# Al-Imam Muhammad Bin Saud Islamic University

## College of Computer and Information Sciences,

## Department of Computer Science

---

# Compiler project

## By:

| Students names | ID |
|---|---|
| | |
| | |
| | |
| Hanan AlMutairi | |
| | |

**Course: CS445**

**Section: 373**

**Instruction:**

**Submission Date: 05-11-2022**

# TABLE OF CONTENT




# TABLE OF CONTENT

# Part (1)

## Introduction

Our project is builds an NFA from a given RE, converts a giving NFA into DFA and build a DFA from a given RE directly.

We will further discuss the terms used and the working of the project.

**Finite Automata.[1]**

A finite automaton is a collection of 5-tuples (Q, ∑, δ, q0, F), where:

Q: a finite set of states

∑: a finite set of the input symbol

q0: initial state

F: final state

δ: Transition function

 Finite automata are used to recognize patterns.

 It takes the string of symbols as input and changes its state accordingly. When the

The desired symbol is found, then the transition occurs.

 At the time of transition, the automata can either move to the next state or stay in

the same state.

 Finite automata have two states, accept state or reject state. When the input string

is processed successfully, and the automata have reached their final state, then they will

accept.

**Types of Finite Automata**

There are two types of Finite Automata:

1. DFA (Deterministic Finite Automata)

2. NFA (Non-Deterministic Finite Automata)

## Non-Deterministic Finite Automata (NFA)

- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than
- DFA for a given regular language.
- The finite automata are called NFA when there exist many paths for specific input
- from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated into DFA.
- NFA is defined in the same way as DFA but with the following two exceptions, it
- contains multiple next states, and it contains ε transition.

## Deterministic Finite Automata (DFA)

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness
- of the computation. The finite automata are called deterministic finite automata if
- the machine is read an input string one symbol at a time.
- In DFA, there is only one path for specific input from the current state to the next
- state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any
- input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.
- In the following diagram, we can see that from state q0 for input a, there is only one
- path that is going to q1. Similarly, from q0, there is only one path for input b going
- to q2.

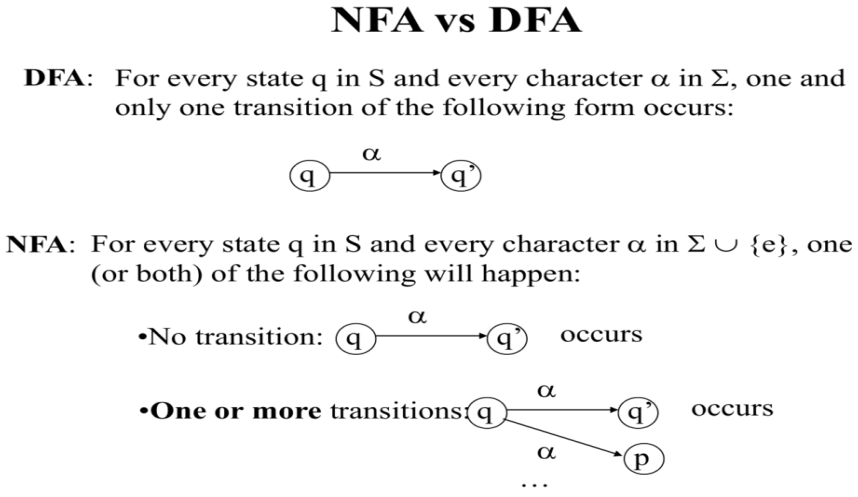



***Figure 1:finite automata***

## Regular expression to NFA:

A Regular Expression is a representation of Tokens. But, to recognize a token, it can need a token Recognizer, which is nothing but a Finite Automata (NFA). So, it can convert Regular Expression into NFA.[2]

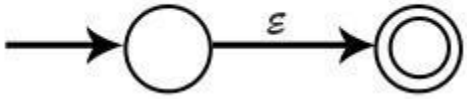
### Algorithm for the conversion of Regular Expression to NFA

**Input** − A Regular Expression R

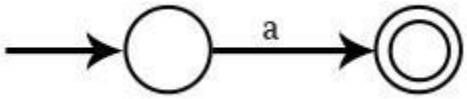
**Output** − NFA accepting language denoted by R

**Method**

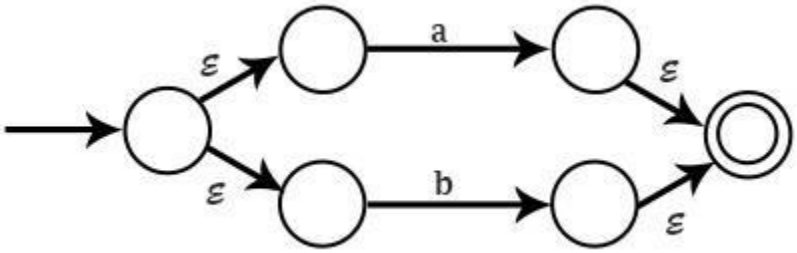
For ε, NFA is



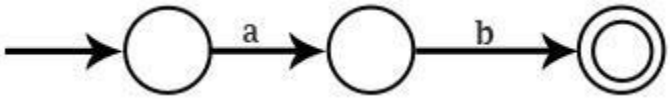

For a NFA is




For a + b, or a | b NFA is




For ab, NFA is




For a*, NFA is

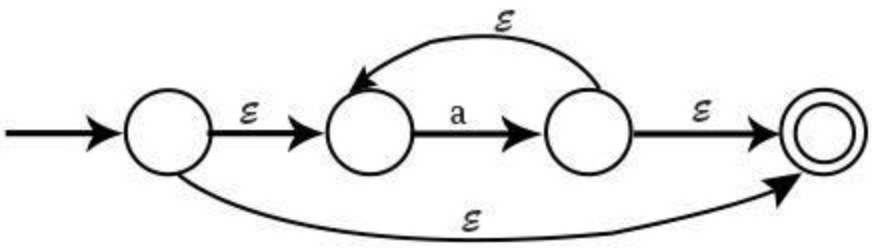



*Figure 2: the conversion of Regular Expression to NFA*

## The programming language used:

We used C++ language to apply the first part of the project for several reasons:

- We looked for code to build the project on, and the first good code we found was in C++[3].
- Since we study this language during the summer vacation, we wanted to apply it to this project.
- C++ is fast because it is translated directly into machine language without an intermediary translation required at runtim.

