ADITYA

COLLEGE OF ENGINEERING & TECHNOLOGY

AN AUTONOMOUS INSTITUTION

ADB Road, Surampalem. Kakinada.Dist., (A.P.)

Department of

.....

Name: ..... PIN No. | | | | | | | | | | |

Certified that this is the bonafide record of practical work done by

Mr./Ms. .....

a student of ..... ***with PIN No.*** .....

in the ..... ***Laboratory during the year*** .....

No. of Practicals Conducted [ ] No. of Practicals Attended [ ]

Signature - of the Incharge Signature - Head of the Department

Submitted for the practical examination held on .....

Examiner-1 Examiner-2

VISION & MISSION OF THE INSTITUTE

VISION

To induce higher planes of learning by imparting technical education with

- International standards
- Applied research
- Creative Ability
- Value based instruction and to emerge as a premiere institute.

MISSION

Achieving academic excellence by providing globally acceptable technical education by forecasting technology through

- Innovative Research and development
- Industry Institute Interaction
- Empowered Manpower

VISION & MISSION OF THE DEPARTMENT

VISION

To be a recognized Computer Science and Engineering hub striving to meet the growing needs of the Industry and Society.

MISSION

M1: Imparting Quality Education through state-of-the-art infrastructure with industry Collaboration

M2: Enhance Teaching Learning Process to disseminate knowledge.

M3: Organize Skill based, Industrial and Societal Events for overall Development.

Pointer

S No	Date	Name of the Experiment	Page No	Remark

Pointer

S No	Date	Name of the Experiment	Page No	Remark

Pointer

S No	Date	Name of the Experiment	Page No	Remark

Exercise -1

AIM: Simulate the following CPU scheduling algorithms:

(a) Round Robin (b) SJF (c) FCFS (d) Priority

a) ROUND ROBIN: AIM: To simulate the CPU scheduling algorithm round-robin.

Source code:

```
#include<stdio.h>
void main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
 float awt=0,att=0,temp=0;
clrscr();
 printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]); ct[i]=bu[i];
}
printf("\nEnterno of processes the size of time slice -- ");
 scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<nil++)
```

```
 if(bu[i]!=0)
if(bu[i]<=t)
{
tat[i]=temp+bu[i]; temp=temp+bu[i]; bu[i]=0;
}
else
{
bu[i]=bu[i]-t; temp=temp+t;
}
for(i=0;i<n;i++)
{
wa[i]=tat[i]-ct[i];
att+=tat[i];
awt+=wa[i];
}
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
getch();
}
```

Output:

b). SHORTEST JOB FIRST: AIM: To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Preemption)

Source code:

```
#include<stdio.h>
#include<conio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n); for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
 for(k=i+1;k<n;k++)
 if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
```

```
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
 wtavg = wtavg + wt[i];
 tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

Output:

c) FIRST COME FIRST SERVE: AIM: To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

Source code:

```
#include<stdio.h>
#include<conio.h>
main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
 for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
 scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
 {
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
```

```
for(i=0;i<n;i++)

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

getch();

}
```

Output:

d) PRIORITY: AIM: To write a c program to simulate the CPU scheduling priority algorithm.

Source code:

```
#include<stdio.h>
 main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
clrscr();
printf("Enter the no of processes -- ");
 scanf("%d",&n); for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]); ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0]; for(i=1;i<n;i++)
if(max<bu[i]) max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t)
{
tat[i]=temp+bu[i]; temp=temp+bu[i]; bu[i]=0;
}
else
```

```
{
bu[i]=bu[i]-t; temp=temp+t;
}
for(i=0;i<n;i++) { wa[i]=tat[i]-ct[i]; att+=tat[i]; awt+=wa[i];
}
printf("\nThe Average Turnaround time is -- %f",att/n); printf("\nThe Average Waiting time is --
%f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]); getch();
}
```

Output:

Exercise -2

AIM: Simulate the following

a) Multiprogramming with a fixed number of tasks (MFT)

b) Multiprogramming with a variable number of tasks(MVT)

a) Multiprogramming with a fixed number of tasks (MFT)

Source code:

```
#include<stdio.h>
main()
{
int ms, bs, nob, ef,n, mp[10],tif=0;
int i,p=0;
printf("Enter the total memory available (in Bytes) -- ");
scanf("%d",&ms);
printf("Enter the block size (in Bytes) -- ");
scanf("%d", &bs);
nob=ms/bs;
ef= ms-nob*bs;
printf("\nEnter the number of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter memory required for process %d (in Bytes)-- ",i+1);
scanf("%d",&mp[i]);
}
printf("\nNo. of Blocks available in memory -- %d",nob);
```

