DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
School of Engineering Sciences and Technology
Jamia Hamdard

Mohd Shahique Qamar Siddiqi
2016-310-060

YEAR 4th SEMESTER 8th

Lab Based on Electives (Compiler Design) Lab
[BTCSE-802]

LAB MANUAL

Dr. Parul Agarwal & Pooja Gupta
[ASST. PROFESSOR]

NEW DELHI-110062

LIST OF EXPERIMENTS

Sl. No.	Name of the Experiments
1.	Practice of Lex/Yacc of Compiler Writing.
2.	Write a Java program to check whether a string belongs to the grammar or not.
3.	Write a program to generate a parse tree.
4.	Write a program to find the FIRST of non-terminals.
5.	Write a program to compute the FOLLOW of non-terminals.
6.	Write a program to check whether the grammar is left recursive or not.
7.	Write a program to remove left factoring.
8.	Write a program to check whether a grammar is Operator precedent.
9.	Write a program to check whether a string satisfies the condition for ab*.
10.	Write a program to check whether a string belongs to the grammar aa*.

Ques. 1) Practice of Lex/Yacc of Compiler Writing

Ans. 1)

Lex

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to Lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to Lex has an associated action. Typically, an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyser that scans for identifiers.

```
letter(letter|digit)*
```

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

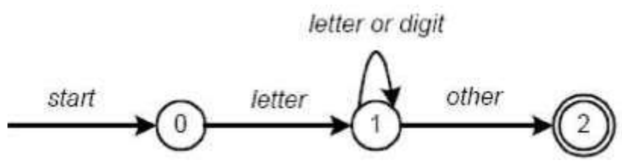


Figure 1: Finite State Automaton

In Figure 1 state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. Any FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start: goto state0
state0: read c
        if c = letter goto state1
        goto state0
state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2
state2: accept string
```

This is the technique used by Lex. Regular expressions are translated by Lex to a computer program that mimics an FSA. Using the next input character and current state the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of Lex's limitations. For example, Lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "(" we push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However, Lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

Yacc

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

```
E -> E + E
E -> E * E
E -> id
```

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

```
E -> E * E      (r2)
  -> E * z      (r3)
  -> E + E * z  (r1)
  -> E + y * z  (r3)
  -> x + y * z  (r3)
```

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression, we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

```
1     .x + y * z     shift
2     x . + y * z    reduce(r3)
3     E . + y * z    shift
4     E + . y * z    shift
5     E + y . * z    reduce(r3)
6     E + E . * z    shift
7     E + E * . z    shift
8     E + E * z .    reduce(r3)
9     E + E * E .    reduce(r2)     emit multiply
10    E + E .                  reduce(r1)     emit add
11    E .            accept
```

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words, the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle and we are reducing the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to change x to E. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly, the add instruction is emitted in step 10. Consequently, multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a shift-reduce conflict. Our grammar is ambiguous because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply Yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section.

The following grammar has a reduce-reduce conflict. With an id on the stack we may reduce to T, or E.

E -> T

E -> id

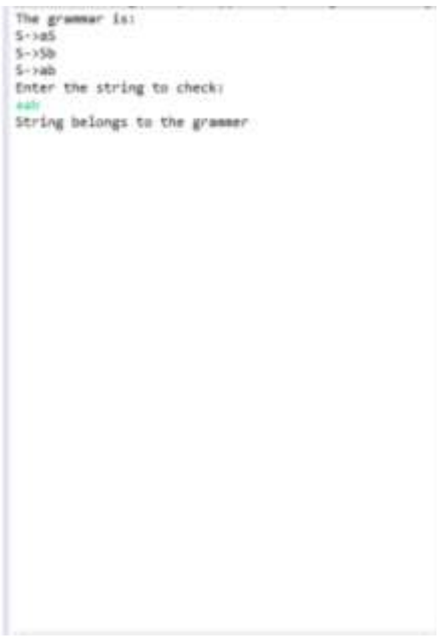
T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

Ques. 2) Write a Java program to check whether a string belongs to the grammar or not.

Ans. 2)

