CSL411 COMPILER LAB COURSE OUTCOMES

CSL411.1	Implement lexical analyzer using the tool LEX. (Cognitive Knowledge Level: Apply)
CSL411.2	Implement Syntax analyzer using the tool YACC. (Cognitive Knowledge Level: Apply)
CSL411.3	Design NFA and DFA for a problem and write programs to perform operations on it. (Cognitive Knowledge Level: Apply)
CSL411.4	Design and Implement Top-Down parsers. (Cognitive Knowledge Level: Apply)
CSL411.5	Design and Implement Bottom-Up parsers. (Cognitive Knowledge Level: Apply)
CSL411.6	Develop intermediate code for simple expressions. (Cognitive Knowledge Level: Apply)

CO-PO MAPPING

Course outcomes	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO 9	PO1O	PO 11	PO 12
CSL411.1	3	3	2	2	0			0		0		2
CSL411.2	3	3	2	3	0			0		0		2
CSL411.3	3	3	3	3				0		0		2
CSL411.4	3	3	3	3				0		0		2
CSL411.5	3	3	2	2				0		0		2
CSL411.6	3	3	3	3				0		0		2

CO-PSO MAPPING

Course outcomes	PSO 1	PSO 2	PSO 3
CSL411.1	3	3	
CSL411.2	3	3	
CSL411.3	3	3	
CSL411.4	3	3	
CSL411.5	3	3	
CSL411.6	3	3	

INDEX

SL. No.	NAME OF EXPERIMENT	DATE	PAGE No.	REMARKS
1	Epsilon(ε) - Closure of States of NFA			
2	Convert NFA with ε-Transitions to NFA			
3	NFA to DFA Conversion			
4	DFA Minimization			
5	Study of LEX and YACC Tools			
6	Implementation of a Lexical Analyzer			
7	Lex Program to Exclude Name Prefix			
8	YACC Program to Recognize Valid Variables			
9	Calculator with LEX and YACC			
10	BNF to YACC Conversion			
11	FOR Statement Syntax Validator			
12	Operator Precedence Parser			
13	Simulation of First and Follow of Grammar			
14	Recursive Descent Parser			
15	Shift Reduce Parser			
16	Intermediate Code Generator			

PROGRAM

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

struct node
{
   int st;
   struct node *link;
};

void findclosure(int,int);
void insert_trantbl(int ,char, int);
int findalpha(char);
void print_e_closure(int);
static int
set[20],nostate,noalpha,s,notransition,c,r,buffer[20
];
char alphabet[20];
static int e_closure[20][20]={0};
struct node * transition[20][20]={NULL};

void main()
{
   int i,j,k,m,t,n;
   struct node *temp;
   printf("Enter the number of alphabets\n");
   scanf("%d",&noalpha);
   getchar();
   printf("NOTE:- [ use letter e as epsilon]\n");
   printf("NOTE:- [e must be last character ,if it is
present]\n");
   printf("\nEnter alphabets?\n");
   for(i=0;i<noalpha;i++)
   {
      alphabet[i]=getchar();
      getchar();
   }
   printf("\nEnter the number of states?\n");
   scanf("%d",&nostate);
   printf("\nEnter no of transition?\n");
   scanf("%d",&notransition);
   printf("NOTE:- [Transition is in the form–>
qno alphabet qno]\n",notransition);
   printf("NOTE:- [States number must be greater
than zero]\n");
   printf("\nEnter transition?\n");
   for(i=0;i<notransition;i++)
   {
      scanf("%d %c%d",&r,&c,&s);
      insert_trantbl(r,c,s);
   }
   printf("\n");
   printf("e-closure of states……\n");
   printf("———————————\n");
   for(i=1;i<=nostate;i++)
   {
      c=0;
      for(j=0;j<20;j++)
      {
         buffer[j]=0;
         e_closure[i][j]=0;
      }
      findclosure(i,i);
      printf("\ne-closure(q%d): ",i);
      print_e_closure(i);
      }
}

void findclosure(int x,int sta)
{
   struct node *temp;
   int i;
   if(buffer[x])
      return;
   e_closure[sta][c++]=x;
   buffer[x]=1;
   if(alphabet[noalpha-1]=='e' &&
transition[x][noalpha-1]!=NULL)
   {
      temp=transition[x][noalpha-1];
      while(temp!=NULL)
      {
         findclosure(temp->st,sta);
         temp=temp->link;
      }
   }
}

void insert_trantbl(int r,char c,int s)
{
   int j;
   struct node *temp;
   j=findalpha(c);
   if(j==999)
   {
```

Date :

EXPERIMENT - 1

EPSILON(ε) - CLOSURE OF STATES OF NFA

AIM

To write a C program to find ε – closure of all states of any given NFA with ε transition.

ALGORITHM

1. Start.
2. Declare necessary headers.
3. Create an empty 2D array ‘states’ to store the states of the NFA.
4. Read the number of states ‘n’.
5. Enter all the states and store them in ‘states[]’.
6. Open the transition table file ‘input.txt’. The file contains transitions in the format: ‘state1 input_symbol state2’ (where ‘input_symbol’ can be ‘ε’).
7. For each state in ‘states[]’:
    a. Set ‘i = 0’ as an index for ‘result[]’.
    b. Initialize a copy of the current state to ‘copy’.
    c. Create an empty array ‘result[]’ to store the epsilon closure for the current state.
    d. Add the current state itself to its epsilon closure
    e. For each transition in the file (‘state1’, ‘input’, ‘state2’):
        i. If ‘state1’ matches the current state and the ‘input’ is ‘ε’ (epsilon transition):
        ii. Add ‘state2’ to the current state's epsilon closure (‘result[]’).
        iii. Update the current state to ‘state2’ and repeat, adding all states reachable by epsilon transitions.
    f. Call display function to print the set of states in ‘result[]’.
    g. Use ‘rewind()’ to reset the file pointer to the beginning of the file so that it can be read again for the next state.
8. Close the file.
9. Stop.

```
        printf("error\n");
        exit(0);
    }
    temp=(struct node *)malloc(sizeof(struct
node));
    temp->st=s;
    temp->link=transition[r][j];
    transition[r][j]=temp;
}

int findalpha(char c)
{
    int i;
    for(i=0;i<noalpha;i++)
        if(alphabet[i]==c)
            return i;
    return(999);
}

void print_e_closure(int i)
{
    int j;
    printf("{");
    for(j=0;e_closure[i][j]!=0;j++)
        printf("q%d,",e_closure[i][j]);
    printf("}");
}
```

OUTPUT

```
asma@asma:~/Cycle 1$ gcc 2_eclosure.c -w
asma@asma:~/Cycle 1$ ./a.out
Enter the number of alphabets
3
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
a
b
e

Enter the number of states?
3

Enter no of transition?
5
NOTE:- [Transition is in the form-> qno alphabet qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1 a 1
1 b 2
1 e 2
2 b 3
2 e 3

e-closure of states……
————————

e-closure(q1): {q1,q2,q3,}
e-closure(q2): {q2,q3,}
e-closure(q3): {q3,}asma@asma:~/Cycle 1$
```

RESULT

Successfully found ε – closure of all states of any given NFA with ε transition.

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	int st;
	struct node *link;
};

void findclosure(int,int);
void insert_trantbl(int ,char, int);
int findalpha(char);
void findfinalstate(void);
void unionclosure(int);
void print_e_closure(int);
static int
set[20],nostate,noalpha,s,notransition,nofinal,start,
finalstate[20],c,r,buffer[20];
char alphabet[20];
static int e_closure[20][20]={0};
struct node * transition[20][20]={NULL};

void main()
{
	 int i,j,k,m,t,n;
	 struct node *temp;
	 printf("enter the number of alphabets?\n");
	 scanf("%d",&noalpha);
	 getchar();
	 printf("NOTE:- [ use letter e as
epsilon]\n");
	printf("NOTE:- [e must be last character ,if
it is present]\n");

	 printf("\nEnter alphabets?\n");
	 for(i=0;i<noalpha;i++)
	{

		alphabet[i]=getchar();
		getchar();
	}
	printf("Enter the number of states?\n");
	scanf("%d",&nostate);
	printf("Enter the start state?\n");
	scanf("%d",&start);
	printf("Enter the number of final states?\n");
	scanf("%d",&nofinal);
	printf("Enter the final states?\n");
	for(i=0;i<nofinal;i++)
		scanf("%d",&finalstate[i]);
	 printf("Enter no of transition?\n");
	scanf("%d",&notransition);
	printf("NOTE:- [Transition is in the form-->
qno  alphabet  qno]\n",notransition);
	printf("NOTE:- [States number must be
greater than zero]\n");
	printf("\nEnter transition?\n");
	for(i=0;i<notransition;i++)
	{
		scanf("%d %c%d",&r,&c,&s);
		insert_trantbl(r,c,s);

	}
	printf("\n");
	for(i=1;i<=nostate;i++)
	{
		c=0;
		for(j=0;j<20;j++)

		{
			buffer[j]=0;
			 e_closure[i][j]=0;
		}
		findclosure(i,i);
	}
	printf("Equivalent NFA without epsilon\n");
	printf("-------------------------------\n");
	printf("start state:");
	print_e_closure(start);
	printf("\nAlphabets:");
	for(i=0;i<noalpha;i++)
		printf("%c ",alphabet[i]);
	printf("\nStates :" );
	for(i=1;i<=nostate;i++)
		print_e_closure(i);
	printf("\nTransitions are...:\n");
	for(i=1;i<=nostate;i++)
	{
		for(j=0;j<noalpha-1;j++)
		{
			for(m=1;m<=nostate;m++)
				set[m]=0;
			for(k=0;e_closure[i][k]!=0;k++)
			{
				t=e_closure[i][k];
```

Date :

EXPERIMENT - 2
CONVERT NFA WITH ε-TRANSITIONS TO NFA

AIM

To write a C program to convert NFA with ε transition to NFA without ε transition.

