**Date:**

**1. Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines**

**Aim:** ToWrite a Program to Design Lexical Analyzer.

**Description:**

The first phase of compiler, the lexical analyzer or scanner separates characters of the source language into groups that logically belong together called tokens.

The input to lexical analyzer is the source program and the output of lexical analyzer is the stream of tokens.

Reasons for separating lexical analysis from syntax analysis:

1)To reduce the complexity in each phase.

2)To improve the efficiency of the compiler.

**Algorithm For Lexical Analyzer:**

function lexan :integer;

var lexbuf:array[1……………………100] of char;

c:char;

begin

loop begin

read a character into c;

If c is a blank or a tab then

do nothing

else if c is a newline then

lineno:=lineno+1

else if c is a digit then begin

set tokenval to the value of this and

following digits;

return NUM

end

else if c is a letter then begin

place c and successive letters and digits into

lexbuf;

p:=lookup(lexbuf);

If p=0 then

p:=insert(lexbuf,ID);

tokenval:=p;

return the token field of table entry p

end

else begin

set tokenval to NONE;

return integer encoding of character c

end

end

end

**Program:**

#include<string.h>

#include<ctype.h>

#include<stdio.h>

void keyword(char str[10])

{

if(strcmp("main",str)==0||

strcmp("for",str)==0||

strcmp("void",str)==0||

strcmp("while",str)==0||

strcmp("do",str)==0||

strcmp("int",str)==0||

strcmp("float",str)==0||

strcmp("char",str)==0||

strcmp("double",str)==0||

strcmp("static",str)==0||

strcmp("switch",str)==0||

strcmp("case",str)==0)

printf("\n %s is a keyword",str);

else

printf("\n %s is an identifier",str);

}

void main()

{

FILE \*f1,\*f2,\*f3,\*f4;

char c,str[10];

int a,num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;

printf("\nEnter the c program:");

printf("\nEnter '@' to end the program :\n");

f1=fopen("input","w");

while((c=getchar())!='@')

putc(c,f1);

fclose(f1);

f1=fopen("input","r");

f2=fopen("identifier","w");

f3=fopen("specialcharacter","w");

f4=fopen("const","w");

while((c=getc(f1))!=EOF)

{

if(isdigit(c))

{

putc(c,f4);

c=getc(f1);

while(isdigit(c))

{

putc(c,f4);

c=getc(f1);

}

num[i++]=tokenvalue;

putc(' ',f4);

ungetc(c,f1);

}

else

if(isalpha(c))

{

putc(c,f2);

c=getc(f1);

while(isdigit(c)||isalpha(c)||c=='\_'=='$')

{

putc(c,f2);

c=getc(f1);

}

putc(' ',f2);

ungetc(c,f1);

}

else

if(c==' '||c=='\t')

printf(" ");

else

if(c=='\n')

lineno++;

else

putc(c,f3);

}

fclose(f2);

fclose(f3);

fclose(f1);

fclose(f4);

f2=fopen("identifier","r");

k=0;

printf("the keywords and identifier are:");

while((c=getc(f2))!=EOF)

{

if(c!=' ')

str[k++]=c;

else

{

str[k]='\0';

keyword(str);

k=0;

}

}

fclose(f2);

f3=fopen("specialcharacter","r");

printf("\nspecial character are");

while((c=getc(f3))!=EOF)

printf("%c",c);

printf("\n");

fclose(f3);

f4=fopen("const","r");

printf("Constatnts are:");

while(((char)a=getc(f4))!=EOF)

printf("%c",a);

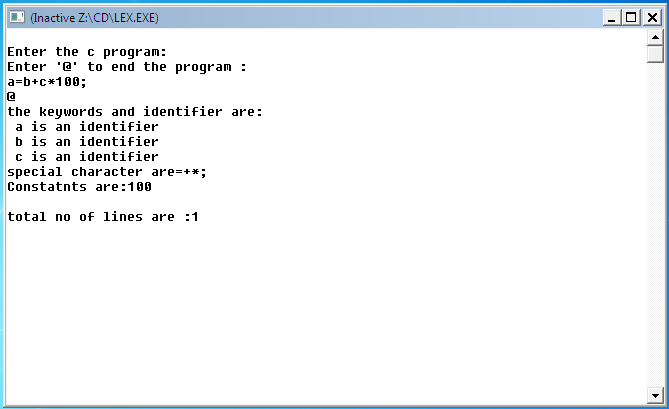
printf("\n");

fclose(f4);

printf("\ntotal no of lines are :%d",lineno);

}

**Output:**

****

**VIVA QUESTIONS**

1. What is compiler?
2. What is interpreter ?
3. Define Translator and bootstrapping?
4. What do you mean by linker and loader ?
5. What are the phases of compilers?

6. Define Role of Assembler?

7. What is symbol table?

8. Define lexeme, token , patterns?

9. Explain about error detection and correction?

10. Differentiate between phase and a pass?

**Date:**

**2. Simulate FIRST and FOLLOW of a grammar**

**a) Simulate FIRST function.**

**Aim:** Write a program to compute FIRST function.

**Description:**

The FIRST**()** of the Non-Terminal is to represent the terminals.If α is any string of grammar symbols, let FIRST(α) be the set of terminals thet begin the strings derived from α. If α->ε, then ε is also in FIRST(α).

**Algorithm:**

Step 1: Start

Step 2: Declare FILE \*fp

Step 3: open the file in.txt in write mode

Step 4: Read the Productions

Step 5: Compute Follow function

Step 6: stop the productions

**Program:**

#include<stdio.h>

#include<ctype.h>

void First(char);

int count,n=0;

char prod[10][10],f[10];

int main()

{

int i,choice;

char c,ch;

printf("No. of productions:");

scanf("%d",&count);

printf("Enter %d productions (Epsilon=$):\n",count);

for(i=0;i<count;i++)

{

scanf("%s%c",&prod[i],&ch);

}

do

{

n=0;

printf("Element:");

scanf("%c",&c);

First(c);

printf("\nFirst(%c)={",c);

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

printf(" %c",f[i]);

printf(" }\n");

printf("Press 1 to continue:");

scanf("%d%c",&choice,&ch);

//printf("%c",ch);

}while(choice==1);

return 0;

}

void First(char c)

{

int j;

if(!isupper(c))

f[n++]=c;

for(j=0;j<count;j++)

{

if(prod[j][0]==c)

{

if(prod[j][1]=='$')

f[n++]=='$';

else if(islower(prod[j][2]))

f[n++]=prod[j][2];

else

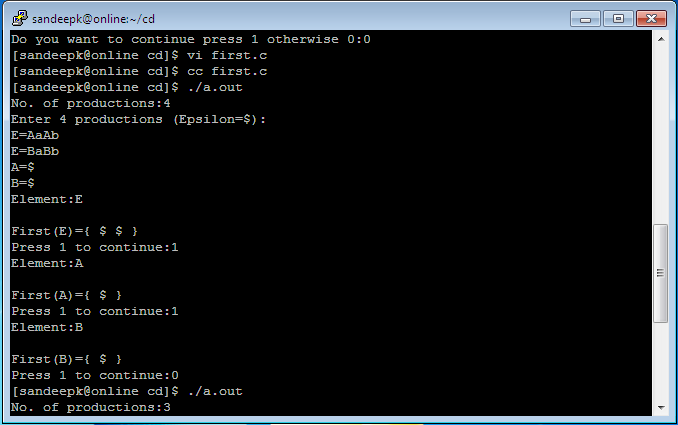
First(prod[j][2]);

}

}

}

**Output:**



**b) Simulate FOLLOW function.**

**Aim:** Write a program to compute FOLLOW(A)

**Description:**

FOLLOW(A) , for non terminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is , the set of terminals a such that there exits a derivation of the form S->αAaβ for some α and β.If A can be the rightmost symbol in sentential form ,then $ is in FOLLOW(A).