```
import java.util.*;
public class stringCheck {
        public static void main(String[] args) {
                System.out.println("The grammar is:\nS->aS \nS->Sb \nS->ab");
                System.out.println("Enter the string to check:");
                Scanner s=new Scanner(System.in);
                String string=s.next();
                int count=1;
                if(string.charAt(0)=='a') {
                        int flag=0;
                        for (count=1;count<string.length();count++) {
                                if(string.charAt(count)=='b') {
                                        flag=1;
                                        if(count!=string.length()-1)
                                                continue;
                                        else
                                                System.out.println("String belongs to the grammer");
                                }
                                else if((flag==1) && (string.charAt(count)=='a')) {
                                        System.out.println("The string does not belong to the specified
grammar");
                                        break;
                                }
                                else if(string.charAt(count)=='a')      {
                                        if(count!=string.length()-1)
                                                continue;
                                        else
                                                System.out.println("The string does not belong to the
specified grammar");
                                }
                                else if((flag==1) && (count==string.length()-1)) {
                                        System.out.println("String belongs to the grammer");
                                        break;
                                        }
                                else {
                                        System.out.println("String does not belong to the specified
grammer ");
                                }
                        }
                }
        }
}
```



```
The grammar is:
S->aS
S->Sb
S->ab
Enter the string to check:
aab
String belongs to the grammer
```

Ques. 3) Write a program to generate a parse tree.
Ans. 3)
```
%{#include "y.tab.h"
 /* very silly implementation, just sufficient for this example */
 static char findname (char *yytext) {return yytext [0];}
 %}
 %%
 [0-9]+                         {yylval.a_number = atoi(yytext); return number;}
 [A-Za-z][A-Za-z0-9]*           {yylval.a_variable = findname (yytext);
                                 return variable;}
 [ \t\n]                        ;
 [-+*/( );]                     return yytext[0];
 .                              {ECHO; yyerror ("unexpected character");}
 %%
 int yywrap (void) {return 1;}

 %{
 #include <stdio.h>
 enum treetype {operator_node, number_node, variable_node};
 typedef struct tree {
   enum treetype nodetype;
   union {
     struct {struct tree *left, *right; char operator;} an_operator;
     int a_number;
     char a_variable;
   } body;
 } tree;
 static tree *make_operator (tree *l, char o, tree *r) {
   tree *result= (tree*) malloc (sizeof(tree));
   result->nodetype= operator_node;
   result->body.an_operator.left= l;
   result->body.an_operator.operator= o;
   result->body.an_operator.right= r;
   return result;
 }
 static tree *make_number (int n) {
   tree *result= (tree*) malloc (sizeof(tree));
   result->nodetype= number_node;
   result->body.a_number= n;
   return result;
 }
 static tree *make_variable (char v) {
   tree *result= (tree*) malloc (sizeof(tree));
   result->nodetype= variable_node;
   result->body.a_variable= v;
   return result;
 }
 static void printtree (tree *t, int level) {
 #define step 4
   if (t)
     switch (t->nodetype)
     {
       case operator_node:
        printtree (t->body.an_operator.right, level+step);
        printf ("%*c%c\n", level, ' ', t->body.an_operator.operator);
        printtree (t->body.an_operator.left, level+step);
        break;
       case number_node:
        printf ("%*c%d\n", level, ' ', t->body.a_number);
        break;
       case variable_node:
        printf ("%*c%c\n", level, ' ', t->body.a_variable);
     }
 }
 %}

