BANARAS HINDU UNIVERSITY

M.SC SEMESTER III EXAMINATION 2017

COMPILER DESIGN
PRACTICAL ASSIGNMENT FILE

NAME: - ASHISH GUPTA
ROLL NO: - 16419CMP008
ENROLMENT: - 390222

INDEX

LEXICAL ANALYSER USING LEX TOOL/C.

PROBLEM DEFINITION

Design a Lexical Analyser using C Lex Tool or C language

SOURCE CODE

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

#define L_T 1
#define L_E 2
#define G_T 3
#define G_E 4
#define NOT 5
#define NOT_E 6
#define Assign 7
#define EQUAL 8
#define XOR 9
#define MOD 10
#define PLUS 11
#define MINUS 12
#define STAR 13
#define DIVIDE 14
#define B_AND 15
#define B_OR 16
#define ERROR -1
#define L_PAREN 17
#define R_PAREN 18
#define L_BRAC 19
#define R_BRAC 20
#define L_BIG_BRAC 21
#define R_BIG_BRAC 22
#define SEMI_COLON 23
#define COMMA 24
```

```
#define CHAR_SET_SIZE 128
#define KEY_WORDS 26

int first_op[CHAR_SET_SIZE], second_op[CHAR_SET_SIZE];

int delimiter[CHAR_SET_SIZE];

char lex[200];

char *key_words[KEY_WORDS] = {
        "void", "int", "double", "char", "long", "float", "switch", "case", "short",
        "if", "else", "for", "while", "break", "return", "continue", "default",
"static",
        "sizeof", "struct", "union", "default", "signed", "unsigned", "const",
"do"};

void memory_set_call(void);
void my_gets(char *, int);
void lexical_analysis(char *);
int iskeyword(char *);

int main(int argc, char **argv[])
{
        memory_set_call();
        char file_name[100];
        printf("\nEnter file name : "), my_gets(file_name, 100);
        printf("\n\n");
        lexical_analysis(file_name);
        return 0;
}

void lexical_analysis(char *file_name)
{
        FILE *fp1 = NULL;
        fp1 = fopen(file_name, "r+");
        if(fp1 == NULL)
        {
                fprintf(stderr, "\n\nUnable to open file : %s\n\n", file_name);
                return;
        }
```

```
	int i, c1, c2, line_number = 1, d_flag = 0, e_flag;
	int k_words = 0, identifiers = 0, num = 0, ch = 0, str = 0, del = 0, op = 0;
	while(!feof(fp1))
	{
		c1 = fgetc(fp1);
		if(c1 == EOF)
			break;
		if(isdigit(c1)) // for numeric constants
		{
			i = 0, e_flag = 0;
			while(1)
			{
				lex[i++] = c1;
				c1 = fgetc(fp1);
				if(isdigit(c1))
					continue;
				else if(delimiter[c1] != ERROR || c1 == ' ' || c1 == '\t'
|| c1 == '\n')
				{
					fseek(fp1, -1, SEEK_CUR);
					lex[i] = '\0';
					break;
				}
				else
				{
					if(c1 == '.' && d_flag == 0)
					{
						d_flag = 1;
						continue;
					}
					else
					{
						e_flag = 1;
						break;
					}
				}
			}
			if(!e_flag)
			{
```

