter(pr[j][strlen(pr[j])-1]));
else
{
for(k=(strlen(pr[j])-1);k>=3;k--)
{ if(searchnt(pr[j][k])==-1)
{
installt(searchnt(pr[j][0]),searchter(pr[j][k]));
break; }
}
}
}
} } while(top!=0) { b=pop(); c=pop();
for(s=0;s<n;s++) { if(pr[s][3]==b)
installt(searchnt(pr[s][0]),searchter(c));
} } for(i=0;i<nt;i++) {
printf("Trailing[%c]\t{",NT[i]);
for(j=0;j<t;j++) {
if(tr[i][j]=='t')
printf("%c,",T[j]); }
printf("}\n");
}
getch(); return
0;
}
```

OUTPUT:



```
Enter the no of productions:6
Enter the productions one by one
E->E+E
E->T
T->T*F
F->(E)
F->i
T->F
Leading[E]      {+,*,(,i,}
Leading[T]      {*,(,i,}
Leading[F]      {(,i,}
Trailing[E]     {+,*,),i,}
Trailing[T]     {*,),i,}
Trailing[F]     {),i,}
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 9

Aim: **Write a program in c to implement operator precedence.**

Theory:

Operator precedence parser – An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. Ambiguous grammars are not allowed in any parser except operator precedence parser. There are two methods for determining what precedence relations should hold between a pair of terminals:

Use the conventional associativity and precedence of operator.

The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

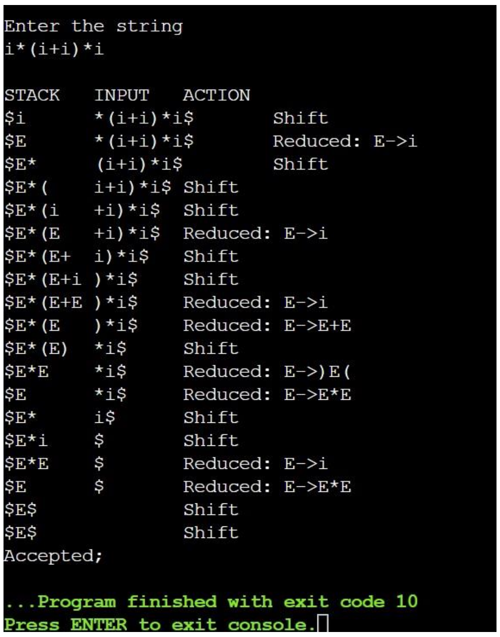
Code:

```
#include<stdio.h> #include<string.h> char *input; int i=0; char

lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"}; int

top=0,l; char prec[9][9]={

                    /*input*/

        /*stack    +   -   *   /   ^   i   (   )   $  */

        /*  + */  '>', '>','<','<','<','<','<','>','>',

        /*  - */  '>', '>','<','<','<','<','<','>','>',

        /*  * */  '>', '>','>','>','<','<','<','>','>',

        /*  / */  '>', '>','>','>','<','<','<','>','>',

        /*  ^ */  '>', '>','>','>','<','<','<','>','>',

        /*  i */  '>', '>','>','>','>','e','e','>','>',

        /*  ( */  '<', '<','<','<','<','<','<','>','e',

        /*  ) */  '>', '>','>','>','>','e','e','>','>',

        /*  $ */  '<', '<','<','<','<','<','<','<','>',

           };

int getindex(char c)

{

switch(c)
```

```
   {    case
'+':return 0;    case
'-':return 1;    case
'*':return 2;    case
'/':return 3;    case
'^':return 4;    case
'i':return 5;    case
'(':return 6;    case
')':return 7;    case
'$':return 8;
   }
} int shift() {
stack[++top]=*(input+i++);
stack[top+1]='\0'; } int reduce() {
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
   {
   len=strlen(handles[i]);
if(stack[top]==handles[i][0]&&top+1>=len)
      {
      found=1;
for(t=0;t<len;t++)
         {
         if(stack[top-t]!=handles[i][t])
            {
            found=0;
            break;
}
         }
      if(found==1)
         {          stack[top-t+1]='E';
top=top-t+1;
strcpy(lasthandle,handles[i]);
stack[top+1]='\0';          return
1;//successful reduction
         }
      }
  } return 0; }
void dispstack()
{
int j;
for(j=0;j<=top;j++)
printf("%c",stack[j]);
} void
dispinput() {
int j; for(j=i;j<l;j++)
printf("%c",*(input+j));
} void
main() {
int j;
input=(char*)malloc(50*sizeof(char)); printf("\nEnter
the string\n"); scanf("%s",input);
input=strcat(input,"$"); l=strlen(input);
strcpy(stack,"$");
printf("\nSTACK\tINPUT\tACTION"); while(i<=l)
         {
```

```
        shift();        printf("\n");   dispstack();
printf("\t");   dispinput();    printf("\tShift");
if(prec[getindex(stack[top])][getindex(input[i])]=='>')
                {
                while(reduce())
                        {
                        printf("\n");
dispstack();                    printf("\t");
        dispinput();
                        printf("\tReduced: E->%s",lasthandle);
                        }
                }
        }
if(strcmp(stack,"$E$")==0)
printf("\nAccepted;"); else
printf("\nNot Accepted;");
}
```

