## LAB OUTCOMES

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

2. Implement the lexical analyzer using JLex, flex or lex tools.

3. Implement the syntax analyzer using yacc tool.

4. Construct a recursive descent parser for the given an expression.

5. Develop an operator precedence parser for a given language.

# INTRODUCTION ABOUT LAB

There are 60 systems installed in this Lab. Their configurations are as follows:

| | |
|---|---|
| Processor | : Intel third generation |
| RAM | : 2GB |
| Hard Disk | : 500 GB |
| Network | : LAN |

- All systems are configured in DUAL BOOT mode i.e., Students can boot from Windows 8 or Linux as per their lab requirement. This is very useful for students because they are familiar with different Operating Systems so that they can execute their programs in different programming environments.
- In all the systems TurboC3 and Putty are installed.
- Systems are provided for students in the ratio of 1:1.
- Systems are assigned numbers and same system is allotted for students when they do the lab.

## GUIDELINES TO STUDENTS

- Students are required to carry their observation / record with completed exercises while entering the lab.
- Students are supposed to occupy the machines allotted to them and are not supposed to talk or make noise in the lab. The allocation is put up on the lab notice board.
- Lab can be used in free time / lunch hours by the students who need to use the systems should take prior permission from the lab in-charge.
- Lab records need to be submitted on or before date of submission.
- Students are not supposed to use pen drives.

**Writing of the experiment in the Observation Book:**

The students will write the experiment in the Record as per the following format:

a) Name of the experiment/Aim.

b) Summary of commands required for this experiment.

c) Source Program.

d) Results.

e) Viva-Voce Questions and Answers.

f) Errors observed (if any) during compilation/execution.

g) Signature of the Faculty.

## LIST OF LAB EXERCISES

| Sl. No | Name of the Program |
|---|---|
| 1 | Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines |
| 2 | Simulate First and Follow of a Grammar. |
| 3 | Develop an operator precedence parser for a given language. |
| 4 | Construct a recursive descent parser for an expression. |
| 5 | Construct a LL(1) parser for an expression |
| 6 | Design predictive parser for the given language |
| 7 | Implementation of shift reduce parsing algorithm. |
| 8 | Design a LALR bottom up parser for the given language. |
| 9 | Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools |
| 10 | Write a program to perform loop unrolling. |
| 11 | Convert the BNF rules into YACC form and write code to generate abstract syntax tree. |
| 12 | Write a program for constant propagation. |

## SOLUTIONS FOR PROGRAMS

### 1) Lexical Analyzer implimentation by using C program

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

void main()
{
FILE *f1;
char c,str[10];
int lineno=1,num=0,i=0;
clrscr();
printf("\nEnter the c program\n");
f1=fopen("input.txt","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input.txt","r");
while((c=getc(f1))!=EOF)  // TO READ THE GIVEN FILE
{
        if(isdigit(c)) // TO  RECOGNIZE NUMBERS
        {
           num=c-48;
           c=getc(f1);
           while(isdigit(c))
           {
                   num=num*10+(c-48);
                   c=getc(f1);
           }
           printf("%d is a number \n",num);
           ungetc(c,f1);
        }
        else if(isalpha(c))  // TO  RECOGNIZE KEYWORDS AND IDENTIFIERS
        {
           str[i++]=c;
           c=getc(f1);
           while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
           {
                   str[i++]=c;
                   c=getc(f1);
           }
           str[i++]='\0';
           if(strcmp("for",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) // TYPE 32 KEYWORDS
           printf("%s is a keyword\n",str);
```

```
            else
            printf("%s is a identifier\n",str);
            ungetc(c,f1);
            i=0;
        }
        else if(c==' '||c=='\t')   // TO IGNORE THE SPACE
            printf("\n");
        else if(c=='\n')   // TO COUNT LINE NUMBER
            lineno++;
        else      // TO FIND SPECIAL SYMBOL
            printf("%c is a special symbol\n",c);

}
printf("Total no. of lines are: %d\n",lineno);
fclose(f1);
getch();
}
```

OUTPUT

```
Enter the c program
int main()
{
int a=10,20;
charch;
float f;
}^Z

The numbers in the program are: 10  20
The keywords and identifiersare:
int is a keyword
main is an identifier
int is a keyword
a is an identifier
char is a keyword
ch is an identifier
float is a keyword
f is an identifier
Special characters are (  )  {  =  ,  ;  ;  ;  }
Total no. of lines are:5
```

## 2) Simulate First and Follow of a Grammar.

### a) FIRST

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void result(char[],char);
intnop;
char prod[10][10];
void main()
{
int i;
char choice;
char c;
char res1[20];
clrscr();
printf("How many number of productions ? :");
scanf(" %d",&nop);
printf("enter the production string like E=E+T\n");
for(i=0;i<nop;i++)
   {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",prod[i]);
   }
do
   {
        printf("\n Find the FIRST of  :");
        scanf(" %c",&c);
        memset(res1,'0',sizeof(res));
        FIRST(res1,c);
        printf("\n FIRST(%c)= { ",c);
        for(i=0;res1[i]!='\0';i++)
        printf(" %c ",res1[i]);
        printf("}\n");
        printf("press 'y' to continue : ");
        scanf(" %c",&choice);
   }
while(choice=='y'||choice =='Y');
}

void FIRST(char res[],char c)
{
inti,j,k;
char subres[5];
int eps;
subres[0]='\0';
res[0]='\0';
memset(res,'0',sizeof(res));
memset(subres,'0',sizeof(res));
if(!(isupper(c)))
   {
```

```
        result(res,c);
        return ;
    }
for(i=0;i<nop;i++)
    {
        if(prod[i][0]==c)
        {
        if(prod[i][2]=='$')
        result(res,'$');
        else
            {
                j=2;
                while(prod[i][j]!='\0')
                {
                eps=0;
                FIRST(subres,prod[i][j]);
                for(k=0;subres[k]!='\0';k++)
                        result(res,subres[k]);
                for(k=0;subres[k]!='\0';k++)
                if(subres[k]=='$')
                    {
                        eps=1;
                        break;
                    }
                if(!eps)
                break;
                j++;
                }
            }
    }
}
return ;
}
void result(char res[],char val)
{
int k;
for(k=0 ;res[k]!='\0';k++)
        if(res[k]==val)
        return;
res[k]=val;
res[k+1]='\0';
}
```