```
                        printf("Token : %s\tLexeme : %s\n", "Numeric
Constant", lex);
                        num++;
                }
                else
                {
                        fprintf(stderr, "\nError found at line %d\n",
line_number);
                        break;
                }
        }
        else if(isalpha(c1)) // for identifiers
        {
                i = 0;
                while(1)
                {
                        lex[i++] = c1;
                        c1 = fgetc(fp1);
                        if(isalnum(c1))
                                continue;
                        else
                        {
                                fseek(fp1, -1, SEEK_CUR);
                                lex[i] = '\0';
                                break;
                        }
                }
                if(iskeyword(lex))
                {
                        k_words++;
                        printf("Token : %s\tLexeme : %s\n", "Key Word", lex);
                }
                else
                {
                        identifiers++;
                        printf("Token : %s\tLexeme : %s\n", "Identifier", lex);
                }
        }
        else if(c1 == '\'') // character constant
        {
```

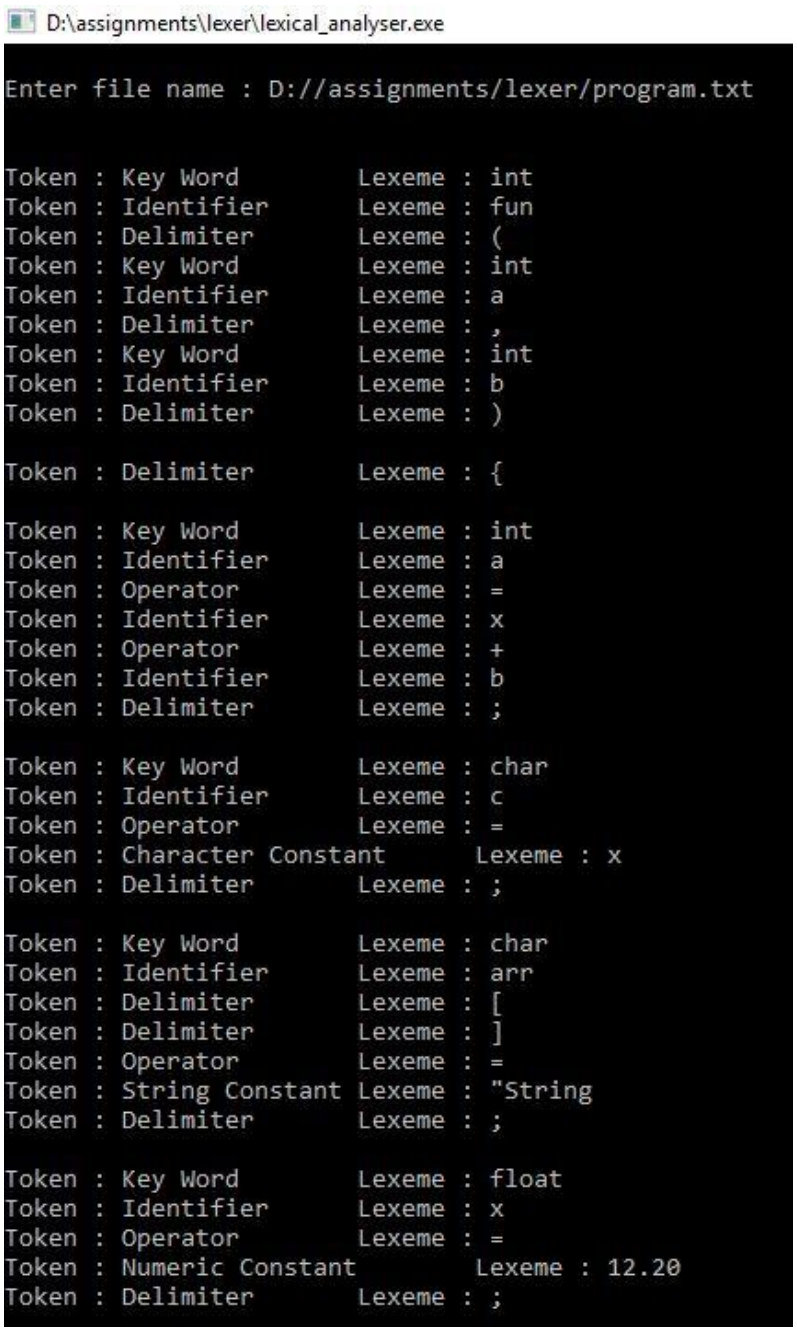
```
                        c2 = fgetc(fp1);
                        c1 = fgetc(fp1);
                        if(c1 == '\'')
                        {
                                printf("Token : %s\tLexeme : %c\n", "Character
Constant", c2);
                                ch++;
                        }
                        else
                        {
                                fprintf(stderr, "\nError found at line %d\n",
line_number);
                                break;
                        }
                }
                else if(c1 == '\"') // string constant
                {
                        i = 0;
                        while(1)
                        {
                                lex[i++] = c1;
                                c1 = fgetc(fp1);
                                if(c1 == '\"')
                                {
                                        lex[i] = '\0';
                                        break;
                                }
                        }
                        str++;
                        printf("Token : %s\tLexeme : %s\n", "String Constant", lex);
                }
                else
                {
                        if(first_op[c1] != ERROR) //for operators
                        {
                                c2 = fgetc(fp1);
                                if(second_op[c2] == ERROR)
                                {
                                        fseek(fp1, -1, SEEK_CUR); // SEEK_END(Last
point), SEEK_CUR (Current position), SEEK_SET (Starting point)
```

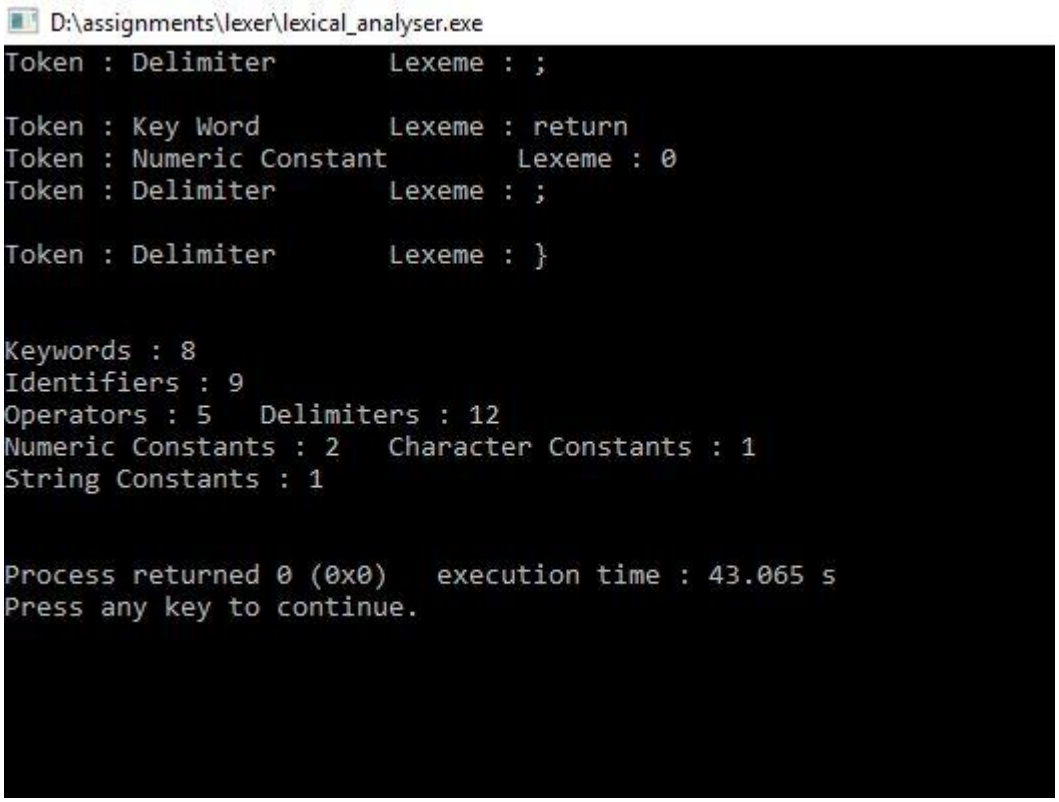
```
                                lex[0] = c1, lex [1] = '\0';
                                printf("Token : %s\tLexeme : %s\n",
"Operator", lex);
                        }
                        else
                        {
                                lex[0] = c1, lex[1] = c2, lex[2] = '\0';
                                printf("Token : %s\tLexeme : %s\n",
"Operator", lex);
                        }
                        op++;
                }
                else if(delimiter[c1] != ERROR) //for delimiters
                {
                        lex[0] = c1, lex[1] ='\0';
                        printf("Token : %s\tLexeme : %s\n", "Delimiter", lex);
                        del++;
                }
                else //for errors
                {
                        if(c1 == ' ' || c1 == '\n' || c1 == '\t') //not an error
                        {
                                if(c1 == '\n')
                                {
                                        line_number++;
                                        printf("\n");
                                }
                                continue;
                        }
                        else //error
                        {
                                fprintf(stderr, "\nError found at line %d\n",
line_number);
                                break;
                        }
                }
        }
    }
    printf("\nKeywords : %d\nIdentifiers : %d\n", k_words, identifiers);
    printf("Operators : %d\tDelimiters : %d\n", op, del);
```

```
	printf("Numeric Constants : %d\tCharacter Constants : %d\n", num, ch);
	printf("String Constants : %d\n\n", str);
	fclose(fp1), fp1 = NULL;
}

int iskeyword(char *arr)
{
	int i, f = 0;
	for(i = 0; i < KEY_WORDS; i++)
	{
		if(!strcmp(key_words[i], arr))
		{
			f = 1;
			break;
		}
	}
	return f;
}
void my_gets(char *arr, int size)
{
	int i = 0, c;
	while((c = getchar()) != '\n')
	{
		if(i > size - 2)
			break;
		arr[i++] = c;
	}
	arr[i] = '\0';
	return;
}
void memory_set_call(void)
{
	memset(first_op, -1, CHAR_SET_SIZE * sizeof(int));
	memset(second_op, -1, CHAR_SET_SIZE * sizeof(int));
	memset(delimiter, -1, CHAR_SET_SIZE * sizeof(int));
	first_op['<'] = L_T, first_op['>'] = G_T, first_op['!'] = NOT, first_op['='] =
Assign,
	first_op['%'] = MOD, first_op['+'] = PLUS, first_op['-'] = MINUS,
first_op['*'] = STAR,
	first_op['/'] = DIVIDE, first_op['&'] = B_AND, first_op['|'] = B_OR;
```