ALGORITHM

main:

1. Input the number of alphabets ‘noalpha’.
2. Store the alphabet in ‘alphabet[]’, ensuring that if epsilon (‘e’) is present, it is the last alphabet.
3. Input the number of states ‘nostate’ and the starting state ‘start’.
4. Input the number of final states ‘nofinal’ and store the states in ‘finalstate[]’.
5. Input the number of transitions ‘notransition’.
6. For each transition (r, c, s), call the ‘insert_trantbl()’ function to store the transition.
7. For each state from 1 to ‘nostate’, reset the buffer and epsilon closure arrays.
8. Call ‘findclosure()’ for each state to compute its epsilon closure.
9. For each state and alphabet, compute the new transitions using ‘unionclosure()’ to merge the epsilon closures.
10. Print the computed transitions.
11. Call ‘findfinalstate()’ to print the final states in the equivalent NFA.

insert_trantbl(int r, char c, int s):

1. Call ‘findalpha(c)’ to find the index of the alphabet ‘c’.
2. If the alphabet does not exist, print an error and exit.
3. Create a new node to represent the transition (with state ‘s’).
4. Insert this node at the head of the linked list for ‘transition[r][index]’ (where ‘index’ is the index of ‘c’ in ‘alphabet[]’).

findalpha(char c):

1. Loop through ‘alphabet[]’ to find the character ‘c’.
2. If found, return its index.
3. If not found, return 999 to indicate an error.

```
                        temp=transition[t][j];
                        while(temp!=NULL)
                        {
                                unionclosure(temp->st);
                                temp=temp->link;
                        }
                }
                printf("\n");
                print_e_closure(i);
                printf("%c\t",alphabet[j]    );
                printf("{");
                for(n=1;n<=nostate;n++)
                {
                        if(set[n]!=0)
                                printf("q%d,",n);
                }
                printf("}");
        }
    }
    printf("\nFinal states:");
    findfinalstate();
}

void findclosure(int x,int sta)
{
        struct node *temp;
        int i;
        if(buffer[x])
                return;
        e_closure[sta][c++]=x;
        buffer[x]=1;
        if(alphabet[noalpha-1]=='e' && transition[x][noalpha-1]!=NULL)
        {
                temp=transition[x][noalpha-1];
                while(temp!=NULL)
                {
                        findclosure(temp->st,sta);
                        temp=temp->link;
                }
        }
 }

void insert_trantbl(int r,char c,int s)
{
        int j;
        struct node *temp;
        j=findalpha(c);
        if(j==999)
        {
                printf("error\n");
                exit(0);
        }
        temp=(struct node *) malloc(sizeof(struct node));
        temp->st=s;
        temp->link=transition[r][j];
        transition[r][j]=temp;
}

int findalpha(char c)
{
        int i;
        for(i=0;i<noalpha;i++)
                if(alphabet[i]==c)
                        return i;
        return(999);
}

void unionclosure(int i)
{
        int j=0,k;
        while(e_closure[i][j]!=0)
        {
                k=e_closure[i][j];
                set[k]=1;
                j++;
        }
}
void findfinalstate()
{
        int i,j,k,t;
        for(i=0;i<nofinal;i++)
        {
                for(j=1;j<=nostate;j++)
                {
                        for(k=0;e_closure[j][k]!=0;k++)
                        {
                        if(e_closure[j][k]==finalstate[i])
                                {
                                        print_e_closure(j);
                                }
                        }
                }
```

findclosure(int x, int sta):

1. If the state ‘x’ has already been visited (‘buffer[x] == 1’), return.
2. Add state ‘x’ to the epsilon closure of ‘sta’.
3. Mark ‘buffer[x] = 1’ to indicate that this state has been visited.
4. If the last alphabet is ‘e’ (epsilon) and there is an epsilon transition from state ‘x’:
5. Follow the epsilon transition and recursively call ‘findclosure()’ for the target states of the epsilon transition.

unionclosure(int i):

1. For each state in ‘e_closure[i]’, add it to the ‘set[]’ array.
2. Increment ‘newstate’ to track new states being added.

findfinalstate():

For each final state:

1. For each state ‘j’ (1 to ‘nostate’), check if any state in ‘e_closure[j]’ matches the final states.
2. If there is a match, print the epsilon closure for state ‘j’.

print_e_closure(int i)’:

1. Loop through the states in ‘e_closure[i][]’.
2. Print the states in the form ‘{q1, q2, ...}’.

```
        }
    }

void print_e_closure(int i)
{
      int j=0;
      printf("{");
     if(e_closure[i][j]!=0)
                printf("q%d,",e_closure[i][0]);
      printf("}\t");
}
```

OUTPUT

```
asma@asma:~/Cycle 1$ gcc 3_enfa_to_nfa.c -w
asma@asma:~/Cycle 1$ ./a.out
enter the number of alphabets?
4
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
a
b
c
e
Enter the number of states?
3
Enter the start state?
1
Enter the number of final states?
1
Enter the final states?
3
Enter no of transition?
5
NOTE:- [Transition is in the form--> qno   alphabet   qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1 a 1
1 e 2
2 b 2
2 e 3
3 c 3

Equivalent NFA without epsilon
-----------------------------------
start state:{q1,}
Alphabets:a b c e
States :{q1,}   {q2,}   {q3,}
Transitions are...:

{q1,}   a       {q1,q2,q3,}
{q1,}   b       {q2,q3,}
{q1,}   c       {q3,}
{q2,}   a       {}
{q2,}   b       {q2,q3,}
{q2,}   c       {q3,}
{q3,}   a       {}
{q3,}   b       {}
{q3,}   c       {q3,}
asma@asma:~/Cycle 1$
```

RESULT

Successfully converted epsilon NFA to NFA without epsilon.

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
 int st;
 struct node *link;
};
struct node1
{
 int nst[20];
};

void insert(int ,char, int);
int findalpha(char);
void findfinalstate(void);
int insertdfastate(struct node1);
int compare(struct node1,struct node1);
void printnewstate(struct node1);
static int
set[20],nostate,noalpha,s,notransition,nofinal,start,
finalstate[20],c,r,buffer[20];
int complete=-1;
char alphabet[20];
static int eclosure[20][20]={0};
struct node1 hash[20];
struct node * transition[20][20]={NULL};
void main()
{
 int i,j,k,m,t,n,l;
 struct node *temp;
 struct node1 newstate={0},tmpstate={0};
 printf("Enter the number of alphabets?\n");
 printf("NOTE:- [ use letter e as epsilon]\n");
 printf("NOTE:- [e must be last character ,if it is
present]\n");
 scanf("%d",&noalpha);
 getchar();
 printf("\nEnter the alphabets?\n");
 for(i=0;i<noalpha;i++)
 {
 alphabet[i]=getchar();
 getchar();
 }
 printf("Enter the number of states?\n");
 scanf("%d",&nostate);
 printf("Enter the start state?\n");
 scanf("%d",&start);
 printf("Enter the number of final states?\n");
 scanf("%d",&nofinal);
 printf("Enter the final states?\n");
 for(i=0;i<nofinal;i++)
 scanf("%d",&finalstate[i]);
 printf("Enter no of transition?\n");
 scanf("%d",&notransition);
 printf("NOTE:- [Transition is in the form–> qno
alphabet qno]\n",notransition);
 printf("NOTE:- [States number must be greater
than zero]\n");
 printf("\nEnter transition?\n");
 for(i=0;i<notransition;i++)
 {
  scanf("%d %c%d",&r,&c,&s);
  insert(r,c,s);
 }
 for(i=0;i<20;i++)
 {
  for(j=0;j<20;j++)
  hash[i].nst[j]=0;
 }
 complete=-1;
 i=-1;
 printf("\nEquivalent DFA.....\n");
 printf("Trnsitions of DFA\n");
 newstate.nst[start]=start;
 insertdfastate(newstate);
 while(i!=complete)
 {
  i++;
  newstate=hash[i];
  for(k=0;k<noalpha;k++)
  {
   c=0;
   for(j=1;j<=nostate;j++)
   set[j]=0;
   for(j=1;j<=nostate;j++)
   {
    l=newstate.nst[j];
    if(l!=0)
    {
     temp=transition[l][k];
     while(temp!=NULL)
     {
      if(set[temp->st]==0)
      {
```

Date :

EXPERIMENT - 3
NFA TO DFA CONVERSION

AIM

To write a C program to convert NFA to DFA

ALGORITHM

main()

1. Prompt the user to input the number of alphabets (‘noalpha’).
2. Read and store the alphabets in ‘alphabet[]’. (Ensure ‘e’ is used for epsilon and it should be the last character).
3. Input the number of states (‘nostate’), starting state (‘start’), number of final states (‘nofinal’), and the final state numbers in ‘finalstate[]’.
4. Input the number of transitions (‘notransition’).
5. For each transition (state ‘r’, alphabet ‘c’, and state ‘s’), call the ‘insert()’ function to store the transition.
6. Initialize the hash table ‘hash[]’ where DFA states are stored.
7. Create the initial DFA state from the NFA start state and add it to the DFA state list using ‘insertDfaState()’.
8. Loop through each DFA state, processing transitions for each alphabet.
9. For each new DFA state found, insert it using ‘insertDfaState()’ and print the transitions using ‘printnewstate()’.
10. Print the DFA states and their transitions.
11. Identify and print the final states using ‘findfinalstate()’.

insert(int r, char c, int s):

1. Call ‘findalpha(c)’ to find the index of alphabet ‘c’.
2. If the alphabet is not found, print an error and exit.
3. Create a new node representing the transition (‘s’).
4. Insert the node at the head of the linked list for ‘transition[r][index]’.

findalpha(char c):

1. Iterate through the ‘alphabet[]’ array to find the character ‘c’.
2. If found, return its index.
3. If not found, return 999 as an error indicator.

```
      c++;
      set[temp->st]=temp->st;
     }
     temp=temp->link;
    }
   }
  }
  printf("\n");
  if(c!=0)
  {
   for(m=1;m<=nostate;m++)
    tmpstate.nst[m]=set[m];
   insertdfastate(tmpstate);
   printnewstate(newstate);
   printf("%c\t",alphabet[k]);
   printnewstate(tmpstate);
   printf("\n");
  }
  else
  {
   printnewstate(newstate);
   printf("%c\t", alphabet[k]);
   printf("NULL\n");
  }
 }
  }
 printf("\nStates of DFA:\n");
 for(i=0;i<=complete;i++)
 printnewstate(hash[i]);
 printf("\n \nAlphabets:\n");
 for(i=0;i<noalpha;i++)
 printf("\t%c\t",alphabet[i]);
 printf("\n\n Start State:\n");
 printf("\tq%d",start);
 printf("\n\nFinal states:\n");
 findfinalstate();
printf("\n");
}
int insertdfastate(struct node1 newstate)
{
 int i;
 for(i=0;i<=complete;i++)
 {
  if(compare(hash[i],newstate))
   return 0;
 }
 complete++;
 hash[complete]=newstate;
 return 1;
}
int compare(struct node1 a,struct node1 b)
{
 int i;
  for(i=1;i<=nostate;i++)
  {
   if(a.nst[i]!=b.nst[i])
    return 0;
  }
 return 1;
}

void insert(int r,char c,int s)
{
      int j;
      struct node *temp;
      j=findalpha(c);
      if(j==999)
      {
 printf("error\n");
 exit(0);
      }
      temp=(struct node *) malloc(sizeof(struct
node));
      temp->st=s;
      temp->link=transition[r][j];
      transition[r][j]=temp;
}

int findalpha(char c)
{
 int i;
 for(i=0;i<noalpha;i++)
 if(alphabet[i]==c)
  return i;
 return(999);
}

void findfinalstate()
{
 int i,j,k,t;
 for(i=0;i<=complete;i++)
 {
  for(j=1;j<=nostate;j++)
  {
   for(k=0;k<nofinal;k++)
   {
```

insertDfaState(struct node1 newstate):

1. Loop through the existing DFA states in ‘hash[]’.
2. Call ‘compare()’ to check if ‘newstate’ matches an existing DFA state.
3. If ‘newstate’ does not already exist, increment ‘complete’ and add the new state to ‘hash[]’.

compare(struct node1 a, struct node1 b):

1. Loop through each state in ‘a.nst[]’ and ‘b.nst[]’.
2. If any state differs between ‘a’ and ‘b’, return 0 (not equal).
3. If all states match, return 1 (equal).

printnewstate(struct node1 state):

1. Loop through the ‘state.nst[]’ array.
2. For each non-zero state, print the state in the form ‘q<number>‘.

findfinalstate():

1. For each DFA state in ‘hash[]’, loop through its constituent NFA states.
2. For each NFA state in the DFA state, check if it matches any of the NFA final states in ‘finalstate[]’.
3. If a DFA state contains any NFA final state, print that DFA state using ‘printnewstate()’.