**Algorithm:**

Step 1: Start

Step 2: Declare FILE \*fp

Step 3: open the file in.txt in write mode

Step 4: Read the Productions

Step 5: Compute Follow function

Step 6: stop the productions**.**

**Program**

#include<stdio.h>

#include<string.h>

#include<ctype.h>

int n,m=0,p,i=0,j=0;

char a[10][10],f[10];

void Follow(char c);

void First(char c);

int main()

{

int i,z;

char c,ch;

printf("Enter no. of productions:");

scanf("%d",&n);

printf("Enter the production(Epsilon=$):\n");

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

{

scanf("%s%c",a[i],&ch);

}

do

{

m=0;

printf("Enter the elements whose Follow is to be found:");

scanf("%c",&c);

printf("Follow(%c)={",c);

for(i=0;i<m;i++)

printf(" %c",f[i]);

printf(" }\n");

printf("Do you want to continue press 1 otherwise 0:");

scanf("%d%c",&z,&ch);

}while(z==1);

}

void Follow(char c)

{

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

{

for(j=2;j<strlen(a[i]);j++)

{

if(a[i][j]==c)

{

if(a[i][j+1]!='\0')

First(a[i][j+1]);

if((a[i][j+1]=='\0')&&(c!=a[i][0]))

Follow(a[i][0]);

}

}

}

if(a[0][0]==c)

f[m++]='$';

}

void First(char c)

{

int k;

if(!isupper(c))

f[m++]=c;

for(k=0;k<n;k++)

{

if(a[k][0]==c)

{

if(a[k][2]=='$')

Follow(a[i][0]);

else if(islower(a[k][2]))

f[m++]=a[k][2];

else

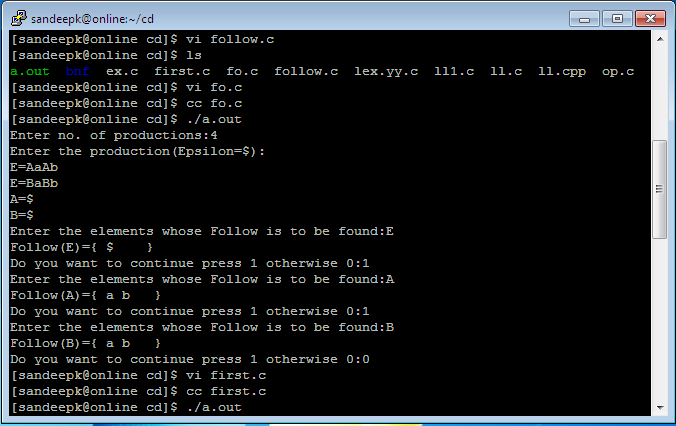
First(a[k][2]);

}

}

}

**Output:**

****

**VIVA QUESTIONS**

1. Define left factoring?

2. Define left recursion?

3. Explain role of parser?

4. What do you mean by CFG?

5. What is FIRST()?

**6. What is FOLLOW()**

**Date:**

**3. Develop an Operator precedence parser for a given grammar.**

**Aim:** Write a program to implement operator precedence parsing

**Description:**

In operator precedence parser , the parser selects the four actions shift ,reduce ,accept ,reject. This table relates the precedence of one operator over the other. Therefore the parser needs an input buffer , an output buffer, a stack the operator precedence table, and the parse routine. It stores the precedence relations between adjacent terminals.

**Algorithm:**

Step 1: start.

Step 2: Declare the prototypes for functions.

Step 3: Enter the String like id\*id+id

Step 4: Read the string and analyze tokens, identifiers, variables.

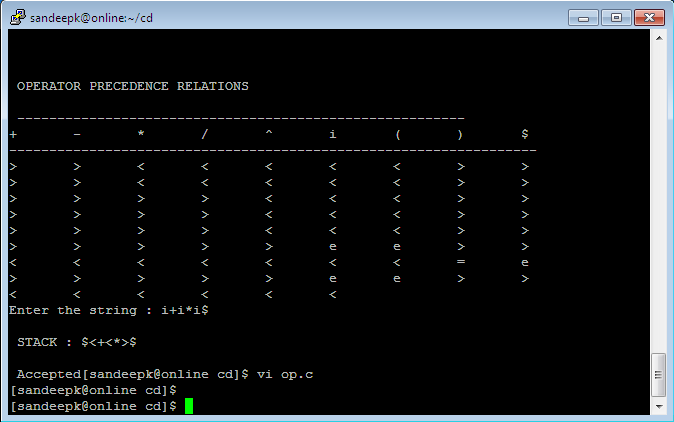
Step 5: Display the operator precedence table.

Step 6: stop.

**Program:**

#include<stdio.h>   
#include<string.h>   
#include<stdlib.h>   
char stack[20],stack1[20],next,s[10];   
int top=-1;   
char prod[9][10]={   
">><<<<<>>",   
">><<<<<>>",   
">>>><<<>>",   
">>>><<<>>",   
">>>><<<>>",   
">>>>>ee>>",   
"<<<<<<<=e",   
">>>>>ee>>",   
"<<<<<<  
};   
char G[7][6]={   
"E->E+E",   
" /E-E",   
" /E\*E",   
" /E/E",   
" /(E)",   
" /i "   
};   
  
  
int main()   
{   
char symbol;   
int i=0,flag=0;   
int j,k;   
clrscr();   
printf("Grammar\n");   
for(j=0;j<7;j++)   
{   
for(k=0;k<6;k++)   
printf("%c",G[j][k]);   
printf("\n");   
}   
printf("\n\n OPERATOR PRECEDENCE RELATIONS \n");   
printf("\n -------------------------------------------------------- \n");   
printf("%c\t%c\t%c\t%c\t%c\t%c\t%c\t%c\t%c\t",'+','-','\*','/','^','i','(',')','$');   
printf("\n------------------------------------------------------------------\n");   
for(j=0;j<9;j++)   
{   
for(k=0;k<10;k++)   
printf("%c\t",prod[j][k]);   
printf("\n");   
}   
printf("Enter the string : ");   
gets(s);   
++top;   
stack[top]='$';   
next=s[i];   
while(1)   
{   
if(stack[top]=='$'&& next=='$'||next=='\0')   
break;   
else   
{   
symbol=prod[f(stack[top])][f(next)];   
if(symbol=='<'||symbol=='=')   
{   
stack[++top]=symbol;   
stack[++top]=next;   
}   
else if(symbol=='>')   
{   
do   
{   
top--;   
}while(stack[top]!='<');   
stack[++top]=next;   
if(next!='$')   
{   
for(j=0;j<=top;j++)   
stack1[j]=stack[j];   
stack1[j]=symbol;   
}   
}   
else   
flag=1;   
next=s[++i];   
}   
}   
  
printf("\n STACK : ");   
for(j=0;j<=top;j++)   
printf("%c",stack1[j]);   
printf("%c",'$');   
if(flag==0)   
printf("\n\n Accepted");   
else   
printf("Rejected");   
return 0;   
}   
  
  
int f(char ch)   
{   
switch(ch)   
{   
case '+':return 0;   
case '-':return 1;   
case '\*':return 2;   
case '/':return 3;   
case '^':return 4;   
case 'i':return 5;   
case '(':return 6;   
case ')':return 7;   
case '$':return 8;   
default :   
{   
printf("\n ERROR ");   
exit(0);   
}   
}   
} 

**Output:**



**VIVA QUESTIONS**

1. **Classification of parsing techniques?**

**2. Define Ambiguity?**

**3. What is the purpose of OPP?**

**4. What do you mean by associative rule?**

**5. What are the relations used in OPP**

**6. Define Simple parsers?**

**Date:**

**4. Construct a recursive descent parser for an expression.**

**Aim:** To write a program on recursive descendent parser.

**Description:**

Recursive descent parsing is the form of any top-down parser.This is a recursive process of adding descendants to the nodes of parse tree.In these parsing a sentence is parsed using CFG.This process begins by placing the start symbol at the end root of the parse tree and expanding it on encountering each non-terminal with a sub tree equivalent to RHS of the grammar rule.This process continues until all the terminals present in the parse tree.

