EXPERIMENT - 1

ROLL NO:

Aim : To write a program to implement Lexical Analysis using C.

Description: Lexical analysis is the starting phase of the compiler. It gathers modified source code that is written in the form of sentences from the language preprocessor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code. To read the input character in the source code and produce a token is the most important task of a lexical analyzer.

The lexical analyzer goes through with the entire source code and identifies each token one by one. The scanner is responsible to produce tokens when it is requested by the parser. The lexical analyzer avoids the whitespace and comments while creating these tokens. If any error occurs, the analyzer correlates these errors with the source file and line number.

Algorithm:

Step 1 : Start the program

Step 2 : Include necessary header files.

Step 3: The ctype header file is to load the file with predicate isdigit.

Step 4 : The define directive defines the buffer size, numerics, assignment operator, relational operator.

Step 5 : Initialize the necessary variables.

Step 6: To return index of new string S, token t using insert() function.

Step 7 : Initialize the length of every string. Step 8: Check the necessary condition.

Step 9: Call the initialize() function. This function loads the keywords into the symbol table.

Step 10 : Check the conditions such as white spaces, digits, letters and alphanumerics.

Step 11 : To return index of entry for string S, or 0 if S is not found using lookup( ) function.

Step 12 : Check this until EOF is found.

Step 13 : Otherwise initialize the token value to be none.

Step 14 : In the main function if lookahead equals numeric then the value of attribute num is given by the global variable tokenval.

Step 15 : Check the necessary conditions such as arithmetic operators , parenthesis , identifiers, assignment operators and relational operators.

Step 16 : Stop the program.

Program:

```
#include<stdio.h>

#include<conio.h>
```

```
#include<ctype.h>

#include<string.h>

void main()

{

FILE *input, *output;

int t=0,j=0,i=0,flag=0,l=1;

char ch,str[20];

char keyword[30][30] = {"int","void","if","else","do","while"};

input=fopen("input.txt","r");

output=fopen("output.txt","w");

fprintf(output,"Line no. \t Token no. \t Token \t Lexeme\n\n");

while(!feof(input))

{

ch=fgetc(input);

if( ch=='+' || ch== '-' || ch=='*' || ch=='/'|| ch=='>' || ch=='<' || ch=='=' )

{

fprintf(output,"%7d\t\t %7d\t\t Operator\t %7c\n",l,t,ch);

t++;


}

else if( ch==';' || ch=='{' || ch=='}' || ch=='(' || ch==')' || ch=='?' || ch=='@' || ch=='!' || ch=='%' ||
ch=='.'||

ch=='"' || ch==',')

{

fprintf(output,"%7d\t\t %7d\t\t Special symbol\t %7c\n",l,t,ch);

t++;

}

else if(isdigit(ch))

{
```

```
fprintf(output,"%7d\t\t %7d\t\t Digit\t\t %7c\n",l,t,ch);
t++;
}
else if(isalpha(ch))
{
str[i]=ch;
i++;
ch=fgetc(input);
while(isalnum(ch) && ch!=' ')
{
str[i]=ch;
i++;
ch=fgetc(input);
}
str[i]='\0';
for(j=0;j<=30;j++)
{
if(strcmp(str,keyword[j])==0)
{
flag=1;
break;
}
}
if(flag==1)
{
fprintf(output,"%7d\t\t %7d\t\t Keyword\t %7s\n",l,t,str);
t++;
}
else
```

```
{
fprintf(output,"%7d\t\t %7d\t\t Identifier\t %7s\n",l,t,str); t++;
}
}
else if(ch=='\n')
{
l++;
}
}
fclose(input);
fclose(output);
getch();
}
```

Input.txt:

```
#include < stdio . h >
void main ( )
{
printf ( " Hello World " ) ;
}
```

Output.txt:

```
Line no.        Token no.       Token              Lexeme


    1               0           Identifier         include
    1               1           Operator              <
    1               2           Identifier          stdio
    1               3           Special symbol        .
```

```
1          4          Identifier        h
1          5          Operator          >
2          6          Keyword          void
2          7          Identifier        main
2          8          Special symbol     (
2          9          Special symbol     )
3          10         Special symbol     {
4          11         Identifier        printf
4          12         Special symbol      (
4          13         Special symbol      "
4          14         Identifier         Hello
4          15         Identifier        World
4          16         Special symbol      "
4          17         Special symbol      )
4          18         Special symbol      ;
5          19         Special symbol      }
```

Result: The Lexical Analyzer program is successfully executed.