```
    if(hash[i].nst[j]==finalstate[k])
    {
     printnewstate(hash[i]);
     printf("\t");
     j=nostate;
     break;
    }
   }
  }
 }
}

void printnewstate(struct node1 state)
{
 int j;
 printf("{");
  for(j=1;j<=nostate;j++)
  {
   if(state.nst[j]!=0)
    printf("q%d,",state.nst[j]);
  }
  printf("}\t");
}
```

OUTPUT

```
asma@asma:~/Cycle 1$ gcc 4_nfa_to_dfa.c -w
asma@asma:~/Cycle 1$ ./a.out
Enter the number of alphabets?
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]
2

Enter the alphabets?
a
b
Enter the number of states?
4
Enter the start state?
1
Enter the number of final states?
1
Enter the final states?
4
Enter no of transition?
5
NOTE:- [Transition is in the form-> qno alphabet qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1 a 1
1 b 1
1 a 2
2 b 3
3 b 4

Equivalent DFA.....
Trnsitions of DFA

{q1,}   a       {q1,q2,}

{q1,}   b       {q1,}

{q1,q2,}        a       {q1,q2,}

{q1,q2,}        b       {q1,q3,}

{q1,q3,}        a       {q1,q2,}

{q1,q3,}        b       {q1,q4,}

{q1,q4,}        a       {q1,q2,}

{q1,q4,}        b       {q1,}

States of DFA:
{q1,}   {q1,q2,}        {q1,q3,}        {q1,q4,}

Alphabets:
        a               b

 Start State:
        q1

Final states:
{q1,q4,}
asma@asma:~/Cycle 1$
```

RESULT

Successfully converted NFA to DFA.

PROGRAM

```
#include <stdio.h>
#include <string.h>
#define STATES  99
#define SYMBOLS 20
int N_symbols;
int N_DFA_states;
char *DFA_finals;
int DFAtab[STATES][SYMBOLS];
char StateName[STATES][STATES+1];
int N_optDFA_states;
int OptDFA[STATES][SYMBOLS];
char NEW_finals[STATES+1];

void print_dfa_table(
   int tab[][SYMBOLS], /* DFA table */
   int nstates,    /* number of states */
   int nsymbols,   /* number of input symbols */
   char *finals)
{
   int i, j;
   puts("\nDFA: STATE TRANSITION TABLE");
   printf("     | ");
   for (i = 0; i < nsymbols; i++) printf("  %c  ",
'0'+i);
   printf("\n-----+--");
   for (i = 0; i < nsymbols; i++) printf("-----");
   printf("\n");
   for (i = 0; i < nstates; i++) {
      printf("  %c  | ", 'A'+i);
      for (j = 0; j < nsymbols; j++)
         printf("  %c  ", tab[i][j]);
      printf("\n");
   }
   printf("Final states = %s\n", finals);
}

void load_DFA_table()
{
   DFAtab[0][0] = 'B'; DFAtab[0][1] = 'C';
   DFAtab[1][0] = 'E'; DFAtab[1][1] = 'F';
   DFAtab[2][0] = 'A'; DFAtab[2][1] = 'A';
   DFAtab[3][0] = 'F'; DFAtab[3][1] = 'E';
   DFAtab[4][0] = 'D'; DFAtab[4][1] = 'F';
   DFAtab[5][0] = 'D'; DFAtab[5][1] = 'E';
   DFA_finals = "EF";
   N_DFA_states = 6;
   N_symbols = 2;
}

void get_next_state(char *nextstates, char
*cur_states,
   int dfa[STATES][SYMBOLS], int symbol)
{
   int i, ch;
   for (i = 0; i < strlen(cur_states); i++)
      *nextstates++ = dfa[cur_states[i]-
'A'][symbol];
   *nextstates = '\0';
}

char equiv_class_ndx(char ch, char
stnt[][STATES+1], int n)
{
   int i;
   for (i = 0; i < n; i++)
      if (strchr(stnt[i], ch)) return i+'0';
   return -1;  /* next state is NOT defined */
}

char is_one_nextstate(char *s)
{
   char equiv_class;
   while (*s == '@') s++;
   equiv_class = *s++;
   while (*s) {
      if (*s != '@' && *s != equiv_class) return 0;
      s++;
   }
   return equiv_class;
}

int state_index(char *state, char
stnt[][STATES+1], int n, int *pn,
   int cur)
{
   int i;
   char state_flags[STATES+1];
   if (!*state) return -1;
   for (i = 0; i < strlen(state); i++)
      state_flags[i] = equiv_class_ndx(state[i], stnt,
n);
   state_flags[i] = '\0';

   printf("   %d:[%s]\t--> [%s] (%s)\n",
```

Date :

EXPERIMENT - 4
DFA MINIMIZATION

AIM

To write a C program to minimize the given DFA.

ALGORITHM

main()

1. Call the `load_DFA_table()` function to initialize the DFA transition table `DFAtab[]`, the number of DFA states (`N_DFA_states`), and the number of symbols (`N_symbols`).
2. Call `print_dfa_table(DFAtab, N_DFA_states, N_symbols, DFA_finals)` to print the initial DFA transition table.
3. Call `optimize_DFA(DFAtab, N_DFA_states, N_symbols, DFA_finals, StateName, OptDFA)` to perform state minimization of the DFA.
4. The result is stored in `OptDFA[]`, and the number of optimized DFA states is returned as `N_optDFA_states`.
5. Call `get_NEW_finals(NEW_finals, DFA_finals, StateName, N_optDFA_states)` to get the final states of the optimized DFA.
6. Call `print_dfa_table(OptDFA, N_optDFA_states, N_symbols, NEW_finals)` to print the minimized DFA transition table.

load_DFA_table()

1. Initialize the DFA transition table `DFAtab[][]` with specific state transitions.
2. Set the DFA final states `DFA_finals`.
3. Set the number of DFA states (`N_DFA_states`) and the number of symbols (`N_symbols`).

optimize_DFA()

1. Call `init_equiv_class()` to divide DFA states into two equivalence classes: final states and non-final states.
2. Store the equivalence classes in `stnt[][]` and return the number of initial equivalence classes `n`.
3. Repeat until no further splitting of equivalence classes occurs:
    a. Print the current equivalence class candidates using `print_equiv_classes()`.
    b. Call `get_optimized_DFA()` to compute the transitions for the current equivalence classes.
    c. Call `set_new_equiv_class()` to split and refine the equivalence classes if necessary.
4. Return the number of minimized DFA states.

```
        cur, stnt[cur], state, state_flags);
    if (i=is_one_nextstate(state_flags))
        return i-'0';
    else {
        strcpy(stnt[*pn], state_flags);
        return (*pn)++;
    }
}
int init_equiv_class(char statename[][STATES+1],
int n, char *finals)
{
    int i, j;

    if (strlen(finals) == n) {
        strcpy(statename[0], finals);
        return 1;
    }
    strcpy(statename[1], finals);
    for (i=j=0; i < n; i++) {
        if (i == *finals-'A') {
            finals++;
        } else statename[0][j++] = i+'A';
    }
    statename[0][j] = '\0';

    return 2;
}

int get_optimized_DFA(char stnt[][STATES+1],
int n,
    int dfa[][SYMBOLS], int n_sym, int
newdfa[][SYMBOLS])
{
    int n2=n;
    int i, j;
    char nextstate[STATES+1];
    for (i = 0; i < n; i++) {
        for (j = 0; j < n_sym; j++) {
            get_next_state(nextstate, stnt[i], dfa, j);
            newdfa[i][j] = state_index(nextstate, stnt,
n, &n2, i)+'A';
        }
    }
    return n2;
}

void chr_append(char *s, char ch)
{
    int n=strlen(s);
    *(s+n) = ch;
    *(s+n+1) = '\0';
}

void sort(char stnt[][STATES+1], int n)
{
    int i, j;
    char temp[STATES+1];
    for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++)
            if (stnt[i][0] > stnt[j][0]) {
                strcpy(temp, stnt[i]);
                strcpy(stnt[i], stnt[j]);
                strcpy(stnt[j], temp);
            }
}

int split_equiv_class(char stnt[][STATES+1],
    int i1,
    int i2,    int n,
    int n_dfa)
{
    char *old=stnt[i1], *vec=stnt[i2];
    int i, n2, flag=0;
    char newstates[STATES][STATES+1];
    for (i=0; i < STATES; i++) newstates[i][0] =
'\0';
    for (i=0; vec[i]; i++)
        chr_append(newstates[vec[i]-'0'], old[i]);
    for (i=0, n2=n; i < n_dfa; i++) {
        if (newstates[i][0]) {
            if (!flag) {
                strcpy(stnt[i1], newstates[i]);
                flag = 1;
            } else
                strcpy(stnt[n2++], newstates[i]);
        }
    }

    sort(stnt, n2);

    return n2;
}

int set_new_equiv_class(char stnt[][STATES+1],
int n,
    int newdfa[][SYMBOLS], int n_sym, int n_dfa)
```

init_equiv_class()

1. Divide DFA states into two groups:
    a. Group 1: Non-final states.
    b. Group 2: Final states.
2. Store these groups as equivalence classes in `statename[][]`.
3. Return the number of initial equivalence classes.

get_optimized_DFA()

1. For each pseudo-DFA state in `stnt[][]`:
    a. For each input symbol, compute the next states.
    b. Call `get_next_state()` to compute the next states for the current state.
    c. Call `state_index()` to check whether the next states belong to a single equivalence class or need further splitting.
    d. Update the `newdfa[][]` transition table with the new equivalence class for each state.
2. Return the new number of equivalence classes after the transition computation.

set_new_equiv_class()

1. For each equivalence class in `stnt[][]`:
    a. For each input symbol, check whether the next states belong to a unique equivalence class using the `newdfa[][]` transition table.
    b. If a state needs further splitting, call `split_equiv_class()` to divide the equivalence class into subclasses.
2. Return the new number of equivalence classes.

split_equiv_class()

1. For the selected equivalence class:
    a. Divide the states in the class into multiple subclasses based on their transitions.
2. Update the equivalence class list `stnt[][]` to include the new subclasses.
3. Sort the equivalence classes and return the updated number of classes.