```
        delimiter['{'] = L_PAREN, delimiter['}'] = R_PAREN, delimiter['('] =
L_BRAC, delimiter[')'] = R_BRAC,
        delimiter['['] = L_BIG_BRAC, delimiter[']'] = R_BIG_BRAC, delimiter[';'] =
SEMI_COLON, delimiter[','] = COMMA;
        second_op['='] = EQUAL;
        return;
}
```

OUTPUT




D:\assignments\lexer\lexical_analyser.exe
Token : Delimiter Lexeme : ;
Token : Key Word Lexeme : return
Token : Numeric Constant Lexeme : 0
Token : Delimiter Lexeme : ;
Token : Delimiter Lexeme : }
Keywords : 8
Identifiers : 9
Operators : 5 Delimiters : 12
Numeric Constants : 2 Character Constants : 1
String Constants : 1
Process returned 0 (0x0) execution time : 43.065 s
Press any key to continue.

PROGRAM TO PARSE THE STRING USING OPERATOR PRECEDENCE PARSER.

PROBLEM DEFINITION

Design an Operator Precedence Parser and parse a string using the parser

SOURCE CODE

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

int getindex(char);

char stack[200];

int top = -1, i = 0, l;

char prec[][9]={

                    /*input*/

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

        /*  + */ {'>', '>','<','<','<','<','<','>','>'},

        /*  - */ {'>', '>','<','<','<','<','<','>','>'},

        /*  * */ {'>', '>','>','>','<','<','<','>','>'},

        /*  / */ {'>', '>','>','>','<','<','<','>','>'},

        /*  ^ */ {'>', '>','>','>','<','<','<','>','>'},

        /*  i */ {'>', '>','>','>','>','e','e','>','>'},

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

        /*  ) */ {'>', '>','>','>','>','e','e','>','>'},
```

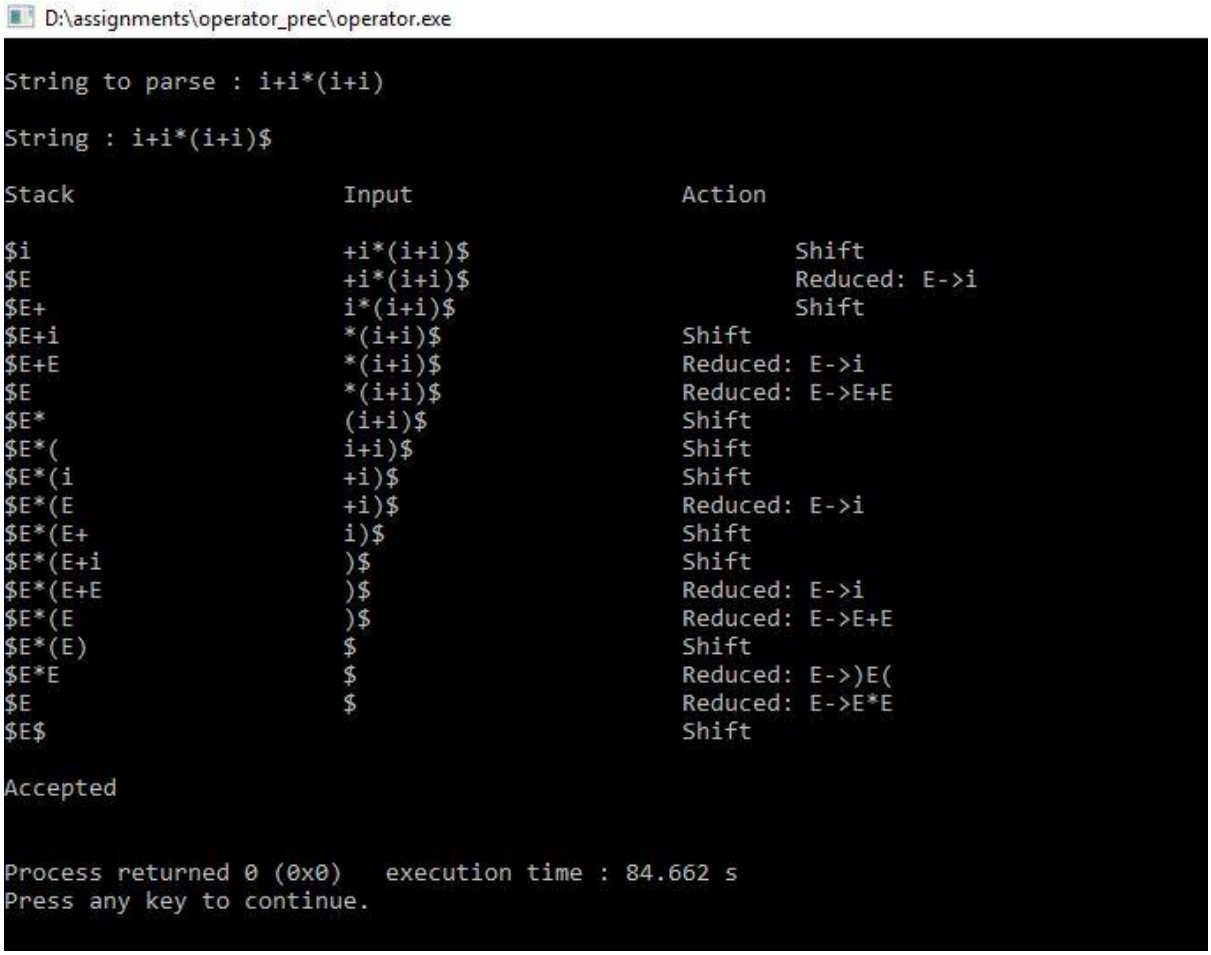
```
        /*  $ */  {'<', '<','<','<','<','<','<','<','>'},

           };

char *handles[] = {")E(", "E*E", "E+E", "E-E", "E/E", "i", "E^E"};
//(E) becomes )E( when pushed to stack

void my_gets(char *, int);

char input[200];

void parse(void);
void shift(void);
void show_stack(void);
void show_input(void);
int reduce(void);
int check();

int h_index = -1;

int main()
{
        printf("\nString to parse : "), my_gets(input, 200);
        printf("\nString : %s\n", input);
        parse();
        return 0;
}

int reduce(void)
{
        int i, len, found, t;
        for(i = 0; i < 7; i++)
        {
                len = strlen(handles[i]);
                if(stack[top] == handles[i][0] && top + 1 >= len)
                {
                        found = 1;
                        for(t = 1; 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;
                        h_index = i;
                        stack[top + 1] = '\0';
                        return 1;
                }
        }
    }
    return 0;
}

void show_stack(void)
{
    int j;
    for(j = 0; j <= top; j++)
        printf("%c", stack[j]);
    return;
}

void show_input(void)
{
    int j;
    for(j = i; j < l; j++)
        printf("%c", input[j]);
    return;
}

int check()
{
    if(top != 2)
        return 0;
    if(stack[0] == '$' && stack[1] == 'E' && stack[2] == '$')
```

```
                return 1;
        return 0;
}

void parse(void)
{
        l = strlen(input);
        stack[++top] = '$';
        printf("\nStack\t\t\tInput\t\t\tAction\n");
        int m, n;
        while(i < l)
        {
                shift();
                printf("\n"), show_stack(), printf("\t\t\t");
                show_input(), printf("\t\t\t"), printf("Shift");
                if(top >= 0 && i < l)
                {
                        m = getindex(stack[top]), n = getindex(input[i]);
                        if(prec[m][n] == '>')
                        {
                                while(reduce())
                                {
                                        printf("\n"), show_stack(), printf("\t\t\t");
                                        show_input(), printf("\t\t");
                                        printf("\tReduced: E->%s", handles[h_index]);
                                }
                        }
                }
        }
        if(check())
                printf("\n\nAccepted\n\n");
        else
                printf("\n\nNot Accepted\n\n");
}

void shift(void)
{
        stack[++top] = input[i++];
}
```

```
void my_gets(char *arr, int n)
{
	int i = 0, c;
	while((c = getchar()) != '\n')
	{
		if(i > n - 3)
			break;
		arr[i++] = c;
	}
	arr[i++] = '$';
	arr[i] = '\0';
}

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;
   }
   return -1;
}
```