## Problems and solutions

1- The difficulty to deal with pointers in C++ language, where it was showing an unknown error when representing the left and right of the node to build the parser tree

Solution: We had to use rai to build the tree and represent the node correctly

2- Representation of the concatenation between the input symbols

Solution :We used the preprocessor method, which returns the string after adding the dots to represent the concatenation between the input signals

## Our implementation:

We first explain our implementation using the figure1 below,

the regular expression is received from the user and then sent to the preprocessor() method, which returns string.

We added to this string the '#' symbol and sent to the preprocessor() method again.

the string entered as an input to the postfix() method, which also returns string.

Then it is sent to reg_nfa() method, which returns an integer representing the number of stattes after converting the regular expression to NFA

This number is used to print an matrix that represents the NFA table and finally the initial and final states are printed.

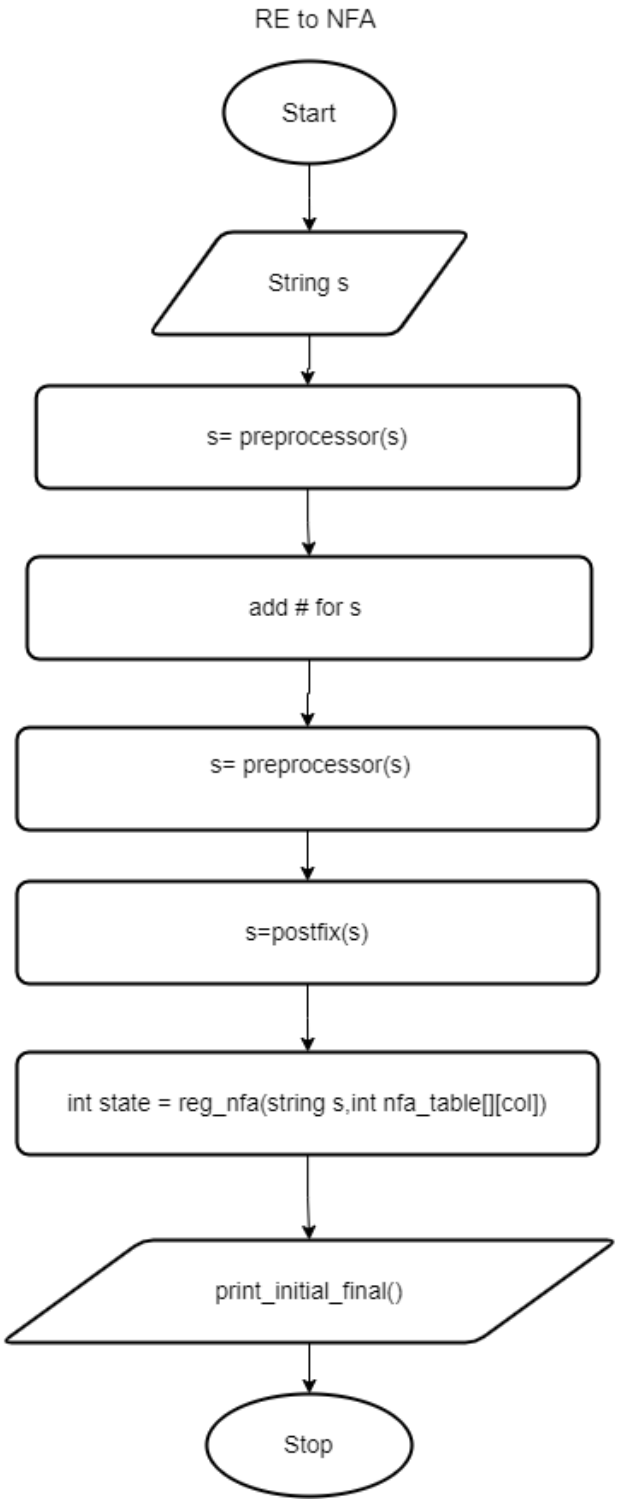



***Figure 3:illustrative Diagram of the conversion RE to NFA***

## Code:

### Preprocessor():

The expression is made simpler by the preprocessor function, which adds a dot between each letter to signify a concatenation operation and parentheses for priority. starting by adding open parenthesis, inserting the char of the input string in x[j], checking letters or symbols, if it is letters then putting a dot, close the parentheses we opened earlier , reformat the string to be with dot and parentheses

```
string preprocessor(string s){
    char x[5000];
    int l=s.length();
    int j=0;
    x[j]='(';
    j += 1;
    for(int i=0;i<l;i++){
            x[j]=s[i];
            j += 1;
        if(((s[i]>=97&&s[i]<=122)||s[i]==35)&&((s[i+1]>=97&&s[i+1]<=122)||s[i+1]==35)){/*checking if s[i] is a letter or operation if it letter then but d
            x[j]='.';
            j += 1;
        }else if(s[i]==')'&&s[i+1]=='('){
            x[j]='.';
            j += 1;
        }else if(((s[i]>=97&&s[i]<=122)||s[i]==35)&&s[i+1]=='('){
            x[j]='.';
            j += 1;
        }else if(s[i]==')'&&(s[i+1]>=97&&s[i+1]<=122||s[i+1]==35)){
            x[j]='.';
            j += 1;
        }else if(s[i]=='*'&&(s[i+1]=='(' || ((s[i+1]>=97&&s[i+1]<=122)||s[i+1]==35))){
            x[j]='.';
            j += 1;
        }
    }
    x[j] = ')';
    j += 1;
    string p;
    for(int i=0;i<j;i++)
       p += x[i];
    return p;
}
```