```
{
   int i, j, k;
   for (i = 0; i < n; i++) {
      for (j = 0; j < n_sym; j++) {
         k = newdfa[i][j]-'A';
         if (k >= n)
            return split_equiv_class(stnt, i, k, n,
n_dfa);
      }
   }
   return n;
}

void print_equiv_classes(char stnt[][STATES+1],
int n)
{
   int i;
   printf("\nEQUIV. CLASS CANDIDATE ==>");
   for (i = 0; i < n; i++)
      printf(" %d:[%s]", i, stnt[i]);
   printf("\n");
}

int optimize_DFA(
   int dfa[][SYMBOLS],
   int n_dfa,
   int n_sym,
   char *finals,
   char stnt[][STATES+1],
   int newdfa[][SYMBOLS])
{
   char nextstate[STATES+1];
   int n;
   int n2;
   n = init_equiv_class(stnt, n_dfa, finals);
   while (1) {
      print_equiv_classes(stnt, n);
      n2 = get_optimized_DFA(stnt, n, dfa, n_sym,
newdfa);
      if (n != n2)
         n = set_new_equiv_class(stnt, n, newdfa,
n_sym, n_dfa);
      else break;
   }
   return n;
}

int is_subset(char *s, char *t)
{
   int i;
   for (i = 0; *t; i++)
      if (!strchr(s, *t++)) return 0;
   return 1;
}

void get_NEW_finals(
   char *newfinals,
   char *oldfinals,
   char stnt[][STATES+1],
   int n)
{
   int i;
   for (i = 0; i < n; i++)
      if (is_subset(oldfinals, stnt[i])) *newfinals++
= i+'A';
   *newfinals++ = '\0';
}

void main()
{
   load_DFA_table();
   print_dfa_table(DFAtab, N_DFA_states,
N_symbols, DFA_finals);
   N_optDFA_states = optimize_DFA(DFAtab,
N_DFA_states,
         N_symbols, DFA_finals, StateName,
OptDFA);
   get_NEW_finals(NEW_finals, DFA_finals,
StateName, N_optDFA_states);
   print_dfa_table(OptDFA, N_optDFA_states,
N_symbols, NEW_finals);
}
```

get_NEW_finals()

1. For each equivalence class in `stnt[][]`, check whether all states in the class are final states.
2. If the class contains final states, add the new state to `newfinals[]`.
3. Return the final states for the minimized DFA.

print_dfa_table()

1. Print the header for the DFA transition table (states and input symbols).
2. For each state in the DFA, print the transitions for all input symbols.
3. Print the final states of the DFA.

OUTPUT

```
asma@asma:~/Cycle 1$ gcc 5_dfa_min.c
asma@asma:~/Cycle 1$ ./a.out

DFA: STATE TRANSITION TABLE
     |   0    1
-----+------------
  A  |   B    C
  B  |   E    F
  C  |   A    A
  D  |   F    E
  E  |   D    F
  F  |   D    E
Final states = EF

EQUIV. CLASS CANDIDATE ==> 0:[ABCD] 1:[EF]
   0:[ABCD]     --> [BEAF] (0101)
   0:[ABCD]     --> [CFAE] (0101)
   1:[EF]       --> [DD] (00)
   1:[EF]       --> [FE] (11)

EQUIV. CLASS CANDIDATE ==> 0:[AC] 1:[BD] 2:[EF]
   0:[AC]       --> [BA] (10)
   0:[AC]       --> [CA] (00)
   1:[BD]       --> [EF] (22)
   1:[BD]       --> [FE] (22)
   2:[EF]       --> [DD] (11)
   2:[EF]       --> [FE] (22)

EQUIV. CLASS CANDIDATE ==> 0:[A] 1:[BD] 2:[C] 3:[EF]
   0:[A]        --> [B] (1)
   0:[A]        --> [C] (2)
   1:[BD]       --> [EF] (33)
   1:[BD]       --> [FE] (33)
   2:[C]        --> [A] (0)
   2:[C]        --> [A] (0)
   3:[EF]       --> [DD] (11)
   3:[EF]       --> [FE] (33)

DFA: STATE TRANSITION TABLE
     |   0    1
-----+------------
  A  |   B    C
  B  |   D    D
  C  |   A    A
  D  |   B    D
Final states = D
asma@asma:~/Cycle 1$
```

RESULT

Successfully minimized the given DFA.

Date :

EXPERIMENT – 5

STUDY OF LEX AND YACC TOOLS

AIM

To study about LEX and YACC tools.

THEORY

What is Lex?

Lex is a computer program that generates lexical analyzers (“scanners” or “lex ers”). It is commonly used with the Yacc parser generator and is the standard lexical analyzer generator on many Unix and Unix-like systems. Lex reads an input stream specifying the lexical analyzer and writes source code which implements the lexical analyzer in the C programming language.

STRUCTURE OF LEX PROGRAMS

A Lex program is organized into three main sections: Declarations, Rules, and Auxiliary Functions. Each section serves a distinct purpose in defining the behavior of the lexical analyzer.

1. **Declarations**

The Declarations section comprises two parts:

- **Regular Definitions**: Define macros and shorthand notations for regular expressions to simplify rule definitions.
- **Auxiliary Declarations**: Contains C code such as header file inclu sions, function prototypes, and global variable declarations. This code is enclosed within %{ and %} and is copied verbatim into the generated lex.yy.c file.

2. **Rules**

The Rules section consists of pattern-action pairs, where each pair defines a regular expression pattern to match and the corresponding C code to execute upon a successful match.

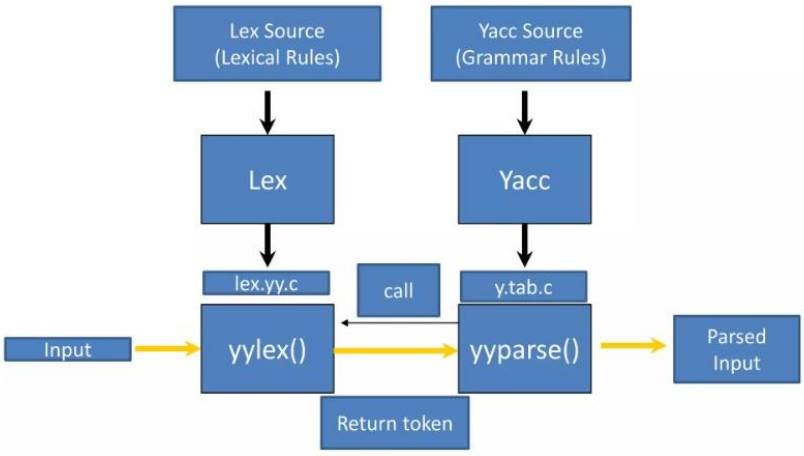
3. **Auxiliary Functions**

The Auxiliary Functions section includes additional C code that is not part of the rules. This typically contains the main function and any helper functions required by the lexer. This code is placed after the second %% and is directly copied into the lex.yy.c file.

yylex()

The `yylex()` function, defined by Lex in the `lex.yy.c` file, reads the input stream, matches it against regular expressions, and executes the corresponding actions for each match. It also generates tokens

Compilation of LEX and YACC



for further processing by parsers or other components, though the programmer must explicitly invoke `yylex()` within the auxiliary functions of the Lex program.

yywrap() :The function `yywrap()` is called by `yylex()` when the end of an input file is reached. If `yywrap()` returns 0, scanning continues; if it returns a non-zero value, `yylex()` terminates and returns 0.

(Comments enclosed in /*…..*/ may appear in any section)

COMPILATION STEPS

1. Create a file with a .l extension and write your Lex program
2. Use Lex to process the .l file and generate a C source file. lex simple_calc .l
3. Use GCC (GNU Compiler Collection) to compile the generated C code. gcc -o simple_calc lex . yy .c - lfl
4. Execute the compiled program using: ./ simple_calc

What is Yacc?

Yacc (Yet Another Compiler Compiler) generates LALR parsers from formal grammar specifications. It is often used with Lex for building compilers and interpreters, handling the syntactic analysis phase. Yacc reads a grammar file and produces a C source parser that processes token sequences (typically from Lex) to check if they follow the grammar. The modern counterpart of Yacc is Bison.

STRUCTURE OF YACC PROGRAM

```
%{
        C Declarations
%}
        Yacc Declarations
%%
        Grammar Rules
%%
        Additional Code
```

COMPILATION STEPS

Step 1: Create a file with a .y and write your yacc .

Step 2: give the command yacc -d simple_calc.y This command generates a C source file named y.tab.c.

Step 3: Use GCC (GNU Compiler Collection) to compile the generated C code. gcc-o simple_calc lex.yy.c -lfl

Step 4: Running the Executable Execute the compiled program using: ./simple_calc

RESULT

Familiarized the working of LEX and YACC.

PROGRAM

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

char line[100];

int is_operator(char c)
{
   switch (c)
   {
      case '+':
      case '-':
      case '*':
      case '/':
      case '=':
         printf("%c - Operator\n", c);
         return 1;
   }
   return 0;
}

int is_delimiter(char c)
{
   switch (c)
   {
      case '{':
      case '}':
      case '(':
      case ')':
      case '[':
      case ']':
      case ',':
      case ';':
         printf("%c - Delimiter\n", c);
         return 1;
   }
   return 0;
}

int is_keyword(char buffer[]){
  char keywords[32][10] =
{"auto","break","case","char","const","continue",
"default","do","double","else","enum","extern","
float","for","goto","if","int","long","register","ret
urn","short","signed","sizeof","static","struct","s
witch","typedef","union","unsigned","void","vola
tile","while"};
  int i;
  for(i = 0; i < 32; ++i){
            if(strcmp(keywords[i], buffer) == 0){
                     return 1;
            }
  }
return 0;
}

void main()
{
   char c;
   FILE *f = fopen("input.txt", "r");

   while (fgets(line, sizeof(line), f))
   {
      int flag1 = 0;
      for (int i = 0; i < strlen(line); i++)
      {
         if (line[i] == '/' && line[i + 1] == '/')
         {
            flag1 = 1;
            break;
         }
      }
      if (flag1)
         continue;
      int flag2 = 0;
      for (int i = 0; i < strlen(line); i++)
      {
         if (line[i] == '/' && line[i + 1] == '*')
         {
            while (fgets(line, sizeof(line), f))
            {
               for (int j = 0; j < strlen(line); j++)
               {
                  if (line[j] == '*' && line[j + 1] == '/')
                     flag2 = 1;
               }
               if (flag2)
                  break;
            }
```

Date :

EXPERIMENT – 6

IMPLEMENTATION OF A LEXICAL ANALYSER

AIM

Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and newlines.