OUTPUT


D:\assignments\operator_prec\operator.exe
String to parse : i+i*(i+i)
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
$E+E *(i+i)$ Reduced: E->i
$E *(i+i)$ Reduced: E->E+E
$E* (i+i)$ Shift
$E*(i+i)$ Shift
$E*(i +i)$ Shift
$E*(E +i)$ Reduced: E->i
$E*(E+ i)$ Shift
$E*(E+i)$ Shift
$E*(E+E)$ Reduced: E->i
$E*(E)$ Reduced: E->E+E
$E*(E) $ Shift
$E*E $ Reduced: E->)E(
$E $ Reduced: E->E*E
E Shift
Accepted
Process returned 0 (0x0) execution time : 84.662 s
Press any key to continue.

PROGRAM TO PARSE THE STRING USING SHIFT REDUCE BOTTOM UP PARSING.

PROBLEM DEFINITION

Design an Shift Reduce Parser and parse a string using the parser

SOURCE CODE

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

char lhs[100][100], rhs[100][200];

void my_gets(char *, int);
void parse(void);
void parsing(void);
int check(void);

char t1[100], t2[100], input[300], temp[300], stack[300];
char start_symbol;

int n, top = -1;

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

int check(void)
{
        if(strlen(stack) != 1)
                return 0;
        if(stack[0] == start_symbol)
                return 1;
        return 0;
}

void parsing(void)
```

```
{
        int len = strlen(input), i, l = 0, k;
        input[len] = ' ', input[len + 1] = '\0';
        len = strlen(input);
        printf("String %s\n", input);
        while(l < len)
        {
                i = 0;
                while(input[l] != ' ')
                {
                        temp[i] = input[l];
                        i++, l++;
                }
                temp[i] = '\0';
                for(k = 0; k < n; k++)
                {
                        if(!strcmp(temp, rhs[k]))
                        {
                                strcpy(temp, lhs[k]);
                                break;
                        }
                }
                strcat(stack, temp);
                printf("\nStack -> %s", stack);
                for(k = 0; k < n; k++)
                {
                        if(!strcmp(stack, rhs[k]))
                        {
                                strcpy(stack, lhs[k]);
                                break;
                        }
                }
                l++;
        }
        printf("\nStack -> %s", stack);
        if(check())
        {
                printf("\n\nAccepted\n\n");
        }
        else
```

```
	{
		printf("\n\nRejected\n\n");
	}
}

void parse(void)
{
	//printf("\nNumber of productions : "), scanf("%d", &n);
	FILE *fp1 = NULL;
	char file_name[100];
	printf("\nFile name containing productions : "), my_gets(file_name,
100);
	printf("\nFile name : %s\n", file_name);
	fp1 = fopen(file_name, "r+");
	if(fp1 == NULL)
	{
		fprintf(stderr, "\n\nUnable to open file : %s\n\n", file_name);
		return;
	}
	int i = 0;
	n = 0;
	fscanf(fp1, "%c\n", &start_symbol);
	while(!feof(fp1))
	{
		fscanf(fp1, "%s %s\n", t1, t2);
		//printf("\n%s\t%s\n", t1, t2);
		strcpy(lhs[i], t1), strcpy(rhs[i], t2);
		i++, n++;
	}
	printf("\nStart Symbol : %c", start_symbol);
	printf("\n\nProductions\n\n");
	for(i = 0; i < n; i++)
	{
		printf("%s -> %s\n", lhs[i], rhs[i]);
	}
	printf("\nInput String : "), my_gets(input, 100);
	printf("\nInput String : %s\n\n", input);
	parsing();
	fclose(fp1), fp1 = NULL;
}
```

```
void my_gets(char *arr, int n)
{
	int i = 0, c;
	while((c = getchar()) != '\n')
	{
		if(i > n - 2)
			break;
		arr[i++] = c;
	}
	arr[i] = '\0';
}
```

OUTPUT

```
File name containing productions : productions.txt

File name : productions.txt

Start Symbol : E

Productions

E -> E+E
E -> E*E
E -> i
E -> E-E
E -> E/E

Input String : i + i * i

Input String : i + i * i

String i + i * i

Stack -> E
Stack -> E+
Stack -> E+E
Stack -> E*
Stack -> E*E
Stack -> E

Accepted


Process returned 0 (0x0)   execution time : 43.263 s
Press any key to continue.
```

PROGRAM TO GENERATE PARSING TABLE FOR LL(1) PARSING

PROBLEM DEFINITION

Write a Program to Generate Parsing Table For LL(1) Grammar

SOURCE CODE

```
#include<stdio.h>
#define max 100

char NonTerminals[max],Terminals[max],production[max][max];
char follow[max];
int x=0,n;
void find_follow(char);
void find_first(char);

int main()
{

   int i,j,k=0,length,flag,l,matrix[max][max];

   printf("\nEnter the number of rules:- ");
   scanf("%d",&n);

   printf("\nEnter the rules one by one\n");
   for(i=0;i<n;i++)
   {
      printf("%d:- ",i+1);
      scanf("%s",production[i]);
      flag=0;
      for(l=0;l<k;l++)
      {
         if(NonTerminals[l]==production[i][0])
            flag=1;
      }
      if(flag==0)
         NonTerminals[k++]=production[i][0];
   }
```

```
NonTerminals[k]='\0';
k=0;
printf("\n\n");
for(i=0;i<n;i++)
{
   length=strlen(production[i]);
   for(j=3;j<length;j++)
   {
      if(!(production[i][j]>=65 && production[i][j]<=90))
      {
         flag=0;
         for(l=0;l<k;l++)
         {
            if(Terminals[l]==production[i][j])
               flag=1;
         }
         if(flag==0)
            Terminals[k++]=production[i][j];
      }

   }
}
Terminals[k]='\0';

for(i=0;i<strlen(NonTerminals);i++)
{
   for(j=0;j<n;j++)
   {
      if(NonTerminals[i]==production[j][0])
      {
         if(production[j][3]=='$')
         {
            Array_Manipulation('$');

            find_follow(production[j][0]);
         }
         else
         {
            find_first(production[j][3]);
```

```
            }

            for(l=0;l<strlen(Terminals);l++)
            {
               for(k=0;k<x;k++)
               {
                  if(follow[k]==Terminals[l])
                     matrix[i][l]=j+1;
               }
            }
            x=0;
         }
      }
      x=0;
   }
   printf("    LL(1) PARSING TABLE\n\n");

   for(i=0;i<strlen(Terminals);i++)
      printf("     %c",Terminals[i]);
   printf("\n\n");

   for(i=0;i<strlen(NonTerminals);i++)
   {
      printf("%c    ",NonTerminals[i]);
      for(j=0;j<strlen(Terminals);j++)
         if(matrix[i][j]==0)
            printf("      ");
         else
            printf("%d     ",matrix[i][j]);
      printf("\n");
   }