```
printf("\n\nPROCESS\tMEMORY REQUIRED\t ALLOCATED\tINTERNAL FRAGMENTATION");
for(i=0;i<n && p<nob;i++)
{
printf("\n %d\t\t%d",i+1,mp[i]);
if(mp[i] > bs)
printf("\t\tNO\t\t---");
else
{
printf("\t\tYES\t%d",bs-mp[i]);
tif = tif + bs-mp[i];
p++;
}
}
if(i<n)
printf("\nMemory is Full, Remaining Processes cannot be accomodated");
printf("\n\nTotal Internal Fragmentation is %d",tif+ef);
}
```

Output:

b) Multiprogramming with a variable number of tasks (MVT)

Source code:

```
#include<stdio.h>
main()
{
int ms,mp[10],i, temp,n=0;
char ch = 'y';
printf("\nEnter the total memory available (in Bytes)-- ");
scanf("%d",&ms); 50
10 temp=ms;
for(i=0;ch=='y';i++,n++) 2,2
{
printf("\nEnter memory required for process %d (in Bytes) -- ",i+1);
scanf("%d",&mp[i]); 25,15
if(mp[i]<=temp) 15<=25
{
printf("\nMemory is allocated for Process %d ",i+1);
temp = temp - mp[i];
}
else
{
printf("\nMemory is Full");
break;
}
printf("\nDo you want to continue(y/n) -- ");
scanf(" %c", &ch);
```

```
}
printf("\n\nTotal Memory Available -- %d", ms);
printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED ");
for(i=0;i<n;i++)
printf("\n \t%d\t\t%d",i+1,mp[i]);
printf("\n\nTotal Memory Allocated is %d",ms-temp);
printf("\nTotal External Fragmentation is %d",temp);
}
```

Output:

Exercise -3

AIM: Simulate the following page replacement algorithms:

a) FIFO b) LRU c) LFU

Source code:

```
#include<stdio.h>
int p[30],a[30],m,a[30];
int fifo(int);
main()
{
int i,n1,n2,pf1,pf2;
printf("***FIFO***");
printf("enter number of pages\n");
scanf("%d",&m);
printf("enter first number of frames\n");
scanf("%d",&n1);
printf("enter second number of frames\n");
scanf("%d",&n2);
printf("enter pages inorder to be loaded\n");
for(i=0;i<m;i++)
scanf("%d",&p[i]);
printf("\nthe pagefaults for %d pageframe is \n",n1);
pf1=fifo(n1);
printf("\nthe pagefaults for %d pageframe is \n",n2);
pf2=fifo(n2);
if(pf1<pf2)
printf("\nbeladys anamoly exists\n");
```

```
else
printf("\nbeladys anamoly doesnot exists\n");
}
int fifo(int n)
{
int i,j,flag,pfault=0;
for(i=0;i<n;i++)
a[i]=-1;
for(i=0;i<m;i++)
{
flag=0;
for(j=0;j<n;j++)
if(a[j]==p[i])
{
flag=1;
break;
}
if(flag==1)
continue;
else
{
for(j=0;j<n-1;j++)
{
a[j]=a[j+1];
}
a[j]=p[i];
```

```
pfault++;
}
printf("\na= ");
for(j=0;j<n;j++)
printf("%3d",a[j]);
}
printf("\n number of pagefaults are %d",pfault);
return(pfault);
}
```

Output:

b) Write a program to simulate page replacement algorithm for LRU

Source code:

```
#include<stdio.h>
int findLRU(int time[], int n)
{
 int i, minimum = time[0], pos = 0;
 for(i = 1; i < n; ++i)
 {
 if(time[i] < minimum){
 minimum = time[i];
 pos = i;
 }
 }
 return pos;
}
int main()
{
 int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j,
pos, faults = 0;
 printf("Enter number of frames: ");
 scanf("%d", &no_of_frames);
 printf("Enter number of pages: ");
 scanf("%d", &no_of_pages);
 printf("Enter reference string: ");
 for(i = 0; i < no_of_pages; ++i){
 scanf("%d", &pages[i]);
```