***Figure 4: Preprocessor method***

**Postfix():**

This method transforms the regular expression that has already been processed into the corresponding postfix notation and return postfix string.

```
string postfix(string s){
    int l=s.length();
    char a[5000];
    stack <char> ch;
    int j=0;
    for(int i=0;i<l;i++){
        char x = s[i];
        switch(x){
            case 'a':   a[j]='a';
                        j += 1;
                        break;
            case 'b':   a[j]='b';
                        j+=1;
                        break;
            case '#':   a[j]='#';
                        j+=1;
                        break;
            case '(':   ch.push('(');
                        break;
            case ')':   while(!ch.empty()){
                            if(ch.top()=='('){
                                ch.pop();
                                break;
                            }else{
                                a[j]=ch.top();
                                    ch.pop();
                                    j += 1;
                            }
                        }
            case '.':   if(ch.empty()){
                            ch.push('.');
                        }else {
                            char temp = ch.top();
                            if(temp=='('
                                ch.push('.');
                            else if(temp=='*'){
                                a[j]=ch.top();
                                ch.pop();
                                j += 1;
                                if(ch.top()== '.'){
                                    a[j] = '.';
                                    j += 1;
                                }else{
                                    ch.push('.');
                                }
                            }else if(temp=='.'){
                                a[j]=ch.top();
                                ch.pop();
                                j += 1;
                                ch.push('.');
                            }else if(temp == '|'){
                                ch.push('.');
                            }
                        }
                        break;
            case '|':   if(ch.empty()){
                            ch.push('|');
                        }else{
                            char temp = ch.top();
                            if(temp == '('
                                ch.push('|');
                            else if(temp == '*'){
                                a[j] = ch.top();
                                ch.pop();
                                j += 1;
                                ch.push('|');
                            }else if(temp == '.'){
                                a[j] = ch.top();
                                j += 1;
                                ch.pop();
                                ch.push('|');
                            }
            case '*':   if(ch.empty()){
                            ch.push('*');
                        }else{
                            char temp = ch.top();
                            if(temp == '(' || temp == '.' || temp == '|' )
                                ch.push('*');
                            else{
                                a[j] = ch.top();
                                ch.pop();
                                j += 1;
                                ch.push('*');
                            }
                        }
                        break;
        }
    }
    string p;
    for(int i=0;i<j;i++){
        p += a[i];
    }
    return p;
}
```

***Figure 5:Postfix method code***

**reg_nfa():**

This function[3] converts the given postfix of a regular expression into its corresponding NFA, where it receives the postfix string and fills the NFA table, which is a matrix with the states as rows and five-column represent the state, a, b, epsilon, epsilon respectively

```
int reg_nfa(string s,int nfa_table[][col]){ // receive the postfix strign and  table as matrix with five columns representing states, a, b, epsilon ,epsilo
    int l = s.length();
    int states = 1;
    int m,n,j,counter;
    for(int i=0;i<l;i++){
        char x = s[i];  // traverse over postfix string char by car
        switch(x){
            case 'a': nfa_table[states][0] = states;     // in case x is a or b then initiate the state(state=1) in nfa_table and add it to array init whic
                      init[a] = states;
                       a += 1;
                          states += 1;
                     nfa_table[states-1][1] = states; // link the initial state to the final state(state=2) of the input string and then add the final sta
                       fin[b] = states;
                       b += 1;
                     nfa_table[states][0] = states; // initiate the new state(state=2) to the nfa_table
                          states +=1;
                     break;
            case 'b': nfa_table[states][0] = states; // the same process as a
                       init[a] = states;
                        a += 1;
                          states += 1;
                     nfa_table[states-1][2] = states;
                        fin[b] = states;
                        b += 1;
                     nfa_table[states][0] = states;
                          states +=1;
                     break;
            case '.': m = fin[b-2]; // in case x is . then we first link the two operand between the dot by link the first final state in fin array and the
                      n = init[a-1]; //and the second initial state in array init
                      nfa_table[m][3]=n; // add the n to epsilon col in the m'th raw
                      reduce_fin(b-2);  //reduces final state , or delete the m since we link it with another state in the table
                      a -= 1;           //reduces initial states, or delete the n since we link it with another state in the table
                     break;
            case '|': for(j=a-1,counter=0;counter<2;counter++){ // in case x is | then we initiate an initial state that connects the initial states we in
                       m = init[j-counter];
                       nfa_table[states][3+counter]=m;// with epsilon as input for each of them
                      }
                      a=a-2; // after we connected the initial states now it's time to delete them from the init array
                      init[a]=states; // and add the new initial state to the array
                      a += 1;
                      nfa_table[states][0] = states; // add the new state to the table
                      states += 1;
                      //We now connect the final states that were previously linked
                      // to the initial states with the new states to become the current final states
                      for(j=b-1,counter=0;counter<2;counter++){
                        m = fin[j-counter];
                        nfa_table[m][3]=states;
                      }
                      b=b-2; // after we connected the final states now it's time to delete them from the fin array
                      fin[b]=states; // and add the new final satate to array
                      b += 1;
                      nfa_table[states][0] = states; // add the new state to the table
                      states += 1;
                      break;
            case '*': m = init[a-1]; // in case x is * then first we take the current initial state and connecte with last state added which the current f
                      nfa_table[states][3] = m; // initial state connected to the final with epsilon
                      nfa_table[states][0] = states; // add the current final state to the table
                      init[a-1] = states; //after we connected the initial states now it's time to delete them from the init array

                      states += 1;
                      n = fin[b-1]; // now we take the current final and connecting it in reverse dirction to current initial state
                      nfa_table[n][3]=m; // with epsilon
                      nfa_table[n][4]=states; // then the current final state is connect to last state which is the final now
                      nfa_table[states-1][4]=states;
                      fin[b-1]=states;
                      nfa_table[states][0]=states;
                      states += 1;
                        break;
        }
    }
```

***Figure 6:Regular Expression to NFA method code***

## NFA to DFA:

An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.[4]

**Conversion from NFA to DFA**

Suppose there is an NFA $N < Q, \sum, q0, \delta, F >$ which recognizes a language L. Then the DFA $D < Q', \sum, q0, \delta', F' >$ can be constructed for language L as:

Step 1: Initially $Q' = \phi$.

Step 2: Add q0 to Q'.

Step 3: For each state in Q', find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q', add it to Q'.

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

## Our implementation:

We explain our implementation using the figure2 below,

the states, NFA table, and DFA table are received and then sent to the nfa_dfa() method

Then it is sent to print_dfa_table() and finally, the initial and final states are printed.