Aim : To write a program to Eliminate Left Recursion from a given grammar.

Description: A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

$A \rightarrow A\,\alpha\,|\beta$.

The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$A \rightarrow \beta A'$

$A \rightarrow \alpha A'|\varepsilon$

Program:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

#define SIZE 20

void main()

{

char pro[SIZE], alpha[SIZE], beta[SIZE];

int nont_terminal,i,j, index=3;

printf("Enter the Production as E->E|A: ");

scanf("%s", pro);

nont_terminal=pro[0];

if(nont_terminal==pro[index]) //Checking if the Grammar is LEFT RECURSIVE

{

//Getting Alpha

for(i=++index,j=0;pro[i]!='|';i++,j++){

alpha[j]=pro[i];

//Checking if there is NO Vertical Bar (|)

if(pro[i+1]==0)
{
printf("This Grammar CAN'T BE REDUCED.\n");
exit(0); //Exit the Program
}
}
alpha[j]='\0'; //String Ending NULL Character
if(pro[++i]!=0) //Checking if there is Character after Vertical Bar (|)
{
//Getting Beta
for(j=i,i=0;pro[j]!='\0';i++,j++){
beta[i]=pro[j];
}
beta[i]='\0'; //String Ending NULL character
//Showing Output without LEFT RECURSION
printf("\nGrammar Without Left Recursion: \n\n");
printf(" %c->%s%c'\n", nont_terminal,beta,nont_terminal);
printf(" %c'->%s%c'|#\n", nont_terminal,alpha,nont_terminal);
}
else
printf("This Grammar CAN'T be REDUCED.\n");
}
else
printf("\n This Grammar is not LEFT RECURSIVE.\n");
getch();
}
```

Input & Output:

```
Enter the Production as E->E|A:

E->E+T|T

Grammar Without Left Recursion:

 E->TE'

 E'->+TE'|#
```

Result: The Left Recursion program is successfully executed.

EXPERIMENT - 3 ROLL NO:

Aim : To write a program to Eliminate Left Factoring from a given grammar.

Description: Left factoring is used to convert a left-factored grammar into an equivalent grammar to remove the uncertainty for the top-down parser. In left factoring, we separate the common prefixes from the production rule.

The following algorithm is used to perform left factoring in the grammar

Suppose the grammar is in the form:

A ⇒ αβ1 | αβ2 | αβ3 | ...... | αβn | γ

Where A is a non-terminal and α is the common prefix.

We will separate those productions with a common prefix and then add a new production rule in which the new non-terminal we introduced will derive those productions with a common prefix.

A ⇒ αA`

A` ⇒ β1 | β2 | β3 | ...... | βn

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];
int i,j=0,k=0,pos;
printf("Enter Production : A->");
gets(gram);
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';
for(i=0;i<strlen(part1)||i<strlen(part2);i++)
{
if(part1[i]==part2[i])
{
modifiedGram[k]=part1[i];
k++;
pos=i+1;
}
}
for(i=pos,j=0;part1[i]!='\0';i++,j++)
{
newGram[j]=part1[i];
}
newGram[j++]='|';
for(i=pos;part2[i]!='\0';i++,j++)
{
newGram[j]=part2[i];
}
modifiedGram[k]='X';
modifiedGram[++k]='\0';
newGram[j]='\0';
printf("\nGrammar Without Left Factoring : \n");
printf(" A->%s",modifiedGram);
printf("\n X->%s\n",newGram);
getch();
}
```

Input & Output:

```
Enter Production : A->bE+acF|bE+f
Grammar Without Left Factoring :
 A->bE+X
 X->acF|f
```

Result: The Left Factoring program is successfully executed.

Aim : To write a program to compute First and Follow sets for given grammar.

Description: First and Follow sets are used in the process of syntax analysis.

The First set of non-terminal symbols in grammar is the set of terminal symbols that can be derived as the first symbol of a string generated by the non-terminal. In other words, it is the set of all possible tokens that can be encountered at the beginning of a string derived from that non-terminal.