```
}
for(i = 0; i < no_of_frames; ++i){
frames[i] = -1;
}
for(i = 0; i < no_of_pages; ++i)
{
flag1 = flag2 = 0;
for(j = 0; j < no_of_frames; ++j)
{
if(frames[j] == pages[i])
{
counter++;
time[j] = counter;
flag1 = flag2 = 1;
break;
}
}
if(flag1 == 0)
{
for(j = 0; j < no_of_frames; ++j)
{
if(frames[j] == -1)
{
counter++;
faults++;
frames[j] = pages[i];
```

```
time[j] = counter;
printf("\ntime = %d\n",time[j]);
flag2 = 1;
break;
}
}
}
if(flag2 == 0){
pos = findLRU(time, no_of_frames);
counter++;
faults++;
frames[pos] = pages[i];
time[pos] = counter;
}
printf("\n");
for(j = 0; j < no_of_frames; ++j)
{
printf("%d\t", frames[j]);
}
}
printf("\n\nTotal Page Faults = %d", faults);
return 0;
}
```

Output

c) AIM: Write a program to simulate page replacement algorithm for optimal by least recently

Source code:

```
#include<stdio.h>
int main()
{
 int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k,
pos, max, faults = 0;
 printf("Enter number of frames: ");
 scanf("%d", &no_of_frames);
 printf("Enter number of pages: ");
 scanf("%d", &no_of_pages);
 printf("Enter page reference string: ");
 for(i = 0; i < no_of_pages; ++i){
 scanf("%d", &pages[i]);
 }
for(i = 0; i < no_of_frames; ++i)
 {
 frames[i] = -1;
 }

 for(i = 0; i < no_of_pages; ++i)
 {
 flag1 = flag2 = 0;
 for(j = 0; j < no_of_frames; ++j)
 {
```

```
if(frames[j] == pages[i])
{
flag1 = flag2 = 1;
break;
}
}
if(flag1 == 0)
{
for(j = 0; j < no_of_frames; ++j)
{
if(frames[j] == -1)
{
faults++;
frames[j] = pages[i];
flag2 = 1;
break;
}
}
}

if(flag2 == 0)
{
flag3 =0;
for(j = 0; j < no_of_frames; ++j)
{
temp[j] = -1;
```

```
for(k = i + 1; k < no_of_pages; ++k)
{
if(frames[j] == pages[k])
{
temp[j] = k;
break;
} } }
for(j = 0; j < no_of_frames; ++j)
{
if(temp[j] == -1)
{
pos = j;
flag3 = 1;
break;
} }
if(flag3 ==0)
{
max = temp[0];
pos = 0;
for(j = 1; j < no_of_frames; ++j)
{
if(temp[j] > max)
{
max = temp[j];
pos = j;
}
```

```
}
}
frames[pos] = pages[i];
faults++;
}
printf("\n");
for(j = 0; j < no_of_frames; ++j)
{
printf("%d\t", frames[j]);
}
}
printf("\n\nTotal Page Faults = %d", faults);
return 0;
}
```

Output:

Exercise -4

AIM: Write a C program that illustrates two processes communicating using shared memory

Source code:

```
//Program 1: This program creates a shared memory segment, attaches itself to it and then writes some content into the shared memory segment.
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT); //creates shared memory segment with key 2345, having size 1024 bytes. IPC_CREAT is used to create the shared segment if it does not exist. 0666 are the permisions on the shared segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory); //this prints the address where the segment is attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
```

```
printf("You wrote : %s\n",(char *)shared_memory);
}
```

Output

//Program 2: This program attaches itself to the shared memory segment created in Program 1. Finally, it reads the content of the shared memory

Source code:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
```

```
shmid=shmget((key_t)2345, 1024, 0666);

printf("Key of shared memory is %d\n",shmid);

shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment

printf("Process attached at %p\n",shared_memory);

printf("Data read from shared memory is : %s\n",(char *)shared_memory);

}
```

How it works?

shmget() here generates the identifier of the same segment as created in Program 1. Remember to

give the same key value. The only change is, do not write IPC_CREAT as the shared memory segment is already created. Next, shmat() attaches the shared segment to the current process.

After that, the data is printed from the shared segment. In the output, you will see that it is the same data that you have written while executing the Program 1.