}

void find_follow(char ch)
{
     int i, j,length;

     if(production[0][0] == ch)
     {
```

```
        Array_Manipulation('$');
    }
    for(i = 0; i < n; i++)
    {
        length = strlen(production[i]);
        for(j = 3; j < length; j++)
        {
            if(production[i][j] == ch)
            {
                if(production[i][j + 1] != '$')
                {
                    find_first(production[i][j + 1]);
                }
                if(production[i][j + 1] == '$' && ch != production[i][0])
                {
                    find_follow(production[i][0]);
                }
            }
        }
    }
}
void find_first(char ch)
{
    int k;
    if(!(isupper(ch)))
    {
        Array_Manipulation(ch);
    }
    for(k = 0; k <n; k++)
    {
        if(production[k][0] == ch)
        {
            if(production[k][3] == '$')
            {
                find_follow(production[k][0]);;
            }
            else if(islower(production[k][3]))
            {
                Array_Manipulation(production[k][3]);
            }
```

```
            else
            {
                find_first(production[k][3]);
            }
        }
    }
}
void Array_Manipulation(char ch)
{
    int i;
    for(i= 0;i< x;i++)
    {
        if(follow[i] == ch)
        {
            return;
        }
    }
    follow[x++] = ch;
}
```

OUTPUT

```
D:\E\Compiler\LL(1)\LL1.exe

Enter the number of rules:- 6

Enter the rules one by one
1:- S->dA$
2:- S->aB$
3:- A->bA$
4:- A->c$
5:- B->bB$
6:- B->c$


    LL(1) PARSING TABLE

     d     $     a     b     c

S    1           2
A                      3     4
B                      5     6

Process returned 3 (0x3)   execution time : 20.533 s
Press any key to continue.
```

PROGRAM TO CHECK WHETHER THE GIVEN GRAMMAR IS LR(0) OR NOT

PROBLEM DEFINITION

Design a program to check whether the given grammar is LR(0) or not.

SOURCE CODE

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define row 100
#define col 100
#define NULL 0

struct data
{
   int state_no;
   int count;
   char items[row][col];
   struct data *link2;
};

typedef struct data node;

void closure(char []);
void Goto(char [],int);
void CreateTable(int,char,char[],char[],char[]);

char Agumented[col]={'Q','-','>','.','S','$'};
int n,state=0,result=0;
char
Grammar[row][col],P1[row][col],symbols[col],table[row][col],GrammarFinal[ro
w][col];
node *start=NULL;

int main()
{
```

```
char str[30],stack[30],temp[col];
node *ptr1;

int i,j,NoOfSymbols,flag=0,k,size;

printf("\nEnter the number of rules:- ");
scanf("%d",&n);
fflush(stdin);
printf("\nEnter the Start Symbol:- ");
scanf("%c",&Agumented[4]);
fflush(stdin);
printf("Enter the terminal first Followed by Non terminals \n");
scanf("%s",symbols);
fflush(stdin);
printf("\nEnter the rules one by one\n");
for(i=0;i<n;i++)
{
   printf("%d:- ",i+1);
   scanf("%s",temp);
   size=strlen(temp);

   strcpy(GrammarFinal[i],temp);
   GrammarFinal[i][size-1]='.';
   GrammarFinal[i][size]='$';
   GrammarFinal[i][size+1]='\0';

   for(j=0;j<=size+1;j++)
   {
      if(j==3)
         Grammar[i][j]='.';
      if(j>3)
         Grammar[i][j]=temp[j-1];
      if(j<3)
         Grammar[i][j]=temp[j];
      if(j==strlen(temp)+1)
        Grammar[i][j]='\0';
   }
}

closure(&Agumented[0]);
```

```
ptr1=start;                          // GOTO Function call
while(ptr1!=NULL)
{
   for(i=0;i<ptr1->count;i++)
   {
      Goto(ptr1->items[i],ptr1->state_no);
   }
   ptr1=ptr1->link2;
}

ptr1=start;
printf("\n");
while(ptr1!=NULL)
{
   printf("<----------STATE %d---------->\n",ptr1->state_no);
   for(i=0;i<ptr1->count;i++)
   {
      printf("%s\n",ptr1->items[i]);
   }
   ptr1=ptr1->link2;
}
printf("\n PARSING TABLE  \n"); // Printing Parsing Table
for(j=0;j<strlen(symbols);j++)
{
   printf("%c    ",symbols[j]);
}
printf("\n");

for(i=0;i<=state;i++)
{
   for(j=0;j<strlen(symbols);j++)
   {
      printf("%c",table[i][j*4]);
      printf("%c",table[i][(j*4)+1]);
      printf("%c  ",table[i][(j*4)+2]);
   }
   printf("\n");
}
if(result==0)
```

```
      printf("The Given Grammar is LR(0)\n");
   else
      printf("The Given Grammar is Not LR(0)\n");


return 0;
}



void closure(char a[])
{
  int i,j,k,c1=0,n1=0,flag=0,z;
  node *ptr,*ptr1=start;

  ptr=(node *)malloc(sizeof(node));
  ptr->state_no=state;
  ptr->link2=NULL;
  strcpy(ptr->items[c1],a);
  c1++;

  strcpy(P1[n1],a);
   n1++;

  for(k=0;k<n1;k++)
  {
     flag=0;
     for(i=0;i<strlen(P1[k]);i++)
     {
        if(P1[k][i]=='.' && P1[k][i+1]!='$')
        {
           for(z=1;z<n1;z++)
           {
              if(P1[z][0]==P1[k][i+1])
              {
                 flag=1;
                 break;
              }
           }
           if(flag==0)
```