The Follow set of non-terminal symbols in grammar is the set of terminal symbols that can come immediately after a string derived from that non-terminal in any valid sentence of the grammar. In other words, it is the set of all possible tokens that can appear immediately after the non-terminal in a valid sentence.

Program:

```
#include <ctype.h>

#include <stdio.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);

#define MAX_COUNT 10

int count, n = 0;

// Stores the final result of the First Sets

char calc_first[MAX_COUNT][100];

// Stores the final result of the Follow Sets

char calc_follow[MAX_COUNT][100];

int m = 0;

// Stores the production rules

char production[MAX_COUNT][10];

char f[10], first[10];

int k;

char ck;
```

```
int e;
int main()
{
int jm = 0;
int km = 0;
int i;
char c, ch;
count = 8; // You can change this if needed
// The Input grammar
strcpy(production[0], "X=TnS");
strcpy(production[1], "X=Rm");
strcpy(production[2], "T=q");
strcpy(production[3], "T=#");
strcpy(production[4], "S=p");
strcpy(production[5], "S=#");
strcpy(production[6], "R=om");
strcpy(production[7], "R=ST");
char done[MAX_COUNT];
int ptr = -1;
// Initializing the calc_first array
for (k = 0; k < count; k++) {
for (int 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;
// Checking if First of c has
// already been calculated
for (int kay = 0; kay <= ptr; kay++)
if (c == done[kay])
xxx = 1;
if (xxx == 1)
continue;
// Function call
findfirst(c, 0, 0);
ptr += 1;
// Adding c to the calculated list
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;
// Printing the First Sets of the grammar
for (i = 0 + jm; i < n; i++) {
int lark = 0, 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");
printf("-----------------------------------------------" "\n\n");
char donee[MAX_COUNT];
ptr = -1;
// Initializing the calc_follow array
for (k = 0; k < count; k++) {
for (int kay = 0; kay < 100; kay++) {
calc_follow[k][kay] = '!';
}
}
point1 = 0;
int land = 0;
for (e = 0; e < count; e++) {
ck = production[e][0];
point2 = 0;
xxx = 0;
// Checking if Follow of ck
// has already been calculated
for (int kay = 0; kay <= ptr; kay++)
if (ck == donee[kay])
xxx = 1;
if (xxx == 1)
continue;
land += 1;
// Function call
```

```
follow(ck);
ptr += 1;
// Adding ck to the calculated list
donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;
// Printing the Follow Sets of the grammar
for (i = 0 + km; i < m; i++) {
int lark = 0, 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++;
}
return 0;
}
void follow(char c)
{
int i, j;
```

```
// Adding "$" to the follow
// set of the start symbol
if (production[0][0] == c) {
f[m++] = '$';
}
for (i = 0; i < 10; i++) {
for (j = 2; j < 10; j++) {
if (production[i][j] == c) {
if (production[i][j + 1] != '\0') {
// Calculate the first of the next
// Non-Terminal in the production
followfirst(production[i][j + 1], i,
(j + 2));
}
if (production[i][j + 1] == '\0'
&& c != production[i][0]) {
// Calculate the follow of the
// Non-Terminal in the L.H.S. of the
// production
follow(production[i][0]);
}
}
}
}
}
void findfirst(char c, int q1, int q2)
{
int j;
// The case where we
// encounter a Terminal
if (!(isupper(c))) {
first[n++] = c;
}
for (j = 0; j < count; j++) {
if (production[j][0] == c) {
if (production[j][2] == '#') {
if (production[q1][q2] == '\0')
first[n++] = '#';
else if (production[q1][q2] != '\0'
&& (q1 != 0 || q2 != 0)) {
// Recursion to calculate First of New
// Non-Terminal we encounter after
// epsilon
findfirst(production[q1][q2], q1,(q2 + 1));
}
else
first[n++] = '#';
}
else if (!isupper(production[j][2])) {
first[n++] = production[j][2];
}
else {
// Recursion to calculate First of
// New Non-Terminal we encounter
// at the beginning
findfirst(production[j][2], j, 3);
}
}
```

```
}
}
void followfirst(char c, int c1, int c2)
{
int k;
// The case where we encounter
// a Terminal
if (!(isupper(c)))
f[m++] = c;
else {
int i = 0, j = 1;
for (i = 0; i < count; i++) {
if (calc_first[i][0] == c)
break;
}
// Including the First set of the
// Non-Terminal in the Follow of
// the original query
while (calc_first[i][j] != '!') {
if (calc_first[i][j] != '#') {
f[m++] = calc_first[i][j];
}
else {
if (production[c1][c2] == '\0') {
// Case where we reach the
// end of a production
follow(production[c1][0]);
}
else {
```

```
// Recursion to the next symbol
// in case we encounter a "#"
followfirst(production[c1][c2], c1,
c2 + 1);
}
}
j++;
}
}
}
```

