```
/*

Compiler's Lab :

*  - Abhishek Goyal - 120101004

*  - Dheeraj Khatri - 120101021

*  - Ojas Deshpande - 120101046

*

*

*

* Input format - Regex must be well paranthesized. No need to give any symbol for

concatenation.

*                Only 'or' (|) and 'kleene star' (*) operations are supported. Alphabet - {a,b}

*

*/
```

```
#include <bits/stdc++.h>

using namespace std;

struct trans {

    int vertex_from;

    int vertex_to;

    char trans_symbol;

};

class NFA {

public:

    vector<int> vertex;

    vector<trans> transitions;

    int final_state;

    NFA() {
```

```
}

int get_vertex_count() {

    return vertex.size();

}

void set_vertex(int no_vertex) {

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

        vertex.push_back(i);

    }

}

void set_transition(int vertex_from, int vertex_to, char trans_symbol) {

    trans new_trans;

    new_trans.vertex_from = vertex_from;

    new_trans.vertex_to = vertex_to;
```

```
    new_trans.trans_symbol = trans_symbol;

    transitions.push_back(new_trans);

}

void set_final_state(int fs) {

    final_state = fs;

}

int get_final_state() {

    return final_state;

}

void display() {

    trans new_trans;

    cout<<"\n";

    for(unsigned int i = 0; i < transitions.size(); i++) {
```

```
            new_trans = transitions.at(i);

            cout<<"q"<<new_trans.vertex_from<<" --> q"<<new_trans.vertex_to<<" :
Symbol - "<<new_trans.trans_symbol<<endl;

        }

        cout<<"\nThe final state is q"<<get_final_state()<<endl;

    }

    set<int> epsilon_closure(int index){

        set<int> myclous;

        myclous.insert(index);

        for(vector<trans> ::iterator it = transitions.begin(); it!=transitions.end(); ++it){

            if(it->vertex_from == index && it->trans_symbol == '^'){

                myclous.insert(it->vertex_to);

                set<int>temps = this->epsilon_closure(it->vertex_to);
```

```
            for(set<int>::iterator itt = temps.begin();itt!=temps.end();++itt){

                myclous.insert(*itt);

            }

        }

    }

    return myclous;

  }

};

class DFA {

public:

  int vertex;   // number of vertex

  vector<trans> transitions;

  vector<int> final_state;
```

```
void set_vertex(int count){

    vertex = count;

}

int get_vertex_count(){

    return vertex;

}

void set_final_state(int state){

    final_state.push_back(state);

}

bool is_final(int state){

    for(vector<int>::iterator it = final_state.begin();it!=final_state.end();++it){

        if(state == *it){

            return true;
```

```
        }

    }

    return false;

}

void set_transition(int vertex_from,int vertex_to,char trans_symbol){

    trans new_trans;

    new_trans.vertex_from = vertex_from;

    new_trans.vertex_to = vertex_to;

    new_trans.trans_symbol = trans_symbol;

    transitions.push_back(new_trans);

}

void display() {

    trans new_trans;
```

```
        cout<<"\n";

        for(unsigned int i = 0; i < transitions.size(); i++) {

            new_trans = transitions.at(i);

            cout<<"q"<<new_trans.vertex_from<<" --> q"<<new_trans.vertex_to<<" :
Symbol - "<<new_trans.trans_symbol<<endl;

        }

        // also print here final states

        cout<<"final states of dfa are"<<endl;

        for(vector<int>::iterator ift = final_state.begin();ift!=final_state.end();ift++){

            cout<<*ift<<" ";

        }cout<<endl;

    }

};
```

```
DFA nfa_to_dfa(NFA nfa){

    DFA dfa;

    int i,stateCounter,curState;

    stateCounter = curState = 0;

    map<set<int>,int >found_mapping;  // map to record sets found till now

    queue<set<int> > todo;            //queue to maintain all the set yet to be proceed

    set<int> a_set;

    set<int> b_set;

    set<int> a_epsilon;

    set<int> b_epsilon;

    set<int> lol;

    bool isdone[20000];

    for(i=0;i<20000;i++)
```

```
    isdone[i] = false;

set<int>temps = nfa.epsilon_closure(0);

//todoSet.push_back(temps);

found_mapping.insert(make_pair(temps,curState));

todo.push(temps);

while(!todo.empty()){

    temps.clear();

    a_set.clear();

    b_set.clear();

    a_epsilon.clear();

    b_epsilon.clear();

    lol.clear();

    temps = todo.front();
```

```
        curState = found_mapping[temps];

        todo.pop();

        if(isdone[curState]==true)continue;

        isdone[curState] = true;

        // finding a_set and b_set

         for(vector<trans>::iterator its =