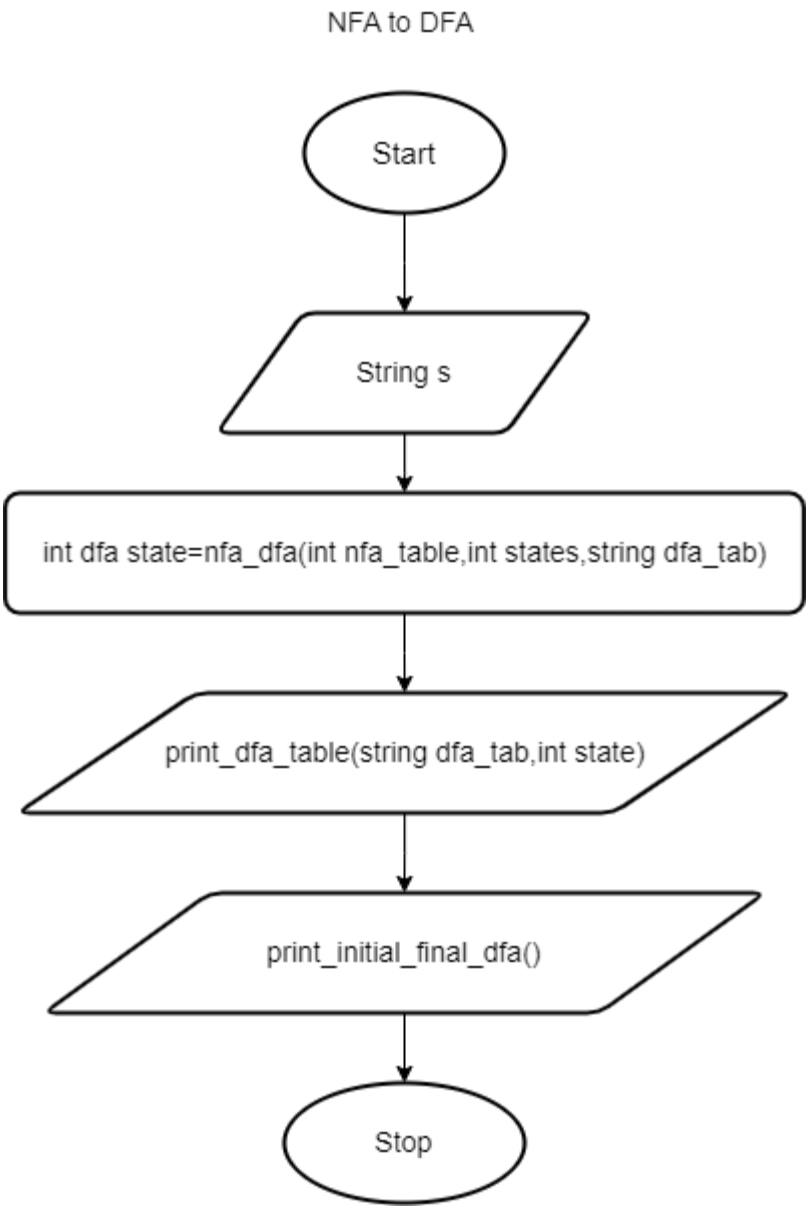



***Figure 7:illustrative Diagram of the conversion NFA to DFA***

**Code**:

**check_init():**

find the initial state for DFA

**check_fin():**

find final state for DFA

**nfa_dfa():**

```
int nfa_dfa(int nfa_table[][col],int states,string dfa_tab[][3]){
    bool flag[states];
void check_init(vector <int> v, string s){
    for(int i=0;i<v.size();i++){
        if(v[i] == init[0]){
```

*Figure 8: check the initial state and check the final state methods*

```
void check_fin(vector <int> v, string s){
    for(int i=0;i<v.size();i++){
        if(v[i] == fin[0]){
            fin_dfa[_b] = s;
            _b += 1;
        }
    }
}
    // transition of e-closure to over input symbol a
    int counter = 0;
    while(true){

      while(!st.empty()){
       vector <int> v ;
       v   = st.top();
            st.pop();
       counter += 1;
       dfa_tab[state][0] = map_state[v];       //find transition of a state over input symbol 'a' and 'b'

       for(int i=0;i<v.size();i++){
           flag[v[i]] = false;
           int temp = nfa_table[v[i]][1];      //over input symbol a
           int temp1 = nfa_table[v[i]][2];     //over input symbol b
           if(temp>=0)
               v1.push_back(temp);
           if(temp1>=0)
               v3.push_back(temp1);
       }
```

*Figure 9:conversion NFA to DFA method code*

first sure all states E-closure found then find the  e-closure for the initial state and mark the initial as visited, add links initial e-closure with a name then check current state is initial or not; check current state is final or not.

```
945
946     map <int,int> map_temp,map_temp1;  //to remove duplicate state
947     map <int,int> ::iterator it;
948
949     for(int i=0;i<v1.size();i++){
950         v2 = eclosure(nfa_table,v1[i]);
951         for(int j=0;j<v2.size();j++){
952             map_temp[v2[j]] = 1;
953         }
954         v2.clear();
955     }
956
957     for(int i=0;i<v3.size();i++){
958         v4 = eclosure(nfa_table,v3[i]);
959         for(int j=0;j<v4.size();j++){
960             map_temp1[v4[j]] = 1;
961         }
962         v4.clear();
```

***Figure 10:conversion NFA to DFA method code***

```
968     }
969
970     for(it = map_temp1.begin(); it != map_temp1.end(); it++){
971         v4.push_back(it->first);
972         flag[it->first] = false;
973     }
```

Map used to remove duplicate state traverse over each state that receive a as input

v2 find the e-closure of each state that receive a as input, add it to map to prevent duplicate states and marked as visited then push the states receives a as input to v2.

```
974
975     if(v2.empty()){
976         dfa_tab[state][1] = "--";
977     } else {
978         string t = map_state[v2];
979         char flag1 = t[0];
980         if( flag1 == 'q'){
981             dfa_tab[state][1] = map_state[v2];   //checking v2 has already been mapped or not
982         } else {
983             dfa_tab[state][1] = state_name(j++);
984             map_state[v2] = dfa_tab[state][1];
985             check_init(v2,map_state[v2]);
986             check_fin(v2,map_state[v2]);
987         /*-    cout<<endl<<map_state[v2]<<" represents :- ";
988             for(int i=0;i<v2.size();i++)
989                 cout<<v2[i]<<" ";
990             cout<<endl; */
991             st.push(v2);               //not mapped state will be pushed into stack
992         }
993     }
994
995     if(v4.empty()){
996         dfa_tab[state][2] = "--";
997     } else {
998         string t = map_state[v4];
999         char flag1 = t[0];
1000        if( flag1 == 'q'){
1001            dfa_tab[state][2] = map_state[v4];
1002        } else {
```