Output:

```
First(X) = { q, n, o, p, #, }
First(T) = { q, #, }
First(S) = { p, #, }
First(R) = { o, p, q, #, }
-----------------------------------------------
Follow(X) = { $, }
Follow(T) = { n, m, }
Follow(S) = { $, q, m, }
Follow(R) = { m, }
```

Result: The First and Follow program is successfully executed.

Aim : To write a program to Construct Predictive Parsing Table.

Description: It is a Non-Recursive Descent which is also known as LL(1) Parser.
LL(1) Parsing: Here the 1st L represents that the scanning of the Input will be done from the Left to Right manner and the second L shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.
Essential conditions to check first are as follows:

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

These conditions are necessary but not sufficient for proving a LL(1) parser.

Program:

```
#include <stdio.h>

#include<conio.h>

#include <string.h>

char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };

char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };

char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@" };

char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };

char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };

char table[5][6][10];

int numr(char c)

{

switch (c)

{

case 'S':

return 0;

case 'A':

return 1;

case 'B':
```

```
return 2;
case 'C':
return 3;
case 'a':
return 0;
case 'b':
return 1;
case 'c':
return 2;
case 'd':
return 3;
case '$':
return 4;
}
return (2);
}
void main()
{
int i, j, k;
for (i = 0; i < 5; i++)
for (j = 0; j < 6; j++)
strcpy(table[i][j], " ");
printf("The following grammar is used for Parsing Table:\n");
for (i = 0; i < 7; i++)
printf("%s\n", prod[i]);
printf("\nPredictive parsing table:\n");
fflush(stdin);
for (i = 0; i < 7; i++)
{
```

```
k = strlen(first[i]);
for (j = 0; j < 10; j++)
if (first[i][j] != '@')
strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);
}
for (i = 0; i < 7; i++)
{
if (strlen(pror[i]) == 1)
{
if (pror[i][0] == '@')
{

k = strlen(follow[i]);
for (j = 0; j < k; j++)
strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);
}
}
}
strcpy(table[0][0], " ");
strcpy(table[0][1], "a");
strcpy(table[0][2], "b");
strcpy(table[0][3], "c");
strcpy(table[0][4], "d");
strcpy(table[0][5], "$");
strcpy(table[1][0], "S");
strcpy(table[2][0], "A");
strcpy(table[3][0], "B");
strcpy(table[4][0], "C");
printf("\n--------------------------------------------------------\n");
```

```
for (i = 0; i < 5; i++)
for (j = 0; j < 6; j++)
{
printf("%-10s", table[i][j]);
if (j == 5)
printf("\n--------------------------------------------------------\n");
}
getch();
}
```

Output:

```
The following grammar is used for Parsing Table:
S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Predictive parsing table:

--------------------------------------------------------
a b c d $
--------------------------------------------------------
S S->A S->A S->A S->A
--------------------------------------------------------
A A->Bb A->Bb A->Cd A->Cd
--------------------------------------------------------
```

```
B B->aB B->@ B->@ B->@
--------------------------------------------------------
C C->@ C->@ C->@
--------------------------------------------------------
```

Result: The Construction of Predictive Parsing Table program is successfully executed.

Aim : To write a program to Implement Shift Reduce Parsing.

Description: 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.
- A stack for storing and accessing the production rules.