OUTPUT:



```
Enter the string
i*(i+i)*i

STACK   INPUT   ACTION
$i      *(i+i)*i$       Shift
$E      *(i+i)*i$       Reduced: E->i
$E*     (i+i)*i$        Shift
$E*(    i+i)*i$ Shift
$E*(i   +i)*i$  Shift
$E*(E   +i)*i$  Reduced: E->i
$E*(E+  i)*i$   Shift
$E*(E+i )*i$    Shift
$E*(E+E )*i$    Reduced: E->i
$E*(E   )*i$    Reduced: E->E+E
$E*(E)  *i$     Shift
$E*E    *i$     Reduced: E->)E(
$E      *i$     Reduced: E->E*E
$E*     i$      Shift
$E*i    $       Shift
$E*E    $       Reduced: E->i
$E      $       Reduced: E->E*E
$E$             Shift
$E$             Shift
Accepted;

...Program finished with exit code 10
Press ENTER to exit console.
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 10

Aim: Write a program in c to implement shift reduce parsing.

Theory:

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of the shift-reduce parser is the LR parser.

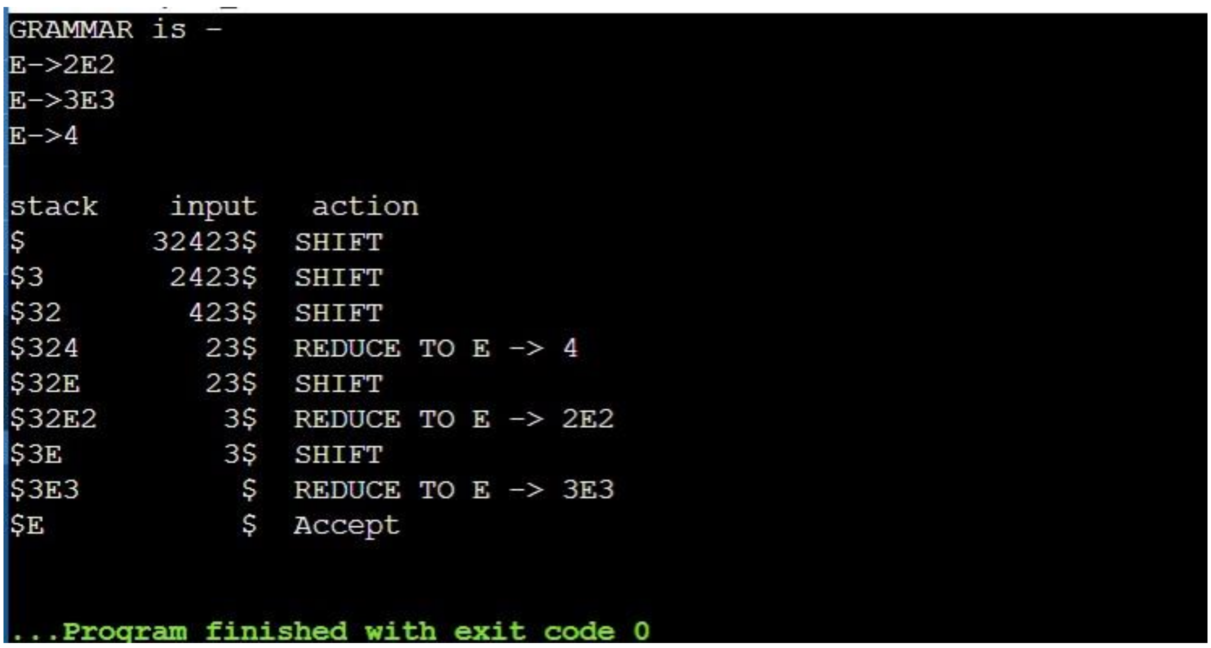
This parser requires some data structures i.e. An input buffer for storing the input string, stack for storing and accessing the production rules.

Code:

```
#include<stdio.h>

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

int z = 0, i = 0, j = 0, c = 0; char a[16],

ac[20], stk[15], act[10];  // Rules can be

E->2E2 , E->3E3 , E->4 void check()

{

        strcpy(ac,"REDUCE TO E -> ");

for(z = 0; z < c; z++)

        {

                if(stk[z] == '4')

                {

                        printf("%s4", ac);

        stk[z] = 'E';                   stk[z + 1] =

'\0';                   printf("\n$%s\t%s$\t", stk,

a);

                }

        }

                for(z = 0; z < c - 2; z++)

        {

                if(stk[z] == '2' && stk[z + 1] == 'E' &&

                                                    stk[z + 2] == '2')
                {
                        printf("%s2E2", ac);
        stk[z] = 'E';                   stk[z + 1] =
'\0';                   stk[z + 2] = '\0';
        printf("\n$%s\t%s$\t", stk, a);
                        i = i - 2;
                }
        }
        for(z=0; z<c-2; z++)
        {
                if(stk[z] == '3' && stk[z + 1] == 'E' && stk[z + 2] == '3')
                {
                        printf("%s3E3", ac);
                        stk[z]='E';
                        stk[z + 1]='\0';
                        stk[z + 1]='\0';
                        printf("\n$%s\t%s$\t", stk, a);
                        i = i - 2;
                }
        }
        return ;
} int
main()
{
        printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");
strcpy(a,"32423");      c=strlen(a);    strcpy(act,"SHIFT");
        printf("\nstack \t input \t
action");       printf("\n$\t%s$\t", a);
for(i = 0; j < c; i++, j++)
        {
         printf("%s", act);
```

```
                stk[i] = a[j];
        stk[i + 1] = '\0';
                a[j]=' ';
                printf("\n$%s\t%s$\t", stk, a);
                check();
        }
        check();
        if(stk[0] == 'E' && stk[1] == '\0')
                printf("Accept\n");
printf("Reject\n");
}
```

OUTPUT:



```
GRAMMAR is -
E->2E2
E->3E3
E->4

stack    input   action
$        32423$  SHIFT
$3        2423$  SHIFT
$32        423$  SHIFT
$324        23$  REDUCE TO E -> 4
$32E        23$  SHIFT
$32E2        3$  REDUCE TO E -> 2E2
$3E          3$  SHIFT
$3E3          $  REDUCE TO E -> 3E3
$E            $  Accept