***Figure 11:conversion NFA to DFA method code***

```
1008            for(int i=0;i<v4.size();i++)
1009                cout<<v4[i]<<" ";
1010            cout<<endl; */
1011            st.push(v4);
1012        }
1013    }
```

if there is now state in v2 then a col is has no transition for a if there is not take the name of the state that receive a as input checking the state has already been mapped to other state which send a, and call print_dfa_table () to print DFA table

## Regular expression to DFA:

Is a Direct method used to convert given regular expression directly into DFA[5].

**Steps**:

- Uses augmented regular expression r#.
- Important states of NFA correspond to positions in regular expression that hold symbols of the alphabet.
- Regular expression is represented as syntax tree where interior nodes correspond to operators representing union, concatenation and closure operations.
- Leaf nodes corresponds to the input symbols
- Construct DFA directly from a regular expression by computing the functions nullable(n), firstpos(n), lastpos(n) andfollowpos(i) from the syntax tree.

- nullable (n): Is true for * node and node labeled with Ɛ. For other nodes it is false.
- firstpos (n): Set of positions at node ti that corresponds to the first symbol of the sub-expression rooted at n.
- lastpos (n): Set of positions at node ti that corresponds to the last symbol of the sub-expression rooted at n.
- followpos (i): Set of positions that follows given position by matching the first or last symbol of a string generated by sub-expression of the given regular expression.

| Node n | nullable *(n)* | firstpos *(n)* | lastpos *(n)* |
|---|---|---|---|
| A leaf labeled Ɛ | True | Ø | Ø |
| A leaf with position i | False | {i} | {i} |
| An *or* node *n* = c1 \| c2 | Nullable (c1 ) or Nullable (c2 ) | firstpos (c1) U firstpos (c2) | lastpos (c1) U lastpos (c2) |
| A cat node *n* = c1c2 | Nullable (c1 ) and Nullable (c2 ) | If (Nullable (c1 )) firstpos (c1) U firstpos (c2) else firstpos (c1) | If (Nullable (c2 )) lastpos (c1) U lastpos (c2) else lastpos (c1) |
| A star node *n* = c1* | True | firstpos (c1) | lastpos (c1) |

***Figure 12:Rules for computing nullable, firstpos and lastpos***

Computation of followpos:
The position of regular expression can follow another in the following ways:

- If n is a cat node with left child c1 and right child c2, then for every position i in lastpos(c1), all positions in firstpos(c2) are in followpos(i).
- For cat node, for each position i in lastpos of its left child, the firstpos of its right child will be in followpos(i).
- If n is a star node and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i).
- For star node, the firstpos of that node is in f ollowpos of all positions in lastpos of that node.

## Our implementation:

explain our implementation using the figure3 below,

the regular expression is received from the user and then sent to the preprocessor() method, which returns a string.

We added to this string the '#' symbol and sent to the preprocessor() method again.

the string entered as an input to the postfix() method, which also returns the string.

This string is sent to the treebu() method, to build a tree by initializing each symbol as a node from a class node.

then finding the position, nullable, follow position, first and the last position.

Then it is sent to DFA() to build a DFA table by starting from the start state that is saved in the followpos[1] by finding all the possible sets from 'a' and 'b' and printing DFA table