**Algorithm:**

Step 1: start.

Step 2: Declare the prototype functions E() , EP(),T(), TP(),F()

Step 3: Read the string to be parsed.

Step 4: Check the productions

Step 5: Compare the terminals and Non-terminals

Step 6: Read the parse string.

Step 7: stop the production

**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

{

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++;

}

int 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:");

gets(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++)

if(op[i]!='e')

tmp[n++]=op[i];

strcpy(op,tmp);

printf("\nE=%-25s",op);

}

return 0;

}

**Output:**



**VIVA QUESTIONS**

1. **What is RDP**
2. **What is Predictive parser?**
3. **What is Syntax analysis?**
4. **What are the problems with Top Down parser?**
5. **What is Non Recursive Predictive parsing?**

**6. How to recover the errors in predictive parser?**

7. What is LMD?

8. What is RMD?

**Date:**

**5. Construct a LL(1) parser for an expression**

**Aim:** To write a program on LL(1) parser.

**Description:** With a LL(1) grammar, it is possible, by reading tokens from left to right, to know the derivation step by step. A recursive descendant parser could be generated automatically. The main advantage of this kind of parser is that it enables a programmer to easily understand and modify the generated code. Even though LL(1) grammars are less expressive than LALR grammar for which yacc-like tools have been created, they seem to me rich enough to express good-looking programming languages.

**Algorithm:**

while the firsts table has changed in the precedent loop cycle

for each rule

for each symbol in the right part of the rule

if symbol is terminal

then

firsts[rule.left,symbol] = true

leave symbol loop

else

for each terminal

if firsts[symbol,terminal]

then

firsts[rule.left,terminal] = true

if not(firsts[symbol,epsilon])

then

leave symbol loop

else

if symbol is the last of the right part of the rule

then

firsts[rule.left,epsilon] = true

**Program:**

#include <stdio.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

int main()

{

int len,i=0,j=0,k=0,m=0,n=0,o=0,o1=0,var=0,l=0,f=0,c=0,f1=0;

char str[30],str1[40]="E",temp[20],temp1[20],temp2[20],tt[20],t3[20];

char t[20];

char array[6][5][10]= {

"NT", "<id>","+","\*",";",

"E", "Te","Error","Error","Error",

"e", "Error","+Te","Error","\0",

"T", "Vt","Error","Error","Error",

"t", "Error","\0","\*Vt","\0",

"V", "<id>","Error","Error","Error"

};

temp1[0]='\0';

temp2[0]='\0';

printf("\n\tLL(1) PARSER TABLE \n");

for(i=0;i<6;i++)

{

for(j=0;j<5;j++)

{

printf("%10s",array[i][j]);

}

printf("\n");

}

printf("\n");

printf("\n\tENTER THE STRING :");

gets(str);

if(str[strlen(str)-1] != ';')

{

printf("END OF STRING MARKER SHOULD BE ';'");

//getch();

exit(1);

}

printf("\n\tCHECKING VALIDATION OF THE STRING ");

printf("%s\n\t",str1);

i=0;

while(i<strlen(str))

{

again:

if(str[i] == ' ' && i<strlen(str))

{

printf("\n\tSPACES IS NOT ALLOWED IN SOURSE STRING ");

//getch();

exit(1);

}

temp[k]=str[i];

temp[k+1]='\0';

f1=0;

again1:

if(i>=strlen(str))

{

//getch();

exit(1);

}

for(l=1;l<=4;l++)

{

if(strcmp(temp,array[0][l])==0)

{

f1=1;

m=0,o=0,var=0,o1=0;

temp1[0]='\0';

temp2[0]='\0';

len=strlen(str1);

while(m<strlen(str1) && m<strlen(str))

{

if(str1[m]==str[m])

{

var=m+1;

temp2[o1]=str1[m];

m++;

o1++;

}

else

{

if((m+1)<strlen(str1))

{

m++;

temp1[o]=str1[m];

o++;

}

else

m++;

}

}

temp2[o1] = '\0';

temp1[o] = '\0';

t[0] = str1[var];

t[1] = '\0';

for(n=1;n<=5;n++)

{

if(strcmp(array[n][0],t)==0)

break;

}

strcpy(str1,temp2);

strcat(str1,array[n][l]);

strcat(str1,temp1);

printf("%s\n\t",str1);

//getch();

t[0]='\0';

if(strcmp(array[n][l],t)==0)

{

if(i==(strlen(str)-1))

{

int len=strlen(str1);

str1[len-1]='\0';

printf("%s\n\t",str1);

printf("\n\n\tENTERED STRING IS VALID");

//getch();

exit(1);

}

temp1[0]='\0';

temp2[0]='\0';

t[0]='\0';

goto again1;

}

if(strcmp(array[n][l],"Error")==0)

{

printf("\n\tERROR IN YOUR SOURCE STRING");

//getch();

exit(1);

}

tt[0]='\0';

strcpy(tt,array[n][l]);

t3[0]='\0';

f=0;

for(c=0;c<strlen(tt);c++)

{

t3[c]=tt[c];

t3[c+1]='\0';

if(strcmp(t3,temp)==0)

{

f=0;

break;

}

else

f=1;

}

if(f==0)

{

temp[0]='\0';

temp1[0]='\0';

temp2[0]='\0';

t[0]='\0';

i++;

k=0;

goto again;

}

else

{

temp1[0]='\0';

temp2[0]='\0';

t[0]='\0';

goto again1;

}

}

}

i++;

k++;

}

if(f1==0)

printf("\nENTERED STRING IS INVALID");

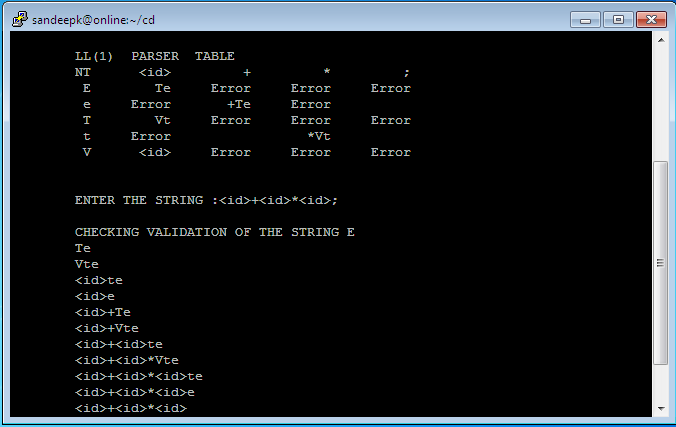
else

printf("\n\n\tENTERED STRING IS VALID");

return 0;

}

**Output:**



**VIVA QUESTIONS**

1. **What is LL(1)?**
2. **What Parameter passing mechanisms?**
3. **What is Intermediate code generator ?**
4. **Which is an optional phase ?**
5. **What is DAG?**

**6. What is three address code?**

**Date:**

**6. Design predictive parser for the given language.**

**Aim:** To Write a Program for Implementation of Predictive Parser.

**Description:**

Predictive parsing is special form of recursive descent parsing called predictive parsing. In which the look-ahead symbol unambiguously determine the procedure selected for each non terminal. The sequence of procedures called in processing the input implicitly define a parse tree for input.

In many cases by carefully writing a grammar ,we can obtain a grammar that can be parsed by a recursive descent parser that needs no back tracking, i.e., a predictive parse. Flow of control constricts in most programming languages with their distinguishing

are usually detectable in these way .