Output

Exercise -5

AIM: Write a C program to simulate producer and consumer problem using semaphores

Source code:

```
#include<stdio.h>
int mutex=1,full=0,empty=2,x=0;
main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
while(1)
{
printf("\nENTER YOUR CHOICE\n");
scanf("%d",&n);
switch(n)
{
case 1: if((mutex==1)&&(empty!=0))
producer();
else
printf("BUFFER IS FULL");
break;
case 2: if((mutex==1)&&(full!=0))
consumer();
```

```
else
printf("BUFFER IS EMPTY");
break;
case 3: exit(0);
break;
}
 }
 }
int wait(int s)
{
return(--s);
}
int signal(int s)
{
return(++s);
}
void producer()
{
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("\nproducer produces the item%d",x);
mutex=signal(mutex);
}
void consumer()
```

```
{
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\n consumer consumes item%d",x);
x--;
mutex=signal(mutex);
}
```

Output:

Exercise -6

AIM: Simulate Bankers Algorithm for DeadLock Avoidance

Source code:

```
#include<stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n,r;
void input();
void show();
void cal();
int main()
{
int i,j;
printf("********** Banker's Algo ************\n");
input();
show();
cal();
return 0;
}
void input()
{
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
```

```
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&max[i][j]);
}
}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&alloc[i][j]);
}
}
printf("Enter the available Resources\n");
for(j=0;j<r;j++)
{
scanf("%d",&avail[j]);
}
}
void show()
{
```

```
int i,j;
printf("Process\t Allocation\t Max\t Available\t");
for(i=0;i<n;i++)
{
printf("\nP%d\t ",i+1);
for(j=0;j<r;j++)
{
printf("%d ",alloc[i][j]);
}
printf("\t");
for(j=0;j<r;j++)
{
printf("%d ",max[i][j]);
}
printf("\t");
if(i==0)
{
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}
}
}
void cal()
{
int finish[100],temp,need[100][100],flag=1,k,c1=0;
int safe[100];
```

```
int i,j;
for(i=0;i<n;i++)
{
finish[i]=0;
}
//find need matrix
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
need[i][j]=max[i][j]-alloc[i][j];
}
}
printf("\n");
while(flag)
{
flag=0;
for(i=0;i<n;i++)
{
int c=0;
for(j=0;j<r;j++)
{
if((finish[i]==0)&&(need[i][j]<=avail[j]))
{
c++;
if(c==r)
```

```
{
for(k=0;k<r;k++)
{
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1;
}
printf("P%d->",i);
if(finish[i]==1)
{
i=n;
}
}
}
}
}
}
for(i=0;i<n;i++)
{
if(finish[i]==1)
{
c1++;
}
else
{
printf("P%d->",i);
```

```
}
}
if(c1==n)
{
printf("\n The system is in safe state");
}
else
{
printf("\n Process are in dead lock");
printf("\n System is in unsafe state");
}
}
```

Output:

Exercise -7

AIM: Simulate Bankers Algorithm for Deadlock Prevention.

Source code:

```
#include< stdio.h>
void main()
{
int allocated[15][15],max[15][15],need[15][15],avail[15],tres[15],work[15],flag[15];
int pno,rno,i,j,prc,count,t,total;
count=0;
printf("\n Enter number of process:");
scanf("%d",&pno);
printf("\n Enter number of resources:");
scanf("%d",&rno);
for(i=1;i<=pno;i++)
{
flag[i]=0;
}
printf("\n Enter total numbers of each resources:");
for(i=1;i<= rno;i++)
scanf("%d",&tres[i]);
printf("\n Enter Max resources for each process:");
for(i=1;i<= pno;i++)
{
printf("\n for process %d:",i);
for(j=1;j<= rno;j++)
scanf("%d",&max[i][j]);
```

```
}
printf("\n Enter allocated resources for each process:");
for(i=1;i<= pno;i++)
{
printf("\n for process %d:",i);
for(j=1;j<= rno;j++)
scanf("%d",&allocated[i][j]);
}
printf("\n available resources:\n");
for(j=1;j<= rno;j++)
{
avail[j]=0;
total=0;
for(i=1;i<= pno;i++)
{
total+=allocated[i][j];
}
avail[j]=tres[j]-total;
work[j]=avail[j];
printf(" %d \t",work[j]);
}
do
{
for(i=1;i<= pno;i++)
{
for(j=1;j<= rno;j++)
```

```
{
need[i][j]=max[i][j]-allocated[i][j];
}
}
printf("\n Allocated matrix Max need");
for(i=1;i<= pno;i++)
{
printf("\n");
for(j=1;j<= rno;j++)
{
printf("%4d",allocated[i][j]);
}
printf("|");
for(j=1;j<= rno;j++)
{
printf("%4d",max[i][j]);
}
printf("|");
for(j=1;j<= rno;j++)
{
printf("%4d",need[i][j]);
}
}
prc=0;
for(i=1;i<= pno;i++)
{
```

```
if(flag[i]==0)
{
prc=i;
for(j=1;j<= rno;j++)
{
if(work[j]< need[i][j])
{
prc=0;
break;
}
}
}
if(prc!=0)
break;
}
if(prc!=0)
{
printf("\n Process %d completed",i);
count++;
printf("\n Available matrix:");
for(j=1;j<= rno;j++)
{
work[j]+=allocated[prc][j];
allocated[prc][j]=0;
max[prc][j]=0;
flag[prc]=1;
```

```
printf(" %d",work[j]);
}
}
}while(count!=pno&&prc!=0);
if(count==pno)
printf("\nThe system is in a safe state!!");
else
printf("\nThe system is in an unsafe state!!");
}
```