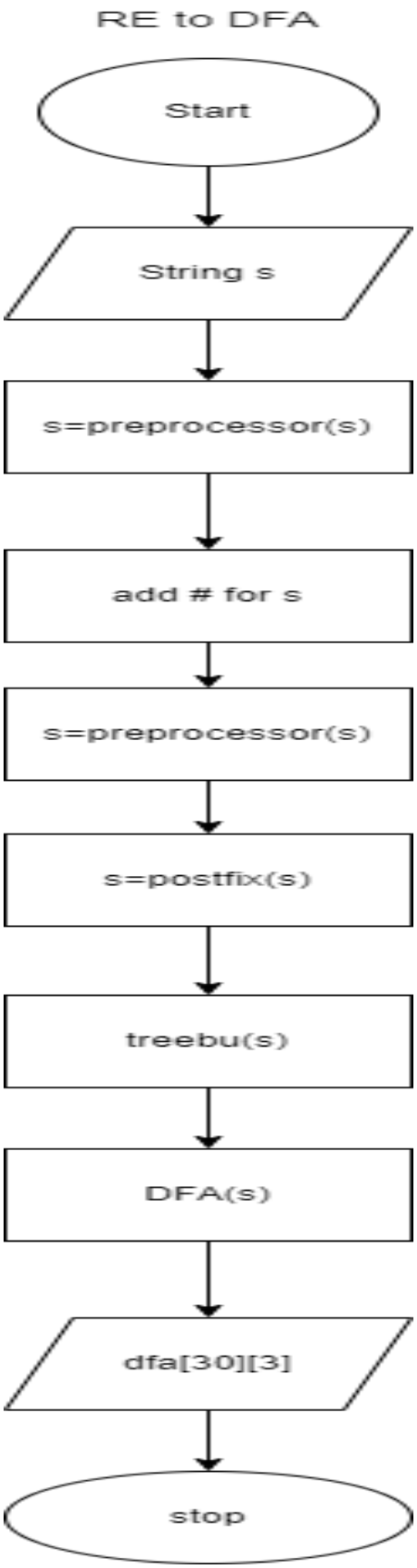



***Figure 13:conversion the RE to DFA***

## code:

### treebu():

function to bulid the symbol tree[6] for RE with '#'
The right child and the left child will be determined for each operator,
In the "." symbol, not in all cases, the left child is i-2. There are subtrees in these children, we use calculate in another way.
node_pos():function to find the position for each symbol
node_null(): function to find the nullable for each symbol
node_firstpos(): function to find the firstpos for each symbol
node_Laststpos(): function to find the lastpos for each symbol
node_followPos(): function to find the followpos for '.' symbol, and '*' symbol

```
507
508 void treebu(string s){
509 int l=s.length()-1;
510 for(int i=0;i<s.length();i++){
511     if(s[i]=='.'){
512
513         tree[i]=new Node(s[i]);
514         if(s[i-1]=='*'&&(s[i-2]=='a'||s[i-2]=='b')){
515            tree[i-3]=new Node(s[i-3]); //left
516            tree[i-1]=new Node(s[i-1]); //right
517         }else if(s[i-1]=='*'&&s[i-2]==')'){
518            tree[i-5]=new Node(s[i-5]); //left
519            tree[i-1]=new Node(s[i-1]); //right
520         }else{
521             tree[i-2]=new Node(s[i-2]); //left
522             tree[i-1]=new Node(s[i-1]); //right
523         }
524     }
525     else if(s[i]=='|'){
526         tree[i]=new Node(s[i]);
527         tree[i-2]=new Node(s[i-2]); //left
528         tree[i-1]=new Node(s[i-1]); //right
529         }
530     else if(s[i]=='*'){
531
532         tree[i]=new Node(s[i]);
533         tree[i-1]=new Node(s[i-1]); //left
534     }
535 }
536 node_pos();
537 node_null();
538 node_firstpos();
539 node_Laststpos();
540 node_followPos();
541
542 }
543
```

***Figure 14: Build tree method***

## DFA():

build a DFA table by starting from the start state that is saved in the followpos and print DFA table

initdfa(): calling this method to initiate the dfa list with null

arr: new list to save all the new states that will be calculated latter

for : to initiate the arr list with null
set: start set
counter=1 to save, how many sets do we have
new_set_a : variable to save the new set that found from generate_set_a
new_set_b: variable to save the new set that found from generate_set_b
ex_a: variable to save the set that generated the new set that found from generate_set_a
ex_b : variable to save the set that generated the new set that found from generate_set_b

```
int DFA(string s){
    initdfa();

    string arr[30][2];
    for(int m =0; m<30; m++){
    arr[m][0]="null";
    arr[m][1]="null";
    }
    string set=followpos[1];
    arr[0][0]="states";
    arr[0][1]="sets";
    arr[1][0]='A';
    arr[1][1]=set;
    int i=1;

    int counter=1;
    while(arr[i][1]!="null"){
        string new_set_a;
        string new_set_b;
        string ex_a="null";
        string ex_b="null";
        new_set_a= generate_set_a(arr[i][1]);
        new_set_b= generate_set_b(arr[i][1]);
            for(int a=0; a<30; a++)
            {if(new_set_a==arr[a][1]){
              ex_a=arr[a][0];
            }if(new_set_b==arr[a][1]){
              ex_b=arr[a][0];
            }

            }
            if(ex_a!="null"){
                dfa[i][0]=arr[i][0][0];
                dfa[i][1]=ex_a[0];
            }
            else {
                counter++;
                arr[counter][0]=arr[counter-1][0][0]+1;
                arr[counter][1]=new_set_a;
                dfa[i][0]=arr[i][0][0];
                dfa[i][1]=arr[counter][0][0];
            }
            if(ex_b!="null"){
                dfa[i][0]=arr[i][0][0];
                dfa[i][2]=ex_b[0];
            }
            else{
                counter++;
                arr[counter][0]=arr[counter-1][0][0]+1;
                arr[counter][1]=new_set_b;
                dfa[i][0]=arr[i][0][0];
                dfa[i][2]=arr[counter][0][0];}

        i++;}

    dfa[1][0]='A';
    cout<<endl<<endl;
    for(int i=0;i<40;i++){
      cout<<"*";
      }cout<<endl<<endl;
    cout<<setw(30)<<"TRANSITION TABLE FOR DFA"<<endl<<endl;
    cout<<setw(10)<<"States"<<setw(10)<<"a"<<setw(10)<<"b"<<endl;
    for(int i=0;i<40;i++){
        cout<<"-";}
    cout<<endl;
    int n=1;
    while(dfa[n][2]!='0'){
        for(int j=0;j<3;j++){
        cout<<setw(10)<<dfa[n][j];
    }
    cout<<endl;
    n++;
   }

return 0;
    }
```

***Figure 15:conversion the Regular Expression to DFA***

On the main, the user's choice is received by the variable X

If the variable is equal to 1,

then the string wanted to convert to NFA and then to DFA.

first, the string was taken and then sent to the preprocessor method, which will modify the string by adding dots on it to represent the concatenation.

```
int main(){
    cout<<"Welcome :)  "<<endl;
    cout<<"You have two choices :-"<<endl;
    cout<<"from regular expression to NFA to DFA enter 1"<<endl;
    cout<<"from regular expression to DFA enter 2"<<endl;
    cout<<"to exit enter 3"<<endl;
    int x;
    cin>>x;

    if(x==1){

            cout<<"Enter a regular expression :-  "<<endl<<"e.g. (a|b)*abb"<<endl;
    string s;
    cin>>s;

    s=preprocessor(s);
    cout<<"after preprocessed "<<s;
    cout<<endl;

    s=postfix(s);
    cout<<"postfix "<<s<<endl;

            int nfa_table[1000][col];
            initialise(nfa_table);
            int states=0;

            states = reg_nfa(s,nfa_table);
            print_nfa_table(nfa_table,states);

            string dfa_tab[states][3];
            int dfa_state = 0;
            dfa_state = nfa_dfa(nfa_table,states,dfa_tab);
    }
```

***Figure 16: Main method***

The string will be sent, after taking it from the user to the preprocessor method, and then adding the # to the end of the string. The string will be sent after adding the # to the preprocessor method again to represent the connection between the dash and the string.

```
        v1.clear();
        v2.clear();
        v3.clear();
        v4.clear();
       state += 1;
    }

        int k = 1;
        for(k=1;k<states;k++){
            if(flag[k]){
                v = eclosure(nfa_table,k);
                map_state[v] = state_name(j++);
                check_init(v,map_state[v]);
                check_fin(v,map_state[v]);
                cout<<endl<<map_state[v]<<" represents :- ";
                for(int i=0;i<v.size();i++)
                    cout<<v[i]<<" ";
                cout<<endl;
                st.push(v);
                break;
            }
        }

        if(k == states)
                break;

    }

    print_dfa_table(dfa_tab,state);     //function to print dfa table

    return state;

}
```

