**1. WAP in C to implement the combined transition diagram for i) identifiers: BEGIN, END, IF, THEN, ELSE ii) integer constants and iii) relational operators: <, <=, =, < >, >, >= that are commonly used in any high level language.**

#include<iostream>

using namespace std;

int main()

{

string s;

cin>>s;

int n = s.length();

int i=0;

if(s[i]>='0' && s[i]<='9')

{

i++;

while(i<n)

{

if(s[i]>='0' && s[i]<='9')

i++;

else

break;

}

if(i==n)

{

cout<<"Integer Constant"<<endl;

}

else

{

cout<<"Not a valid token"<<endl;

}

}

else

{

switch(s[i])

{

case 'b':

{

i++;

if(s[i]=='e')

{

i++;

if(s[i]=='g')

{

i++;;

if(s[i]=='i')

{

i++;

if(s[i]=='n')

{

i++;

}

}

}

}

if(i==n)

{

cout<<"Valid Keyword"<<endl;

break;

}

else

{

cout<<"Not a valid token"<<endl;

break;

}

}

case 'e':

{

i++;

if(s[i]=='l')

{

i++;

if(s[i]=='s')

{

i++;;

if(s[i]=='e')

{

i++;

}

}

}

else if(s[i]=='n')

{

i++;

if(s[i]=='d')

{

i++;

}

}

if(i==n)

{

cout<<"Valid Keyword"<<endl;

break;

}

else

{

cout<<"Not a valid token"<<endl;

break;

}

}

case 'i':

{

i++;

if(s[i]=='f')

{

i++;

}

if(i==n)

{

cout<<"Valid Keyword"<<endl;

break;

}

else

{

cout<<"Not a valid token"<<endl;

break;

}

}

case 't':

{

i++;

if(s[i]=='h')

{

i++;

if(s[i]=='e')

{

i++;

if(s[i]=='n')

{

i++;

}

}

}

if(i==n)

{

cout<<"Valid Keyword"<<endl;

break;

}

else

{

cout<<"Not a valid token"<<endl;

break;

}

}

case '<':

{

i++;

if(i==n)

{

cout<<"Valid Relational Operator"<<endl;

break;

}

else if(i!=n && (s[i]=='=' || s[i]=='>'))

{

cout<<"Valid Relational Operator"<<endl;

break;

}

else

{

cout<<"Not a valid token"<<endl;

break;

}

}

case '=':

{

i++;

if(i==n)

{

cout<<"Valid Relational Operator"<<endl;

break;

}

else

{

cout<<"Not a valid token"<<endl;

break;

}

}

case '>':

{

i++;

if(i==n)

{

cout<<"Valid Relational Operator"<<endl;

break;

}

else if(s[i]=='=')

{

cout<<"Valid Relational Operator"<<endl;

break;

}

else

{

cout<<"Not a valid token"<<endl;

break;

}

}

default:

{

cout<<"Not a valid token"<<endl;

break;

}

}

}

return 0;

}

**2. Using flex, write a lexical analyzer for the following specifications of the tokens:**

**a. Comments are surrounded by /\* and \*/**

**b. Blanks between tokens are optional, with the exception that keywords must be surrounded by blanks and newlines.**

**c.Identifier:**

**letter → [a-z, A-Z]**

**digit → [0-9]**

**id → letter (letter | digit)\***

**The lexer shall recognize identifiers. An identifier is a sequence of letters and digits, starting with a letter. The underscore ‘\_’ counts as a letter.**

**d. Keywords:**

**begin, end, if, then, else, for , do , while, switch, case, default, break, continue, goto**

%{  
%}  
  
DIGIT [0-9]  
ID [a-z][a-zA-Z0-9]\*  
NOTID [0-9]\*[a-zA-Z0-9]\*  
COMMENTMULTI [/]{1}[\*]{1}[^\*]\*{1}[\*]{1}[/]  
COMMENTSINGLE [/]{1}[/]{1}[^\*]\*  
FRACTION [0-9]\*{1}[.]{1}[0-9]\*  
SINGLEDIGIT [0-9]  
%%  
{DIGIT}+ {printf("An Integer: %s(%d)\n",yytext);}  
  
if|then|begin|end|procedure|function { printf("A Keyword: %s\n",yytext); }  
  
{ID} { printf("An identifier: %s\n",yytext);}  
  
"<"|"<="|">"|">+"|"=="|"!=" {printf("A relational operator: %s\n",yytext); }  
  
{COMMENTMULTI} {printf("Comment MultiLine: %s\n",yytext);}  
{COMMENTSINGLE} {printf("Comment Single Line: %s\n",yytext);}  
{FRACTION} {printf("Fraction: %s\n",yytext);}  
  