**Algorithm:**

procedure match(t:token);

begin

if lookahead=t then

lookahead:=nexttoken

else error

end; V

procedure type;

begin

if lookahead is in {integer,char,num} then

simple

else if lookahead=’-> ‘ then begin

match(‘->’); match(id)

end

else if lookahead=array then begin

match(array);match(‘[‘);simple;match(‘]’);match(of);type

end

else error

end;

procedure simple;

begin

if lookahead=integer then

match(integer)

else if lookahead=char then

match(char)

else if lookahead=num then begin

match(num);match(dotdot);match(num)

end

else error

end;

**Program**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

int main()

{

char pro[10][10],first[10][10],follow[10][10],nt[10],ter[10],res[10][10][10],temp[10];

int npro,noter=0,nont=0,i,j,k,flag=0,count[10][10],row,col,l,m,n,index;

//clrscr();

for(i=0;i<10;i++)

{

for(j=0;j<10;j++)

{

count[i][j]=NULL;

for(k=0;k<10;k++){

res[i][j][k]=NULL; }

}

}

printf("Enter the no of productions:");

scanf("%d",&npro);

printf("Enter the productions:");

for(i=0;i<npro;i++)

{

scanf("%s",pro[i]);

}

for(i=0;i<npro;i++)

{

flag=0;

for(j=0;j<nont;j++)

{

if(nt[j]==pro[i][0])

{

flag=1;

}

}

if(flag==0)

{

nt[nont]=pro[i][0];

nont++;

}

}

printf("\nEnter the first values:\n");

for(i=0;i<nont;i++)

{

printf("First value(%c):",nt[i]);

scanf("%s",first[i]);

}

printf("\nEnter the follow values:\n");

for(i=0;i<nont;i++)

{

printf("Follow value(%c):",nt[i]);

scanf("%s",follow[i]);

}

for(i=0;i<nont;i++)

{

flag=0;

for(j=0;j<strlen(first[i]);j++)

{

for(k=0;k<noter;k++)

{

if(ter[k]==first[i][j])

{

flag=1;

}

}

if(flag==0)

{

if(first[i][j]!='#')

{

ter[noter]=first[i][j];

noter++;

}

}

}

}

for(i=0;i<nont;i++)

{

flag=0;

for(j=0;j<strlen(follow[i]);j++)

{

for(k=0;k<noter;k++)

{

if(ter[k]==follow[i][j])

{

flag=1;

}

}

if(flag==0)

{

ter[noter]=follow[i][j];

noter++;

}

}

}

for(i=0;i<nont;i++)

{

for(j=0;j<strlen(first[i]);j++)

{

flag=0;

if(first[i][j]=='#')

{

col=i;

for(m=0;m<strlen(follow[col]);m++)

{

for(l=0;l<noter;l++)

{

if(ter[l]==follow[col][m])

{

row=l;

}

}

temp[0]=nt[col];

temp[1]='-' ;

temp[2]='>';

temp[3]='#';

temp[4]='\0';

printf("temp %s",temp);

strcpy(res[col][row],temp);

count[col][row]+=1;

for(k=0;k<10;k++){

temp[k]=NULL; }

}

}

else{

for(l=0;l<noter;l++)

{

if(ter[l]==first[i][j])

{

row=l;

}

}

for(k=0;k<npro;k++){

if(nt[i]==pro[k][0])

{

col=i;

if((pro[k][3]==first[i][j])&&(pro[k][0]==nt[col]))

{

strcpy(res[col][row],pro[k]);

count[col][row]+=1;

}

else

{

if((isupper(pro[k][3]))&&(pro[k][0]==nt[col]))

{

flag=0;

for(m=0;m<nont;m++)

{

if(nt[m]==pro[k][3]){index=m;flag=1;}

}

if(flag==1){

for(m=0;m<strlen(first[index]);m++)

{if(first[i][j]==first[index][m])

{strcpy(res[col][row],pro[k]);

count[col][row]+=1;}

}

}

}}}}}

}}

printf("\nParsing Table\n\n");

flag=0;

for(i=0;i<noter;i++)

{

printf("\t%c",ter[i]);

}

for(j=0;j<nont;j++)

{

printf("\n\n%c",nt[j]);

for(k=0;k<noter;k++)

{

printf("\t%s",res[j][k]);

if(count[j][k]>1){flag=1;}

}

}

if(flag==1){

printf("\nThe given grammar is not LL1");}

else{

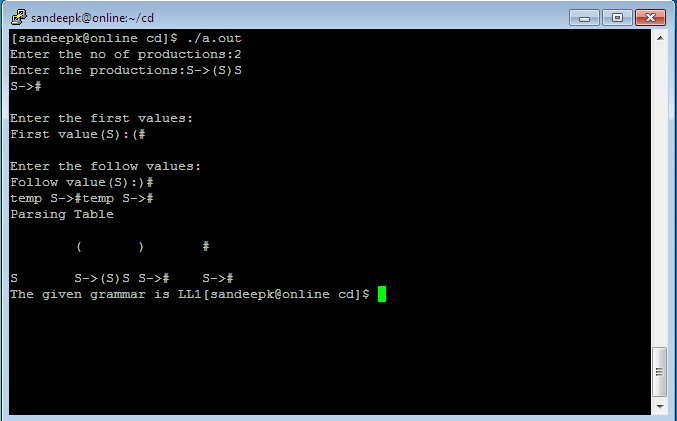
printf("\nThe given grammar is LL1");

}

return 0;

}

**Output:**

****

**VIVA QUESTIONS**

1. **What is a keyword?**
2. **What Is the difference between identifier and a variable?**
3. Explain about Syntax analysis?
4. **What is LEX Specification**

**5. What is String, sequence ,comment?**

**6. What is the use of symbol table in LEX?**

**Date:**

**7.** **Implementation of shift reduce parsing algorithm.**

**Aim:** Program to implement shift reduce parser

**Description:** A shift-reduce parser is a class of efficient, table-driven [bottom-up parsing](https://en.wikipedia.org/wiki/Bottom-up_parsing) methods for computer languages and other notations formally defined by a [grammar](https://en.wikipedia.org/wiki/Grammar). The parsing methods most commonly used today, [LR parsing](https://en.wikipedia.org/wiki/LR_Parser) and its variations, are shift-reduce methods. The [precedence parsers](https://en.wikipedia.org/wiki/Simple_precedence_parser) used before the invention of LR parsing are also shift-reduce methods. All shift-reduce parsers have similar outward effects, in the incremental order in which they build a parse tree or call specific output actions. The outward actions of an LR parser are best understood by ignoring the arcane mathematical details of how LR parser tables are generated, and instead looking at the parser as just some generic shift-reduce method.

**Algorithm:**

loop forever:

for top-of-stack symbol, s, and next input symbol, a

case action of T[s,a]

shift x: (x is a STATE number)

push a, then x on the top of the stack and

advance ip to point to the next input symbol.

reduce y: (y is a PRODUCTION number)

Assume that the production is of the form

A ==> beta

pop 2 \* |beta| symbols of the stack. At this point the top of the stack should be a state number, say s’.

push A, then goto of T[s’,A] (a state number) on the top of the stack. Output the production A ==> beta.

accept:

return --- a successful parse.

default:

error --- the input string is not in the language.