 %union {
   int a_number;
```

```
  char a_variable;
  tree* a_tree;
}
%start input
%token <a_number> number
%token <a_variable> variable
%type <a_tree> exp term factor
%%
input  : exp ';'               {printtree ($1, 1);}
       ;
exp    : '+' term              {$$ = $2;}
       | '-' term              {$$ = make_operator (NULL, '~', $2);}
       | term                  {$$ = $1;}
       | exp '+' term          {$$ = make_operator ($1, '+', $3);}
       | exp '-' term          {$$ = make_operator ($1, '-', $3);}
       ;
term   : factor                {$$ = $1;}
       | term '*' factor       {$$ = make_operator ($1, '*', $3);}
       | term '/' factor       {$$ = make_operator ($1, '/', $3);}
       ;
factor : number                {$$ = make_number ($1);}
       | variable              {$$ = make_variable ($1);}
       | '(' exp ')'           {$$ = $2;}
       ;
%%
void yyerror (char *s) {fprintf (stderr, "%s\n", s);}
int main (void) {return yyparse();}
```

Example input expression, and output equivalent fully-bracketed expression and parse tree:

((1 + 2) * 3 / (- a) - (+ b)) * (c + d + e * 4) ;

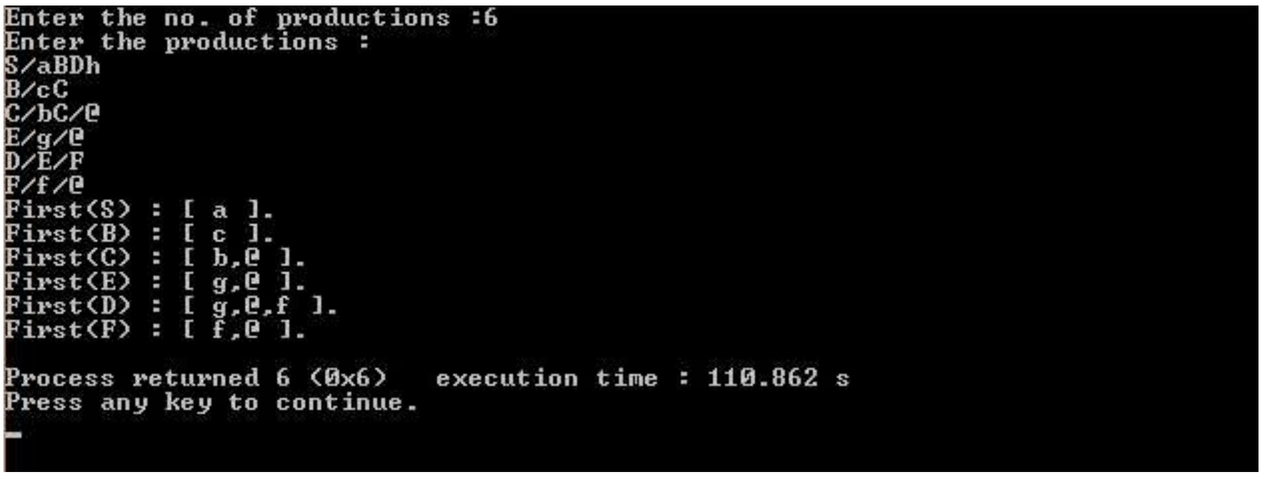
(((((1+2)*3)/(-a))-b)*((c+d)+(e*4)))

```
             4
         *
             e
     +
             d
         +
             c
 *
         b
     -
                 a
             ~
         /
                 3
             *
                     2
                 +
                     1
```

Ques. 4) Write a program to find the FIRST of non-terminals.
Ans. 4)

```c
#include<stdio.h>
#include<conio.h>
char array[10][20],temp[10];
int c,n;
void fun(int,int[]);
int fun2(int i,int j,int p[],int );

void main()
{
	int p[2],i,j;
	printf("Enter the no. of productions :");
	scanf("%d",&n);
	printf("Enter the productions :\n");
	for(i=0;i<n;i++)
		scanf("%s",array[i]);
	for(i=0;i<n;i++)
	{
		c=-1,p[0]=-1,p[1]=-1;
		fun(i,p);
		printf("First(%c) : [ ",array[i][0]);
		for(j=0;j<=c;j++)
			printf("%c,",temp[j]);
		printf("\b ].\n");
		getch();
	}
}

int fun2(int i,int j,int p[],int key)
{
	int k;
	if(!key)
	{
		for(k=0;k<n;k++)
			if(array[i][j]==array[k][0])
				break;
		p[0]=i;p[1]=j+1;
		fun(k,p);
		return 0;
	}
	else
	{
		for(k=0;k<=c;k++)
		{
			if(array[i][j]==temp[k])
				break;
		}
		if(k>c)return 1;
		else return 0;
	}
}
void fun(int i,int p[])
{
	int j,k,key;
	for(j=2;array[i][j] != NULL; j++)
	{
		if(array[i][j-1]=='/')
		{
			if(array[i][j]>= 'A' && array[i][j]<='Z')
			{
				key=0;
				fun2(i,j,p,key);
			}
			else
```