OUTPUT

```
How many number of productions ?:8
enter the production string like E=E+T
Enter productions Number 1 : E=TX
Enter productions Number 2 : X=+TX
Enter productions Number 3 : X=$
Enter productions Number 4 : T=FY
```

```
Enter productions Number 5 : Y=*FY
Enter productions Number 6 : Y=$
Enter productions Number 7 : F=(E)
Enter productions Number 8 : F=i

 Find the FIRST of  :X

 FIRST(X)= {  +  $ }
press 'y' to continue : Y

 Find the FIRST of  :F

FIRST(F)= {  (  i }
press 'y' to continue : Y

 Find the FIRST of  :Y

 FIRST(Y)= {  *  $ }
press 'y' to continue : Y

 Find the FIRST of  :E

FIRST(E)= {  (  i }
press 'y' to continue : Y

 Find the FIRST of  :T

FIRST(T)= {  (  i }
press 'y' to continue : N
```

b) **FOLLOW**

```
#include<stdio.h>
#include<string.h>
intnop,m=0,p,i=0,j=0;
char prod[10][10],res[10];
void FOLLOW(char c);
void first(char c);
void result(char);

void main()
{
int i;
int choice;
charc,ch;
printf("Enter the no.of productions: ");
scanf("%d", &nop);
printf("enter the production string like E=E+T\n");
for(i=0;i<nop;i++)
   {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",prod[i]);
   }
do
 {
  m=0;
memset(res,'0',sizeof(res));
printf("Find FOLLOW of -->");
scanf(" %c",&c);
  FOLLOW(c);
printf("FOLLOW(%c) = { ",c);
for(i=0;i<m;i++)
printf("%c ",res[i]);
printf(" }\n");
printf("Do you want to continue(Press 1 to continue....)?");
scanf("%d%c",&choice,&ch);
 }
while(choice==1);
}

void FOLLOW(char c)
{
if(prod[0][0]==c)
        result('$');
for(i=0;i<nop;i++)
 {
for(j=2;j<strlen(prod[i]);j++)
  {
if(prod[i][j]==c)
   {
```

```
if(prod[i][j+1]!='\0')
        first(prod[i][j+1]);
if(prod[i][j+1]=='\0'&&c!=prod[i][0])
        FOLLOW(prod[i][0]);
   }
  }
 }
}

void first(char c)
{
int k;
                if(!(isupper(c)))
                result(c);
                for(k=0;k<nop;k++)
                 {
                        if(prod[k][0]==c)
                         {
                        if(prod[k][2]=='$')
                                FOLLOW(prod[i][0]);
                        else if(islower(prod[k][2]))
                                result(prod[k][2]);
                        else
                                first(prod[k][2]);
                         }
                 }
}

void  result(char c)
{
int i;
for( i=0;i<=m;i++)
        if(res[i]==c)
        return;
res[m++]=c;
}
```

OUTPUT

```
Enter the no.of productions: 8
enter the production string like E=E+T
Enter productions Number 1 : E=TX
Enter productions Number 2 : X=+TX
Enter productions Number 3 : X=$
Enter productions Number 4 : T=FY
Enter productions Number 5 : Y=*FY
Enter productions Number 6 : Y=$
Enter productions Number 7 : F=(E)
Enter productions Number 8 : F=i
Find FOLLOW of -->X
FOLLOW(X) = { $ )  }
```

Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->E
FOLLOW(E) = {$ )  }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->Y
FOLLOW(Y) = { + $ )  }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->T
FOLLOW(T) = { +$ )  }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->F
FOLLOW(F) = { * + $ )  }
Do you want to continue(Press 1 to continue....)?2

**3) Develop an operator precedence parser for a given language.**

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

void main()
{
        char stack[20],ip[20],opt[10][10][1],ter[10];
        inti,j,k,n,top=0,row,col;
        clrscr();
        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:");
        scanf("%d",&n);
        printf("\nEnter the terminals:");
        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("\nOPERATOR PRECEDENCE TABLE:\n");
        for(i=0;i<n;i++)
        {
                printf("\t%c",ter[i]);
        }
        printf("\n_____");
        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("\n\nEnter the input string(append with $):");
        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])
                        row=k;
                        if(ip[i]==ter[k])
                        col=k;
                }
                if((stack[top]=='$')&&(ip[i]=='$'))
                {
                        printf("String is ACCEPTED");
                        break;
                }
                else if((opt[row][col][0]=='<') ||(opt[row][col][0]=='='))
                {
                        stack[++top]=opt[row][col][0];
                        stack[++top]=ip[i];
                        ip[i]=' ';
                        printf("Shift %c",ip[i]);
                        i++;
                }
                else
                {
                        if(opt[row][col][0]=='>')
                        {
                                while(stack[top]!='<')
                                {
                                        --top;
                                }
                                top=top-1;
                                printf("Reduce");
                        }
                        else
                        {
                                printf("\nString is not accepted");
                                break;
                        }
                }
                printf("\n");
                printf("%s\t\t\t%s\t\t\t",stack,ip);
        }
getch();
}
```

**OUTPUT**