ALGORITHM

1. Start
2. Create a character array line[100] to hold each line of input.
3. Define functions to identify operators, delimiters, keywords, integers, and floating-point numbers.
4. Open the input file input.txt for reading.
5. While there are lines to read in the file:
    - 5.1 Read a line into the line array.
    - 5.2 Initialize flags for comment detection (flag1 for single-line comments, flag2 for multi-line comments) to false.
    - 5.3 For each character in the line, if the character is '/' and the next character is '/', set flag1 to true and skip processing the line.
    - 5.4 For each character in the line, if the character is '/' and the next character is '*', skip lines until you find the closing '*/' and set flag2 to true and skip processing the line.
    - 5.5 Initialize an empty token string and an index counter set to 0.
    - 5.6 For each character in the line:
        - 5.6.1 If token is a keyword, print it as a keyword.
        - 5.6.2 Else if token is an integer or floating-point number, print it as a number.
        - 5.6.3 Else, print it as an identifier.
        - 5.6.4 Clear the token string and reset the index counter.
        - 5.6.5 If the character is whitespace (space, tab, or newline), continue to the next character.
        - 5.6.6 If the character is not an operator, delimiter, or whitespace, add it to the token string and increment the index.
6. Close the input file after processing all lines.
7. Stop

```
        }
      }
      if (flag2)
        continue;
      printf("\n%s\n", line);
      char token[100];
      int index = 0;
      strcpy(token, "");

      for (int i = 0; i < strlen(line); i++)
      {
        if (is_operator(line[i]) ||
is_delimiter(line[i]) || line[i] == ' ' || line[i] == '\t'
|| line[i] == '\n')
        {
          if (strcmp(token, "") != 0)
          {
            if (is_keyword(token))
              printf("%s - Keyword\n", token);
            else
              printf("%s - Identifier\n", token);
            strcpy(token, "");
            index = 0;
          }
        }
        else
        {
          token[index++] = line[i];
          token[index] = '\0';
        }
      }
    }
    fclose(f);
}
```

Input.txt

```
void main()
{
   int c=22;
   int a=15;
   int b = 21;
   float c = 3.1415;
   double d = 2;
   int a[4] = {1, 2, 3, 4};
}
```

OUTPUT

```
asma@asma:~/Cycle 1/lexical_analyser$ gcc 1_lexical_analyser.c
asma@asma:~/Cycle 1/lexical_analyser$ ./a.out

void main()

void - Keyword
( - Delimiter
main - Identifier
) - Delimiter

{

{ - Delimiter

    int c=22;

int - Keyword
= - Operator
c - Identifier
; - Delimiter
22 - Identifier

    int a=15;

int - Keyword
= - Operator
a - Identifier
; - Delimiter
15 - Identifier

    int b = 21;

int - Keyword
b - Identifier
= - Operator
; - Delimiter
21 - Identifier

    float c = 3.1415;

float - Keyword
c - Identifier
= - Operator
; - Delimiter
3.1415 - Identifier

    double d = 2;

double - Keyword
d - Identifier
= - Operator
; - Delimiter
2 - Identifier

    int a[4] = {1, 2, 3, 4};

int - Keyword
[ - Delimiter
a - Identifier
] - Delimiter
4 - Identifier
= - Operator
{ - Delimiter
, - Delimiter
1 - Identifier
, - Delimiter
2 - Identifier
, - Delimiter
3 - Identifier
} - Delimiter
4 - Identifier
; - Delimiter

}
} - Delimiter
asma@asma:~/Cycle 1/lexical_analyser$
```

RESULT

Successfully run the program and obtained desired output.

PROGRAM

Substring.l

```
%{
#include <stdio.h>
#include <string.h>

char name[100];
char prefix[100];

void recognize_string(const char *input_string)
{
   if (strstr(input_string, prefix) == NULL) {
      printf("%s is recognized\n", input_string);
   } else {
      printf("%s is not recognized\n",
input_string);
   }
}
%}

%%
[A-Za-z]+  { recognize_string(yytext); }
\n         { /* Ignore newlines */ }
.          { /* Ignore any other character */ }
%%

int main() {
   printf("Enter the name: ");
   fgets(name, 100, stdin);
   name[strcspn(name, "\n")] = 0;
   int prefix_len;
   prefix_len=4;
   strncpy(prefix, name, prefix_len);
   prefix[prefix_len] = '\0';
   printf("Enter input strings (Ctrl+D to stop):\n");
   yylex();
   return 0;
}

int yywrap() {
   return 1;
}
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 2/1$ lex 1_substring.l
asma@asma:~/Downloads/Cycle 2/1$ gcc lex.yy.c
asma@asma:~/Downloads/Cycle 2/1$ ./a.out
Enter the name: arun
Enter input strings (Ctrl+D to stop):
arundathi
arundathi is not recognized
asma
asma is recognized
varun
varun is not recognized
aruna
aruna is not recognized
achu
achu is recognized
asma@asma:~/Downloads/Cycle 2/1$
```

EXPERIMENT - 7

LEX PROGRAM TO EXCLUDE NAME PREFIX

AIM

To write a lex program to recognize all strings which does not contain first four characters of your name as a substring.

ALGORITHM

1. The program begins execution in the main function, which prompts the user to enter a string and calls yylex() to start scanning input.
2. Lex Rule: The rule [a-zA-Z]* matches any sequence of alphabetic characters (including an empty string).
3. Substring Check: The program loops through the current token yytext to see if it contains the first four letters of your name as a substring.
4. If the first four letters of your name is found within the input string, a flag is set to indicate its presence. The matched string is then stored in the name variable for later use.
5. After processing the string, the program checks the flag and displays messages based on whether the first four letters of your name were found or not.

RESULT

Successfully run the program and obtained desired output.

PROGRAM

Valid.y

```
%{
   #include<stdio.h>
   int valid=1;
   int yyerror();
%}

%token digit letter

%%
start : letter s
s :    letter s
     | digit s
     |
     ;
%%

int yyerror(){
   printf("\nIts not a identifier!\n");
  valid=0;
  return 0;
}

int main(){
    printf("\nEnter a name to tested for identifier
");
   yyparse();
   if(valid){
   printf("\nIt is an identifier!\n");
   }
}
```

Valid.l

```
%{
   #include "y.tab.h"
%}

%%
[a-zA-Z_][a-zA-Z_0-9]* {return letter;}
[0-9] {return digit;}
.  {return yytext[0];}
\n {return 0;}
%%

int yywrap(){
return 1;
}
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 2/2$ yacc -d valid.y
asma@asma:~/Downloads/Cycle 2/2$ lex valid.l
asma@asma:~/Downloads/Cycle 2/2$ gcc lex.yy.c y.tab.c -o valid -ll -w
asma@asma:~/Downloads/Cycle 2/2$ ./valid

Enter a name to tested for identifier asma

It is an identifier!
asma@asma:~/Downloads/Cycle 2/2$ ./valid

Enter a name to tested for identifier 1asma

Its not a identifier!
asma@asma:~/Downloads/Cycle 2/2$ ./valid

Enter a name to tested for identifier _water

It is an identifier!
asma@asma:~/Downloads/Cycle 2/2$
```

Date :

EXPERIMENT - 8

YACC PROGRAM TO RECOGNIZE VALID VARIABLES

AIM

To write a YACC program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

ALGORITHM

1. The program begins in main() where the user is prompted to input a string.
2. yyparse() is called to start the parsing process.
3. The lexical analyzer identifies tokens based on patterns: letter for identifiers and digit for numbers.
4. The lexer returns letter for [a-zA-Z_][a-zA-Z_0-9]* and digit for [0-9] to the YACC parser.
5. The parser uses the CFG rules:
   - start → letter s
   - s → letter s | digit s | ε
6. The parser checks if the input follows these rules, defining valid identifiers starting with a letter.
7. If the input matches the CFG, the parsing continues successfully.
8. If the input violates the CFG, yyerror() is called, setting valid = 0 and printing an error message.
9. After parsing, main() checks the valid flag.
10. If valid, it prints "It is an identifier"; otherwise, it prints an error message.

RESULT

Successfully run the program and obtained desired output.

PROGRAM

Calculator.l

```
%{
#include "y.tab.h"
%}

%%
[0-9]+    { yylval = atoi(yytext); return NUMBER;
}
"+"       { return '+'; }
"-"       { return '-'; }
"*"       { return '*'; }
"/"       { return '/'; }
"("       { return '('; }
")"       { return ')'; }
[ \t]+    { /* Ignore spaces and tabs */ }
.         { printf("Unexpected character: %s\n",
yytext); }
%%

int yywrap() {
   return 1;
}
```

Calculator.y

```
  %{
  #include <stdio.h>
  #include <stdlib.h>
  void yyerror(char *s);
  int yylex(void);
  %}

  %token NUMBER

  %%
  start:  expression            { printf(" Result is :
  %d\n", $1); }
         ;
  expression:
       expression '+' term        { $$ = $1 + $3; }
     | expression '-' term        { $$ = $1 - $3; }
     | term                     { $$ = $1; }
     ;
  term:
       term '*' factor           { $$ = $1 * $3; }
     | term '/' factor           { $$ = $1 / $3; }
     | factor                    { $$ = $1; }
     ;
factor:
      '(' expression ')'         { $$ = $2; }
    | NUMBER                       { $$ = $1;
    ;
%%

int main() {
   printf("Enter an arithmetic expression: ");
   yyparse();
   return 0;
}

void yyerror(char *s) {
   fprintf(stderr, "Error: %s\n", s);
}
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 2/3$ yacc -d calculator.y
asma@asma:~/Downloads/Cycle 2/3$ lex calculator.l
asma@asma:~/Downloads/Cycle 2/3$ gcc lex.yy.c y.tab.c -o calculator -ll -w
asma@asma:~/Downloads/Cycle 2/3$ ./calculator
Enter an arithmetic expression: 1*9/7+(9*5+1/25*(9+3-1)/2)+6

 Result is : 52
asma@asma:~/Downloads/Cycle 2/3$ ./calculator
Enter an arithmetic expression: 1+2*(3/1)+5

 Result is : 12
asma@asma:~/Downloads/Cycle 2/3$
```

Date :

EXPERIMENT - 9

CALCULATOR WITH LEX AND YACC

AIM

To implement a calculator using lex and yacc.

ALGORITHM

1. The program begins in main(), prompting the user to input an arithmetic expression.
2. yyparse() is called to start parsing the input.
3. The lexical analyzer matches patterns for numbers ([0-9]+) and arithmetic operators (+, -, *, /) and returns tokens to the parser.
4. The parser uses the following CFG:
    - start → expression
    - expression → expression '+' term | expression '-' term | term
    - term → term '*' factor | term '/' factor | factor
    - factor → '(' expression ')' | NUMBER
5. The parser processes tokens based on the CFG rules, performing arithmetic operations as it matches expressions.
6. Intermediate results are calculated and stored using the semantic actions associated with each rule.
7. If an invalid token or character is encountered, the lexer prints an error message.
8. After successfully parsing, the result of the expression is printed.
9. If any syntax errors occur, yyerror() is invoked, printing an error message.The program terminates after displaying the result or an error message.

RESULT

Successfully run the program and obtained desired output.