{NOTID} {printf("Invalid Identifier: %s\n",yytext);}  
%%  
int main()  
{  
 yylex();  
}

**16. Write a YACC program that recognizes strings with balanced parenthesis.Consider all the three types of braces i.e. “{“, “]” and “(“ .**

Lex

%{

#include "y.tab.h"

%}

%%

[\t] {}

"(" return OPEN1;

")" return CLOSE1;

"{" return OPEN2;

"}" return CLOSE2;

"[" return OPEN3;

"]" return CLOSE3;

\n|. { return yytext[0];}

:%

Yacc

%{

#include<ctype.h>

#include<stdio.h>

#include "y.tab.h"

extern int yydebug;

%}

%token OPEN1 OPEN2 OPEN3 CLOSE1 CLOSE2 CLOSE3

%%

lines :s '\n' {printf("Balanced\n"); }

;

s :

|OPEN1 s CLOSE1 s

|OPEN2 s CLOSE2 s

|OPEN3 s CLOSE3 s

;

%%

void yyerror(char\* s)

{

 printf("error !\n");

}

int yywrap() {return 1; }

int main ()

{

yydebug = 1;

return yyparse();

}

**14. Write a YACC program to parse if-then-else statement following the grammar**

**S → iCtS| iCtSeS |a**

**C →b**

Lex

%{  
#include<stdio.h>  
#include"y.tab.h"  
extern int yylval;  
%}  
  
%%  
[a] {  
        yylval = atoi(yytext);  
        return a;  
   }  
[b] {  
        yylval = atoi(yytext);  
        return b;  
   }  
[i] {return 'i';}  
[t] {return 't';}  
[e] {return 'e';}  
[\t];  
[\n] return 0;  
. return yytext[0];  
  
%%  
  
int yywrap()  
{  
        return 1;  
}

Yacc

%{  
        #include<stdio.h>  
%}  
  
%token a b  
%left 'i' 't' 'e'  
  
%%  
  
stmt:S {printf("Statement belongs to this grammer\n");  }  
  
S:'i'C't'S {}  
 |'i'C't'S'e'S {}  
 |S1 {}  
  
S1:a {}  
  
C:b {}  
 ;  
  
%%  
  
main()  
{  
        printf("Enter statement for the grammer\n");  
        yyparse();  
}  
  
yyerror()  
{  
        printf("Invalid Statement\n");  
}

**5. Consider the following regular expressions:**

**a) ((a + b)\*(c+d)\*)+ + ab\*c\*d**

**b) (0 + 1)\* + 0\*1\***

**c) (01\*2 + 0\*2+1)+**

**Write flex programs for above regular expressions mentioned above.**

%{  
%}  
  
A [ab]\*[cd]\*  
E [a][b]\*[c]\*[d]  
B [01]\*  
C [0]\*[1]\*  
D [0][1]\*[2]|[0]\*[2]|[1]  
  
%%  
  
{A}+|{E} {printf("String Pattern valid for given R.E-1: %s(%d)\n", yytext); }  
{B}|{C} {printf("String Pattern valid for given R.E-2: %s(%d)\n", yytext); }  
{D}+ {printf("String Pattern valid for given R.E-3: %s(%d)\n", yytext); }  
  
%%  
  
int main()  
{  
    yylex();  
}

**13. Write a YACC program to parse an arithmetic expression following the grammar:**

**E → E + E | E – E | E \* E | E / E | E ↑ E | (E) | - E | id**

**Also, evaluate the arithmetic expression.**

Yacc

%{  
    #include<stdio.h>  
%}  
%token NUM  
%left '+' '-'  
%left '\*' '/'  
%left '(' ')'  
%left '^'  
%%  
expr: e{  
         printf("result:%d\n",$$);  
         return 0;  
        }  
e:e'+'e {$$=$1+$3;}  
 |e'-'e {$$=$1-$3;}  
 |e'\*'e {$$=$1\*$3;}  
 |e'/'e {$$=$1/$3;}  
 |'('e')' {$$=$2;}  
 |e'^'e {$$=$1^$3;}  
 | NUM {$$=$1;}  
;  
%%  
  
main()  
{  
   printf("\n enter the arithematic expression:\n");  
   yyparse();  
   printf("\nvalid expression\n");  
}  
yyerror()  
{  
   printf("\n invalid expression\n");  
   exit(0);  
}