...Program finished with exit code 0
```

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 11

Aim: Write a program in c to implement SLR parser.

Theory:

SLR is simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states. We can solve this problem by entering 'reduce' corresponding to FOLLOW of LHS of production in the terminating state. This is called SLR(1) collection of items

Steps for constructing the SLR parsing table :

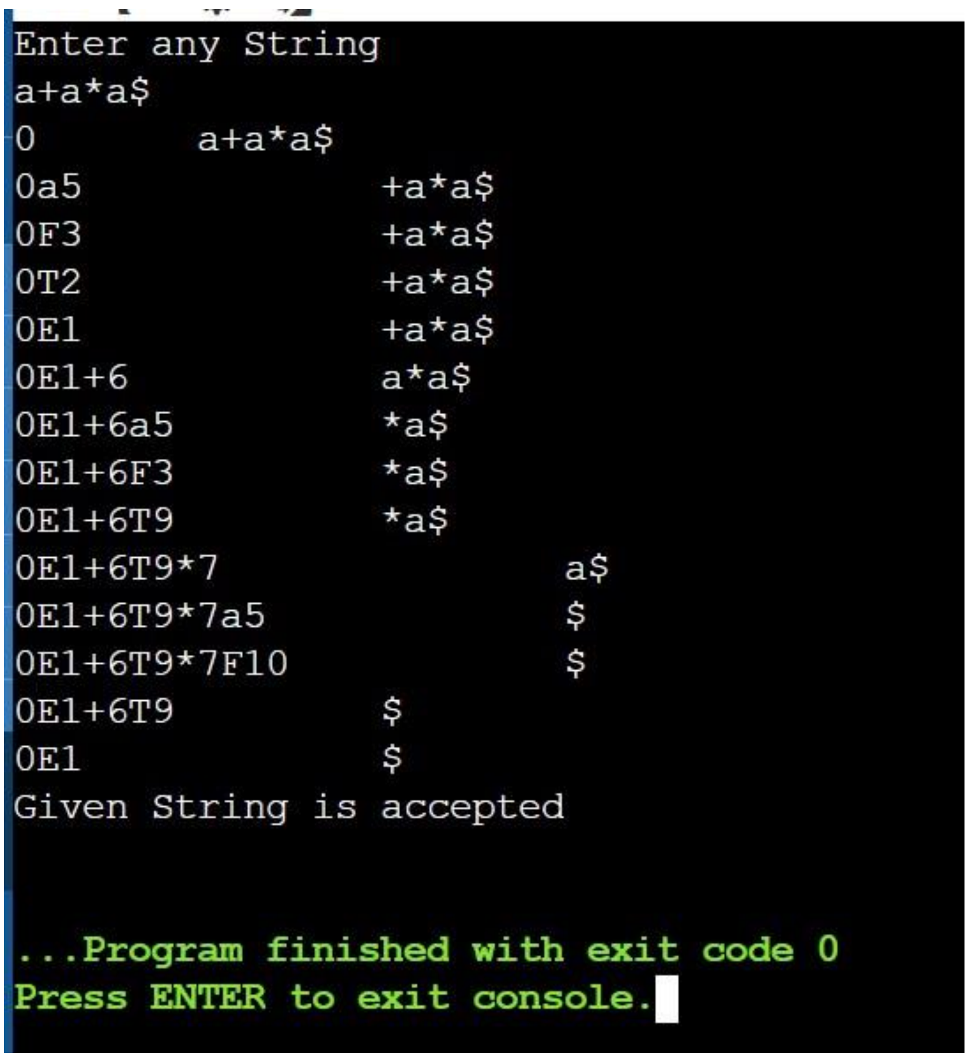
- Writing augmented grammar
- LR(0) collection of items to be found
- Find FOLLOW of LHS of production
- Defining 2 functions:goto[list of terminals] and action[list of non-terminals] in

the parsing table **Code:**

```
#include<stdio.h>

#include<string.h> int

axn[][6][2]={

    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

    {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},

    {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},

    {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},

    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

    {{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},

    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

    {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,1},{-1,-1}},

    {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},

    {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},

    {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}
```

```
        };//Axn  Table  int  gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-
1,-1,8,2,3,-1,-1,-1,     -1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1};  //GoTo table  int a[10];

char b[10]; int top=-
1,btop=-1,i; void
push(int k)
{   if(top<9)
a[++top]=k;
} void pushb(char
k)
{   if(btop<9)
b[++btop]=k;
} char TOS() {
return a[top];
} void pop()
{
if(top>=0)
    top--; } void
popb() {
if(btop>=0)
b[btop--]='\0';
} void display() {
for(i=0;i<=top;i++)
printf("%d%c",a[i],b[i]);
}
void display1(char p[],int m) //Displays The Present Input String {
  int l;   printf("\t\t");
for(l=m;p[l]!='\0';l++)
printf("%c",p[l]);
printf("\n"); } void
error() {
printf("Syntax Error");
} void reduce(int p) {
int len,k,ad;    char
src,*dest;    switch(p)
   {
 case 1:dest="E+T";
        src='E';
break;  case
2:dest="T";
        src='E';
break;
 case 3:dest="T*F";
        src='T';
break;  case
4:dest="F";
        src='T';
break;  case
5:dest="(E)";
        src='F';
break;  case
6:dest="i";