```
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(append with $):i+i*i$

STACK              INPUT STRING              ACTION

$                    i+i*i$                  Shift
$<i                   +i*i$                  Reduce
$<i                   +i*i$                  Shift
$<+                    i*i$                  Shift
$<+<i                   *i$                  Reduce
$<+<i                   *i$                  Shift
$<+<*                    i$                  Shift
$<+<*<i                   $                  Reduce
$<+<*<i                   $                  Reduce
$<+<*<i                   $                  Reduce
$<+<*<i                   $                  String is ACCEPTED
```

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

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

char input[10];
int i=0,error=0;
void E();
void T();
voidEprime();
voidTprime();
void F();

void main()
{
        clrscr();
        printf("Enter an arithmetic expression   :\n");
        gets(input);
        E();
        if(strlen(input)==i&&error==0)
                printf("\nAccepted..!!!");
        else
                printf("\nRejected..!!!");
        getch();
}
void E()
{
T();
Eprime();
}
voidEprime()
{
if(input[i]=='+')
    {
i++;
T();
Eprime();
    }
}
void T()
{
F();
Tprime();
}
voidTprime()
{
if(input[i]=='*')
    {
                i++;
```

```
                F();
                Tprime();
    }
}
void F()
{
        if(input[i]=='(')
        {
                i++;
                E();
                if(input[i]==')')
                i++;
        }
        else if(isalpha(input[i]))
        {
                i++;
                while(isalnum(input[i])||input[i]=='_')
                i++;
        }
        else
                error=1;
}
```

OUTPUT

1)

```
Enter an arithmetic expression   :
sum+month*interest

Accepted..!!!
```

2)

```
Enter an arithmetic expression   :
sum+avg*+interest

Rejected..!!!
```

## 5) Construct a LL(1) parser for an expression

```
#include<stdio.h>
#include<string.h>
char str[25],st[25],*temp,v,ch,ch1;
char t[5][6][10]={"$","$","TX","TX","$","$",
                  "+TX","$","$","$","e","e",
                  "$","$","FY","FY","$","$",
                  "e","*FY","$","$","e","e",
                  "$","$","i","(E)","$","$"};
int i,k,n,top=-1,r,c,m,flag=0;
void push(char t)
{
        top++;
        st[top]=t;
}
char pop()
{
        ch1=st[top];
        top--;
        return ch1;
}
main()
{
        printf("enter the string:\n");
        scanf("%s",str);
        n=strlen(str);
        str[n++]='$';
        i=0;
        push('$');
        push('E');
        printf("stack\tinput\toperation\n");
        while(i<n)
        {
                for(k=0;k<=top;k++)
                printf("%c",st[k]);
                printf("\t");
                for(k=i;k<n;k++)
                printf("%c",str[k]);
                printf("\t");
                if(flag==1)
                printf("pop");
                if(flag==2)
                printf("%c->%s",ch,t[r][c]);
                if(str[i]==st[top])
                {
                        flag=1;
```

```
                ch=pop();
                i++;
            }
            else
            {
                flag=2;
                if(st[top]=='E')
                    r=0;
                else if(st[top]=='X')
                    r=1;
                else if(st[top]=='T')
                    r=2;
                else if(st[top]=='Y')
                    r=3;
                else if(st[top]=='F')
                    r=4;
                else
                    break;
                if(str[i]=='+')
                    c=0;
                else if(str[i]=='*')
                    c=1;
                else if(str[i]=='i')
                    c=2;
                else if(str[i]=='(')
                    c=3;
                else if(str[i]==')')
                    c=4;
                else if(str[i]=='$')
                    c=5;
                else
                    break;
                if(strcmp(t[r][c],"$")==0)
                    break;
                ch=pop();
                temp=t[r][c];
                m=strlen(temp);
                if(strcmp(t[r][c],"e")!=0)
                {
                for(k=m-1;k>=0;k--)
                    push(temp[k]);
                }
            }
        printf("\n");
        }
if(i==n)
        printf("\nparsed successfully");
```

```
else
        printf("\nnot parsed");
}
```

**OUTPUT**

1)

Enter any String(Append with $)i+i*i$

| Stack | Input | Output |
|---|---|---|
| $E | i+i*i$ | |
| $HT | i+i*i$ | E->TH |
| $HUF | i+i*i$ | T->FU |
| $HUi | i+i*i$ | F->i |
| $HU | +i*i$ | POP |
| $H | +i*i$ | U->ε |
| $HT+ | +i*i$ | H->+TH |
| $HT | i*i$ | POP |
| $HUF | i*i$ | T->FU |
| $HUi | i*i$ | F->i |
| $HU | *i$ | POP |
| $HUF* | *i$ | U->*FU |
| $HUF | i$ | POP |
| $HUi | i$ | F->i |
| $HU | $ | POP |
| $H | $ | U->ε |
| $ | $ | H->ε |

Given String is accepted

2)