**Lex**

%{  
#include<stdio.h>  
#include"y.tab.h"  
extern int yylval;  
%}  
  
%%  
[0-9]+ {  
          yylval=atoi(yytext);  
          return NUM;  
       }  
[\t] ;  
\n return 0;  
. return yytext[0];  
%%

**4. Write a program to generate precedence function for the following grammar assuming that the precedence table is given:**

**E → E+E | E-E | E\*E | E/E | E↑E | (E) | id**

#include<vector>

using namespace std;

vector<int> vec[10];

bool visit[10];

int dfs(int v)

{

int i,val;

int ln = vec[v].size();

int maxm = 0;

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

{

val = dfs(vec[v][i]);

if(val>maxm)

{

maxm = val;

}

}

return maxm+1;

}

int main()

{

int p\_table[5][5],i,j,n;

cin>>n;

cout<<"Enter precedence table\n";

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

{

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

{

cin>>p\_table[i][j];

}

}

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

{

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

{

if(p\_table[i][j] == 2)

{

vec[i].push\_back(n+j);

{

maxm = val;

}

}

return maxm+1;

}

int main()

{

int p\_table[5][5],i,j,n;

cin>>n;

cout<<"Enter precedence table\n";

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

{

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

{

cin>>p\_table[i][j];

}

}

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

{

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

{

if(p\_table[i][j] == 2)

{

vec[i].push\_back(n+j);

}

else if(p\_table[i][j] == 1)

{

vec[n+j].push\_back(i);

}

}

}

int f[5],g[5];

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

{

f[i] = dfs(i)-1;

}

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

{

g[i] = dfs(i+n)-1;

}

cout<<"f: ";

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

cout<<f[i]<<" ";

cout<<"\n";

cout<<"g: ";

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

cout<<g[i]<<" ";

}

**7. Write a program for FIRST and FOLLOW computations for the following grammar:**

**E → E + T | T**

**T → T \* F | F**

**F → (E) | id**

   #include<stdio.h>

#include<ctype.h>

char a[8][8];

struct firTab

{

    int n;

    char firT[5];

};

struct folTab

{

    int n;

    char folT[5];

};

struct folTab follow[5];

struct firTab first[5];

int col;

void findFirst(char,char);

void findFollow(char,char);

void folTabOperation(char,char);

void firTabOperation(char,char);

 main()

{

    int i,j,c=0,cnt=0;

    char ip;

    char b[8];

    printf("\nFIRST AND FOLLOW SET \n\nenter 8 productions in format A->B+T\n");

    for(i=0;i<8;i++)

    {

    scanf("%s",&a[i]);

    }

    for(i=0;i<8;i++)

    {   c=0;

    for(j=0;j<i+1;j++)

    {

        if(a[i][0] == b[j])

        {

            c=1;

            break;

        }

        }

    if(c !=1)

    {

      b[cnt] = a[i][0];

      cnt++;

    }

    }

     printf("\n");

    for(i=0;i<cnt;i++)

    {   col=1;

    first[i].firT[0] = b[i];

    first[i].n=0;

    findFirst(b[i],i);

    }

    for(i=0;i<cnt;i++)

    {

    col=1;

    follow[i].folT[0] = b[i];

    follow[i].n=0;

    findFollow(b[i],i);

     }

    printf("\n");

   for(i=0;i<cnt;i++)

   {

    for(j=0;j<=first[i].n;j++)

    {

            if(j==0)

            {

                printf("First(%c) : {",first[i].firT[j]);

            }

            else

            {

                printf(" %c",first[i].firT[j]);

            }

    }

    printf(" } ");

    printf("\n");

    }

     printf("\n");

   for(i=0;i<cnt;i++)

   {

    for(j=0;j<=follow[i].n;j++)

    {

            if(j==0)

            {

                printf("Follow(%c) : {",follow[i].folT[j]);

            }

            else

            {

                printf(" %c",follow[i].folT[j]);

            }

    }

    printf(" } ");

    printf("\n");

    }

}

void findFirst(char ip,char pos)

{

    int i;

    for(i=0;i<8;i++)

    {

        if(ip == a[i][0])

        {

            if(isupper(a[i][3]))

            {

                findFirst(a[i][3],pos);