**Program:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

char ip\_sym[15],stack[15];

int ip\_ptr=0,st\_ptr=0,len,i;

char temp[2],temp2[2];

char act[15];

void check();

void main()

{

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

printf("\n GRAMMAR\n");

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

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

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

gets(ip\_sym);

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

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

printf("\n\_\_\_\_\_\_\t\t \_\_\_\_\_\_\_\_\_\_\_\_\t\t \_\_\_\_\_\_\n");

printf("\n $\t\t%s$\t\t\t--",ip\_sym);

strcpy(act,"shift ");

temp[0]=ip\_sym[ip\_ptr];

temp[1]='\0';

strcat(act,temp);

len=strlen(ip\_sym);

for(i=0;i<=len-1;i++)

{

stack[st\_ptr]=ip\_sym[ip\_ptr];

stack[st\_ptr+1]='\0';

ip\_sym[ip\_ptr]=' ';

ip\_ptr++;

printf("\n $%s\t\t%s$\t\t\t%s",stack,ip\_sym,act);

strcpy(act,"shift ");

temp[0]=ip\_sym[ip\_ptr];

temp[1]='\0';

strcat(act,temp);

check();

st\_ptr++;

}

st\_ptr++;

check();

}

void check()

{

int flag=0;

temp2[0]=stack[st\_ptr];

temp2[1]='\0';

if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))

{

stack[st\_ptr]='E';

if(!strcmpi(temp2,"a"))

printf("\n $%s\t\t%s$\t\t\tE->a",stack, ip\_sym);

else

printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip\_sym);

flag=1;

}

if((!strcmpi(temp2,"+"))||(!strcmpi(temp2,"\*"))||(!strcmpi(temp2,"/")))

{

flag=1;

}

if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E\E"))||(!strcmpi(stack,"E\*E")))

{

strcpy(stack,"E");

st\_ptr=0;

if(!strcmpi(stack,"E+E"))

printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip\_sym);

else

if(!strcmpi(stack,"E/E"))

printf("\n $%s\t\t %s$\t\t\tE->E/E",stack,ip\_sym);

else

printf("\n $%s\t\t%s$\t\t\tE->E\*E",stack,ip\_sym);

flag=1;

}

if(!strcmpi(stack,"E")&&ip\_ptr==len)

{

printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip\_sym);

exit(0);

}

if(flag==0)

{

printf("\n%s\t\t\t%s\t\t reject",stack,ip\_sym);

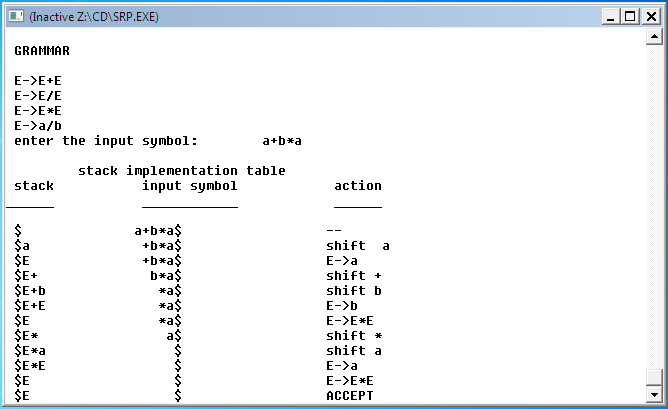
exit(0);

}

return;

}

**Output:**



**VIVA QUESTIONS**

1. **Define SRP**
2. **What is the use of accept?**
3. **What do you mean by shift?**
4. **What is reduce in SRP?**

**5. What is goto?**

**6. Define error?**

7. What is shift-shift confict?

8. When we can say reduce-reduce conflict?

9. What do you mean by shift-reduce conflict?

10. What type of parser SRP?

**Date:**

**8.** **Design a LALR bottom up parser for the given language**.

**Aim:** To Write a program to Design LALR Bottom up Parser.

**Description:**

The LALR parser is smaller than the CLR.The most

programming languages are expressed conveniently b LALR

grammar .The same is almost true for SLR grammars.For a comparison of parser size,the SLR and LALR tables for a grammar always have the same number of states.

**Algorithm:**

Input:An augmented grammar G.

Output:The LALR parsing table functions action and goto for G’.

Method:

1)Construct C={I0**,**I1,…………………………..,In}  **,**the collection of sets of

LR(1) items.

2)The set of LR(1) items,find all sets having that core ,and

replace these sets by their union.

3)Let C’={J0,J1,…………,Jm} be the resulting sets of LR(1) items .The parsing actions for state I are constructed from Ji in the same manner as CLR .If there is a parsing action conflict,the algorithm fails to produce a parser,and the grammar is said not to be LALR(1).

4)If J is the union of one or more sets of LR(1) items,that is,J=I1UI2U,……………….UIk, then the cores of goto(I1,X), goto(I2,X ),………………… goto(Ik,X) are the same, since I1,I2,……………….,Ik all have the same core let k be the union of all sets of items having the same core as goto(I1,X) then goto(J,X)=K.

**Program**

<parser.l>

%{

#include<stdio.h>

#include "y.tab.h"

%}

%%

[0-9]+ {yylval.dval=atof(yytext);

return DIGIT;

}

\n|. return yytext[0];

%%

<parser.y>

%{

/\*This YACC specification file generates the LALR parser for the program

considered in experiment 4.\*/

#include<stdio.h>

%}

%union

{

double dval;

}

%token <dval> DIGIT

%type <dval> expr

%type <dval> term

%type <dval> factor

%%

line: expr '\n' {

printf("%g\n",$1);

}

;

expr: expr '+' term {$$=$1 + $3 ;}

| term

;

term: term '\*' factor {$$=$1 \* $3 ;}

| factor

;

factor: '(' expr ')' {$$=$2 ;}

| DIGIT

;

%%

int main()

{

yyparse();

}

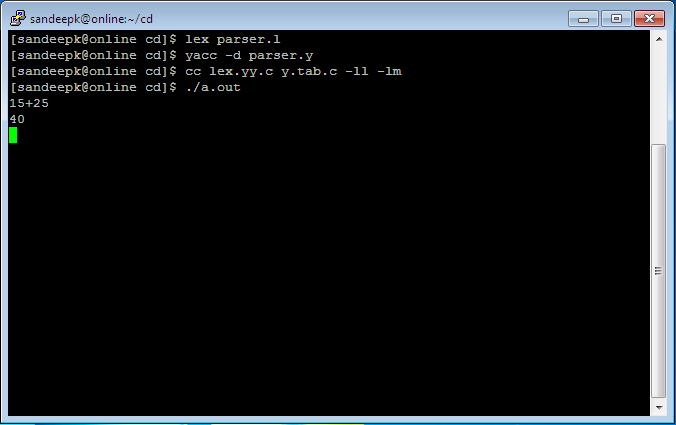
yyerror(char \*s)

{

printf("%s",s);

}

**Output:**

****

**VIVA QUESTIONS**

1. **What is Bottom up parser?**
2. **Why LR parser?**
3. **Why LL parser**
4. **What is the model of LR parser ?**
5. **What are the more powerful LR parser?**
6. Differentiate between LL and LR?

7. Error recovery in LR parser?

8. What is Shift reduce parser?

9. What is LALR?

**Date:**

**9. Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer**

**generating tools**

**Aim:** To Write a program Implement the Lexical Analyzer Using LEX Tool.

**Description:**

Lex program for the tokens translated below.

|  |  |  |
| --- | --- | --- |
| Regular Expression | Token | Attribute |
| Ws | - | - |
| If | If | - |
| Then | Then | - |
| Else | Else | - |
| Id | Id | Ptr to table entry |
| Num | Num | Ptr to table entry |
| < | Relop | LT |
| <= | Relop | LE |
| = | Relop | EQ |
| <> | Relop | NE |
| > | Relop | GT |
| >= | Relop | GE |

**Lex Specification** :

A lex program consists of three parts.

Declaration section

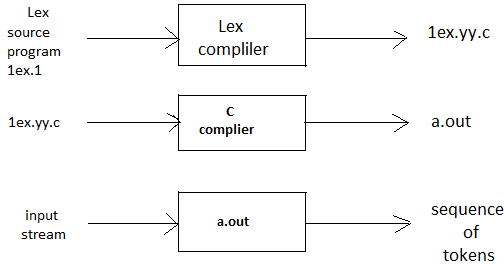
% %

Translation rules

% %

Auxulliary procedures

**Creating a lexical analyzer with lex:**

****

**Algorithm:**

**%{**

**/\* de**finitions of manifest constants

LT,LE,EQ,NE,GT,GE,IF,THEN,ELSE,IDNUMBER,RELOP\*/

%}

/\*Regular definitions\*/

delim [\t\n]

**ws** {delim}

letter [A-Z,a-z]

digit [0-9]

id {letter}({letter}|{digit})\*

number {digit}+(\.digit}+)?(E[+\-]?{digit}+)?