***Figure 17:Main method***

**Run:**

When run the program, the following options appear:

1- To convert regular expression to NFA and NFA to DFA

2- To convert regular expression to DFA

Press any key to exit the program

The user must enter one of them

when choosing 1

The user is asked to enter a regular expression then

The regular expression user entered is printed

Print the regular expression after sending it to the Preprocessor Method

Print the regular expression after converting it into postfix form

After that the NFA table is calculated and printed with the initial state and final state

followed by the DFA table with it's initial state and final state

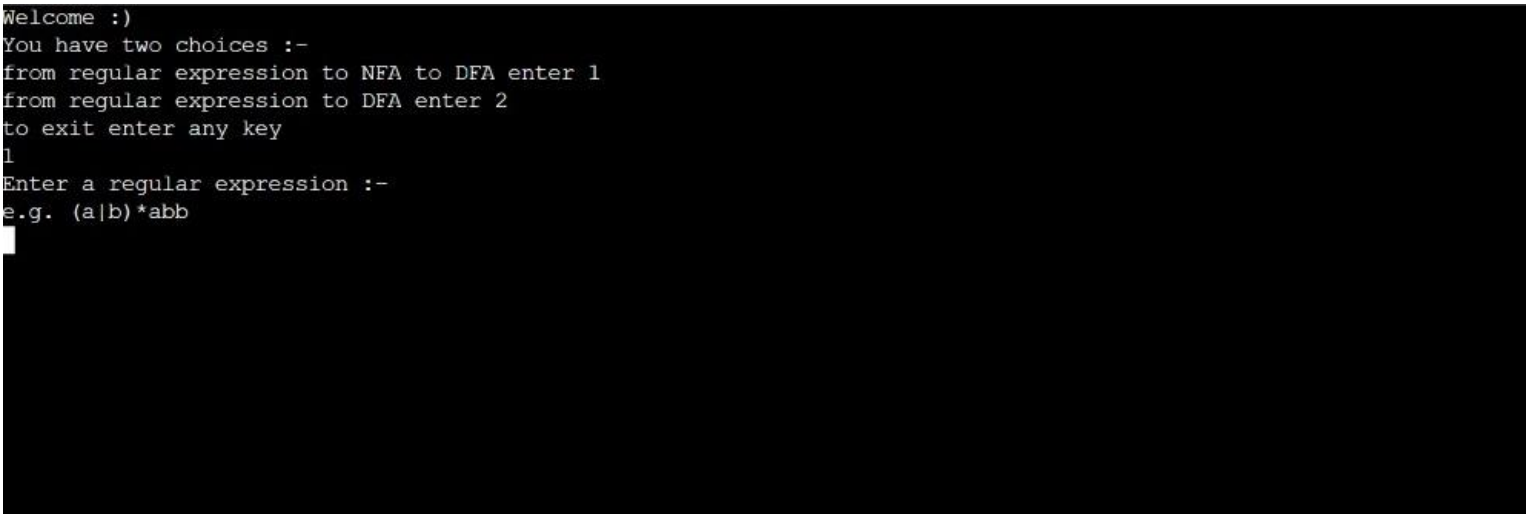



*Figure 18: part1-Screenshot of the code run*

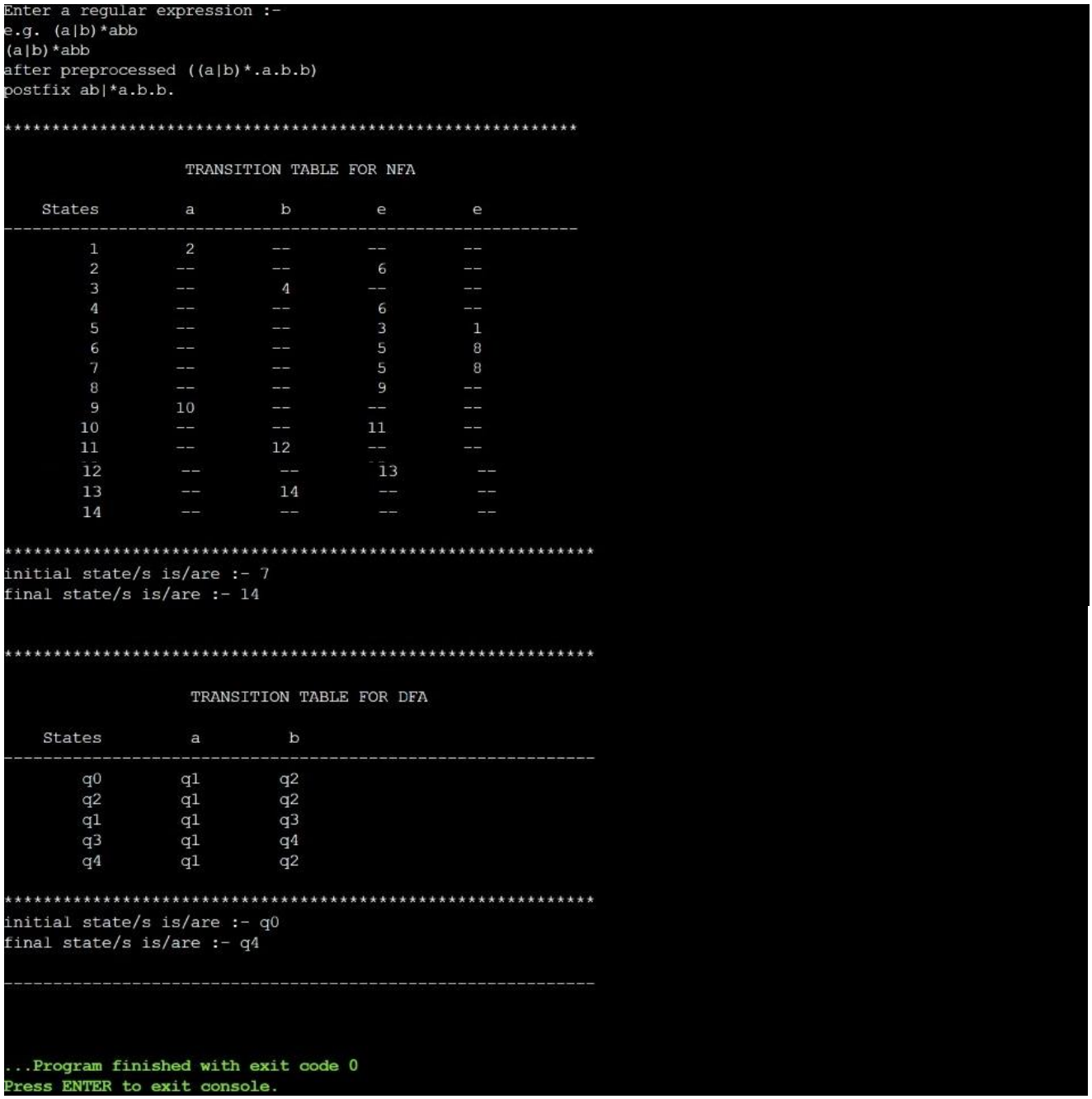



*Figure 19: RE To NFA transiton table and NFA to DFA transiton table Screenshot*

when choosing 2

The regular expression entered by the user is printed

the regular expression after sending it to the Preprocessor method is printed

The regular expression is printed after adding # and sending it to the Preprocessor method

Print the regular expression after converting it to postfix form

And then the direct DFA table is printed

```
Welcome :)
You have two choices :-
from regular expression to NFA to DFA enter 1
from regular expression to DFA enter 2
to exit enter any key
2
Enter a regular expression :-
e.g. (a|b)*abb
(a|b)*abb
after preprocessed ((a|b)*.a.b.b)
after preprocessed (((a|b)*.a.b.b).#)
postfix ab|*a.b.b.#.


****************************************

      TRANSITION TABLE FOR DFA

    States         a         b
-----------------------------------------
        A         B         A
        B         B         C
        C         B         D
        D         B         A


...Program finished with exit code 0
Press ENTER to exit console.
```

***Figure 20: RE to DFA transition table Screenshot***

And by pressing any key:

The program ends.

# Part1 References

[1] Compilers - Principles Techniques and Tools by Alfred Aho - Monica Lam- Ravi Sethi- Jeffrey Ullman - Second Edition

[2]https://www.tutorialspoint.com/what-is-the-conversion-of-a-regular-expression-to-finite-automata-nfa

[3] https://github.com/atulrulers/Regx_to_Nfa/blob/master/reg_nfa_dfa_v4.cpp

[4] https://www.geeksforgeeks.org/conversion-from-nfa-to-dfa/