Program:

```
#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

#include<string.h>

char ip_sym[15],stack[15];

int ip_ptr=0,st_ptr=0,len,i;

char temp[2],temp2[2];

char act[15];

void check();

void main()

{

clrscr();

printf("\n\t\t SHIFT REDUCE PARSER\n");

printf("\n GRAMMER\n");

printf("\n E->E+E\n E->E/E");

printf("\n E->E*E\n E->a/b");

printf("\n enter the input symbol:\t");

gets(ip_sym);

printf("\n\t stack implementation table");

printf("\n stack \t\t input symbol\t\t action");

printf("\n________\t\t____________\t\t____________\n");

printf("\n $\t\t%s$\t\t\t--",ip_sym);
```

```
strcpy(act,"shift");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
len=strlen(ip_sym);
for(i=0;i<=len-1;i++)
{
stack[st_ptr]=ip_sym[ip_ptr];
stack[st_ptr+1]='\0';
ip_sym[ip_ptr]=' ';
ip_ptr++;
printf("\n $%s\t\t%s$\t\t\t%s",stack,ip_sym,act);
strcpy(act,"shift");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
check();
st_ptr++;
}
st_ptr++;
check();
}
void check()
{
int flag=0;
temp2[0]=stack[st_ptr];
temp2[1]='\0';
if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))
{
```

```
stack[st_ptr]='E';
if(!strcmpi(temp2,"a"))
printf("\n $%s\t\t%s$\t\t\tE->a",stack,ip_sym);
else
printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip_sym);
flag=1;
}
if((!strcmpi(temp2,"+"))||(strcmpi(temp2,"*"))||(!strcmpi(temp2,"/")))
{
flag=1;
}
if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E\E"))||(!strcmpi(stack,"E*E")))
{
strcpy(stack,"E");
st_ptr=0;
if(!strcmpi(stack,"E+E"))
printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
else if(!strcmpi(stack,"E\E"))
printf("\n $%s\t\t%s$\t\t\tE->E\E",stack,ip_sym);
else if(!strcmpi(stack,"E*E"))
printf("\n $%s\t\t%s$\t\t\tE->E*E",stack,ip_sym);
else
printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
flag=1;
}
if(!strcmpi(stack,"E")&&ip_ptr==len)
{
printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip_sym);
getch();
```

```
exit(0);
}
if(flag==0)
{
printf("\n%s\t\t\t%s\t\t reject",stack,ip_sym);
exit(0);
}
return;
}
```

Output:

```
SHIFT REDUCE PARSER
GRAMMER
E->E+E
E->E/E
E->E*E
E->a/b
Enter the input symbol: a+b

Stack Implementation Table
Stack Input Symbol Action
------- ----------------- ---------
$ a+b$ --
$a +b$ shift a
$E +b$ E->a
$E+ b$ shift +
$E+b $ shift b
$E+E $ E->b
$E $ E->E+E
```

```
$E $ ACCEPT
```

Result: The Shift Reduce parser program is successfully executed.

Aim : To write a program to Compute LR(0) items.

Description: The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar. This technique is also called LR(0) parsing.
L stands for the left to right scanning
R stands for rightmost derivation in reverse
0 stands for no. of input symbols of lookahead.
Augmented grammar :
If G is a grammar with starting symbol S, then G' (augmented grammar for G) is a grammar with a new starting symbol S' and productions S'-> .S . The purpose of this new starting production is to indicate to the parser when it should stop parsing. The ' . ' before S indicates the left side of ' . ' has been read by a compiler and the right side of ' . ' is yet to be read by a compiler.

Program:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

int i,j,k,m,n=0,o,p,ns=0,tn=0,rr=0,ch=0;

char read[15][10],gl[15],gr[15][10],temp,templ[15],tempr[15][10],*ptr,temp2[5],dfa[15][15];

struct states

{

char lhs[15],rhs[15][10];

int n;

}I[15];

int compstruct(struct states s1,struct states s2)

{

int t;

if(s1.n!=s2.n)

return 0;

if( strcmp(s1.lhs,s2.lhs)!=0 )

return 0;

for(t=0;t<s1.n;t++)
```

```
if( strcmp(s1.rhs[t],s2.rhs[t])!=0 )
return 0;
return 1;
}
void moreprod()
{
int r,s,t,l1=0,rr1=0;
char *ptr1,read1[15][10];
for(r=0;r<I[ns].n;r++)
{
ptr1=strchr(I[ns].rhs[l1],'.');
t=ptr1-I[ns].rhs[l1];
if( t+1==strlen(I[ns].rhs[l1]) )
{
l1++;
continue;
}
temp=I[ns].rhs[l1][t+1];
l1++;
for(s=0;s<rr1;s++)
if( temp==read1[s][0] )
break;
if(s==rr1)
{
read1[rr1][0]=temp;
rr1++;
}
```