```
                        {
                                key = 1;
                                if(fun2(i,j,p,key))
                                        temp[++c] = array[i][j];
                                if(array[i][j]== '@'&& p[0]!=-1) //taking '@' as null symbol
                                {
                                        if(array[p[0]][p[1]]>='A' && array[p[0]][p[1]] <='Z')
                                        {
                                                key=0;
                                                fun2(p[0],p[1],p,key);
                                        }
                                        else
                                        if(array[p[0]][p[1]] != '/'&& array[p[0]][p[1]]!=NULL)
                                                {
                                                        if(fun2(p[0],p[1],p,key))
                                                                temp[++c]=array[p[0]][p[1]];
                                                }
                                }
                        }
                }
        }
}
```

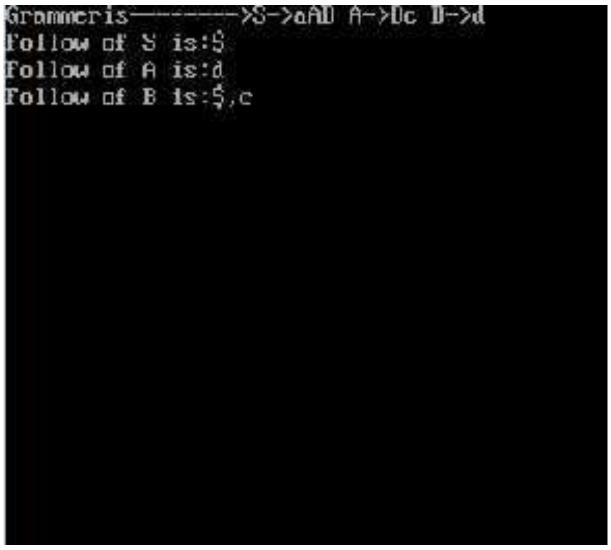


```
Enter the no. of productions :6
Enter the productions :
S/aBDh
B/cC
C/bC/@
E/g/@
D/E/F
F/f/@
First(S) : [ a ].
First(B) : [ c ].
First(C) : [ b,@ ].
First(E) : [ g,@ ].
First(D) : [ g,@,f ].
First(F) : [ f,@ ].

Process returned 6 (0x6)   execution time : 110.862 s
Press any key to continue.
```

Ques. 5) Write a program to compute the FOLLOW of non-terminals.

Ans. 5)
```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void first(char ch);
char p[10][10]={"S->aAB","A->Bc","B->d"};
char stk[10];
int top=-1;
int c=0;

int follow(char ch)
{
	int i,j,c1,count=0,x;
	if(ch=='S')
	{
		printf("$");
		//break;
	}
	for(i=0;i<3;i++)
	{
		if(ch==p[i][0])
			count++;
		for(j=3;p[i][j]!='\0';j++)
		{
			if(p[i][j]==ch && p[i][j+1]!='\0')
			{
				if(p[i][j+1]=='a' || p[i][j+1]=='c' || p[i][j+1]=='d')
					printf(",%c",p[i][j+1]);
				else first(p[i][j+1]);
			}
			if(p[i][j]==ch && p[i][j+1]=='\0')
				x=follow(p[i][0]);
		}
	}
	return count;
}