Enter any String(Append with $)i+i**i$

| Stack | Input | Output |
|---|---|---|
| $E | i+i**i$ | |
| $HT | i+i**i$ | E->TH |
| $HUF | i+i**i$ | T->FU |
| $HUii | +i**i$ | F->i |
| $HU | +i**i$ | POP |
| $H | +i**i$ | U->ε |
| $HT+ | +i**i$ | H->+TH |
| $HT | i**i$ | POP |
| $HUF | i**i$ | T->FU |
| $HUi | i**i$ | F->i |
| $HU | **i$ | POP |
| $HUF* | **i$ | U->*FU |
| $HUF | *i$ | POP |
| $HU$ | *i$ | F->$ |

Syntax Error

Given String is not accepted

## 6) Design predictive parser for the given language

```
#include<stdio.h>
#include<string.h>
# define SIZE 30
charst[100];
int top=-1;
int s1(char),ip(char);
int n1,n2;
charnt[SIZE],t[SIZE];
/*Function to return variable index*/
int s1(char c)
{
int i;
for(i=0;i<n1;i++)
{
if(c==nt[i])
return i;
}
}
/*Function to return terminal index*/
intip(char c)
{
int i;
for(i=0;i<n2;i++)
{
if(c==t[i])
return i;
}
}
void push(char c)
{
top++;
st[top]=c;
return;
}
void pop()
{
top--;
return;
}
main()
{
char table[SIZE][SIZE][10],input[SIZE];
intx,f,s,i,j,u;
printf("Enter the number of variables:");
scanf("%d",&n1);
printf("\nUse single capital letters for variables\n");
for(i=0;i<n1;i++)
{
```

```
printf("Enter the %d nonterminal:",i+1);
scanf("%c",&nt[i]);
}
printf("Enter the number of terminals:");
scanf("%d",&n2);
printf("\nUse single small letters for terminals\n");
for(i=0;i<n2;i++)
{
printf("Enter the %d terminal:",i+1);
scanf("%c",&t[i]);
}
/*Reading the parsing table*/
printf("Please enter only right sides of productions\n");
printf("Use symbol n to denote no entry and e to epsilon\n");
for(i=0;i<n1;i++)
{
for(j=0;j<n2;j++)
{
printf("\nEnter the entry for %c under %c:",nt[i],t[j]);
scanf("%s",table[i][j]);
}
}
/*Printing the parsing taable*/
for(i=0;i<n2;i++)
printf("\t%c",t[i]);
printf("\n----------------------------------------------------------\n");
for(i=0;i<n1;i++)
{
printf("\n%c|\t",nt[i]);
for(j=0;j<n2;j++)
{
if(!strcmp(table[i][j],"n"))
printf("\t");
else
printf("%s\t",Table[i][j]);
}
printf("\n");
}
printf("Enter the input:");
scanf("%s",input);
/*Initialising the stack*/
top++;
st[top]='$';
top++;
st[top]=nt[0];
printf("STACK content INPUT content PRODUCTION used\n");
printf("-----------------------------------------------------------\n");
i=0;
printf("$%c\t\t\t%s\n\n",st[top],input);
while(st[top]!='$')
```

```
{
x=0;
f=s1(st[top]);
s=ip(input[i]);
if(!strcmp(table[f][s],"n"))
{
printf("'String not accepted");
}
else
if(!strcmp(table[f][s],"e"))
{
pop();
}
else
if(st[top]==input[i])
{
x=1;
pop();
i++;
}
else
{
pop();
for(j=strlen(table[f][s])-1;j>0;j--)
{
{
push(table[f][s][j]);
}
}
for(u=0;u<=top;u++)
printf("%c",st[u]);
printf("\t\t\t");
for(u=i;input[u]!='\0';u++)
printf("%c",input[u]);
printf("\t\t\t");
if(x==0)
printf("%c->%s\n\n",nt[f],table[f][s]);
printf("'\n\n");
}
printf("\n\nThus string is accepted");
}
}
```

**OUTPUT:**

```
Enter the number of variables:5
Use single capital letters for the variables
Enter the 1 non terminal:E
Enter the 2 non terminal:A
Enter the 3 non terminal:T
Enter the 4 non terminal:B
Enter the 5 non terminal:F
Enter the number of terminals:6
Use only single small letter for the terminals
Enter the 1 terminal:+
Enter the 2 terminal:*
Enter the 3 terminal:(
Enter the 4 terminal:)
Enter the 5 terminal:i
Enter the 6 terminal:$
Please enter only the right sides of productions.
Use symbol n to denote noentry and e to epsilon
Enter the entry for E under $: n
Enter the entry for E under +: n
Enter the entry for E under *: n
Enter the entry for E under (: TA
Enter the entry for E under ): n
Enter the entry for E under i: TA
Enter the entry for A under +: +TA
Enter the entry for A under *: n
Enter the entry for A under (: n
Enter the entry for A under ): e
Enter the entry for A under i: n
Enter the entry for A under $: e
Enter the entry for T under +: n
Enter the entry for T under *: n
Enter the entry for T under (: FB
Enter the entry for T under ): n
Enter the entry for T under i: FB
Enter the entry for T under $: n
Enter the entry for B under +: e
Enter the entry for B under *: *FB
Enter the entry for B under (: n
Enter the entry for B under ): e
Enter the entry for B under i: n
Enter the entry for B under $: e
Enter the entry for F under +: n
Enter the entry for F under *: n
Enter the entry for F under (: (E)
Enter the entry for F under ): n
Enter the entry for F under i: i
Enter the entry for F under $: n
+ * ( ) i $
--------------------------------------------------------------------------
```