PROGRAM

bnf.l

```
%{
#include"y.tab.h"
#include <stdio.h>
#include<string.h>
int LineNo=1;
%}

identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)

%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int|char|float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%

int yywrap() {
   return 1;
}
```

bnf.y

```
%{
#include<string.h>
 #include<stdio.h>
struct quad { char op[5];
 char arg1[10];
 char arg2[10];
char result[10];
}QUAD[30];
struct stack { int items[100];
 int top; }stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
{
char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT
CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
 strcpy(QUAD[Index].arg1,$3);
  strcpy(QUAD[Index].arg2,"");
  strcpy(QUAD[Index].result,$1);
   strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR
{AddQuadruple("+",$1,$3,$$);}
```

Date :

EXPERIMENT - 10

BNF TO YACC CONVERSION

AIM

To write a program to convert the BNF rules into YACC form and write code to generate abstract syntax tree

ALGORITHM

1. The program begins in the main() function, where input is taken from a file (if provided), and yyparse() is invoked to start parsing.
2. The parser processes the source code, using a set of grammar rules and a CFG, which defines program structures like PROGRAM, BLOCK, CODE, STATEMENT, EXPR, CONDITION, and control structures like IF, ELSE, and WHILE.
3. The lexical analyzer scans the input code and returns tokens such as NUM, VAR, RELOP, and control keywords like IF, WHILE, and TYPE to the YACC parser.
4. During parsing, the program builds quadruples (3-address code) for expressions and conditions using the function AddQuadruple().
5. The quadruples are stored in the global array QUAD[], and temporary variables are generated using tIndex for intermediate expressions.
6. For assignments and expressions (e.g., addition, subtraction), the AddQuadruple() function creates the necessary quadruple, updating QUAD[] with the operator, operands, and result.
7. Control structures like IF, ELSE, and WHILE involve conditional jumps. These are handled by pushing and popping indices of quadruples onto the stack and updating their result fields to indicate where control should jump.
8. After parsing, the program prints all generated quadruples in a tabular format showing the operator, arguments, and result for each operation.
9. The stack functions push() and pop() manage control flow for conditional and loop statements by handling jump addresses.
10. The program concludes by printing any errors encountered, using yyerror() to indicate the line number where an error occurred.

```
| EXPR '-' EXPR {AddQuadruple("-
",$1,$3,$$);}
| EXPR '*' EXPR
{AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR
{AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
 Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
 strcpy(QUAD[Index].arg2,"FALSE");
  strcpy(QUAD[Index].result,"-1");
   push(Index);
Index++;
}
BLOCK { strcpy(QUAD[Index].op,"GOTO");
 strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
 Ind=pop();
  push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR
{AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
 Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
   if(argc>1) {
      fp=fopen(argv[1],"r");
         if(!fp) {
            printf("\n File not found");
            exit(0);
            }
      yyin=fp;
      }
yyparse();
printf("\n\n\t\t ----------------------------","\n\t\t
Pos Operator Arg1 Arg2 Result", "\n\t\t----------
---------");
for(i=0;i<Index;i++)
{
```

```
printf("\n\t\t %d\t %s\t %s\t
%s\t%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i]
.arg2,QUAD[i].result);
}
printf("\n\t\t ----------------------");
printf("\n\n");
return 0; }

void push(int data){
   stk.top++;
      if(stk.top==100)
      {
         printf("\n Stack overflow\n");
         exit(0); }
         stk.items[stk.top]=data;
      }
int pop() {
   int data;
      if(stk.top==-1){
         printf("\n Stack underflow\n");
         exit(0);
         }
   data=stk.items[stk.top--];
   return data;
 }
void AddQuadruple(char op[5],char
arg1[10],char arg2[10],char result[10]){
   strcpy(QUAD[Index].op,op);
   strcpy(QUAD[Index].arg1,arg1);
   strcpy(QUAD[Index].arg2,arg2);
   sprintf(QUAD[Index].result,"t%d",tIndex++);
   strcpy(result,QUAD[Index++].result);
   }

yyerror() {
printf("\n Error on line no:%d",LineNo);
}
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 2/4$ yacc -d bnf.y
asma@asma:~/Downloads/Cycle 2/4$ lex bnf.l
asma@asma:~/Downloads/Cycle 2/4$ gcc lex.yy.c y.tab.c -o bnf -ll -w
asma@asma:~/Downloads/Cycle 2/4$ ./bnf test.c


----------------------------
0       <       a       b       t0
1       ==      t0      FALSE   5
2       +       a       b       t1
3       =       t1              a
4       GOTO                    5
5       <       a       b       t2
6       ==      t2      FALSE   10
7       +       a       b       t3
8       =       t3              a
9       GOTO                    5
10      <=      a       b       t4
11      ==      t4      FALSE   15
12      -       a       b       t5
13      =       t5              c
14      GOTO                    17
15      +       a       b       t6
16      =       t6              c
-----------------------

asma@asma:~/Downloads/Cycle 2/4$
```

test.c

```
main() { int a,b,c;
 if(a<b) { a=a+b; }
  while(a<b) { a=a+b; }
  if(a<=b) { c=a-b; }
   else { c=a+b;
    }
}
```

RESULT

Successfully run the program and obtained desired output.

PROGRAM

for.l

```
%{
#include "y.tab.h"
%}

%%
for                                     return
FOR;
[\(\)]                                  return
PARANTHESIS;
[a-zA-Z0-9]*                            return
OPERAND;
"="|"<"|">"|">="|"<="|"=="|"++"|"--"    return
OPERATOR;
\;                                      return
SEMICOLON;
,                                       return
COMMA;
\n                                      return
NEWLINE;
.                                       ;
%%

int yywrap()
{
        return 1;
}
```

for.y

```
%{
        #include <stdio.h>
        int valid = 1;
%}

%token FOR PARANTHESIS OPERAND
OPERATOR COMMA SEMICOLON
NEWLINE

%%
start: FOR PARANTHESIS A A B
PARANTHESIS NEWLINE;
A: OPERAND OPERATOR OPERAND
SEMICOLON | OPERAND OPERATOR
OPERAND COMMA A | SEMICOLON;
B: OPERAND OPERATOR | OPERAND
OPERATOR COMMA B |;
%%

int yyerror()
{
        valid = 0;
        printf("Invalid.\n");
        return 1;
}

void main()
{
        printf("Enter string:\n");
        yyparse();

        if (valid)
                printf("Valid.\n");
}
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 2/5$ yacc -d for.y
asma@asma:~/Downloads/Cycle 2/5$ lex for.l
asma@asma:~/Downloads/Cycle 2/5$ gcc lex.yy.c y.tab.c -o for -ll -w
asma@asma:~/Downloads/Cycle 2/5$ ./for
Enter string:
for(i=0;i<n;i++)
Valid.
asma@asma:~/Downloads/Cycle 2/5$ ./for
Enter string:
for(i=0;i<n;j<n)
Invalid.
asma@asma:~/Downloads/Cycle 2/5$ ./for
Enter string:
for(i=0;i<n;i++){}
Valid.
asma@asma:~/Downloads/Cycle 2/5$
```

Date :

EXPERIMENT – 11

FOR STATEMENT SYNTAX VALIDATOR

AIM

To write a yacc program to check syntax of for statement

ALGORITHM

1. The program execution begins in the main() function, where the user is prompted to enter a string.
2. The yyparse() function is called to initiate the parsing process, which triggers both the lexical analyzer and the YACC parser.
3. The lexical analyzer scans the input string, recognizing tokens (FOR, PARANTHESIS, OPERAND, OPERATOR, COMMA, SEMICOLON, NEWLINE) based on regular expressions and returns them to the YACC parser.
4. The YACC parser attempts to match the tokens against the predefined context-free grammar (CFG):
   - start → FOR PARANTHESIS A A B PARANTHESIS NEWLINE
   - A → OPERAND OPERATOR OPERAND SEMICOLON | OPERAND OPERATOR OPERAND COMMA A | SEMICOLON
   - B → OPERAND OPERATOR | OPERAND OPERATOR COMMA B
5. If the input conforms to the CFG, the parsing continues smoothly.
6. If any part of the input violates the grammar rules, the yyerror() function is invoked, which sets valid = 0 and prints "Invalid."
7. Once the parsing completes, control returns to the main() function.
8. The valid flag is checked. If it remains 1, indicating no errors, "Valid" is printed.
9. The yywrap() function is called at the end of input processing to signal completion.
10. The program successfully terminates after printing the validation result.

RESULT

Successfully run the program and obtained desired output.

PROGRAM

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

void main() {
  char stack[20], ip[20], opt[10][10][1], ter[10];
  int i, j, k, n, top = 0, col, row;
  for (i = 0; i < 10; i++) {
    stack[i] = NULL;
    ip[i] = NULL;
    for (j = 0; j < 10; j++) {
      opt[i][j][1] = NULL;
    }
  }
  printf("Enter the no.of terminals :\n");
  scanf("%d", &n);
  printf("\nEnter the terminals :\n");
  scanf("%s", &ter);
  printf("\nEnter the table values :\n");
  for (i = 0; i <n; i++) {
    for (j = 0; j < n; j++) {
      printf("Enter the value for %c %c:", ter[i],
ter[j]);
      scanf("%s", opt[i][j]);
    }
  }
  printf("\n**** OPERATOR PRECEDENCE
TABLE ****\n");
  for (i = 0; i < n; i++) {
    printf("\t%c",ter[i]);
  }
  printf("\n");
  for (i = 0; i < n; i++) {
    printf("\n%c", ter[i]);
    for (j = 0; j < n; j++) {
      printf("\t%c", opt[i][j][0]);
    }
  }
  stack[top] = '$';
  printf("\nEnter the input string:");
  scanf("%s", ip);
  i = 0;
  printf("\nSTACK\t\t\tINPUT
STRING\t\t\tACTION\n");
  printf("\n%s\t\t\t%s\t\t\t", stack, ip);
  while (i <= strlen(ip)) {
    for (k = 0; k < n; k++) {
      if (stack[top] == ter[k])
        col = k;
      if (ip[i] == ter[k])
        row = k;
    }
    if ((stack[top] == '$') && (ip[i] == '$')) {
      printf("String is accepted\n");
      break;
    } else if ((opt[col][row][0] == '<') ||
(opt[col][row][0] == '=')) {
      stack[++top] = opt[col][row][0];
      stack[++top] = ip[i];
      printf("Shift %c", ip[i]);
      i++;
    } else {
      if (opt[col][row][0] == '>') {
        while (stack[top] != '<') {
          --top;
        }
        top = top - 1;
        printf("Reduce");
      } else {
        printf("\nString is not accepted");
        break;
      }
    }
    printf("\n");
    for (k = 0; k <= top; k++) {
      printf("%c", stack[k]);
    }
    printf("\t\t\t");
    for (k = i; k < strlen(ip); k++) {
      printf("%c", ip[k]);
    }
    printf("\t\t\t");
  }
  getchar();
}
```

Date :

EXPERIMENT - 12

OPERATOR PRECEDENCE PARSER

AIM

To develop an operator precedence parser for a given language.

ALGORITHM:

1. Initialize arrays stack[20], ip[20], opt[10][10][1], and ter[10]. Set variables i, j, k, n, top = 0, col, row.
2. Set all elements of stack and ip to NULL. Set opt[i][j][1] to NULL in a nested loop.
3. Prompt the user to enter the number of terminals n.
4. Read the terminal symbols into the ter array.
5. In a nested loop, input the precedence table values (<, =, >) for each terminal pair.
6. Store the values in the array opt[i][j].
7. Print the operator precedence table for all terminal pairs.
8. Set stack[0] = '$' (bottom of the stack).
9. Print the initial stack and prompt the user to input the string ip.
10. Start a loop while i <= strlen(ip).
11. Find the column col and row row in the precedence table using stack[top] and ip[i].
12. If stack[top] == '$' and ip[i] == '$', print "String is accepted". Exit the loop.
13. If opt[col][row][0] == '<' or '=', push opt[col][row][0] and ip[i] onto the stack. Print "Shift ip[i]". Increment i.
14. If opt[col][row][0] == '>', pop symbols from the stack until stack[top] == '<'. Pop one more symbol. Print "Reduce".
15. If no valid precedence entry, print "String is not accepted". Exit the loop.
16. Print the current stack and remaining input after every shift or reduce.
17. Repeat steps 10-16 until the input string is either accepted or rejected.
18. Exit after accepting or rejecting the string.

OUTPUT