void first(char ch)
{
	int i;
	for(i=0;i<3;i++)
	{
		if(ch==p[i][0])
		{
			if(p[i][3]=='a' || p[i][3]=='c' || p[i][3]=='d')
				printf("%c",p[i][3]);
			else
			{
				top++;
				stk[top]=p[i][3];
				if(p[i][3]=='B')
				{
					if(p[i][4]=='a' || p[i][4]=='c' ||p[i][4]=='d')
						printf("%c",p[i][4]);
					else
					{
						top++;
						stk[top]=p[i][4];
					}
				}
			}
		}
	}
	while(top!=-1)
```

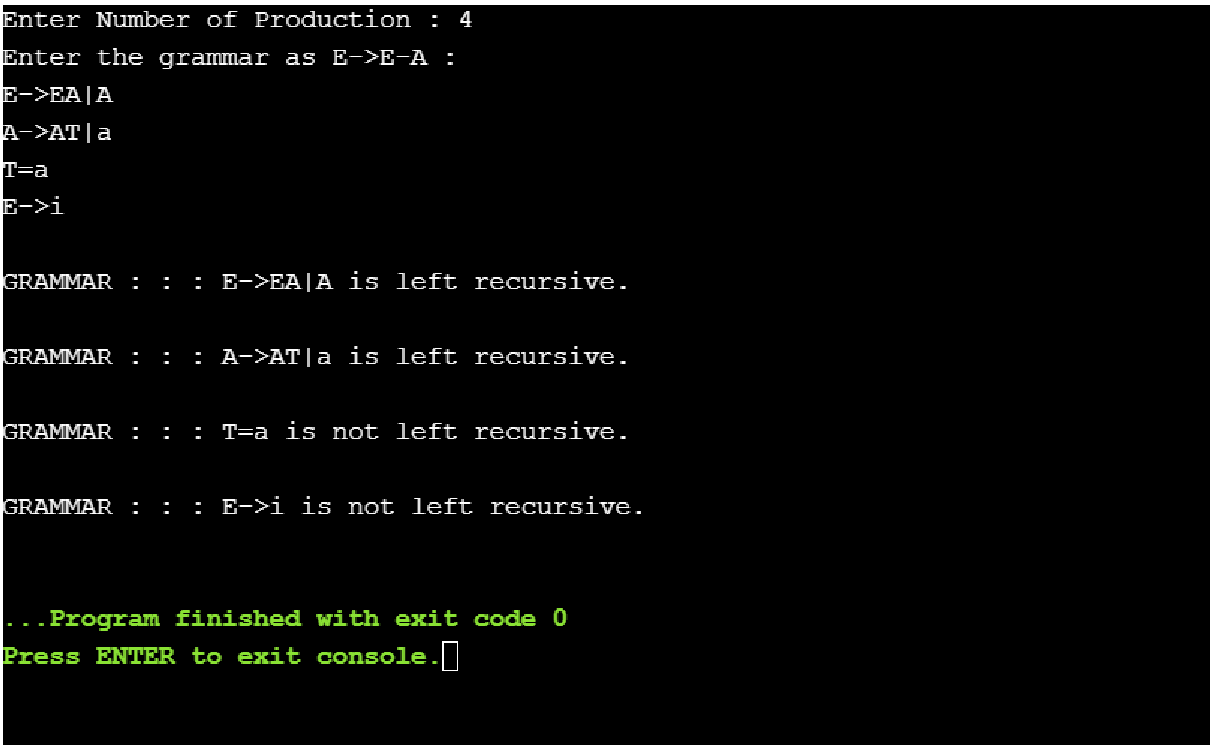
```c
	{
		char ch1;
		ch1=stk[top];
		top--;
		first(ch1);
	}
}

void main()
{
	int i;
	clrscr();
	printf("Grammeris   >S->aAB A->Bc B->d");
	for(i=0;i<3;i=i+c)
	{
		printf("\nFollow of %c is:" ,p[i][0]);
		c=follow(p[i][0]);
	}
	getch();
}
```