```
else
continue;
for(s=0;s<n;s++)
{
if(gl[s]==temp)
{
I[ns].rhs[I[ns].n][0]='.';
I[ns].rhs[I[ns].n][1]=NULL;
strcat(I[ns].rhs[I[ns].n],gr[s]);
I[ns].lhs[I[ns].n]=gl[s];
I[ns].lhs[I[ns].n+1]=NULL;
I[ns].n++;
}
}
}
}
void canonical(int l)
{
int t1;
char read1[15][10],rr1=0,*ptr1;
for(i=0;i<I[l].n;i++)
{
temp2[0]='.';
ptr1=strchr(I[l].rhs[i],'.');
t1=ptr1-I[l].rhs[i];
if( t1+1==strlen(I[l].rhs[i]) )
continue;
temp2[1]=I[l].rhs[i][t1+1];
temp2[2]=NULL;
```

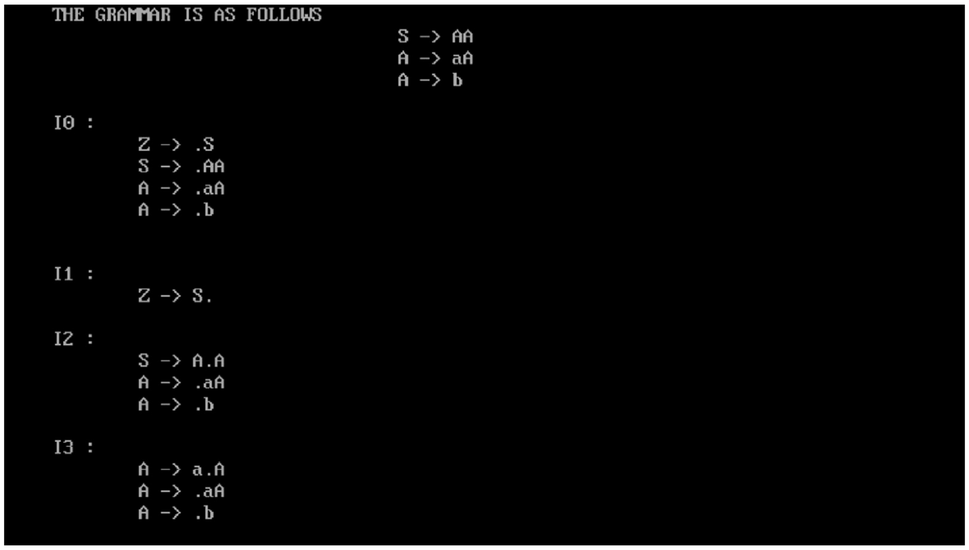
```
for(j=0;j<rr1;j++)
if( strcmp(temp2,read1[j])==0 )
break;
if(j==rr1)
{
strcpy(read1[rr1],temp2);
read1[rr1][2]=NULL;
rr1++;
}
else
continue;
for(j=0;j<I[0].n;j++)
{
ptr=strstr(I[l].rhs[j],temp2);
if( ptr )
{
templ[tn]=I[l].lhs[j];
templ[tn+1]=NULL;
strcpy(tempr[tn],I[l].rhs[j]);
tn++;
}
}
for(j=0;j<tn;j++)
{
ptr=strchr(tempr[j],'.');
p=ptr-tempr[j];
tempr[j][p]=tempr[j][p+1];
tempr[j][p+1]='.';
I[ns].lhs[I[ns].n]=templ[j];
I[ns].lhs[I[ns].n+1]=NULL;
strcpy(I[ns].rhs[I[ns].n],tempr[j]);
I[ns].n++;
}
moreprod();
for(j=0;j<ns;j++)
{
//if ( memcmp(&I[ns],&I[j],sizeof(struct states))==1 )
if( compstruct(I[ns],I[j])==1 )
{
I[ns].lhs[0]=NULL;
for(k=0;k<I[ns].n;k++)
I[ns].rhs[k][0]=NULL;
I[ns].n=0;
dfa[l][j]=temp2[1];
break;
}
}
if(j<ns)
{
tn=0;
for(j=0;j<15;j++)
{
templ[j]=NULL;
tempr[j][0]=NULL;
}
continue;
}
```