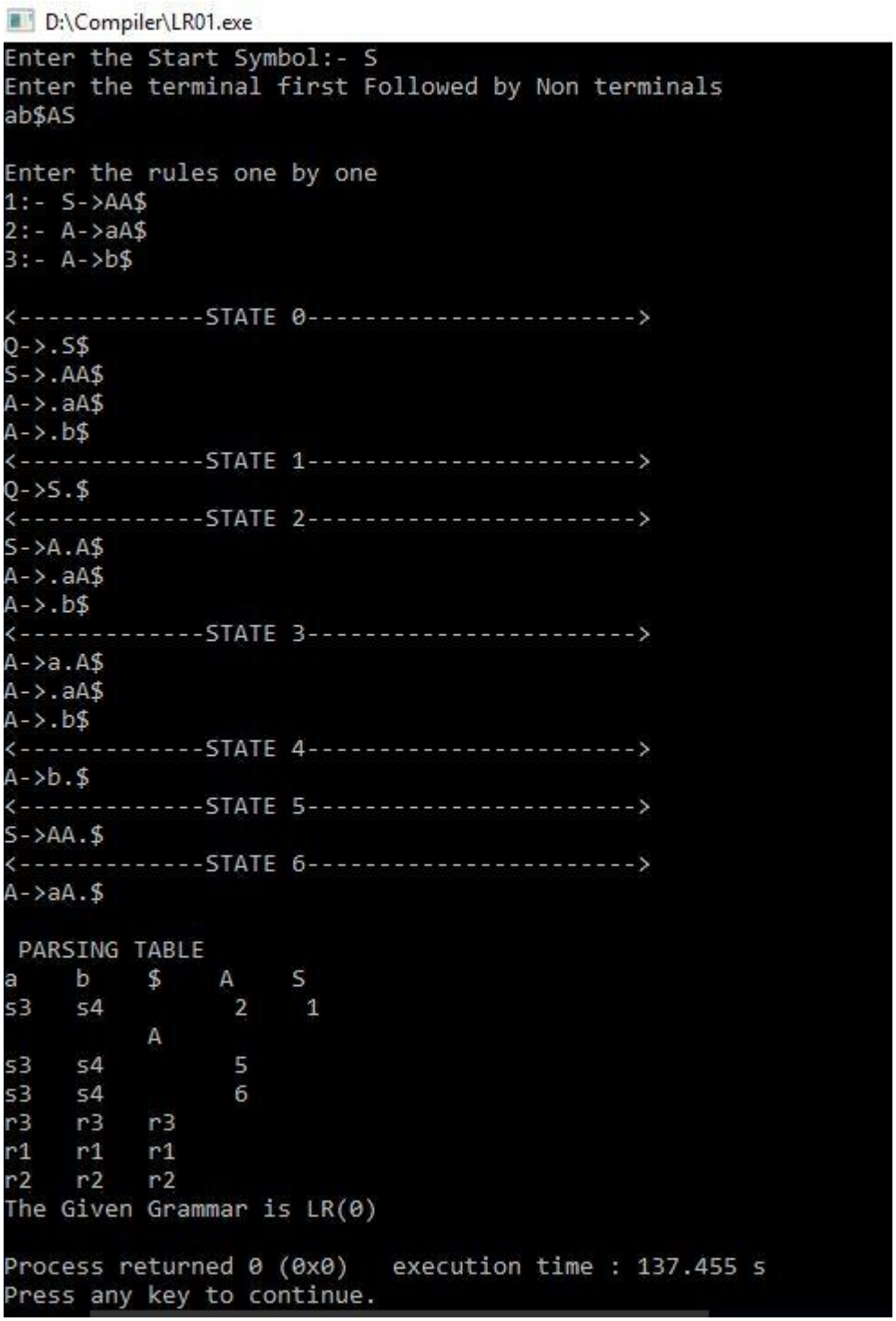
```
            {
              for(j=0;j<n;j++)
              {
                  if(P1[k][i+1]==Grammar[j][0])
                  {
                     strcpy(P1[n1],Grammar[j]);
                     n1++;
                     strcpy(ptr->items[c1],Grammar[j]);
                     c1++;
                  }
               }
            }

            break;
         }
      }
   }

    ptr->count=c1;
    if(start==NULL)
    {
       start=ptr;
    }
    else
    {
      while(ptr1->link2!=NULL)
         ptr1=ptr1->link2;

      ptr1->link2=ptr;
    }
}

void Goto(char a[],int s_no)
{
    char b[col],temp,buffer[3];
    int i,flag=0,j,pos=0,k;
    node *ptr1;

    for(i=0;i<strlen(a);i++)
    {
```

```
        if(a[i]=='.' && a[i+1]!='$')
        {
             b[i]=a[i+1];
             temp=a[i+1];
             b[i+1]='.';
             i=i+1;
        }
        else
        {
          b[i]=a[i];
        }
    }
    b[i]='\0';

    ptr1=start;
    while(ptr1!=NULL)
    {
        if(strcmp(b,ptr1->items[0])==0)
        {
            flag=1;

            sprintf(buffer,"%d",ptr1->state_no);
            CreateTable(s_no,temp,buffer,a,b);

            break;
        }
        ptr1=ptr1->link2;
    }

    if(flag==0)
    {
        state++;

        sprintf(buffer,"%d",state);
        CreateTable(s_no,temp,buffer,a,b);

        closure(&b[0]);
    }
}
```

```
void CreateTable(int s_no,char temp,char buffer[],char a[],char b[])
{
   int k,flag=0,i;

   for(k=0;k<strlen(symbols);k++)
   {
      if(temp==symbols[k])
      {
         flag=1;
         break;
      }
   }
   if(flag==0)
   {
      printf("Enter the symbols correctly \n");
      printf("%s %s",temp,symbols[k]);
      exit(0);
   }


   if(temp>=65 && temp<=90)                //Goto Phase
   {
      table[s_no][k*4]=' ';
      table[s_no][(k*4)+1]=buffer[0];
      table[s_no][(k*4)+2]=buffer[1];
   }
   else if(strcmp(a,b)==0)                 //Reduce Phase
   {
      for(i=0;i<n;i++)
      {
         if(strcmp(a,GrammarFinal[i])==0)
         {
            sprintf(buffer,"%d",i+1);
               break;
         }
      }
      for(k=0;k<strlen(symbols);k++)
      {
         if(a[0]=='Q' && symbols[k]=='$')
         {
```

```
            table[s_no][k*4]='A';
        }
        else if(a[0]!='Q')
        {
           if((symbols[k]>=97 && symbols[k]<=122)||(symbols[k]>=35 &&
symbols[k]<=57))
           {
              if(table[s_no][k*4]=='r'|| table[s_no][k*4]=='s')
                 result=1;
              table[s_no][k*4]='r';
              table[s_no][(k*4)+1]=buffer[0];
              table[s_no][(k*4)+2]=buffer[1];
           }
           else
              break;
        }
     }
  }
  else if(strcmp(a,b)!=0)              //Shift Phase
  {
     if(table[s_no][k*4]=='r'|| table[s_no][k*4]=='s')
        result=1;
    table[s_no][k*4]='s';
    table[s_no][(k*4)+1]=buffer[0];
    table[s_no][(k*4)+2]=buffer[1];
  }

}
```

OUTPUT


D:\Compiler\LR01.exe
Enter the Start Symbol:- S
Enter the terminal first Followed by Non terminals
ab$AS
Enter the rules one by one
1:- S->AA$
2:- A->aA$
3:- A->b$
<-------------STATE 0----------------------->
Q->.S$
S->.AA$
A->.aA$
A->.b$
<-------------STATE 1----------------------->
Q->S.$
<-------------STATE 2----------------------->
S->A.A$
A->.aA$
A->.b$
<-------------STATE 3----------------------->
A->a.A$
A->.aA$
A->.b$
<-------------STATE 4----------------------->
A->b.$
<-------------STATE 5----------------------->
S->AA.$
<-------------STATE 6----------------------->
A->aA.$
PARSING TABLE
a b $ A S
s3 s4 2 1
A
s3 s4 5
s3 s4 6
r3 r3 r3
r1 r1 r1
r2 r2 r2
The Given Grammar is LR(0)
Process returned 0 (0x0) execution time : 137.455 s
Press any key to continue.