E| TA TA
A| +TA e e
T| FB FB
B| e *FB e e
F| (E) i
Enter the input: i+i*i$
Stack content Input content Production used
---
$E i+i*i$
$A i+i*i$ E->TA
$AB i+i*i$ T->FB
$AB i+i*i$ F->i
$A +i*i$
$ +i*i$ B->e
$AT +i*i$ A->+TA
$A i*i$
$AB i*i$ T->FB
$AB i*i$ F->i
$A *i$
$ABF *i$ B->*FB
$AB i$
$AB i$ F->i
$A $
$ $ B->e
$ A->e
Thus string is accepted

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

```
#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 GRAMMER\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++;
}
check();
}
void check()
{
int flag=0;
temp2[0]=stack[st_ptr];
temp2[1]='\0';
if((isalpha(temp2[0])))
{
```

```
 stack[st_ptr]='E';
printf("\n $%s\t\t%s$\t\t\tE->%s",stack,ip_sym,temp2);
 flag=1;
}
if((!strcmp(temp2,"+"))||(!strcmp(temp2,"*"))||(!strcmp(temp2,"/")))
{
 flag=1;
}
if((!strcmp(stack,"E+E"))||(!strcmp(stack,"E/E"))||(!strcmp(stack,"E*E")))
{
if(!strcmp(stack,"E+E"))
{
strcpy(stack,"E");
printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
}
else if(!strcmp(stack,"E/E"))
{
strcpy(stack,"E");
printf("\n $%s\t\t %s$\t\t\tE->E/E",stack,ip_sym);
}
else
{
strcpy(stack,"E");
printf("\n $%s\t\t%s$\t\t\tE->E*E",stack,ip_sym);
}
flag=1;
st_ptr=0;
}
if(!strcmp(stack,"E")&&ip_ptr==len)
{
printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip_sym);
exit(0);
}
if(flag==0)
{
printf("\n $%s\t\t%s$\t\t\tReject",stack,ip_sym);
exit(0);
}
return;
}
```

**OUTPUT:**

1)

```
SHIFT REDUCE PARSER GRAMMER
 E->E+E
 E->E/E
 E->E*E
 E->E-E
 E->id
enter the input symbol:        a+b*c
stack implementation table
```

| stack | input symbol | action |
|---|---|---|
| $ | a+b*c$ | -- |
| $a | +b*c$ | shift a |
| $E | +b*c$ | E->a |
| $E+ | b*c$ | shift + |
| $E+b | *c$ | shift b |
| $E+E | *c$ | E->b |
| $E | *c$ | E->E+E |
| $E* | c$ | shift * |
| $E*c | $ | shift c |
| $E*E | $ | E->c |
| $E | $ | E->E*E |
| $E | $ | ACCEPT |

2) SHIFT REDUCE PARSER GRAMMER

E->E+E
E->E/E
E->E*E
E->E-E
E->id

enter the input symbol: a+b*+c

stack implementation table

| stack | input symbol | action |
|---|---|---|
| $ | a+b*+c$ | -- |
| $a | +b*+c$ | shift a |
| $E | +b*+c$ | E->a |
| $E+ | b*+c$ | shift + |
| $E+b | *+c$ | shift b |
| $E+E | *+c$ | E->b |
| $E | *+c$ | E->E+E |
| $E* | +c$ | shift * |
| $E*+ | c$ | shift + |
| $E*+c | $ | shift c |
| $E*+E | $ | E->c |
| $E*+E | | reject |

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

```
{%
#nclude<stdio.h>
#include<conio.h>
intyylex(void);
%}
%token ID
%start line
%%
line:expr '\n', {printf("%d",S1);}
expr:expr'+'term {SS=S1+S3;}
   |term
term:term'*'factor {SS=S1+S3;}
|factor
factor:'('expr')' {SS=S2;}
|ID
%%

yylex()
{
char c[10],i;
gets(c);
if(isdigit(c))
{
yylval=c;
return ID;
}
return c;
}
```

**Output:**

```
$vi  lalr.y
$yacc –v lalr.y
$vi  y.output
y.output contains the ouput

1  line : expr '\n'
2  expr : expr '+' term
   3       | term
4  term : term '*' factor
   5       | factor
6  factor : '(' expr ')'
   7        | ID
^L
state 0
        $accept : . line $end  (0)
ID  shift 1
        '('  shift 2
        .  error
line  goto 3
```

```
exprgoto 4
term  goto 5
state 1
factor : ID .  (7)
	.  reduce 7
state 2
factor : '(' . expr ')'  (6)
ID  shift 1
	'('  shift 2
	.  error
exprgoto 7
term  goto 5
factor  goto 6
state 3
	$accept : line . $end  (0)
	$end  accept
state 4
line :expr . '\n'  (1)
expr :expr . '+' term  (2)
	'\n'  shift 8
	'+'  shift 9
	.  error
state 5
expr : term .  (3)
term : term . '*' factor  (4)
	'*'  shift 10
	'\n'  reduce 3
	'+'  reduce 3
	')'  reduce 3
state 6
term : factor .  (5)
	.  reduce 5
state 7
expr :expr . '+' term  (2)
factor : '(' expr . ')'  (6)
	'+'  shift 9
	')'  shift 11
	.  error
state 8
line :expr '\n' .  (1)
	.  reduce 1
state 9
expr :expr '+' . term  (2)
ID  shift 1
	'('  shift 2
	.  error
term  goto 12
factor  goto 6
state 10
term : term '*' . factor  (4)
```

```
ID  shift 1
        '('  shift 2
        .  error
factor  goto 13
state 11
factor : '(' expr ')' .  (6)
        .  reduce 6
state 12
expr :expr '+' term .  (2)
term : term . '*' factor  (4)
        '*'  shift 10
        '\n'  reduce 2
        '+'  reduce 2
        ')'  reduce 2
state 13
term : term '*' factor .  (4)
       .  reduce 4
8 terminals, 5 nonterminals
8 grammar rules, 14 states
```