        src='F';
break;
default:dest="\0";
  src='\0';
break;
   }
   for(k=0;k<strlen(dest);k++)
   {
pop();
popb();
   }
pushb(src);
switch(src)
   {
 case 'E':ad=0;
break;  case
'T':ad=1;
break;  case
'F':ad=2;
break;  default:
ad=-1;    break;
   }
  push(gotot[TOS()][ad]);
} int main() {
   int j,st,ic;
   char ip[20]="\0",an;
printf("Enter any String\n");
scanf("%s",ip);

   push(0);
display();
printf("\t%s\n",ip);
for(j=0;ip[j]!='\0';)
   {
 st=TOS();
 an=ip[j];
if(an>='a'&&an<='z') ic=0;
else if(an=='+') ic=1;  else
if(an=='*') ic=2;  else
if(an=='(') ic=3;  else
if(an==')') ic=4;  else
if(an=='$') ic=5;  else {
error();     break;  }
if(axn[st][ic][0]==100)
   {       pushb(an);
push(axn[st][ic][1]);
display();       j++;
display1(ip,j);
   }
  if(axn[st][ic][0]==101)
   {
reduce(axn[st][ic][1]);
display();
display1(ip,j);
   }
  if(axn[st][ic][1]==102)
   {
  printf("Given String is accepted \n");
     break;
   }   }
return 0;
}
```

OUTPUT:



```
Enter any String
a+a*a$
0       a+a*a$
0a5             +a*a$
0F3             +a*a$
0T2             +a*a$
0E1             +a*a$
0E1+6           a*a$
0E1+6a5         *a$
0E1+6F3         *a$
0E1+6T9         *a$
0E1+6T9*7               a$
0E1+6T9*7a5             $
0E1+6T9*7F10            $
0E1+6T9         $
0E1             $
Given String is accepted


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

Programme	B.Tech CSE	Course Name	Compiler Construction
Course code	CSE304	Semester	6
Student name	Archita Singh	Enrollment Number	A2305221505

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

Experiment 12

Aim: Write a program in C to implement CLR.

Theory: CLR (Canonical LR) is a parsing technique used in compiler design to build LR(1) parsing tables. It extendsLR(0) and SLR(1) parsing, offering greater parsing power by considering more lookahead symbols. CLR parsing constructs a canonical collection of LR(1) items and generates action and goto tables from these items to guide parsing. It strikes a balance between parsing efficiency and grammar expressiveness, making it a popular choice incompiler construction.

Code:

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

#define MAX_STATES 100
#define MAX_SYMBOLS 100
#define MAX_PROD 100

int num_states, num_symbols, num_prod;
char symbols[MAX_SYMBOLS];
char productions[MAX_PROD][MAX_SYMBOLS];
int action[MAX_STATES][MAX_SYMBOLS];
int goto_table[MAX_STATES][MAX_SYMBOLS];

void initialize_tables()
{
  memset(action, -1, sizeof(action));
  memset(goto_table, -1, sizeof(goto_table));
}

int main()
{

  initialize_tables();

  num_states = 3;
  num_symbols = 2;
  num_prod = 3;
  strcpy(symbols, "ab");
  strcpy(productions[0], "S=aSb");
  strcpy(productions[1], "S=");
  strcpy(productions[2], "a=b");
  printf("Action Table:\n");
  for (int i = 0; i < num_states; i++)
  {
    for (int j = 0; j < num_symbols; j++)
    {
      printf("%d ", action[i][j]);
    }
    printf("\n");
  }

  printf("\nGoto Table:\n");
  for (int i = 0; i < num_states; i++)
  {
    for (int j = 0; j < num_symbols; j++)
    {
      printf("%d ", goto_table[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

Output:

```
-1  -1
 -1  -1
 -1  -1



 Goto Table:
 -1  -1
 -1  -1
 -1  -1
```

Result: The experiment was successfully performed in C.