(nfa.transitions).begin();its!=(nfa.transitions).end();++its){

            if(temps.find(its->vertex_from)!=temps.end()){

                if(its->trans_symbol == 'a'){

                    a_set.insert(its->vertex_to);

                }else if(its->trans_symbol == 'b'){

                    b_set.insert(its->vertex_to);

                }
```

```
    }

}

//set dfa transitions

if(a_set.empty()){

    dfa.set_transition(curState,-1,'a');

} else{

    // find clousure of all the states in set

    for(set<int>::iterator aits = a_set.begin();aits!=a_set.end();aits++){

        lol.clear();

        lol = nfa.epsilon_closure(*aits);

        for(set<int>::iterator ilol = lol.begin(); ilol!=lol.end();++ilol){

            a_epsilon.insert(*ilol);

        }
```

```
        }

        //HERE ALSO CHECK WHETHER THE STATE IS ALREADY PRESENT OR
NOT THEN INSERT IN VECTOR MAPPING

        if(found_mapping.find(a_epsilon)==found_mapping.end()){

            found_mapping.insert(make_pair(a_epsilon,++stateCounter));

        }

        todo.push(a_epsilon);

        dfa.set_transition(curState,found_mapping[a_epsilon],'a');

    }

    //INSERT EPSILON CLOSURE FOR A_sET AND B_sET

    if(b_set.empty()){

        dfa.set_transition(curState,-1,'b');

    } else{
```

```
        for(set<int>::iterator bits = b_set.begin();bits!=b_set.end();bits++){

            lol.clear();

            lol = nfa.epsilon_closure(*bits);

            for(set<int>::iterator ilols = lol.begin(); ilols!=lol.end();++ilols){

                b_epsilon.insert(*ilols);

            }

        }

        //HERE ALSO CHECK WHETHER THE STATE IS ALREADY PRESENT OR
NOT THEN INSERT IN VECTOR MAPPING

        if(found_mapping.find(b_epsilon)==found_mapping.end()){

            found_mapping.insert(make_pair(b_epsilon,++stateCounter));

        }

        todo.push(b_epsilon);
```

```
            dfa.set_transition(curState,found_mapping[b_epsilon],'b');

        }

        //check for final state  temps is current set, check whether final state of johnson
algo is there ?

        if(temps.find(nfa.get_final_state())!=temps.end()){

            dfa.set_final_state(curState);

        }

    }

    int total_vertex = found_mapping.size();

    cout << total_vertex << "\n";

    int flag1=0;

    for(vector<struct trans>::iterator ajeeb =
dfa.transitions.begin();ajeeb!=dfa.transitions.end();ajeeb++){
```

```
        if(ajeeb->vertex_to == -1){

            ajeeb->vertex_to=total_vertex;

            flag1=1;

        }

    }

    if(flag1==1){

        dfa.set_transition(total_vertex,total_vertex,'a');

        dfa.set_transition(total_vertex,total_vertex,'b');

    }

    if(flag1==1) dfa.set_vertex(total_vertex+1);

    else dfa.set_vertex(total_vertex);

    return dfa;

}
```

```
NFA concatenation(NFA a, NFA b) {

    NFA result;

    result.set_vertex(a.get_vertex_count() + b.get_vertex_count());

    unsigned int i;

    trans new_trans;

    for(i = 0; i < a.transitions.size(); i++) {

        new_trans = a.transitions.at(i);

        result.set_transition(new_trans.vertex_from, new_trans.vertex_to,
new_trans.trans_symbol);

    }

    result.set_transition(a.get_final_state(), a.get_vertex_count(), '^');

    for(i = 0; i < b.transitions.size(); i++) {

        new_trans = b.transitions.at(i);
```

```
        result.set_transition(new_trans.vertex_from + a.get_vertex_count(),
new_trans.vertex_to + a.get_vertex_count(), new_trans.trans_symbol);
    }
    result.set_final_state(a.get_vertex_count() + b.get_vertex_count() - 1);
    return result;
}
NFA kleene_star(NFA a) {
    NFA result;
    unsigned int i;
    trans new_trans;
    result.set_vertex(a.get_vertex_count() + 2);
    result.set_transition(0, 1, '^');
    for(i = 0; i < a.transitions.size(); i++) {
```

```
        new_trans = a.transitions.at(i);

        result.set_transition(new_trans.vertex_from + 1, new_trans.vertex_to + 1,
new_trans.trans_symbol);

    }

    result.set_transition(0, a.get_vertex_count() + 1, '^');

    result.set_transition(a.get_vertex_count(), a.get_vertex_count() + 1, '^');

    result.set_transition(a.get_vertex_count(), 1, '^');

    result.set_final_state(a.get_vertex_count() + 1);

    return result;

}

NFA or_function(NFA n1, NFA n2) {

    NFA result;

    int vertex_count = 2;
```

```
int i;

unsigned int j;

NFA med;

trans new_trans;

vertex_count += n1.get_vertex_count();

vertex_count += n2.get_vertex_count();

result.set_vertex(vertex_count);