```
asma@asma:~/Downloads/Cycle 3$ gcc 1_operator_grammer.c -w
asma@asma:~/Downloads/Cycle 3$ ./a.out
Enter the no.of terminals :
4

Enter the terminals :
i+*$

Enter the table values :
Enter the value for i i:>
Enter the value for i +:>
Enter the value for i *:>
Enter the value for i $:>
Enter the value for + i:<
Enter the value for + +:>
Enter the value for + *:<
Enter the value for + $:>
Enter the value for * i:<
Enter the value for * +:>
Enter the value for * *:>
Enter the value for * $:>
Enter the value for $ i:<
Enter the value for $ +:<
Enter the value for $ *:<
Enter the value for $ $:>

**** OPERATOR PRECEDENCE TABLE ****
        i       +       *       $

i       >       >       >       >
+       <       >       <       >
*       <       >       >       >
$       <       <       <       >
Enter the input string:i+i*i$

STACK                   INPUT STRING                    ACTION

$                       i+i*i$                  Shift i
$<i                     +i*i$                   Reduce
$                       +i*i$                   Shift +
$<+                     i*i$                    Shift i
$<+<i                   *i$                     Reduce
$<+                     *i$                     Shift *
$<+<*                   i$                      Shift i
$<+<*<i                 $                       Reduce
$<+<*                   $                       Reduce
$<+                     $                       Reduce
$                       $                       String is accepted
asma@asma:~/Downloads/Cycle 3$
```

RESULT

Successfully simulated the working of operator precedence parser.

PROGRAM

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

int n;
char prods[50][50];
char firsts[26][50];
int is_first_done[26];
char follows[26][50];
int is_follow_done[26];

int isTerminal(char c)
{
   if (c < 65 || c > 90)
      return 1;
   return 0;
}

void first(char nonterm)
{
   int index = 0;
   char curr_firsts[50];
   for (int i = 0; i < n; i++)
   {
      if (prods[i][0] == nonterm)
      {
         int curr_prod_index = 2;
         int flag = 0;
         while (prods[i][curr_prod_index] != '\0'
&& flag == 0)
         {
            flag = 1;
            if
(isTerminal(prods[i][curr_prod_index]))
            {
               curr_firsts[index] = prods[i][2];
               index++;
               break;
            }
            if
(!is_first_done[prods[i][curr_prod_index] - 65])
               first(prods[i][curr_prod_index]);
            int in = 0;
            while (firsts[prods[i][curr_prod_index] -
65][in] != '\0')
            {
               curr_firsts[index] =
firsts[prods[i][curr_prod_index] - 65][in];
               if (firsts[prods[i][curr_prod_index] -
65][in] == 'e')
               {
                  curr_prod_index++;
                  flag = 0;
               }
               index++;
               in++;
            }
         }
      }
   }
   curr_firsts[index] = '\0';
   index++;
   strcpy(firsts[nonterm - 65], curr_firsts);
   is_first_done[nonterm - 65] = 1;
}

void follow(char nonterm)
{
   int index = 0;
   char curr_follows[50];
   if (nonterm == prods[0][0])
   {
      curr_follows[index] = '$';
      index++;
   }
   for (int j = 0; j < n; j++)
   {
      int k = 2;
      int include_lhs_flag;
      while (prods[j][k] != '\0')
      {
         include_lhs_flag = 0;
         if (prods[j][k] == nonterm)
         {
            if (prods[j][k + 1] != '\0')
            {
               if (isTerminal(prods[j][k + 1]))
               {
                  curr_follows[index] = prods[j][k + 1];
                  index++;
                  break;
               }
               int in = 0;
```

Date :

EXPERIMENT - 13
SIMULATION OF FIRST AND FOLLOW OF GRAMMAR

AIM

To simulate first and follow of any given grammar.

ALGORITHM:

1.Input the number of productions and the grammar rules.

2.Initialize arrays for storing First and Follow sets, and flags to track if the computation for each non-terminal is done.

3. For each non-terminal, find the First set:

    3.1 If the first symbol in the production is a terminal, add it to the First set.

    3.2 If the first symbol is a non-terminal, recursively compute its First set.

    3.3 If the First set of the non-terminal contains 'ε', continue checking the next symbol in the production.

4. Mark the First set computation for the non-terminal as done.

5. For the start symbol, add '$' to the Follow set.

6. For each production, find the Follow set:

    6.1 If a non-terminal is followed by a terminal, add that terminal to the Follow set.

    6.2 If a non-terminal is followed by another non-terminal, add the First set of the latter (excluding 'ε') to the Follow set.

    6.3 If a non-terminal appears at the end of a production or is followed by a non-terminal whose First set contains 'ε', add the Follow set of the left-hand side of the production to the current non-terminal's Follow set.

7. Mark the Follow set computation for the non-terminal as done.

8. Print the First and Follow sets for each non-terminal.

```
            while (firsts[prods[j][k + 1] - 65][in]
!= '\0')
            {
               if (firsts[prods[j][k + 1] - 65][in]
== 'e')
               {
                  include_lhs_flag = 1;
                  in++;
                  continue;
               }
               int temp_flag = 0;
               for (int z = 0; z < index; z++)
                  if (firsts[prods[j][k + 1] - 65][in]
== curr_follows[z])
                  {
                     temp_flag = 1;
                     in++;
                     break;
                  }
               if (temp_flag)
                  continue;
               curr_follows[index] =
firsts[prods[j][k + 1] - 65][in];
               index++;
               in++;
            }
         }
         if (prods[j][k + 1] == '\0' ||
include_lhs_flag == 1)
         {
            if (prods[j][0] != nonterm)
            {
               if (!is_follow_done[prods[j][0] -
65])
                  follow(prods[j][0]);
               int x = 0;
               while (follows[prods[j][0] - 65][x]
!= '\0')
               {
                  int temp_flag = 0;
                  for (int z = 0; z < index; z++)
                     if (follows[prods[j][0] - 65][x]
== curr_follows[z])
                     {
                        temp_flag = 1;
                        x++;
                        break;
                     }
                  if (temp_flag)
                     continue;
                  curr_follows[index] =
follows[prods[j][0] - 65][x];
                  index++;
                  x++;
               }
            }
         }
         k++;
      }
   }
   curr_follows[index] = '\0';
   index++;
   strcpy(follows[nonterm - 65], curr_follows);
   is_follow_done[nonterm - 65] = 1;
}

int main()
{
   printf("Enter the number of productions\n");
   scanf("%d", &n);
   printf("Enter productions: \n");
   for (int i = 0; i < n; i++)
      scanf("%s", prods[i]);
   for (int i = 0; i < 26; i++)
      is_first_done[i] = 0;
   for (int i = 0; i < n; i++)
      if (is_first_done[prods[i][0] - 65] == 0)
         first(prods[i][0]);
   for (int i = 0; i < n; i++)
      if (is_follow_done[prods[i][0] - 65] == 0)
         follow(prods[i][0]);
   printf("Firsts:\n");
   for (int i = 0; i < 26; i++)
      if (is_first_done[i])
         printf("%c : %s\n", i + 65, firsts[i]);
   printf("Follows:\n");
   for (int i = 0; i < 26; i++)
      if (is_follow_done[i])
         printf("%c : %s\n", i + 65, follows[i]);
}
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 3$ gcc 2_first_follow.c
asma@asma:~/Downloads/Cycle 3$ ./a.out
Enter the number of productions
4
Enter productions:
S=A
A=aB/Ad
B=b
C=g
Firsts:
A : a
B : b
C : g
S : a
Follows:
A : $d
B : /
C :
S : $
asma@asma:~/Downloads/Cycle 3$ ./a.out
Enter the number of productions
8
Enter productions:
E=TM
M=aTM
M=e
T=FN
N=mFN
N=e
F=rEl
F=i
Firsts:
E : ri
F : ri
M : ae
N : me
T : ri
Follows:
E : $l
F : ma$l
M : $l
N : a$l
T : a$l
asma@asma:~/Downloads/Cycle 3$
```

RESULT

Successfully simulated first and follow of any given grammar.

PROGRAM

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

char ip_sym[15],ip_ptr=0,op[50],tmp[50];
void e_prime();
void e();
void t_prime();
void t();
void f();
void advance();
int n=0;
void e()
{
 strcpy(op,"TE'");
 printf("E=%-25s",op);
 printf("E->TE'\n");
 t();
 e_prime();
}
void e_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
    if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='E';n++);
if(ip_sym[ip_ptr]=='+')
 {
    i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i<=l);
  op[n++]='+';
  op[n++]='T';
  op[n++]='E';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("E'->+TE'\n");
  advance();
  t();
 e_prime();
 }
 else
 {
    op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("E'->e");
 }
}
void t()
{
 int i,n=0,l;
 for(i=0;i<=strlen(op);i++)
  if(op[i]!='e')
   tmp[n++]=op[i];
 strcpy(op,tmp);
 l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);
 i=n+1;
 do
 {
  op[i+2]=op[i];
  i++;
 }while(i < l);
 op[n++]='F';
 op[n++]='T';
 op[n++]=39;
 printf("E=%-25s",op);
 printf("T->FT'\n");
 f();
 t_prime();
}

void t_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
    if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);
if(ip_sym[ip_ptr]=='*')
 {
    i=n+2;
 do
```

Date :

EXPERIMENT - 14

RECURSIVE DESCENT PARSER

AIM

To construct a recursive descent parser for an expression.

ALGORITHM:

1. Start.
2. Declare necessary headers.
3. Define global variables:
    a) Define an array input[] to store the input string.
    b) Pointer to track the current position in the input string.
    c) Temporary arrays like production[] to store and manipulate intermediate parsing results.
4. Define parse_expression():
    a) Apply the rule E->T E'.
    b) Print the current production .
    c) Call parse_term() to handle T.
    d) Call parse_expression_prime() to process E'.
5. Define parse_expression_prime():
    a) Apply the rule E'->+TE'| ε.
    b) If the current input symbol is +, modify the production to E'->+TE', print it, and recursively call parse_term() and parse_expression_prime().
    c) If no + is found, apply epsilon (ε) and terminate.
6. Define parse_term():
    a) Apply the rule T->FT'.
    b) Print the current production .
    c) Call parse_factor() to handle F.
    d) Call parse_term_prime() to process T'.