Output:

Exercise -8

AIM: Write a C Program to identify different types of tokens in a given program

Source code:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// Returns 'true' if the character is a DELIMITER. bool isDelimiter(char ch)
{
if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == ',' || ch == ';' || ch == '>' || ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
ch == '[' || ch == ']' || ch == '{' || ch == '}'|| ch=='%') return (true);
return (false);
}
// Returns 'true' if the character is an OPERATOR. bool isOperator(char ch)
{
if (ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == '>' || ch == '<' || ch == '=')
return (true); return (false);
}
// Returns 'true' if the string is a VALID IDENTIFIER. bool validIdentifier(char* str)
{
if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
str[0] == '3' || str[0] == '4' || str[0] == '5' ||
str[0] == '6' || str[0] == '7' || str[0] == '8' || str[0] == '9' || isDelimiter(str[0]) == true) return (false);
return (true);
```

```
}
// Returns 'true' if the string is a KEYWORD. bool isKeyword(char* str)
{
if (!strcmp(str, "if") || !strcmp(str, "else") ||
!strcmp(str, "while") || !strcmp(str, "do") ||
!strcmp(str, "break") ||
!strcmp(str, "continue") || !strcmp(str, "int")
|| !strcmp(str, "double") || !strcmp(str, "float")
|| !strcmp(str, "return") || !strcmp(str, "char")
|| !strcmp(str, "case") || !strcmp(str, "char")
|| !strcmp(str, "sizeof") || !strcmp(str, "long")
|| !strcmp(str, "short") || !strcmp(str, "typedef")
|| !strcmp(str, "switch") || !strcmp(str, "unsigned")
|| !strcmp(str, "void") || !strcmp(str, "static")
|| !strcmp(str, "struct") || !strcmp(str, "goto")) return (true);
return (false);
}
// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
int i, len = strlen(str);
if (len == 0)
return (false); for (i = 0; i < len; i++) {
if (str[i] != '0' && str[i] != '1' && str[i] != '2'
&& str[i] != '3' && str[i] != '4' && str[i] != '5'
&& str[i] != '6' && str[i] != '7' && str[i] != '8'
```

```
&& str[i] != '9' || (str[i] == '-' && i > 0)) return (false);
}
return (true);
}
// Returns 'true' if the string is a REAL NUMBER. bool isRealNumber(char* str)
{
int i, len = strlen(str); bool hasDecimal = false;
if (len == 0)
return (false); for (i = 0; i < len; i++) {
if (str[i] != '0' && str[i] != '1' && str[i] != '2'
&& str[i] != '3' && str[i] != '4' && str[i] != '5'
&& str[i] != '6' && str[i] != '7' && str[i] != '8'
&& str[i] != '9' && str[i] != '.' ||
(str[i] == '-' && i > 0)) return (false);
if (str[i] == '.')
hasDecimal = true;
}
return (hasDecimal);
}
// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
int i;
char* subStr = (char*)malloc(
sizeof(char) * (right - left + 2));
for (i = left; i <= right; i++)
```

```
subStr[i - left] = str[i]; subStr[right - left + 1] = '\0'; return (subStr);
}
// Parsing the input STRING. void parse(char* str)
{
int left = 0, right = 0; n int len = strlen(str);
while (right <= len && left <= right) {
if (isDelimiter(str[right]) == false)
right++;
if (isDelimiter(str[right]) == true && left == right) { if (isOperator(str[right]) == true)
printf("'%c' IS AN OPERATOR\n", str[right]);
right++; left = right;
} else if (isDelimiter(str[right]) == true && left != right
|| (right == len && left != right)) { char* subStr = subString(str, left, right - 1);
if (isKeyword(subStr) == true)
printf("'%s' IS A KEYWORD\n", subStr);
else if (isInteger(subStr) == true)
printf("'%s' IS AN INTEGER\n", subStr);
else if (isRealNumber(subStr) == true)
printf("'%s' IS A REAL NUMBER\n", subStr);
else if (validIdentifier(subStr) == true
&& isDelimiter(str[right - 1]) == false) printf("'%s' IS A VALID IDENTIFIER\n", subStr);
else if (validIdentifier(subStr) == false
&& isDelimiter(str[right - 1]) == false) printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
left = right;
}
}
```