int adder_track = 1;

for(i = 0; i < 2; i++) {

    result.set_transition(0, adder_track, '^');

    med = i==0?n1:n2;

    for(j = 0; j < med.transitions.size(); j++) {

        new_trans = med.transitions.at(j);
```

```
            result.set_transition(new_trans.vertex_from + adder_track, new_trans.vertex_to
+ adder_track, new_trans.trans_symbol);
        }
        adder_track += med.get_vertex_count();
        result.set_transition(adder_track - 1, vertex_count - 1, '^');
    }
    result.set_final_state(vertex_count - 1);
    return result;
}
string infix_to_postfix(string re){
    stack<char> operators;
    stack<string> variables;
    int reg_l = re.length();
```

```
for(int i=0;i<reg_l;i++){

    string tem="";

    tem+=re[i];

    if(re[i]=='('){

        operators.push(re[i]);

        variables.push(tem);

    }

    else if(re[i]==')'){

        while(!operators.empty() && operators.top()!='('){

            string t1 = variables.top();

            variables.pop();

            string t2 = variables.top();

            variables.pop();
```

```
    char op = operators.top();

    operators.pop();

    variables.push(t2+t1+op);

}

string temp = variables.top();

variables.pop();

operators.pop();

if(!variables.empty() && variables.top()!="(") temp+='&';

while(!variables.empty() && variables.top()!="("){

    string temp2 = variables.top();

    variables.pop();

    if(variables.top()!="(") temp=temp2+'&'+temp;

    else temp=temp2+temp;
```

```
    }

    variables.pop();

    variables.push(temp);

}

else if(re[i]=='*'){

    string temp = variables.top();

    variables.pop();

    temp+='*';

    variables.push(temp);

}

else if(re[i]=='|') operators.push(re[i]);

else{

    if(!operators.empty() && operators.top()!='(' && !variables.empty()){
```

```
            string temp = variables.top();

            variables.pop();

            temp+=re[i];

            temp+=operators.top();

            operators.pop();

            variables.push(temp);

        }

        else variables.push(tem);

    }

}

if(!operators.empty()){

    while(!operators.empty()){

        string t1 = variables.top();
```

```
        variables.pop();

        string t2 = variables.top();

        variables.pop();

        char op = operators.top();

        operators.pop();

        variables.push(t2+t1+op);

    }

}

re=variables.top();

variables.pop();

if(!variables.empty()) re+='&';

while(!variables.empty()){

    string temp2=variables.top();
```

```
        variables.pop();

        if(!variables.empty()) re = temp2+'&'+re;

        else re = temp2 + re;

    }

    reg_l = re.length();

    cout << re << "\n";

    return re;

}

NFA re_to_nfa(string re) {

    stack<char> operators;

    stack<NFA> operands;

    char cur_sym;

    NFA *new_sym;
```

```
    for(string::iterator it = re.begin(); it != re.end(); ++it) {

        cur_sym = *it;

        if(cur_sym != '(' && cur_sym != ')' && cur_sym != '*' && cur_sym != '|' &&
cur_sym != '.' && cur_sym != '&') {

            new_sym = new NFA();

            new_sym->set_vertex(2);

            new_sym->set_transition(0, 1, cur_sym);

            new_sym->set_final_state(1);

            operands.push(*new_sym);

            delete new_sym;

        } else {

            if(cur_sym == '*') {

                NFA star_sym = operands.top();
```

```
        operands.pop();

        operands.push(kleene_star(star_sym));

    } else if(cur_sym == '.' || cur_sym == '&') {

        NFA op1,op2;

        op2 = operands.top();

        operands.pop();

        op1 = operands.top();

        operands.pop();

        operands.push(concatenation(op1, op2));

    } else if(cur_sym == '|') {

        vector<NFA> selections;

        NFA n2=operands.top();

        operands.pop();
```

```
                NFA n1=operands.top();

                operands.pop();

                operands.push(or_function(n1,n2));

            }

        }

    }

    return operands.top();

}

//minimization using myhill nerode theorem

DFA dfa_minimize(DFA dfa){

    int states,i,j,iatrans,ibtrans,jatrans,jbtrans;  // iatrans = $(i,a) ; ibtrans = $(i,b)

    states = dfa.get_vertex_count();
```

```
    bool change = false; // to check whether is there any change in table if not then stop
there

    bool isfalse;  // to check whether table entry already checked or not

    DFA m_dfa;  // final minimized dfa

    bool table[states][states];