            }

            else

        {

        first[pos].firT[col]=a[i][3];

        first[pos].n++;

        col++;

            }

        }

    }

}

void findFollow(char ip,char row)

{   int i,j;

    if(row==0 && col==1)

    {

        follow[row].folT[col]= '$';

        col++;

        follow[row].n++;

    }

    for(i=0;i<8;i++)

    {

        for(j=3;j<7;j++)

        {

            if(a[i][j] == ip)

            {

                if(a[i][j+1] == '\0')

                {

                    if(a[i][j] != a[i][0])

                    {

                        folTabOperation(a[i][0],row);

                    }

                }

                else if(isupper(a[i][j+1]))

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

                    {

                        firTabOperation(a[i][j+1],row);

                }

                }

                else

                {

                    follow[row].folT[col] = a[i][j+1];

                    col++;

                    follow[row].n++;

                }

            }

        }

    }

}

void folTabOperation(char ip,char row)

{   int i,j;

    for(i=0;i<5;i++)

    {

        if(ip == follow[i].folT[0])

        {

            for(j=1;j<=follow[i].n;j++)

            {

                follow[row].folT[col] = follow[i].folT[j];

                col++;

                follow[row].n++;

            }

        }

    }

}

void firTabOperation(char ip,char row)

{

        int i,j;

    for(i=0;i<5;i++)

    {

        if(ip == first[i].firT[0])

        {

            for(j=1;j<=first[i].n;j++)

            {

                if(first[i].firT[j] != '0')

                {

                    follow[row].folT[col] = first[i].firT[j];

                    follow[row].n++;

                    col++;

                }

                else

                {

                    folTabOperation(ip,row);

                }

            }

        }

    }

}

**11. Write a YACC program to check whether given string a^nb^n is accepted by the grammar. Also write a YACC program to recognize a valid variable which starts with a letter followed by a digit.**

%{

#include<stdio.h>

%}

%token a b

%%

stmt: S {printf("\n string belongs to grammer..\n"); exit(0);}

|error { printf(“\n” string does not belongs to grammer..\n”); exit(0); }

;

S: a S b

      |

      ;

%%

main()

{

printf("Enter String for Grammer a^nb^n:\n");

yyparse();

}

yylex()

{

char ch;

while((ch=getchar())==' ')

if(ch=='a')

return a;

if(ch=='b')

return b;

return ch;

}

yyerror(char \*S)

{

printf("%s",S);

}

**3. Using flex write a lexical analyzer for the following specifications of the tokens:**

**a. Keywords: else, int, void, if, else, while, return. For each one of them, the lexer shall return the tokens INT, CHAR, VOID, IF, ELSE, WHILE, RETURN respectively.**

**b. It recognizes integer numbers. An integer number is a sequence of digits, possibly starting with a + or -.**

**c. It recognizes real numbers. A real number is a sequence of digits, possibly starting with a + or – and / or with . and E notations. For each real number, it shall return the token REAL.**

**d. The lexer shall recognize the operators ‘->’, ‘&&’, ‘||’, ‘.’ for which it shall return the tokens PTR\_OP, AND\_OP, OR\_OP, and DOT\_OP respectively.**

**e. It recognizes operators ‘-’, ‘+’, ‘\*’, ‘/’ for which it shall return the same character as token.**

**f. It recognizes separators ‘;’, ‘{’, ‘}’, ‘,’, ‘=’, ‘(’, ‘)’, ‘&’, ‘~’, , ‘[‘ and ‘]’ for which it shall return the same character as token.**

%{

%}

DIGIT [0-9]

%%

(\+|-)\*({DIGIT}+) {printf("An integer %s\n",yytext);}

(\+|-)\*({DIGIT}+(\.){DIGIT}+) {printf("A fraction %s\n",yytext);}

(\+|-)\*{DIGIT}\*(\.){DIGIT}+(E)\*(\+|-)\*{DIGIT}+ {printf("A fraction %s\n",yytext);}

(-|\\*|\/|\+) {printf("An operator %s\n",yytext);}

(->) {printf("PTR\_OP\n");}

(&&) {printf("AND\_OP\n");}

(\|\|) {printf("OR\_OP");}

(\.) {printf("DOT\_OP");}

%%

int main()

{

yylex();

}

**8. Write a YACC program to check whether given string is palindrome or not by the grammar.**

**Lex**

%{

%}

%option noyywrap

%%

a return A;

b return B;