```
return;
}
// DRIVER FUNCTION
int main()
{
// maximum length of string is 100 here char str[100];


scanf("%[^\n]s",str);
printf("%s",str);
parse(str); // calling the parse function
return (0);
}
```

Output:

Exercise -9

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

Source code:

```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
```

```
goto {printf("\n\t%s is a keyword",yytext);}

"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}

{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}

\{  {if(!COMMENT)printf("\n BLOCK BEGINS");}

\}  {if(!COMMENT)printf("BLOCK ENDS ");}

{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}

\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}

\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}

\( ECHO;

= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}

\<= |

\>= |

\< |

== |

\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%

int main(int argc, char **argv)

{

FILE *file;

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

if(!file)

{

printf("could not open the file");

exit(0);

}
```

```
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

INPUT:

```
//var.c
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

Output:

Exercise -10

AIM: Write a C- program to simulate lexical analyser to validate a given input operator

Source code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s[5];
printf("\n Enter any operator:"); gets(s);
switch(s[0])
{
case'>': if(s[1]=='=')
printf("\n Greater than or equal"); else
printf("\n Greater than"); break;
case'<': if(s[1]=='=')
printf("\n Less than or equal"); else
printf("\nLess than"); break;
case'=': if(s[1]=='=') printf("\nEqual to"); else printf("\nAssignment"); break;
case'!': if(s[1]=='=') printf("\nNot Equal"); else
printf("\n Bit Not"); break;
case'&': if(s[1]=='&')
printf("\nLogical AND"); else
printf("\n Bitwise AND"); break;
case'|': if(s[1]=='|') printf("\nLogical OR"); else
printf("\nBitwise OR"); break;
case'+': printf("\n Addition"); break;
```

```
case'-': printf("\nSubstraction"); break;

case'*': printf("\nMultiplication"); break;

case'/': printf("\nDivision"); break;

case'%': printf("Modulus"); break;

default: printf("\n Not a operator");

}

getch();

}
```

Output:

Exercise -11

AIM: Write a C-program to implement the brute force technique of top down parser

Source code:

```
#include <stdio.h>
#include <string.h>
 #define MAX 100
/* try to find the given pattern in the search string */manu manu
int bruteForce(char *search, char *pattern, int slen, int plen) { int i, j, k;
for (i = 0; i <= slen - plen; i++) {
for (j = 0, k = i; (search[k] == pattern[j]) && (j < plen); j++, k++);
if (j == plen)
return j;
}
return -1;}
int main() {
char searchStr[MAX], pattern[MAX]; int res;
printf("Enter Search String:");
fgets(searchStr, MAX, stdin);
printf("Enter Pattern String:");
fgets(pattern, MAX, stdin);
searchStr[strlen(searchStr) - 1] = '\0';
pattern[strlen(pattern) - 1] = '\0';
res = bruteForce(searchStr, pattern, strlen(searchStr), strlen(pattern)); if (res == -1) {
printf("Search pattern is not available\n");
} else {
printf("Search pattern available at the location %d\n", res);
```

```
}

return 0;

}
```

Output:

Exercise -12

AIM: Write a c program to implement a recurssive descent parser

Source code:

```
 #include<stdio.h>
#include<string.h>
int E(),Edash(),T(),Tdash(),F(); char *ip;
char string[50]; int main()
{
printf("Enter the string\n"); scanf("%s",string); ip=string;
printf("\n\nInput\tAction\n     \n");
if(E() && ip=="\0"){
printf("\n        \n");
printf("\n String is successfully parsed\n");
}
else{
printf("\n        \n"); printf("Error in parsing String\n");
}
}
int E()
{
printf("%s\tE->TE' \n",ip); if(T())
{
if(Edash())
{
return 1;
}
```

```
else return 0;
}
else return 0;
}
int Edash()
{
if(*ip=='+')
{
printf("%s\tE'->+TE' \n",ip); ip++;
if(T())
{
if(Edash())
{
return 1;
}
else return 0;
}
else return 0;
}
else
{
printf("%s\tE'->^ \n",ip); return 1;
}
}
int T()
{
```

```
printf("%s\tT->FT' \n",ip); if(F())
{
if(Tdash())
{
return 1;
}
else return 0;
}
else return 0;
}
int Tdash()
{
if(*ip=='*')
{
printf("%s\tT'->*FT' \n",ip); ip++;
if(F())
{
if(Tdash())
{
return 1;
}
else return 0;
}
else return 0;
}
else
```

```
{
printf("%s\tT'->^ \n",ip); return 1;
}
}
int F()
{
if(*ip=='(')
{
printf("%s\tF->(E) \n",ip); ip++;
if(E())
{
if(*ip==')')
{
ip++; return 0;
}
else return 0;
}
else return 0;
}
else if(*ip=='i')
{
ip++;
printf("%s\tF->id \n",ip); return 1;
}
else return 0;
}
```

Output:

Exercise -13

AIM: Write a C program to compute the first and follow sets of given grammer

```
// C program to calculate the First and
// Follow sets of a given grammar
#include<stdio.h>
#include<ctype.h>
#include<string.h>

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

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

int count, n = 0;
// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100]; int m = 0;

// Stores the production rules
char production[10][10];
```

```
char f[10], first[10]; int k;
char ck; int e;
int main(int argc, char **argv)
{
        int jm = 0; int km = 0; int i, choice; char c, ch; count = 8;
        // The Input grammar
        strcpy(production[0], "E=TR");
        strcpy(production[1], "R=+TR");
        strcpy(production[2], "R=#");
        strcpy(production[3], "T=FY");
        strcpy(production[4], "Y=*FY");
        strcpy(production[5], "Y=#");
        strcpy(production[6], "F=(E)");
        strcpy(production[7], "F=i");

        int kay;
        char done[count]; int ptr = -1;

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

```
int point1 = 0, point2, xxx;
for(k = 0; k < count; k++)
{
        c = production[k][0]; point2 = 0;
        xxx = 0;
        // Checking if First of c has
        // already been calculated
        for(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[count];
        ptr = -1;
        // Initializing the calc_follow array
        for(k = 0; k < count; k++) {
                for(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(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++;
}
}
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 vend 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:

Exercise -14

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

```
//Eliminate Left Recursion in C Program
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define SIZE 20
int 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");
}
```

Output:

//Left Factoring Program in 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("\nGrammar Without Left Factoring : : \n");
    printf(" A->%s",modifiedGram);
    printf("\n X->%s\n",newGram);
}
```

Output:

Exercise -15

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

Source code:

```
#include <stdio.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);
}
 int 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");
    }
}
```

Output:

Exercise -16

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

Aim : To write a C program for implementation of LR parsing to accept a given input string

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int z = 0, i = 0, j = 0, c = 0; //Global Variables// Modify array size to increase
char a[16], ac[20], stk[15], act[10]; // length of string to be parsed
// This Function will check whether
// the stack contain a production rule
// which is to be Reduce.
// Rules can be E->2E2 , E->3E3 , E->4
void check()
{
 // Copying string to be printed as action
 strcpy(ac,"REDUCE TO E -> ");
 // c=length of input string
 for(z = 0; z < c; z++)
 {
 //checking for producing rule E->4
 if(stk[z] == '4')
 {
 printf("%s4", ac);
 stk[z] = 'E';
 stk[z + 1] = '\0';
```

```
//printing action
 printf("\n$%s\t%s$\t", stk, a);
 }
 }
 for(z = 0; z < c - 2; z++)
 {
 //checking for another production
 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++)
 {
 //checking for E->3E3
 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 ; //return to main
}
//Driver Function
int main()
{
 printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");     // a is input string
 strcpy(a,"32423");        // strlen(a) will return the length of a to c
c=strlen(a);       // "SHIFT" is copied to act to be printed
strcpy(act,"SHIFT");       // This will print Labels (column name)
printf("\nstack \t input \t action");
 // This will print the initial
 // values of stack and input
 printf("\n$\t%s$\t", a);
 // This will Run upto length of input string
 for(i = 0; j < c; i++, j++)
 {                                  // Printing action
 printf("%s", act);
// Pushing into stack
 stk[i] = a[j];
stk[i + 1] = '\0';
```

```
// Moving the pointer
 a[j]=' ';
// Printing action
 printf("\n$%s\t%s$\t", stk, a);
// Call check function ..which will
 // check the stack whether its contain
 // any production or not
 check();
 }
// Rechecking last time if contain
 // any valid production then it will
 // replace otherwise invalid
 check();
// if top of the stack is E(starting symbol)
 // then it will accept the input
 if(stk[0] == 'E' && stk[1] == '\0')
 printf("Accept\n");
 else //else reject
 printf("Reject\n");
}
```