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

        for(j=0;j<states;j++){

            table[i][j] = true;

        }

    }

    //first loop for ticking

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

        for(j=0;j<states;j++){
```

```
            if(i<=j){

                continue;   // only lower half table needed to be filled

            } else{          // chceck whether one of i and j found in final state

                if( ( (find(dfa.final_state.begin(),dfa.final_state.end(),i)!=dfa.final_state.end())
&& (find(dfa.final_state.begin(),dfa.final_state.end(),j)==dfa.final_state.end())) ||

                    ( (find(dfa.final_state.begin(),dfa.final_state.end(),j)!=dfa.final_state.end())
&& (find(dfa.final_state.begin(),dfa.final_state.end(),i)==dfa.final_state.end()))

                ){

                    table[i][j] = false;

                }

            }

        }

    }
```

```
    // loop until there is no change in table

    do{

        change = false;

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

            for(j=0;j<states;j++){

                if(i<=j || table[i][j]==false){

                    continue;

                }else{

                    //iterate over all transitions and find i and j's transitions

                    //if pair goes to such state on a or b which is having false in table

                    for(vector<trans>::iterator its =
(dfa.transitions).begin();its!=dfa.transitions.end();its++){

                        if(its->vertex_from == i){
```

```
                if(its->trans_symbol == 'a')

                    iatrans = its->vertex_to;

                else if(its->trans_symbol == 'b')

                    ibtrans = its->vertex_to;

            }else if(its->vertex_from == j){

                if(its->trans_symbol == 'a')

                    jatrans = its->vertex_to;

                else if(its->trans_symbol == 'b')

                    jbtrans = its->vertex_to;

            }

        }   // for pair i j found the a and b transitions

        // FOR SOME STATE I IF THERE IS NO TRANSITION THEN WE ARE
SETTING THAT TRANSITION IN -1 STATE
```

```
                //     check whether any trans lead to FALSE entry in table

                if((iatrans!=-1 && jatrans!=-1) && iatrans!=jatrans){

                   isfalse = iatrans > jatrans ? table[iatrans][jatrans]:table[jatrans][iatrans];

                   if(!isfalse) {

                      change = true;

                      table[i][j] = false;

                   }

                   // if any one (i or j)is -1 and other is final state then set table enrty
FALSE

                }else if((iatrans==-1 && jatrans!=-1 &&
(find(dfa.final_state.begin(),dfa.final_state.end(),jatrans)!=dfa.final_state.end()))||

                   (jatrans==-1 && iatrans!=-1 &&
(find(dfa.final_state.begin(),dfa.final_state.end(),iatrans)!=dfa.final_state.end()))){
```

```
                change = true;

            table[i][j] = false;

        }

        if((ibtrans!=-1 && jbtrans!=-1) && ibtrans!=jbtrans){

            isfalse = ibtrans > jbtrans ? table[ibtrans][jbtrans]:table[jbtrans][ibtrans];

            if(!isfalse) {

                change = true;

                table[i][j] = false;

            }

        }else if((ibtrans==-1 && jbtrans!=-1 &&
(find(dfa.final_state.begin(),dfa.final_state.end(),jbtrans)!=dfa.final_state.end()))||

                (jbtrans==-1 && ibtrans!=-1 &&
(find(dfa.final_state.begin(),dfa.final_state.end(),ibtrans)!=dfa.final_state.end()))){
```

```
                    change = true;

                table[i][j] = false;

            }

        } // end of i>j else

    } // end of j loop

}  // end of i loop

}while(change);

cout<<"FINAL MINIMIZED TABLE IS "<<endl;

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

    for(j=0;j<states;j++){

        if(i<=j)cout<<"- ";

        else cout<<table[i][j]<<" ";

    }
```

```
    cout<<endl;

}

//make sets and maps them to new states

bool checked[states];

map<set<int>,int>final_mapping;  // merged set will be mapped to new given states

int curCount=0; // counter for new merged set to map a new state

set<int>aisehi,temps;

map<set<int>,int>::iterator to_erase;

set<int>::iterator sit;

for(i=0;i<states;i++) checked[i] = false;  // to check all states are checked or not

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

    if(checked[i]==false){

        aisehi.clear();
```

```