%%

{WS} {{/\*no action no return\*/}}

If {return(IF);}

then {return(THEN);}

else {return(ELSE);}

{id} {yylval=install\_id();return(ID);}

{number} {yyval=install\_num();return(NUMBER);}

“<” {yyval=LT;return(RELOP);}

“<=” {yyval=LE;return(RELOP);}

“=” {yyval=EQ;return(RELOP);}

“<>” {yyval=NE;return(RELOP);}

“>” {yyval=GT;return(RELOP);}

“>=” {yyval=GE;return(RELOP);}

%%

install\_ id()

{

/\*procedure to install the lexeme,whose first character is pointed to by yytext and whose length is yyleng,into the symbol table and return a pointer there to\*/.

}

install\_num()

{

/\*similar procedures to install a lexeme that is a number\*/

}

**Program**

%{

/\* program to recognize a c program \*/

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 |

do |

if |

break |

continue |

void |

switch |

case |

long |

struct |

const |

typedef |

return |

else |

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

"/\*" {COMMENT = 1;}

"\*/" {COMMENT = 0;}

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

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

\} {if(!COMMENT) printf("\n 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\t%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)

{

if (argc > 1)

{

FILE \*file;

file = fopen(argv[1],"r");

if(!file)

{

printf("could not open %s \n",argv[1]);

exit(0);

}

yyin = file;

}

yylex();

printf("\n\n");

return 0;

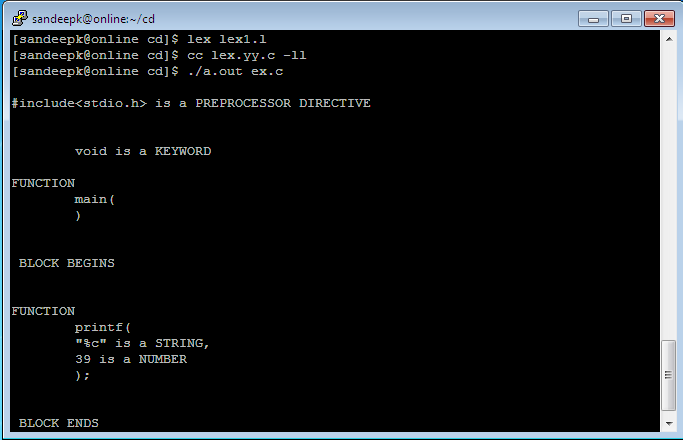
} int yywrap()

{

return 0;

}

**Output:**

****

**VIVA QUESTIONS**

1. What is lexical analysis?
2. Describe the Semantic analysis?.
3. Difference between lexical analysis and parsing?
4. What is Dangling Else ambiguity?
5. Define SDT?

6. What is purpose S-Attribute?

7. What is L-Attribute?

8. What are storage organizations?

**Date:**

**10. Write a program to perform loop unrolling.**

**Aim:** Program to perform loop unrolling.

**Description:** Loop overhead can be reduced by reducing the number of iterations and replicating the body of the loop.

**Algorithm:**

send(to, from, count)

register short \*to, \*from;

register count;

{

register n =(count + 7) /8;

switch (count %8) {

case 0: do { \*to = \*from++;

case 7: \*to = \*from++;

case 6: \*to = \*from++;

case 5: \*to = \*from++;

case 4: \*to = \*from++;

case 3: \*to = \*from++;

case 2: \*to = \*from++;

case 1: \*to = \*from++;

} while (--n >0);

}

}

**Program:**

#include<stdio.h>

int count=0;

void main()

{

int i;

for(i=1;i<=100;i++)

{

printf("%d\t",i);

i++;

printf("%d\t",i);

i++;

printf("%d\t",i);

i++;

printf("%d\t",i);

i++;

printf("%d\t",i);

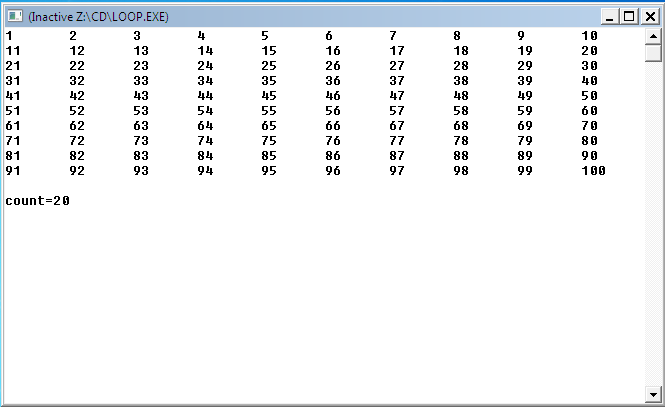
count++;

}

printf("\ncount=%d\n",count);

}

**Output:**



**VIVA QUESTIONS**

1. Define loop unrolling?
2. What is the use of loop unrolling in code optimization?
3. What are the techniques of machine dependent code optimization?
4. Define flow of control optimization?

5. What is the use of strength reduction?

6. Define Instruction scheduling?

7. Define IPO?

8. What is the importance of garbage collection via reference counting?

9. What is quadruples?

**Date:**

**11. Convert the BNF rules into YACC form and write code to generate abstract syntax**

**tree**.

**Aim:** To Convert The BNF rules into Yacc form and write code to generate abstract syntax tree.

**Description:**

Abstract syntax tree is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the abstract syntax tree and inherited attributes of that grammar symbol.

**Algorithm:**

procedure R;

begin

if lookahead = addop then begin

match(addop); T;R

end

else begin

end

end;

**Program:**

**<first.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];

%%

<**second.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,$$);}

| 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);

}

Input:

$vi example.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

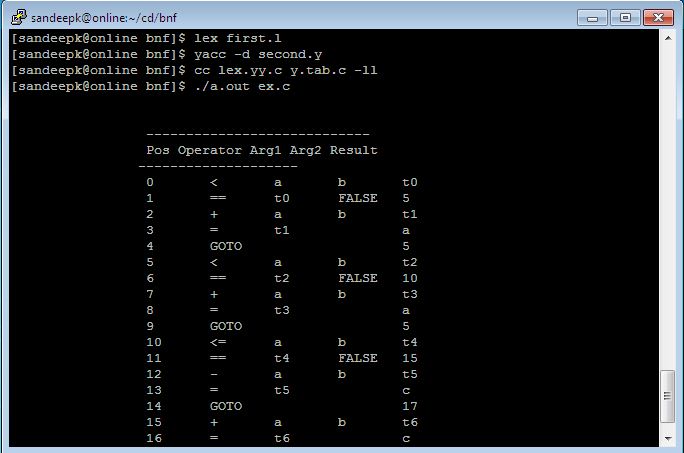
{

c=a+b;

}

}

**Output:**

****

**VIVA QUESTIONS**

1. What is Syntax analysis?

2. What is YACC?

3. What is Automatic parser generator?

4. Define Heap management?

5. What is Explicit allocation and Implicit allocation?

Date:

**12. Write a program for constant propagation.**

**Aim:** Program for constant propagation

**Description:**

Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

**Algorithm:**

1. Assign the mapping [*x ->*0, *y ->*0, *z->*0] to statement 1, look at statement 2.

2. Assign [*x ->*0, *y ->*0, *z->*3] to statement 2, move to statement 3.

3. Assign [*x ->*1, *y ->*0, *z ->*3] to statement 3, move to statement 4.

4. Assign [*x ->*1, *y ->*0, *z ->*3] to statement 4, move to statement 5.

5. Assign [*x ->*1, *y ->*0, *z ->*3] to statement 5, move to statement 8.

6. Assign [*x ->*1, *y ->*7, *z ->*3] to statement 8, move to statement 9.

7. Assign [*x ->*3, *y->*7, *z->*3] to statement 9, move to statement 4.

8. Change the mapping of statement 4 to be [*x ->T*, *y ->T*, *z ->*3], move to 5 (and 7).

9. Change 5 to be [*x ->*1, *y ->T*, *z ->*3] , move to 7 (and 8).

10. Assign [*x ->T*, *y ->T*, *z ->*3] to statement 7, move to 8.

11. Change 8 to be [*x ->T*, *y ->*7, *z ->*3], move to 9.

12. The mapping of statement 9 stays [*x ->*3, *y ->*7, *z ->*3] - the algorithm terminates since

no further changes are possible.