**9) Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools**

```
// Program name as “lexicalfile.l”
%{
#include<stdio.h>
%}

delim [\t]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
num {digit}+(\.{digit}+)?(E[+|-]?{digit}+)?

%%
ws {printf("no action");}
if|else|then {printf("%s is a keyword",yytext);} // TYPE 32 KEYWORDS
{id} {printf("%s is a identifier",yytext);}
{num} {printf(" it is a number");}
"<" {printf("it is a relational operator less than");}
"<=" {printf("it is a relational operator less than or equal");}
">" {printf("it is a relational operator greater than");}
">=" {printf("it is a relational operator greater than");}
"==" {printf("it is a relational operator equal");}
"<>" {printf("it is a relational operator not equal");}
%%

main()
{
yylex();
}
```

OUTPUT

```
lexlexicalfile.l
cc  lex.yy.c  -ll
if
if is a keyword
number
number is a identifier
254
It is a number
<>
it is a relational operator not equal
^Z
```

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

```
#include<stdio.h>
int main()
 {
  int i;
  for(i=0;i<10;i+=2)
    {
       printf("fun(%d)\n",i+1);
       printf("fun(%d)\n",i+2);

    }
  }
```

**OUTPUT**

```
fun(1)
fun(2)
fun(3)
fun(4)
fun(5)
fun(6)
fun(7)
fun(8)
fun(9)
fun(10)
```

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

**<int.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;
}
```
<**int.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 test1.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:**

$lex int.l

$yacc –d –v int.y

$gcc lex.yy.c y.tab.c  –lm

$./a.out test1.c

```
----------------------------
Pos Operator Arg1 Arg2 Result
-----------------
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
-----------------------
```

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

```
#include<stdio.h>

int main()

 {

     int x, y, z;

    x = 10;

    y = x + 45;

   z = y + 4;

   printf("The value of z = %d", z);

   return 0;

}
```

**OUTPUT:**

$ vi test.c

$ cc –c –S test.c

$ vi test.s //before optimization assembly code

```
main:

      pushl   %ebp
      movl    %esp, %ebp
      andl    $-16, %esp
      subl    $32, %esp
      movl    $10, 20(%esp)
      movl    20(%esp), %eax
      addl    $45, %eax
      movl    %eax, 24(%esp)
      movl    24(%esp), %eax
      addl    $4, %eax
      movl    %eax, 28(%esp)
      movl    $.LC0, %eax
      movl    28(%esp), %edx
      movl    %edx, 4(%esp)
      movl    %eax, (%esp)
      call    printf
      movl    $0, %eax
      leave
```

```
        ret
$ cc –c –S  -O2 test.c

$ vi test.s //after optimization assembly code

main:

        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $59, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        xorl    %eax, %eax
        leave
        ret
```

## VIVA QUESTIONS WITH ANSWERS

1. What is a compiler?

A compiler is a program that reads a program written in one language –the source language and translates it into an equivalent program in another language-the target language. The compiler reports to its user the presence of errors in the source program.

2. What are the two parts of a compilation? Explain briefly.

Analysis and Synthesis are the two parts of compilation. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

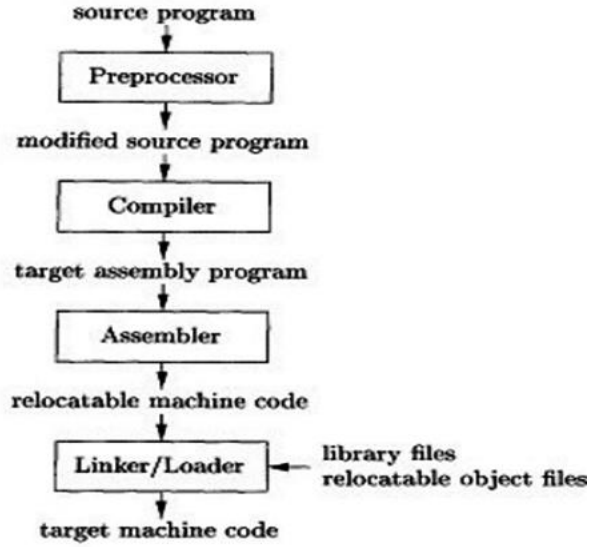
The synthesis part constructs the desired target program from the intermediate representation.

3. List the subparts or phases of analysis part.

Analysis consists of three phases:

Linear Analysis.
Hierarchical Analysis.
Semantic Analysis.

4. Depict diagrammatically how a language is processed.




5. What is linear analysis?

Linear analysis is one in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning. Also called lexical analysis or scanning.

6. List the various phases of a compiler.

The following are the various phases of a compiler:

Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate code generator
Code optimizer
Code generator

7. What are the classifications of a compiler?

Compilers are classified as:
Single- pass
Multi-pass
Load-and-go
Debugging or optimizing

8. What is a symbol table?

A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.

9. Mention some of the cousins of a compiler.

Cousins of the compiler are:

- Preprocessors
- Assemblers
- Loaders and Link-Editors

10. List the phases that constitute the front end of a compiler.

The front end consists of those phases or parts of phases that depend primarily on the source language and are largely independent of the target machine. These include

- Lexical and Syntactic analysis
- The creation of symbol table
- Semantic analysis
- Generation of intermediate code

A certain amount of code optimization can be done by the front end as well. Also includes error handling that goes along with each of these phases.

11. Mention the back-end phases of a compiler.

The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language. These include

- Code optimization
- Code generation, and along with error handling and symbol- table operations.

12. Define compiler-compiler.

Systems to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems.

Largely they are oriented around a particular model of languages , and they are suitable for generating compilers of languages similar model.

13. List the various compiler construction tools.

The following is a list of some compiler construction tools:

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

14. Differentiate tokens, patterns, lexeme.

Tokens- Sequence of characters that have a collective meaning.

Patterns- There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token

Lexeme- A sequence of characters in the source program that is matched by the pattern for a token.

15. List the operations on languages.

Union – L U M ={s | s is in L or s is in M}

Concatenation – LM ={st | s is in L and t is in M}

Kleene Closure – L* (zero or more concatenations of L)

Positive Closure – L+ ( one or more concatenations of L)

16. Write a regular expression for an identifier.

An identifier is defined as a letter followed by zero or more letters or digits. The regular expression for an identifier is given as

letter (letter | digit)*

17. Mention the various notational shorthands for representing regular expressions.

One or more instances (+)

Zero or one instance (?)

Character classes ([abc] where a,b,c are alphabet symbols denotes the regular expressions a | b | c.)

Non regular sets

18. What is the function of a hierarchical analysis?

Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. It also termed as Parsing.

19. What does a semantic analysis do?

Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully. Mainly performs type checking.

20. List the various error recovery strategies for a lexical analysis.

Possible error recovery actions are:

Panic mode recovery

Deleting an extraneous character

Inserting a missing character

Replacing an incorrect character by a correct character

Transposing two adjacent characters

21. Define parser.

Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. It also termed as Parsing.

22. Mention the basic issues in parsing.

There are two important issues in parsing.

Specification of syntax

Representation of input after parsing.

23. Why lexical and syntax analyzers are separated out?

Reasons for separating the analysis phase into lexical and syntax analyzers:

Simpler to design.

Compiler efficiency is improved.

Compiler portability is enhanced.

24. Define a context free grammar.

A context free grammar G is a collection of the following

V is a set of non terminals

T is a set of terminals

S is a start symbol

P is a set of production rules

G can be represented as G = (V,T,S,P)

25. Briefly explain the concept of derivation.

Derivation from S means generation of string w from S. For constructing derivation two things are important.

i) Choice of non-terminal from several others.

ii) Choice of rule from production rules for corresponding non terminal.

Instead of choosing the arbitrary non terminal one can choose

i) either leftmost derivation – leftmost non terminal in a sentinel form

ii) or rightmost derivation – rightmost non terminal in a sentinel form

26. Define ambiguous grammar.

A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language L(G). i.e. both leftmost and rightmost derivations are same for the given sentence.

27. What is a operator precedence parser?

A grammar is said to be operator precedence if it possess the following properties:

1. No production on the right side is ε.
2. There should not be any production rule possessing two adjacent non terminals at the right hand side.

28. List the properties of LR parser.

1. LR parsers can be constructed to recognize most of the programming languages for which the context free grammar can be written.
2. The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
3. LR parsers work using non backtracking shift reduce technique yet it is efficient one.

29. Mention the types of LR parser.

SLR parser- simple LR parser
LALR parser- lookahead LR parser
Canonical LR parser

30. What are the problems with top down parsing?

The following are the problems associated with top down parsing:
Backtracking
Left recursion
Left factoring
Ambiguity

31. Write the algorithm for FIRST and FOLLOW.

FIRST():

1. If X is terminal, then FIRST(X) IS {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non terminal and X → Y1,Y2..Yk is a production, then place a in FIRST(X) if for some i , a is in FIRST(Yi) , and ε is in all of FIRST(Y1),…FIRST(Yi-1);

FOLLOW():

1. Place $ in FOLLOW(S),where S is the start symbol and $ is the input right end-marker.
2. If there is a production A → αBβ, then everything in FIRST(β) except for ε is placed in FOLLOW(B).
3. If there is a production A → αB, or a production A→ αBβ where FIRST(β) contains ε , then everything in FOLLOW(A) is in FOLLOW(B).

32. List the advantages and disadvantages of operator precedence parsing.