PROGRAM TO CHECK WHETHER THE GIVEN GRAMMAR IS SLR(1) OR NOT

PROBLEM DEFINITION

Design a program to check whether the given grammar is SLR(1) or not.

SOURCE CODE

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define row 100
#define col 100
#define NULL 0

struct data
{
   int state_no;
   int count;
   char items[row][col];
   struct data *link2;
};

typedef struct data node;

void closure(char []);
void Goto(char [],int);
void CreateTable(int,char,char[],char[],char[]);
void find_follow(char );
void Array_Manipulation(char );

char Agumented[col]={'Q','-','>','.','S','$'};
char follow[col];
int n,state=0,x=0,result=0;
char
Grammar[row][col],P1[row][col],symbols[col],table[row][col],production[row][
col],follow[col];
char GrammarFinal[row][col];
node *start=NULL;
```

```
int main()
{
   char str[30],stack[30],temp[col];
   node *ptr1;

   int i,j,NoOfSymbols,flag=0,k,size;

   printf("\nEnter the number of rules:- ");
   scanf("%d",&n);
   fflush(stdin);
   printf("\nEnter the Start Symbol:- ");
   scanf("%c",&Agumented[4]);
   fflush(stdin);
   printf("Enter the terminal first Followed by Non terminals \n");
   scanf("%s",symbols);
   fflush(stdin);
   printf("\nEnter the rules one by one\n");
   for(i=0;i<n;i++)
   {
      printf("%d:- ",i+1);
      scanf("%s",temp);
      size=strlen(temp);

      strcpy(production[i],temp);
      strcpy(GrammarFinal[i],temp);
      GrammarFinal[i][size-1]='.';
      GrammarFinal[i][size]='$';
      GrammarFinal[i][size+1]='\0';

      for(j=0;j<=size+1;j++)
      {
         if(j==3)
            Grammar[i][j]='.';
         if(j>3)
            Grammar[i][j]=temp[j-1];
         if(j<3)
            Grammar[i][j]=temp[j];
         if(j==strlen(temp)+1)
           Grammar[i][j]='\0';
```

```
   }
}

closure(&Agumented[0]);

ptr1=start;                          // GOTO Function call
while(ptr1!=NULL)
{
   for(i=0;i<ptr1->count;i++)
   {
      Goto(ptr1->items[i],ptr1->state_no);
   }
   ptr1=ptr1->link2;
}

ptr1=start;
printf("\n");
while(ptr1!=NULL)
{
   printf("<-------------STATE %d---------------------->\n",ptr1->state_no);
   for(i=0;i<ptr1->count;i++)
   {
      printf("%s\n",ptr1->items[i]);
   }
   ptr1=ptr1->link2;
}
printf("\n PARSING TABLE  \n"); // Printing Parsing Table
for(j=0;j<strlen(symbols);j++)
{
   printf("%c    ",symbols[j]);
}
printf("\n");

for(i=0;i<=state;i++)
{
   for(j=0;j<strlen(symbols);j++)
   {
      printf("%c",table[i][j*4]);
      printf("%c",table[i][(j*4)+1]);
      printf("%c",table[i][(j*4)+2]);
```

```
            printf("%c ",table[i][(j*4)+3]);
        }
        printf("\n");
    }
    if(result==0)
        printf("The Given Grammar is SLR\n");
    else
        printf("The Given Grammar is Not SLR\n");


return 0;
}



void closure(char a[])
{
  int i,j,k,c1=0,n1=0,flag=0,z;
  node *ptr,*ptr1=start;

  ptr=(node *)malloc(sizeof(node));
  ptr->state_no=state;
  ptr->link2=NULL;
  strcpy(ptr->items[c1],a);
  c1++;

  strcpy(P1[n1],a);
   n1++;

  for(k=0;k<n1;k++)
  {
      flag=0;
      for(i=0;i<strlen(P1[k]);i++)
      {
          if(P1[k][i]=='.' && P1[k][i+1]!='$')
          {
              for(z=1;z<n1;z++)
              {
                  if(P1[z][0]==P1[k][i+1])
                  {
```

```
                flag=1;
                break;
              }
            }
            if(flag==0)
            {
              for(j=0;j<n;j++)
              {
                if(P1[k][i+1]==Grammar[j][0])
                {
                  strcpy(P1[n1],Grammar[j]);
                  n1++;
                  strcpy(ptr->items[c1],Grammar[j]);
                  c1++;
                }
              }
            }

            break;
          }
        }
      }

    ptr->count=c1;
    if(start==NULL)
    {
       start=ptr;
    }
    else
    {
      while(ptr1->link2!=NULL)
         ptr1=ptr1->link2;

      ptr1->link2=ptr;
    }
}

void Goto(char a[],int s_no)
{
   char b[col],temp,buffer[3];
```

```
int i,flag=0,j,pos=0,k;
node *ptr1;

for(i=0;i<strlen(a);i++)
{
   if(a[i]=='.' && a[i+1]!='$')
   {
        b[i]=a[i+1];
        temp=a[i+1];
        b[i+1]='.';
        i=i+1;
   }
   else
   {
     b[i]=a[i];
   }
}
b[i]='\0';

ptr1=start;
while(ptr1!=NULL)
{
   if(strcmp(b,ptr1->items[0])==0)
   {
      flag=1;

      sprintf(buffer,"%d",ptr1->state_no);
      CreateTable(s_no,temp,buffer,a,b);

      break;
   }
   ptr1=ptr1->link2;
}

if(flag==0)
{
   state++;