```
dfa[l][j]=temp2[1];
printf("\n\nI%d :",ns);
for(j=0;j<I[ns].n;j++)
printf("\n\t%c -> %s",I[ns].lhs[j],I[ns].rhs[j]);
getch();
ns++;
tn=0;
for(j=0;j<15;j++)
{
templ[j]=NULL;
tempr[j][0]=NULL;
}
}
}
void main()
{
FILE *f;
int l;
clrscr();
for(i=0;i<15;i++)
{
I[i].n=0;
I[i].lhs[0]=NULL;
I[i].rhs[0][0]=NULL;
dfa[i][0]=NULL;
}
f=fopen("tab6.txt","r");
while(!feof(f))
{
```

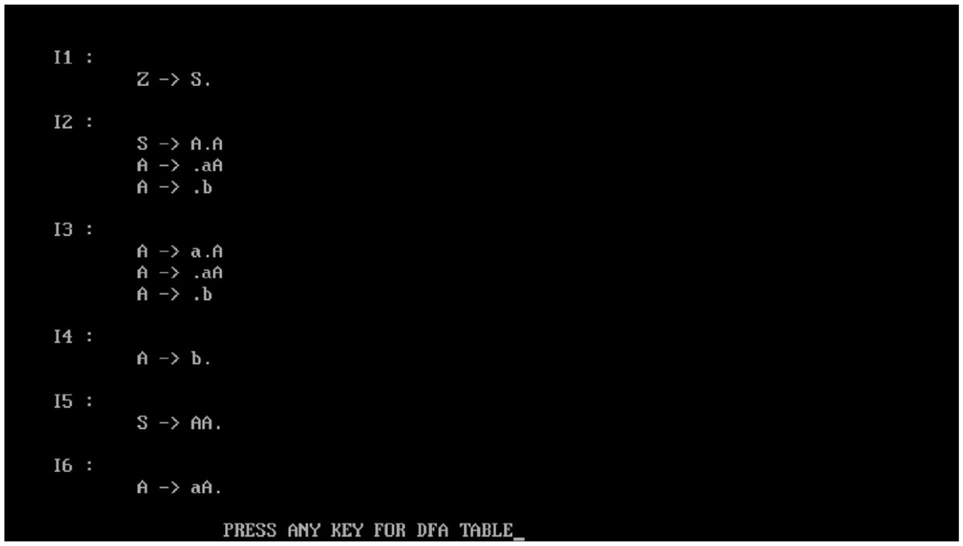
```
fscanf(f,"%c",&gl[n]);
fscanf(f,"%s\n",gr[n]);
n++;
}
printf("THE GRAMMAR IS AS FOLLOWS\n");
for(i=0;i<n;i++)
printf("\t\t\t\t%c -> %s\n",gl[i],gr[i]);
I[0].lhs[0]='Z';
strcpy(I[0].rhs[0],".S");
I[0].n++;
l=0;
for(i=0;i<n;i++)
{
temp=I[0].rhs[l][1];
l++;
for(j=0;j<rr;j++)
if( temp==read[j][0] )
break;
if(j==rr)
{
read[rr][0]=temp;
rr++;
}
else
continue;
for(j=0;j<n;j++)
{
if(gl[j]==temp)
{
```