**Program:**

#include<stdio.h>

void main()

{

int a,b,y=4;

a=3;

b=5; //b=a+2;

printf("a=%d,b=%d\n",a,b);

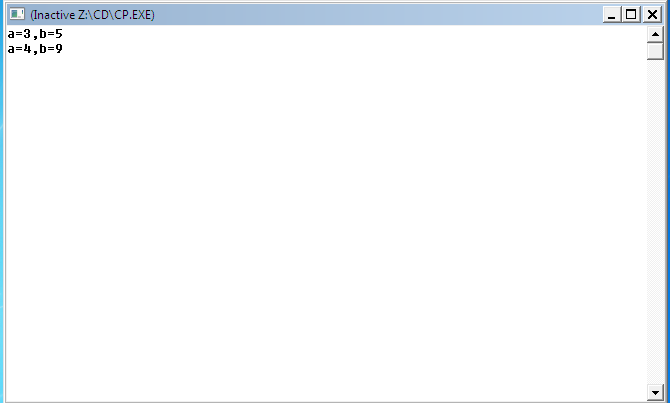
a=y;

b=a+5; //b=a+b;

printf("a=%d,b=%d\n",a,b);

}

**Output:**



**VIVA QUESTIONS**

1. What is constant propogation ?

2. What is constant folding?

3. What is loop optimization ?

4. What is abstract syntax tree ?

5. What is dead code elimination?

6. Define procedure in lining?

7. Define basic blocks?

**ADDITIONAL EXPERIMENT**

**Date:**

**13. Write a program to implement SLR parser.**

**Aim:** Program for SLR(1) parser

**Description:**

A rule of the form *B → y •* within a state of a SLR(1) automaton is said to be irreducible or in a reduced state because it has been completely expanded and is incapable of undergoing any shift transition. Rules in this state will have a dot ( • , the current look-ahead position) located at the rightmost end of its RHS (Right Hand Side).

**Algorithm:**

1. If state *s* contains any item of the form *A → a • Xb*, where *X* is a terminal, and *X* is the next token in the input string, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item *A → aX • b*.
2. If state *s* contains the complete item *A → y •* , and the next token in the input string is in *Follow(A)*, then the action is to reduce by the rule *A → y*. A reduction by the rule *S' → S*, where *S* is the start state, is equivalent to acceptance; this will happen only if the next input token is *$*. In all other cases, the new state in computed as follows. Remove the string *y* and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of *y* began. By construction, this state must contain an item of the form *B → a • Ab*. Push *A* on to the stack, and push the state containing the item *B → aA • b*.
3. If the next input token is such that neither of the above two cases applies, an error is declared.

**Program:**

#include<stdio.h>

#include<ctype.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

#include<iostream.h>

#define epsilon '^'

char prod[20][20],T[20],NT[20],c[10][10],foll[10][10],fir[10][10];

int tt,tnt,tp,a;

int follow[20][20],first[20][20];

void first\_of(char);

int count(int j);

void rhs(int j);

void read\_tnt();

int rhs(int j);

void read\_tnt()

{ cout<<"For SLR parser: ";

cout<<"\nEnter number of terminals: ";

cin>>tt;

cout<<"\nEnter terminals: ";

for(int i=0;i<tt;i++)

T[i]=getche();

getch();

cout<<"\nEnter number of Non-terminals: ";

cin>>tnt;

cout<<"\nEnter Non-terminals: ";

for(i=0;i<tnt;i++)

NT[i]=getche();

getch(); }

void read\_prod()

{ int j;

char x=0;

cout<<"\n\nEnter number of productions: ";

cin>>tp;

cout<<"\n Enter productions: ";

for(int i=0;i<tp;i++)

{ j=x=0;

while(x!='\r')

{ prod[i][j]=x=getche();

j++; }

cout<<"\n"; }

getch(); }

return(i);

if(t=='$')

return(tt);

return(-1); }

int terminal(char x)

{ for(int i=0;i<tt;i++)

if(T[i]==x)

return(1);

return(0); }

int nonterminal(char x)

{ for(int i=0;i<tnt;i++)

if(NT[i]==x)

return(1);

return(0); }

int in\_rhs(char \*s,char x)

{ for(int i=0;i<=strlen(s);i++)

if(\*(s+i)==x)

return(i);

return(-1); }

void find\_first()

{ for(int i=0;i<tnt;i++)

first\_of(NT[i]); }

void first\_of(char n)

{ int t1,t2,p1,cnt=0,i,j;

char x;

static int over[20];

p1=t\_no(epsilon);

if(terminal(n))

return;

t1=nt\_no(n);

if(over[t1])

return;

over[t1]=1;

for(i=0;i<tp;i++)

{ t1=nt\_no(prod[i][0]);

if(prod[i][0]==n)

{ int k=0;

cnt=count(1);

rhs(i);

while(k<cnt)

{ x=c[i][k];

if(terminal(x))

{ t2=t\_no(x);

first[t1][t2]=1;

break; }

else

{ t2=nt\_no(x);

first\_of(x);

for(int j=0;j<tt;j++)

first[t1][p1]=1; } } }

void follow\_of(char n)

{ int f,t1,t2,p1,t,cnt=0;

char x,beta;

static int over[20];

p1=t\_no(epsilon);

t1=nt\_no(n);

if(over[t1])

return;

over[t1]=1;

if(NT[0]==n)

follow[nt\_no(NT[0])][tt]=1;

for(int i=0;i<tp;i++)

{ rhs(i);

cnt=count(i);

t=in\_rhs(c[i],n);

int bno;

for(int j=0;j<tt;j++)

{

bno=nt\_no(beta);

if((first[bno][j]) && (j!=p1))

follow[t1][j]=1; }

if((p1!=-1) && (first[bno][p1]==1))

continue;

else if((t==(cnt-1)||(k>=cnt)))

{ follow\_of(prod[i][0]);

t1=nt\_no(prod[i][0]);

for(int l=0;l<=tt+1;l++)

if(follow[t][l])

follow[t1][l]=1; } } } }

int count(int j)

{ int c1=0;

for(int q=3;prod[j][q]!='\r';q++)

c1++;

return(c1); }

void show\_follow()

{ int b=0;

a=0;

cout<<"\n\n Follow Table For Grammar: \n";

for(int i=0;i<tnt;i++)

{

b=0;

cout<<"\n FOLLOW ("<<NT[i]<<" )= { ";

for(int j=0;j<tt+1;j++)

if(follow[i][j] && j!=tt)

{ foll[a][b]=T[j];

b++;

cout<<T[j]<<" "; }

else

if(j==tt)

{ foll[a][b]='$';

b++;

cout<<'$'; }

a++;

cout<<" } "; }

getch(); }

void show\_first()

{ int b=0;

a=0;

cout<<"\n\n First Table For Grammar: \n";

for(int i=0;i<tnt;i++)

{ b=0;

cout<<"\n FIRST ("<<NT[i]<<" )= { ";

for(int j=0;j<tt+1;j++)

if(first[i][j] && j!=tt)

{ fir[a][b]=T[j];

b++;

cout<<T[j]<<" "; }

a++;

cout<<" } "; }

getch()}}}}

To construct parse table:

#include<stdio.h>

#include<conio.h>

#include<string.h>

#include<ctype.h>

#include<stdlib.h>

#include<iostream.h>

#include"c:\tc\bin\SLR.h"

int S=0,i=0,j=0,state[20];

char TNT[15];

struct node

{ int pno,dpos; };

struct t

{ char s;

int n; };

struct t1

{ struct t lr[10];

int gr[5]; };

struct t1 action[15];

struct node closure[10][10];

int g[15][10];

int l;

void sclosure(int,int);

int added(int);

int t\_into(char);

void print\_table(int);

void parser(void);

void find\_closure(int,int);

void SLR(void);

void main()

{ clrscr();

mainf();

getch();

for(int i=0;i<tnt;i++)

TNT[i]=NT[i];

for(int j=0;j<tt;j++)

{ TNT[i]=T[j];

i++; }

strcat(T,"$");

i=j=0;

SLR();

print\_table(S);

getch(); }

void SLR()

{ int clno,no=0,x,y,z,len,cnt=-1,d=0;

closure[i][j].pno=0;

closure[i][j++].dpos=3;

find\_closure(no,3);

sclosure(i,j);

state[i]=j;

S=0;

do

{ cnt++;

z=state[cnt];

for(int k=0;k<tnt+tt;k++)

{ i++;

j=0;d=0;

for(int l=0;l<z;l++)

{ x=closure[cnt][1].pno;

y=closure[cnt][1].dpos;

if(prod[x][y]==TNT[k])

{ d=1;

closure[i][j].pno=x;

closure[i][j++].dpos=++y;

if((y<strlen(prod[x])) && (isupper(prod[x][y])))

find\_closure(x,y); } }

if(d==0)

{ i--;

continue; }

sclosure(i,j);

else

{ action[cnt].lr[k-tnt].s='S';

action[cnt].lr[k-tnt].n=clno;

}

if(added(i-1)!=-1)

i--;

else

{ S++;

for(l=0;l<state[i];l++)

{ if(closure[i][1].pno==0)

{ action[i].lr[tt].s='A';

continue; }

len=(strlen(prod[closure[i][l].pno])-1);

if(len==closure[i][l].dpos)

{ char v=prod[closure[i][l].pno][0];

int u=nt\_no(v);

for(x=0;x<strlen(foll[u]);x++)

{ int w=t\_ino(foll[u][x]);

action[i].lr[w].s='R';

action[i].lr[w].n=closure[i][l].pno;}}}}}}

while(cnt!=S); }

void print\_table(int states)

{ int lin=5;

cout<<"\n\n Parser Table: \n";

for(int i=0;i<tt;i++)

cout<<"\t"<<T[i];

cout<<"\t$";

for(i=0;i<tnt;i++)

cout<<"\t"<<NT[i];

cout<<"\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

for(i=0;i<=states;i++)

{ gotoxy(l,lin);

cout<<"I"<<i<<"\t";

for(int j=0;j<=tt;j++)

{ if(action[i].lr[j].s!='\x0')

{ if(action[i].lr[j].s=='A')

{ cout<<"Acc";

continue; }

else

cout<<"\t"; }

for(j=0;j<tnt;j++)

if(action[i].gr[j])

{ cout<<action[i].gr[j];

cout<<"\t"; }

else

cout<<"\t";

lin++;

cout<<"\n"; }

cout<<"\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"; }

void sclosure(int clno,int prodno)

{ struct node temp;

for(int i=0;i<prodno-1;i++)

{ for(int j=i+1;j<prodno;j++)

{ if(closure[clno][i].pno>closure[clno][j].pno)

{ temp=closure[clno][i];

closure[clno][i]=closure[clno][j];

closure[clno][j]=temp; }}}

for(i=0;i<prodno-1;i++)

{for(j=i+1;j<prodno;j++)

{if((closure[clno][i].dpos>closure[clno][j].dpos) &&

(closure[clno][i].pno==closure[clno][j].pno))

{ temp=closure[clno][i];

closure[clno][i]=closure[clno][j];

closure[clno][j]=temp;}}}}

int added(int n)

{ int d=1;

for(int k=0;k<=n;k++)

{if(state[k]==state[n+1])

{ d=0;

return(k); } }

return(-1); }

void find\_closure(int no,int dp)

{ int k;

char temp[5];

if(isupper(prod[no][dp]))

{for(k=0;k<tp;k++)

{if(prod[k][0]==prod[no][dp])

{ int t\_ino(char t)

{ for(int i=0;i<=tt;i++)

if(T[i]==t)

return(i);

return(-1); }

char pops2;

struct node1

{ char s2;int s1; };

struct node1 stack[10];

int pops1,top=0;

void parser(void)

{ int r,c;

struct t lr[10];

char t,acc='f',str[10];

cout<<"Enter I/p String To Parse: ";

cin>>str;

strcat(str,"$");

stack[0].s1=0;

stack[0].s2='\n';

cout<<"\n\n STACK";

cout<<"\t\t INPUT";

cout<<"\t\t ACTION";

for(int j=0;j<strlen(str);j++)

cout<<str[j];

do

{r=stack[top].s1;

c=find\_index(str[i]);

if(c==-1)

cout<<"\n Error! Invalid String!";

return; }

while(top!=0);

switch(action[r],lr[c].s)

{case 'S': { push(str[i],action[r].lr[c].n);

i++;

cout<<"\t\t\t Shift";

break; }

case 'R': { t=prod[action[r].lr[c].n][3];

do { pop(); }

while(pops2!=t);

t=prod[action[r].lr[c].n][0];

r=stack[top].s1;

c=find\_index(t);

push(t,action[r].gr[c-tt-1]);

cout<<"\t\t\t Reduce";

break;}

case 'A':{ cout<<"\t\t\t Accept";

cout<<"\n\n\n String accepted";

acc='t';

getch();

return; }

default: { cout<<"\n\n\n Error! String not accepted!";

getch();

exit(0);}}

for(j=0;j<=top;j++)

cout<<stack[j].s2<<stack[j].s1;

if(top<4)

cout<<"\t\t\t";

else

cout<<"\t\t";

for(j=i;j<strlen(str);j++)

cout<<str[j];

if(acc=='t')

return; }

int find\_index(char temp)

{for(int i=0;i<=tt+tnt;i++)

{if(i<=tt)

{ if(T[i]==temp)

return(i);}

else

if(NT[i-tt-1]==temp)

return(i); }

return(-1); }

void push(char t2,int t1)

{++top;

stack[top].s1=t1;

stack[top].s2=t2;

return; }

void pop(void)

{pops1=stack[top].s1;

pops2=stack[top].s2;

--top; getch(); }

**Output:**

Enter number of terminals: 5

Enter terminals:+\*()i

Enter number of non-terminals:3

Enter non-terminals:ETF

Enter number of productions:6

Enter productions:

E->E+T

E->T

T->T\*F

T->F

F->(E)

F->i

Follow table:

FOLLOW(E)={+ ) $}

FOLLOW(F)={+ \* ) $}

FOLLOW(T)={ + \* ) $}

First Table :

FIRST(E)={ ( i }

FIRST(E)={ ( i }

FIRST(E)={ ( i }

Expected parse table:

+ \* ( ) i $ E T F

I0 S4 S5 1 2 3

I1 S6 ACC

I2 R1 S7 R1 R1

I3 R3 R3 R3 R3

I4 S4 S5 ACC 8 2 3

I5 R5 R5 R5 R5

I6 ACC

I7 S4 S5 9

I8 S10 S11 ACC

I9 R2 R2 R2 R2

I10 ACC

I11 R4 R4 R4 R4

Enter i/p string: i+i\*i

STACK INPUT ACTION

0 i+i\*i$ Shift

0i5 +i\*i$ Reduce

0F3 +i\*i$ Reduce

0T2 +i\*i$ Reduce

0E1 +i\*i$ Shift

0E1+6 i\*i$

ERROR! STRING NOT ACCEPTED!

**VIVA QUESTIONS**

1. Define SLR ?

2. Define CLR?

3. Define LALR ?

4. What is the difference between SLR ,CLR ,LALR ?

5. What is the difference between LR(0) and SLR(1)?