\n return '\n';

. {fprintf(stderr, "Error\n"); exit(1);}

%%

**Yacc**

%{

#include <stdio.h>

int i=0;

%}

%token A B

%glr-parser

%%

S : pal '\n' {i=1; return 1 ;}

| error '\n' {i=0; return 1 ;}

pal: A pal A

| B pal B

| A

| B

|

;

%%

#include "lex.yy.c"

int main() {

yyparse();

if(i==1) printf("Valid\n");

else printf("inValid\n");

return 0;

}

int yyerror(char\* s) { return 0; }

**12. Write a YACC program to recognize a valid variable which starts with a letter followed by a letter.**

Lex

%{

%}

%option noyywrap

%%

a {return A; }

b {return B; }

\n {return '\n'; }

. {fprintf(stderr, "Error\n"); }

%%

Yacc

%{

#include<stdio.h>

int i=0;

%}

%token A B

%glr-parser

%%

stmt : S '\n' {i=1; return 1;}

| error '\n' {i=0; return 1;}

S : A S B

|

;

%%

#include "lex.yy.c"

int main()

{

yyparse();

if(i==1)

printf("Valid String\n");

else

printf("Invalid String\n");

return 0;

}

int yyerror(char \*s)

{

return 0;

}

**17. Write a YACC program to implement a top-down parsing by recursive procedures for the grammar.**

**S --> ABC**

**A--> abA | ab**

**B--> b | BC**

**C--> c | cC**

**Lex**

%{

%}

%option noyywrap

%%

a {return X; }

b {return Y; }

c {return Z; }

\n {return '\n'; }

. {fprintf(stderr, "Error\n"); }

%%

Yacc

%{

#include<stdio.h>

#include<math.h>

int i=0;

%}

%token X Y Z

%glr-parser

%%

stmt : S '\n' {i=1; return 1;}

| error '\n' {i=0; return 1;}

S : A B C

A : X Y

| X Y A

B : Y D

D : C D

|

C : Z

| Z C

;

%%

#include "lex.yy.c"

int main()

{

yyparse();

if(i==1)

printf("Valid Grammer\n");

else

printf("InValid Grammer\n");

return 0;

}

int yyerror(char \*s)

{

return 0;

}

**9. Write a program to implement a top-down parsing by recursive procedures for the grammar:**

**S → Aa | b**

**A → Ac | Sd | f**

#include<bits/stdc++.h>

using namespace std;

string str;

int i,n;

bool match(char ch)

{

if(i<n && str[i]==ch)

{

i+=1;

return true;

}

else

{

return false;

}

}

bool AP()

{

int save = i;

if(i<n)

{

if(match('c'))

{

if(AP())

return true;

}

else if(match('a') && match('d'))

{

if(AP())

return true;

}

else

{

i = save;

return true;

}

}

else

return true;

}

bool A()

{

int save = i;

if(i<n)

{

if(match('b') && match('d'))

{

if(AP())

return true;

}

else if(match('f'))

{

if(AP())

return true;

}

else

{

i = save;

return true;

}

}

else

return true;

}

bool S()

{

if(i<n)

{

if(i==n-1 && match('b'))

return true;

else

{

if(A())

{

if(match('a'))

return true;

else

return false;

}

else

return false;

}

}

else

return true;

return false;

}

int main()

{

cin>>str;

i = 0;

n = str.length();

if(S() && i==n)

cout<<"Valid String"<<endl;

else

cout<<"Invalid String"<<endl;

return 0;

}

**10. Write a YACC program to implement a top-down parsing by recursive procedures for the grammar:**

**S → cAd**

**A → ab| a**

**Lex**

%{

%}

%option noyywrap

%%

a {return X; }

b {return Y; }

c {return Z; }

d {return D; }

f {return F; }

\n {return '\n'; }

. {fprintf(stderr, "Error\n"); }

%%

Yacc

%{

#include<stdio.h>

#include<math.h>

int i=0;

%}

%token X Y Z D F

%glr-parser

%%

stmt : S '\n' {i=1; return 1;}

| error '\n' {i=0; return 1;}

S : A X

| Y

A : Y D B

| F B

B : Z B

| X D B

|

;

%%

#include "lex.yy.c"

int main()

{

yyparse();

if(i==1)

printf("Valid Grammer\n");

else

printf("InValid Grammer\n");

return 0;

}

int yyerror(char \*s)

{

return 0;

}