            aisehi.insert(i);

        }

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

            if(table[j][i]==1 && checked[i]==false){

                aisehi.insert(j);

                checked[j] = true;

            }

        }

        if(checked[i]==false)final_mapping[aisehi] = curCount++;

        checked[i] = true;

    }

    m_dfa.set_vertex(curCount);

    set<int>::iterator ist;
```

```
    int atra,btra;

    for(map<set<int>,int>::iterator ift = final_mapping.begin(); ift!=final_mapping.end();
ift++){

        // set final states

        for(ist = ift->first.begin(); ist!=ift->first.end();ist++){

            if(find(dfa.final_state.begin(),dfa.final_state.end(),*ist)!=dfa.final_state.end()){

                m_dfa.set_final_state(ift->second);

                break;  //necessary otherwise multiple copy in final state vector

            }

        }

        // now set transitions

        ist = (ift->first).begin();

        //iterate over dfa.transitions and find the a and b transition state
```

```
        for(vector<trans>::iterator itr = dfa.transitions.begin(); itr!=dfa.transitions.end();
itr++){

            if(itr->vertex_from == *ist && itr->trans_symbol == 'a'){

                atra = itr->vertex_to;

            }

            if(itr->vertex_from == *ist && itr->trans_symbol == 'b'){

                btra = itr->vertex_to;

            }

        }

        if(atra==-1){

            m_dfa.set_transition(ift->second,-1,'a');

        }

        if(btra==-1){
```

```
            m_dfa.set_transition(ift->second,-1,'b');

        }

        // now check in which set atra and btra are and accordingly set the transition of
final dfa

        for(map<set<int>,int>::iterator ifft =
final_mapping.begin();ifft!=final_mapping.end();ifft++){

            for(set<int>::iterator isst = ifft->first.begin();isst!=ifft->first.end();isst++){

                if(atra!=-1 && *isst == atra ){

                    m_dfa.set_transition(ift->second,ifft->second,'a');

                }

                if(btra!=-1 && *isst == btra){

                    m_dfa.set_transition(ift->second,ifft->second,'b');

                }
```

```
            }

        }

    }

    m_dfa.display();

    return m_dfa;

}

bool simulate(DFA dfa, string s){

    unsigned int i;

    int curstate=0;

    if(s!="^"){

        for(i=0;i<s.length();i++){

            for(vector<trans>::iterator its =

dfa.transitions.begin();its!=dfa.transitions.end();its++){
```

```
            if(its->vertex_from == curstate && its->trans_symbol == s[i] && its->vertex_to
== -1)return false;

            else if(its->vertex_from == curstate && its->trans_symbol == s[i]){

                curstate = its->vertex_to;

                break;

            }

        }

    }

    // now check cur state if final or not

    for(vector<int>::iterator it = dfa.final_state.begin(); it!=dfa.final_state.end();it++){

        if(*it == curstate)return true;

    }
```

```
    return false;

}

int main() {

    string re;

    cout<<"\n\nEnter the regular expression - \n\n";

    cin>>re;

    cout<<"------The postfix expression for the regex : -----"<<endl;

    re = infix_to_postfix(re);

    cout<<"------The required NFA has the transitions : -----"<<endl;

    NFA required_nfa;

    required_nfa = re_to_nfa(re);

    required_nfa.display();

    string to_simulate;
```

```
cout<<"--------------------------------------"<<endl;

cout<<"now nfa->dfa will be shown"<<endl;

DFA required_dfa = nfa_to_dfa(required_nfa);

required_dfa.display();

cout<<"--------------------------------------"<<endl;

cout<<"here goes minimization table for required dfa"<<endl;

DFA tempdfa = dfa_minimize(required_dfa);

cout<<"enter the strings to simulate and press q to quit"<<endl;

cin>>to_simulate;

while(to_simulate!="q"){

    bool flag = false;

    if(to_simulate!="^")

        for(unsigned int l = 0; l<to_simulate.length();l++)
```

```
            if(to_simulate[l]!='a' && to_simulate[l]!='b'){

                flag=true;

                break;

            }

        if(flag){

            cout << "rejected\n";

            cin>> to_simulate;

            continue;

        }

        simulate(tempdfa,to_simulate) == true ? cout<<"accepted"<<endl :
cout<<"rejected"<<endl;

        cin>>to_simulate;

    }
```

```
    cout<<"----------------------------------BYE BYE------------------------------"<<endl;


    return 0;


}
```