Output :

Exercise -17

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

Source code:

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
  {
 puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");
    for(k=0,i=0; j<c; k++,i++,j++)
     {
       if(a[j]=='i' && a[j+1]=='d')
        {
          stk[i]=a[j];
          stk[i+1]=a[j+1];
          stk[i+2]='\0';
          a[j]=' ';
          a[j+1]=' ';
          printf("\n$%s\t%s$\t%sid",stk,a,act);
          check();
        }
      else
```

```
        {
          stk[i]=a[j];
          stk[i+1]='\0';
          a[j]=' ';
          printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
          check();
        }
      }

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

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

Output:

Exercise -18

Aim:Simulate the calculator using LEX and YACC tool

Source code:

cal.l

```
DIGIT [0-9]+
 %option noyywrap
%%
{DIGIT}              { yylval=atof(yytext); return NUM;}
 \n|.                { return yytext[0];}
 %%
```

cal.y

```
%{
            #include<ctype.h>
            #include<stdio.h>
            #define YYSTYPE double
%}
%token NUM
%left '+' '-' %left '*' '/'
%right UMINUS
 %%
Statment:E { printf("Answer: %g \n", $$); }
           |Statment '\n'
           ;
E          : E'+'E { $$ = $1 + $3; }
           | E'-'E { $$=$1-$3; }
           | E'*'E { $$=$1*$3; }
            | E'/'E { $$=$1/$3; }
           | NUM
; %%
```

Output:

Additional Experiments

1. Aim: Write a C program to implement FCFS disk scheduling algorithm

Source code:

```
#include  <stdio.h>
#include <math.h>
int size = 8;
void FCFS(int arr[],int head)
{
int seek_count = 0;
        int cur_track, distance;
        for(int i=0;i<size;i++)
        {
        cur_track = arr[i];          // calculate absolute distance
        distance = fabs(head - cur_track);      // increase the total count
        seek_count += distance;         // accessed track is now new head
        head = cur_track;
        }
printf("Total number of seek operations: %d\n",seek_count);
        // Seek sequence would be the same
        // as request array sequence
        printf("Seek Sequence is\n");
for (int i = 0; i < size; i++) {
                printf("%d\n",arr[i]);
        }}
int main()
{                   // request array
int arr[8] = { 176, 79, 34, 60, 92, 11, 41, 114 };
        int head = 50;
        FCFS(arr,head);
        return 0;
}
```

Output:

2. Aim: Write a C program to generate three address code

Source code:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

struct three

{

char data[10],temp[7];

}s[30];

void main()

{

char d1[7],d2[7]="t";

int i=0,j=1,len=0;

FILE *f1,*f2;

clrscr();

f1=fopen("sum.txt","r");

f2=fopen("out.txt","w");

while(fscanf(f1,"%s",s[len].data)!=EOF)

len++;

itoa(j,d1,7);

strcat(d2,d1);

strcpy(s[j].temp,d2);

strcpy(d1,"");

strcpy(d2,"t");

if(!strcmp(s[3].data,"+"))

{
```

```
fprintf(f2,"%s=%s+%s",s[j].temp,s[i+2].data,s[i+4].data);

j++;

}

else if(!strcmp(s[3].data,"-"))

{

fprintf(f2,"%s=%s-%s",s[j].temp,s[i+2].data,s[i+4].data);

j++;

}

for(i=4;i<len-2;i+=2)

{

itoa(j,d1,7);

strcat(d2,d1);

strcpy(s[j].temp,d2);

if(!strcmp(s[i+1].data,"+"))

fprintf(f2,"\n%s=%s+%s",s[j].temp,s[j-1].temp,s[i+2].data);

else if(!strcmp(s[i+1].data,"-"))

fprintf(f2,"\n%s=%s-%s",s[j].temp,s[j-1].temp,s[i+2].data);

strcpy(d1,"");

strcpy(d2,"t");

j++;

}

fprintf(f2,"\n%s=%s",s[0].data,s[j-1].temp);

fclose(f1);

fclose(f2);

getch();

}
```

Input: sum.txt

out = in1 + in2 + in3 - in4

Output :