```
Grammeris--------->S->aAB A->Bc B->d
Follow of S is:$
Follow of A is:d
Follow of B is:$,c
```

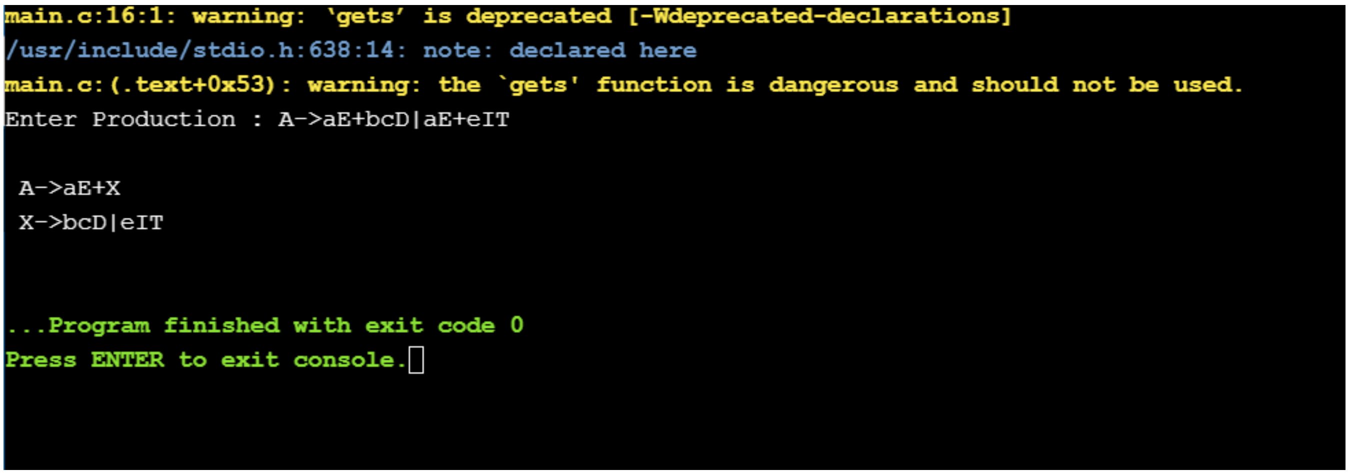
Ques. 6) Write a program to check whether the grammar is left recursive or not.

Ans. 6)
```c
#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++;
                }
                else
                        printf(" is not left recursive.\n");
                index=3;
        }
}
```


```
Enter Number of Production : 4
Enter the grammar as E->E-A :
E->EA|A
A->AT|a
T=a
E->i

GRAMMAR : : : E->EA|A is left recursive.

GRAMMAR : : : A->AT|a is left recursive.

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

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


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

Ques. 7) Write a program to remove left factoring.
Ans. 7)

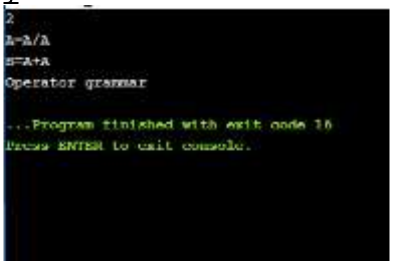
```c
#include<stdio.h>
#include<string.h>

int main()
{
	char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];
	int i,j=0,k=0,l=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("\n A->%s",modifiedGram);
	printf("\n X->%s\n",newGram);
}
```



```
main.c:16:1: warning: ‘gets’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.c:(.text+0x53): warning: the `gets' function is dangerous and should not be used.
Enter Production : A->aE+bcD|aE+eIT

 A->aE+X
 X->bcD|eIT


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

Ques. 8) Write a program to check whether a grammar is Operator precedent.
Ans. 8)

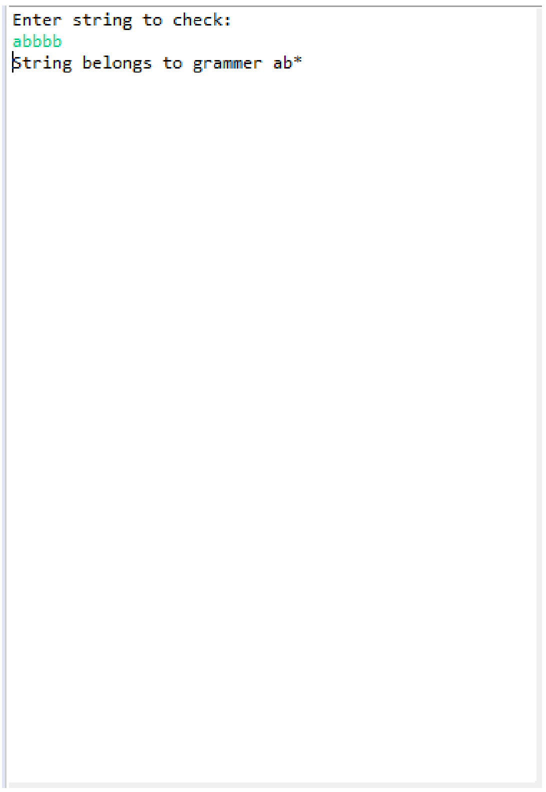
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void f()
{
    printf("Not operator grammar");
    exit(0);
}

void main()
{
    char grm[20][20], c;
    int i, n, j = 2, flag = 0;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%s", grm[i]);
    for (i = 0; i < n; i++) {
        c = grm[i][2];
        while (c != '\0') {
             if (grm[i][3] == '+' || grm[i][3] == '-'|| grm[i][3] == '*' || grm[i][3] == '/')
                    flag = 1;
             else {
                    flag = 0;
                f();
            }
            if (c == '$') {
             flag = 0;
                f();
            }
            c = grm[i][++j];
        }
    }
    if (flag == 1)
        printf("Operator grammar");
}
```



```
2
A=A/A
B=A+A
Operator grammar

...Program finished with exit code 16
Press ENTER to exit console.
```