```
I[0].rhs[I[0].n][0]='.';
strcat(I[0].rhs[I[0].n],gr[j]);
I[0].lhs[I[0].n]=gl[j];
I[0].n++;
}
}
}
ns++;
printf("\nI%d :\n",ns-1);
for(i=0;i<I[0].n;i++)
printf("\t%c -> %s\n",I[0].lhs[i],I[0].rhs[i]);
for(l=0;l<ns;l++)
canonical(l);
printf("\n\n\t\tPRESS ANY KEY FOR DFA TABLE");
getch();
clrscr();
printf("\t\t\tDFA TABLE IS AS FOLLOWS\n\n\n");
for(i=0;i<ns;i++)
{
printf("I%d : ",i);
for(j=0;j<ns;j++)
if(dfa[i][j]!='\0')
printf("'%c'->I%d | ",dfa[i][j],j);
printf("\n\n\n");
}
printf("\n\n\n\t\tPRESS ANY KEY TO EXIT");
getch();
}
```

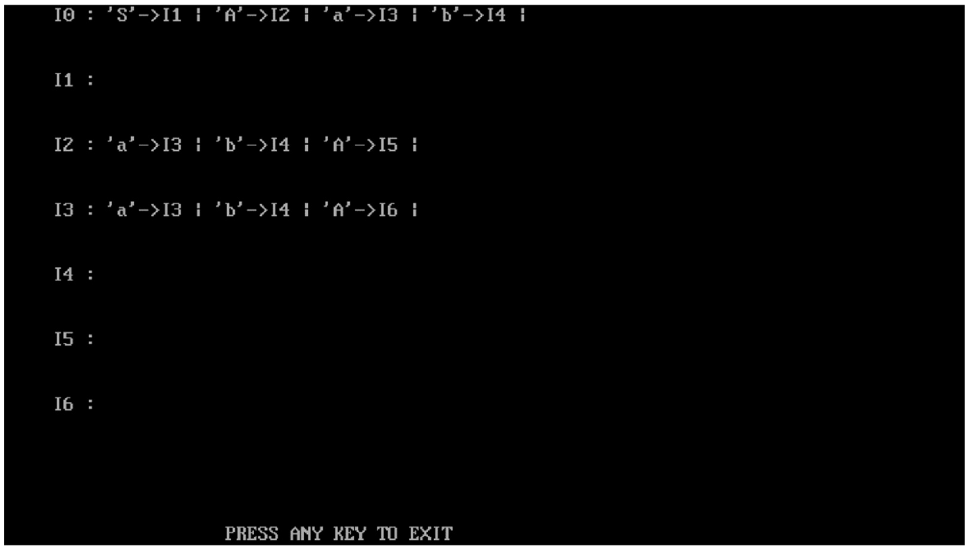
Input & Output:



```
    THE GRAMMAR IS AS FOLLOWS
                                S -> AA
                                A -> aA
                                A -> b

    I0 :
            Z -> .S
            S -> .AA
            A -> .aA
            A -> .b


    I1 :
            Z -> S.

    I2 :
            S -> A.A
            A -> .aA
            A -> .b

    I3 :
            A -> a.A
            A -> .aA
            A -> .b
```



```
    I1 :
            Z -> S.

    I2 :
            S -> A.A
            A -> .aA
            A -> .b

    I3 :
            A -> a.A
            A -> .aA
            A -> .b

    I4 :
            A -> b.

    I5 :
            S -> AA.

    I6 :
            A -> aA.

                    PRESS ANY KEY FOR DFA TABLE_
```



```
I0 : 'S'->I1 ¦ 'A'->I2 ¦ 'a'->I3 ¦ 'b'->I4 ¦

I1 :

I2 : 'a'->I3 ¦ 'b'->I4 ¦ 'A'->I5 ¦

I3 : 'a'->I3 ¦ 'b'->I4 ¦ 'A'->I6 ¦

I4 :

I5 :

I6 :



                PRESS ANY KEY TO EXIT
```

Result: The LR(0) items program is successfully executed.

Aim : To write a program to Generate Intermediate Code.

Description: Intermediate code can translate the source program into the machine program. Intermediate code is generated because the compiler can't generate machine code directly in one pass. Therefore, first, it converts the source program into intermediate code, which performs efficient generation of machine code further. The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph, three address codes, Quadruples, and triples.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char op[2],arg1[5],arg2[5],result[5];
void main()
{
FILE *fp1,*fp2;
fp1=fopen("input.txt","r");
fp2=fopen("output.txt","w");
while(!feof(fp1))
{
fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);
if(strcmp(op,"+")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nADD R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"*")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nMUL R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"-")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nSUB R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"/")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nDIV R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"=")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nMOV %s,R0",result);
}
}
fclose(fp1);
fclose(fp2);
getch();
}
```

Input.txt:

```
+ a b t1
* c d t2
- t1 t2 t
```

```
= t ? x
```

Output.txt:

```
MOV R0,a
ADD R0,b
MOV t1,R0
MOV R0,c
MUL R0,d
MOV t2,R0
MOV R0,t1
SUB R0,t2
MOV t,R0
MOV R0,t
MOV x,R0
```

Result: The Intermediate Code program is successfully executed.