Advantages
This typeof parsing is simple to implement.
Disadvantages

1. The operator like minus has two different precedence(unary and binary).Hence it is hard to handle tokens like minus sign.
2. This kind of parsing is applicable to only small class of grammars.

33. What is dangling else problem?

Ambiguity can be eliminated by means of dangling-else grammar which is show below:

stmt → if expr then stmt

| if expr then stmt else stmt
| other

34. Write short notes on YACC.

YACC is an automatic tool for generating the parser program.

YACC stands for Yet Another Compiler Compiler which is basically the utility available from UNIX. Basically YACC is LALR parser generator. It can report conflict or ambiguities in the form of error messages.

35. What is meant by handle pruning?

A rightmost derivation in reverse can be obtained by handle pruning. If w is a sentence of the grammar at hand, then w = γn, where γn is the nth right-sentential form of some as yet unknown rightmost derivation.

S = γ0 => γ1…=> γn-1 => γn = w

36. Define LR(0) items.

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus, production A → XYZ yields the four items

A→.XYZ
A→X.YZ
A→XY.Z
A→XYZ.

37. What is meant by viable prefixes?

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

38. Define handle.

A handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

A handle of a right – sentential form γ is a production A→β and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ. That is , if S =>αAw =>αβw,then A→β in the position following α is a handle of αβw.

39. What are kernel & non-kernel items?

Kernel items, whish include the initial item, S'→ .S, and all items whose dots are not at the left end.

Non-kernel items: which have their dots at the left-end.

40. What is phrase level error recovery?

Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

41. What are the benefits of intermediate code generation?

A Compiler for different machines can be created by attaching different back end to the existing front ends of each machine. A Compiler for different source languages can be created by proving different front ends for corresponding source languages t existing back end. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

42. What are the various types of intermediate code representation?

There are mainly three types of intermediate code representations.

Syntax tree

Postfix
Three address code

43. Define back-patching.

Back-patching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.In the semantic actions the functions used are mklist(i),merge_list(p1,p2) and backpatch(p,i)

44. What is the intermediate code representation for the expression a or b and not c?

The intermediate code representation for the expression a or b and not c is the three address sequence

t1 := not c
t2 := b and t1
t3 := a or t2

45. What are the various methods of implementing three address statements?

The three address statements can be implemented using the following methods.

Quadruple : a structure with atmost four fields such as operator(OP),arg1,arg2,result.

Triples : the use of temporary variables is avoided by referring the pointers in the symbol table.

Indirect triples : the listing of triples has been done and listing pointers are used instead of using statements.

46. Give the syntax-directed definition for if-else statement.

1. S → if E then S1
   E.true := new_label()
   E.false :=S.next
   S1.next :=S.next
   S.code :=E.code | | gen_code(E.true ‘: ‘) | | S1.code
2. S → if E then S1 else S2
   E.true := new_label()
   E.false := new_label()
   S1.next :=S.next
   S2.next :=S.next
   S.code :=E.code | | gen_code(E.true ‘: ‘) | | S1.code| | gen_code(‘go to’,S.next) | |gen_code(E.false ‘:’) | | S2.code

47. What is a flow graph?

A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes to the flow graph are represented by basic blocks
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block B1 to block B2 if B2 immediately follows B1 in the given sequence. We can say that B1 is a predecessor of B2.

48. What is a DAG? Mention its applications.

Directed acyclic graph (DAG) is a useful data structure for implementing transformations on basic blocks. DAG is used in

- Determining the common sub-expressions.
- Determining which names are used inside the block and computed outside the block.
- Determining which statements of the block could have their computed value outside the block.
- Simplifying the list of quadruples by eliminating the common su-expressions and not performing the assignment of the form x := y unless and until it is a must.

49. Define peephole optimization.

Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence.

50. List the characteristics of peephole optimization.

- Redundant instruction elimination
- Flow of control optimization
- Algebraic simplification
- Use of machine idioms

51. How do you calculate the cost of an instruction?

The cost of an instruction can be computed as one plus cost associated with the source and destination addressing modes given by added cost.

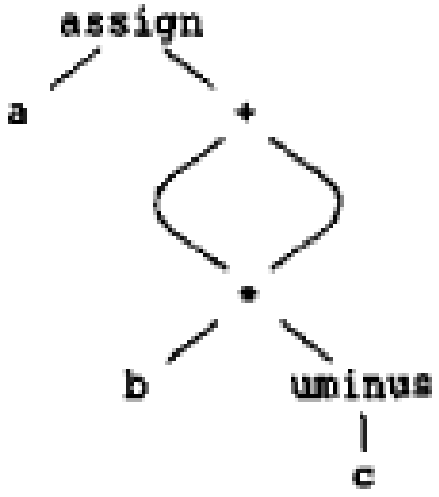
MOV R0,R1 1
MOV R1,M 2
SUB 5(R0),*10(R1) 3

52. What is a basic block?

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

Eg.
t1:=a*5
t2:=t1+7
t3:=t2-5
t4:=t1+t3
t5:=t2+b

53. How would you represent the following equation using DAG?
a:=b*-c+b*-c




54. What are the basic goals of code movement?

To reduce the size of the code(space complexity). To reduce the frequency of execution of code(time complexity).

55. List the different storage allocation strategies.

The strategies are:

Static allocation
Stack allocation
Heap allocation

56. What are the contents of activation record?

The activation record is a block of memory used for managing the information needed by a single execution of a procedure. Various fields f activation record are:

- Temporary variables
- Local variables
- Saved machine registers
- Control link
- Access link
- Actual parameters
- Return values

57. What is dynamic scoping?

In dynamic scoping a use of non-local variable refers to the non-local data declared in most recently called and still active procedure. Therefore each time new findings are set up for local names called procedure. In dynamic scoping symbol tables can be required at run time.

58. Define symbol table.

Symbol table is a data structure used by the compiler to keep track of semantics of the variables. It stores information about scope and binding information about names.

59. What is code motion?

Code motion is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same result independent of the number of times the loop is executed. Such an expression is placed before the loop.

60. What are the properties of optimizing compiler?

The source code should be such that it should produce minimum amount of target code. There should not be any unreachable code. Dead code should be completely removed from source language.

The optimizing compilers should apply following code improving transformations on source language.

i) common subexpression elimination
ii) dead code elimination
iii) code movement
iv) strength reduction

## REFERENCES


1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monica S Lam, Ravi Sethi, Jeffrey D. Ullman,2nd ed, Pearson,2007
2. Compiler Design, K. Muneeswaran, Oxford..