```
 {
 op[i+2]=op[i];
 i++;
 }while(i < l);
  op[n++]='*';
  op[n++]='F';
  op[n++]='T';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("T'->*FT'\n");
  advance();
  f();
  t_prime();
 }
 else
 {
   op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("T'->e\n");
 }
}

void f()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
    if(op[i]!='e')
 tmp[n++]=op[i];
 strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='F';n++);
 if((ip_sym[ip_ptr]=='i')||(ip_sym[ip_ptr]=='I'))
 {
 op[n]='i';
 printf("E=%-25s",op);
 printf("F->i\n");
 advance();
 }
 else
 {
  if(ip_sym[ip_ptr]=='(')
  {
   advance();
   e();
   if(ip_sym[ip_ptr]==')')
   {
    advance();
     i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i<=l);
  op[n++]='(';
  op[n++]='E';
  op[n++]=')';
  printf("E=%-25s",op);
  printf("F->(E)\n");
   }
  }
  else
  {
   printf("\n\t syntax error");
     exit(1);
  }
 }
}

void advance()
{
 ip_ptr++;
}
void main()
{
 int i;
  printf("\nGrammar without left recursion");
 printf("\n\t\t E->TE' \n\t\t E'->+TE'|e \n\t\t T->FT'
");
 printf("\n\t\t T'->*FT'|e \n\t\t F->(E)|i");
 printf("\n Enter the input expression:");
 scanf("%s",ip_sym);
 printf("Expressions");
 printf("\t Sequence of production rules\n");
  e();
  for(i=0;i < strlen(ip_sym);i++)
 {
if(ip_sym[i]!='+'&&ip_sym[i]!='*'&&ip_sym[i]!=
'('&&
ip_sym[i]!=')'&&ip_sym[i]!='i'&&ip_sym[i]!='I')
  {
   printf("\nSyntax error");
   break;
  }
  for(i=0;i<=strlen(op);i++)
```

7. Define parse_term_prime():
    a) Apply the rule T'->*FT'|ε.
    b) If the current input symbol is *, modify the production to T'->*FT', print it, and recursively call parse_factor() and parse_term_prime().
    c) If no * is found, apply epsilon (ε) and terminate.
8. Define parse_factor():
    a) Apply the rule F->(E)|id.
    b) If the current input symbol is id, accept it, update the production, and print it.
    c) If the current input is (, recursively call parse_expression() to handle the contents inside parentheses, followed by checking for the closing).
9. Define advance_input():
    a) Move the input pointer to the next character.
10. In main function():
    a) Read input string from the user.
    b) Call parse_expression() to initiate parsing.
    c) Print syntax errors if the input doesn't match the expected grammar rules.
    d) Output the successful parsing sequence if no errors are found.
11. Stop.

```
    if(op[i]!='e')
  tmp[n++]=op[i];
    strcpy(op,tmp);
    printf("\nE=%-25s",op);
  }
 }
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 3$ gcc 3_recursive_descent.c
asma@asma:~/Downloads/Cycle 3$ ./a.out

Grammar without left recursion
                E->TE'
                E'->+TE'|e
                T->FT'
                T'->*FT'|e
                F->(E)|i
 Enter the input expression:i+i*i
Expressions     Sequence of production rules
E=TE'                       E->TE'
E=FT'E'                     T->FT'
E=iT'E'                     F->i
E=ieE'                      T'->e
E=i+TE'                     E'->+TE'
E=i+FT'E'                   T->FT'
E=i+iT'E'                   F->i
E=i+i*FT'E'                 T'->*FT'
E=i+i*iT'E'                 F->i
E=i+i*ieE'                  T'->e
E=i+i*ie                    E'->e
E=i+i*i                     asma@asma:~/Downloads/Cycle 3$
```

RESULT

Successfully constructed a recursive descent parser for an expression.

PROGRAM

```
#include <stdio.h>
#include <string.h>
#define MAX_INPUT 100
#define MAX_STACK 100

void check();

char input[MAX_INPUT],
stack[MAX_STACK], action[20];
int input_len, stack_top = -1;

int main() {
   printf("GRAMMAR is:\nE -> E+E | E*E | (E)
| id\n");
   printf("Enter input string: ");
   scanf("%s", input);
   input_len = strlen(input);
   printf("\nStack\t\tInput\t\tAction\n");
   for (int i = 0; i < input_len; i++) {
      if (input[i] == 'i' && input[i+1] == 'd') {
         stack[++stack_top] = 'i';
         stack[++stack_top] = 'd';
         stack[stack_top + 1] = '\0';
         printf("$%s\t\t%s$\t\tSHIFT->id\n",
stack, input + i + 2);
         check();
         i++;  // Skip the 'd' in the next iteration
      } else {
         stack[++stack_top] = input[i];
         stack[stack_top + 1] = '\0';
         printf("$%s\t\t%s$\t\tSHIFT->%c\n",
stack, input + i + 1, input[i]);
         check();
      }
   }
   if (stack_top == 0 && stack[0] == 'E') {
      printf("\nInput string is VALID.\n");
   } else {
      printf("\nInput string is INVALID.\n");
   }
   return 0;
}

void check() {
   int i, j, handle_size;
   char *handle;
   while (1) {
      if (stack_top >=1 && stack[stack_top-1]
== 'i' && stack[stack_top] == 'd') {
         handle = "id";
         handle_size = 2;
      } else if (stack_top >= 2 &&
stack[stack_top-2] == 'E' && stack[stack_top-1]
== '+' && stack[stack_top] == 'E') {
         handle = "E+E";
         handle_size = 3;
      } else if (stack_top >= 2 &&
stack[stack_top-2] == 'E' && stack[stack_top-1]
== '*' && stack[stack_top] == 'E') {
         handle = "E*E";
         handle_size = 3;
      } else if (stack_top >= 2 &&
stack[stack_top-2] == '(' && stack[stack_top-1]
== 'E' && stack[stack_top] == ')') {
         handle = "(E)";
         handle_size = 3;
      } else {
         return;
      }
      stack_top -= handle_size - 1;
      stack[stack_top] = 'E';
      stack[stack_top + 1] = '\0';
      printf("$%s\t\t%s$\t\tREDUCE->%s\n",
stack, input + strlen(input), handle);
   }
}
```

Date :

EXPERIMENT - 15

SHIFT REDUCE PARSER

AIM

To simulate the working of a shift reduce parser.

ALGORITHM:

1. Initialization

   1.1. Initialize ‘input[]’, ‘stack[]’, ‘action[]’, and ‘stack_top = -1’.

   1.2. Set ‘input_len’ as the length of the input string.

2. Display Grammar

   2.1. Display the grammar:

   - ‘E -> E + E’, ‘E -> E * E’, ‘E -> (E)’, ‘E -> id’.

3. Input the String

   3.1. Prompt the user for input.

   3.2. Read the input into ‘input[]’.

4. Shift Operation

   4.1. For each character in ‘input[]’:

   4.2. If ‘id’ is found:

   - Push ‘id’ to the stack.
   - Print stack, remaining input, and action ‘SHIFT->id’.
   - Call ‘check()’ and skip ‘d’ by incrementing ‘i’ by 1.

   4.3. For other characters:

   - Push the character to the stack.
   - Print stack, remaining input, and action ‘SHIFT-><character>‘.
   - Call ‘check()’.

OUTPUT

```
asma@asma:~/Downloads/Cycle 3$ gcc 4_shift_reduce.c
asma@asma:~/Downloads/Cycle 3$ ./a.out
GRAMMAR is:
E -> E+E | E*E | (E) | id
Enter input string: id+id*id+id

Stack           Input           Action
$id             +id*id+id$              SHIFT->id
$E              $               REDUCE->id
$E+             id*id+id$               SHIFT->+
$E+id           *id+id$         SHIFT->id
$E+E            $               REDUCE->id
$E              $               REDUCE->E+E
$E*             id+id$          SHIFT->*
$E*id           +id$            SHIFT->id
$E*E            $               REDUCE->id
$E              $               REDUCE->E*E
$E+             id$             SHIFT->+
$E+id           $               SHIFT->id
$E+E            $               REDUCE->id
$E              $               REDUCE->E+E

Input string is VALID.
asma@asma:~/Downloads/Cycle 3$
```

5. Reduction (check() Function)

   5.1. Check for handles on the stack:

   - If ‘id’ → reduce to ‘E’.

   - If ‘E + E’, ‘E * E’, ‘(E)’ → reduce to ‘E’.

   5.2. If a handle is found:

   - Perform reduction and print stack, input, and action ‘REDUCE-><handle>‘.

   5.3. Repeat until no further reductions are possible.

6. Final Validation

   6.1. If stack contains only ‘E’, print "VALID".

   6.2. Otherwise, print "INVALID".

RESULT

Successfully simulated the working of a shift reduce parser.

PROGRAM

```
#include <stdio.h>
#include <string.h>
void gen_code_for_operator(char *inp, char
operator, char * reg)
{    int i = 0, j = 0;
   char temp[100];
   while (inp[i] != '\0')
   {
      if (inp[i] == operator)
      {
         printf("%c\t%c\t%c\t%c\n", operator, *
reg, inp[i - 1], inp[i + 1]);
         temp[j - 1] = *reg;
         i += 2;
         (*reg)--;
         continue;
      }
      temp[j] = inp[i];
      i++;
      j++;
   }
   temp[++j] = '\0';
   strcpy(inp, temp);
}

void gen_code(char *inp)
{
   // Operator precedence - /, *, +, -, =
   char reg = 'Z';
   gen_code_for_operator(inp, '/', &reg);
   gen_code_for_operator(inp, '*', &reg);
   gen_code_for_operator(inp, '+', &reg);
   gen_code_for_operator(inp, '-', &reg);
   gen_code_for_operator(inp, '=', &reg);
}
int main()
{
   char inp[100];
   printf("Enter expression:\n\n");
   scanf("%s", inp);
   printf("Op  \tDestn\tArg1\tArg2\n");
   gen_code(inp);
}
```

OUTPUT

```
asma@asma:~/Downloads/Cycle 4$ gcc 1_intermediate_code.c
asma@asma:~/Downloads/Cycle 4$ ./a.out
Enter expression:

p=a*b+c/d-2
Op      Destn   Arg1    Arg2
/       Z       c       d
*       Y       a       b
+       X       Y       Z
-       W       X       2
=       V       p       W
asma@asma:~/Downloads/Cycle 4$
```

Date :

EXPERIMENT - 16

INTERMEDIATE CODE GENERATOR

AIM

To implement an intermediate code generator for simple expressions.

ALGORITHM

1. Start
2. Declare necessary headers.
3. Declare a variable reg initialized to 'Z' to store intermediate results.
4. Read the input string inp(arithmetic expression)
5. Define the order of operator precedence: /, *, +, -, =
6. For each operator from highest precedence to lowest
    a) Traverse the input string inp character by character
    b) If an operator matches the current operator:
        i) Print the operator and the operands (characters before and after the operator) as three-address code.
        ii) Assign the result to the current register (reg).
        iii) Update the expression by replacing the operands and operator with the register.
        iv) Decrement the register to use the next available register for further operations.
    c) Continue the traversal until the end of the string.
7. Repeat step 6 for the next operator in the precedence order, continuing until all operators are processed
8. Stop

RESULT

Successfully implement an intermediate code generator for simple expressions.