   sprintf(buffer,"%d",state);
   CreateTable(s_no,temp,buffer,a,b);
```

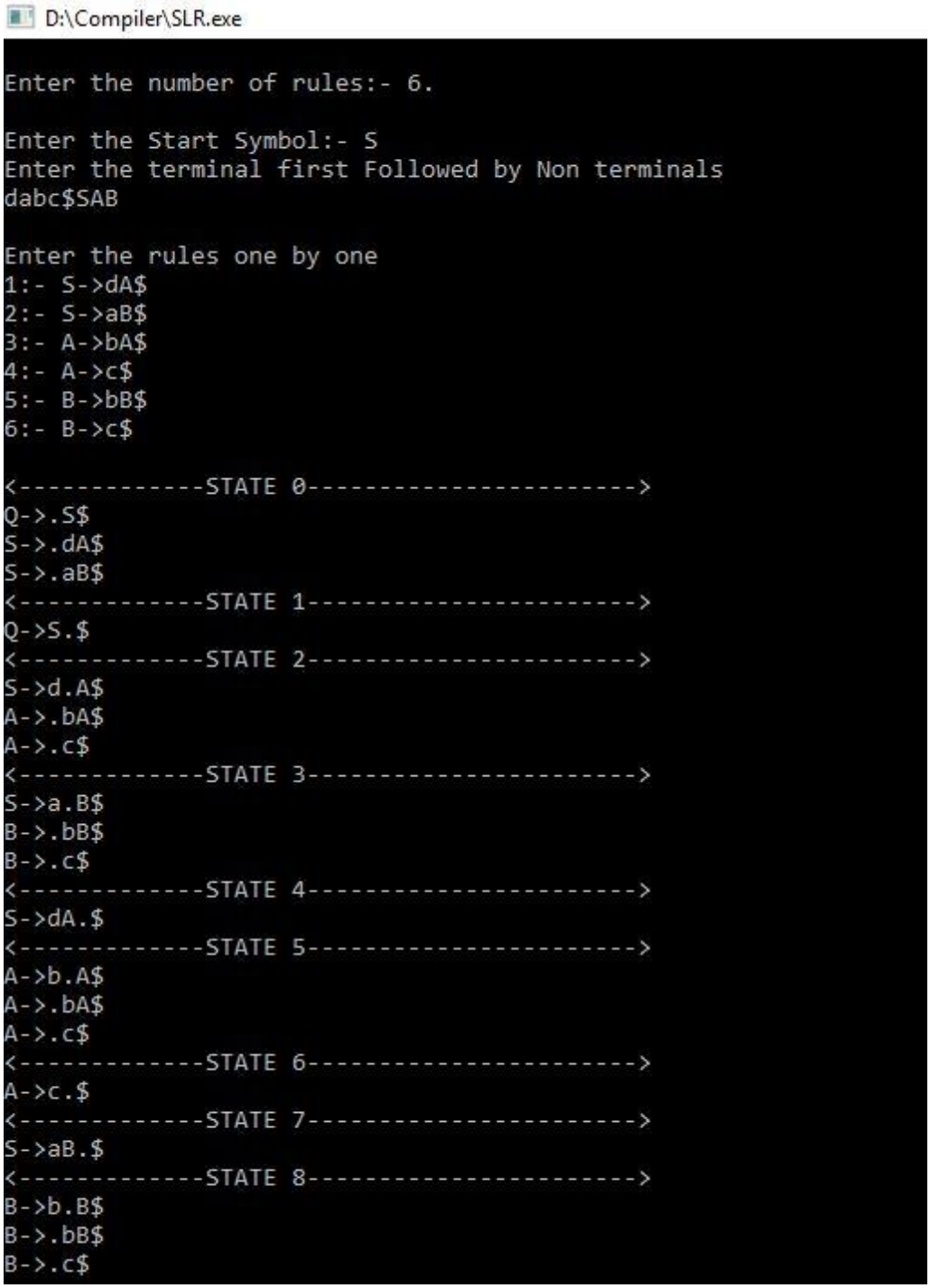
```
      closure(&b[0]);
   }
}

void CreateTable(int s_no,char temp,char buffer[],char a[],char b[])
{
   int k,flag=0,i,j;

   for(k=0;k<strlen(symbols);k++)
   {
      if(temp==symbols[k])
      {
         flag=1;
         break;
      }
   }
   if(flag==0)
   {
      printf("Enter the symbols correctly \n");
      printf("%s %s",temp,symbols[k]);
      exit(0);
   }


   if(temp>=65 && temp<=90)                //Goto Phase
   {
      table[s_no][k*4]=' ';
      table[s_no][(k*4)+1]=buffer[0];
      table[s_no][(k*4)+2]=buffer[1];
      table[s_no][(k*4)+3]=buffer[2];
   }
   else if(strcmp(a,b)==0)                 //Reduce Phase
   {
      for(i=0;i<n;i++)
      {
         if(strcmp(a,GrammarFinal[i])==0)
         {
            sprintf(buffer,"%d",i+1);
            break;
```

```
        }
      }
      find_follow(a[0]);

      for(k=0;k<strlen(symbols);k++)
      {
        if(a[0]=='Q' && symbols[k]=='$')
        {
            table[s_no][k*4]='A';
        }
        else if(a[0]!='Q')
        {
          for(j=0;j<x;j++)
          {
           if(follow[j]==symbols[k])
           {
             if(table[s_no][k*4]=='r'|| table[s_no][k*4]=='s')
              result=1;
             table[s_no][k*4]='r';
             table[s_no][(k*4)+1]=buffer[0];
             table[s_no][(k*4)+2]=buffer[1];
             table[s_no][(k*4)+3]=buffer[2];
             break;
           }
          }
        }
      }
      x=0;
   }
   else if(strcmp(a,b)!=0)              //Shift Phase
   {
     if(table[s_no][k*4]=='r'|| table[s_no][k*4]=='s')
        result=1;
     table[s_no][k*4]='s';
     table[s_no][(k*4)+1]=buffer[0];
     table[s_no][(k*4)+2]=buffer[1];
     table[s_no][(k*4)+3]=buffer[2];
   }

}
```

```
void find_follow(char ch)
{
    int i, j,length;

    if(production[0][0] == ch)
    {
        Array_Manipulation('$');
    }
    for(i = 0; i < n; i++)
    {
        length = strlen(production[i]);
        for(j = 3; j < length; j++)
        {
            if(production[i][j] == ch)
            {
                if(production[i][j + 1] != '$')
                {
                    find_first(production[i][j + 1]);
                }
                if(production[i][j + 1] == '$' && ch != production[i][0])
                {
                    find_follow(production[i][0]);
                }
            }
        }
    }
}

void find_first(char ch)
{
    int k;
    if(!(isupper(ch)))
    {
        Array_Manipulation(ch);
    }
    for(k = 0; k <n; k++)
    {
        if(production[k][0] == ch)
        {
```

```
            if(production[k][3] == '$')
            {
                find_follow(production[k][0]);;
            }
            else if(islower(production[k][3]))
            {
                Array_Manipulation(production[k][3]);
            }
            else
            {
                find_first(production[k][3]);
            }
        }
    }
}

void Array_Manipulation(char ch)
{
    int i;
    for(i= 0;i< x;i++)
    {
        if(follow[i] == ch)
        {
            return;
        }
    }
    follow[x++] = ch;
}
```

OUTPUT


D:\Compiler\SLR.exe
Enter the number of rules:- 6.
Enter the Start Symbol:- S
Enter the terminal first Followed by Non terminals
dabc$SAB
Enter the rules one by one
1:- S->dA$
2:- S->aB$
3:- A->bA$
4:- A->c$
5:- B->bB$
6:- B->c$
<-------------STATE 0----------------------->
Q->.S$
S->.dA$
S->.aB$
<-------------STATE 1----------------------->
Q->S.$
<-------------STATE 2----------------------->
S->d.A$
A->.bA$
A->.c$
<-------------STATE 3----------------------->
S->a.B$
B->.bB$
B->.c$
<-------------STATE 4----------------------->
S->dA.$
<-------------STATE 5----------------------->
A->b.A$
A->.bA$
A->.c$
<-------------STATE 6----------------------->
A->c.$
<-------------STATE 7----------------------->
S->aB.$
<-------------STATE 8----------------------->
B->b.B$
B->.bB$
B->.c$

```
<-------------STATE 9----------------------->
B->c.$
<-------------STATE 10---------------------->
A->bA.$
<-------------STATE 11---------------------->
B->bB.$

 PARSING TABLE
d    a    b    c    $    S    A    B
s2   s3                   1
                    A
          s5   s6              4
          s8   s9                   7
                    r1
          s5   s6              10
                    r4
                    r2
          s8   s9                   11
                    r6
                    r3
                    r5
The Given Grammar is SLR

Process returned 0 (0x0)   execution time : 26.593 s
Press any key to continue.
```