Ques. 9) Write a program to check whether a string satisfies the condition for ab*.
Ans. 9)

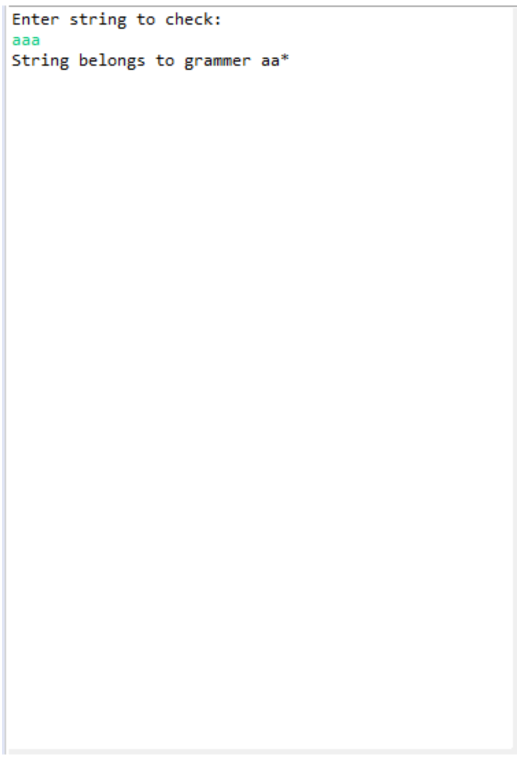
```java
import java.util.*;

public class nine{
	public static void main(String[] args) {
		Scanner s=new Scanner(System.in);
		String str=s.next();
		int count=0;
		for(int i=2;i<str.length();i++)
		{
			if(str.length()<2)
			{
				count++;
				break;
			}
			if(str.charAt(0)!='a' || str.charAt(1)!='b')
			{
				count++;
				break;
			}
			else if(str.charAt(i)!='b')
			{
				count++;
				break;
			}
			else
				count=count;
		}
		if(count==0)
			System.out.println("String belongs to grammer ab*");
		else
			System.out.println("String does not belong to grammer ab*");
	}
}
```



```
Enter string to check:
abbbb
String belongs to grammer ab*
```

Ques. 10) Write a program to check whether a string belongs to the grammar aa*.
Ans. 10)

```
import java.util.*;

public class ten{
	public static void main(String[] args) {
		Scanner s=new Scanner(System.in);
		String str=s.next();
		int count=0;
		for(int i=2;i<str.length();i++)
		{
			if(str.length()<2)
			{
				count++;
				break;
			}
			if(str.charAt(0)!='a' || str.charAt(1)!='a')
			{
				count++;
				break;
			}
			else if(str.charAt(i)!='a')
			{
				count++;
				break;
			}
			else
				count=count;
		}
		if(count==0)
			System.out.println("String belongs to grammer aa*");
		else
			System.out.println("String does not belong to grammer aa*");
	}
}
```



```
Enter string to check:
aaa
String belongs to grammer aa*
```