[5] https://ecomputernotes.com/compiler-design/convert-regular-expression-to-dfa

[6]https://www.geeksforgeeks.org/binary-tree-array-implementation

[7] https://drive.google.com/file/d/1ZMTr44Ydz4uZPu7hlnJGKhuQ5ZHl4bbK/view

[8] https://www.geeksforgeeks.org/compiler-design-ll1-parser-in-python/#:~:text=LL(1)%20grammar,follows%20Top%2Ddown%20parsing%20method

# Part (2)

**Introduction:**

In this part we modify of grammar and create LL1 parser.

**Syntax Analysis**: this is the second phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

**A parser** is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.

Type of parser

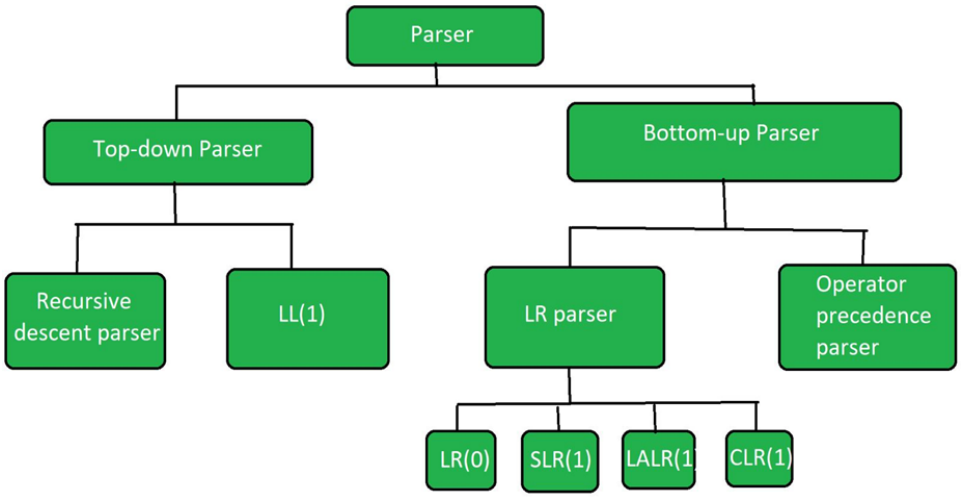



***Figure 21: Parser type***

# LL (1)

In the name LL (1), the first L stands for scanning the input from left to right, the second L stands for producing a leftmost derivation, and the 1 stands for using one input symbol of lookahead at each step to make parsing action decision.

Essential conditions to check first are as follows:

The grammar is free from left recursion.

The grammar should not be ambiguous.

The grammar has to be left factored in so that the grammar is deterministic grammar.

Algorithm to construct LL (1) Parsing Table:

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First () and Follow () for all non-terminals.

First (): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

Follow (): What is the Terminal Symbol which follows a variable in the process of derivation?

Step 3: For each production A –> α. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry A –> α in the table.

If First(α) contains ε (epsilon) as terminal, find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.

If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

To construct the parsing table :

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

## The programming language used:

We used python language to apply the second part of the project for several reasons:

- We looked for code to build the project on, and the first good code we found was in python and the python fixable than another language.
- There is a lot solutions of this problems.

.

## Problems and solutions

We facing problem how to modify this grammar and hot write it a code, how to modify this code with our grammar, how to find first and follow of grammar. And too facing errors “infinite loop. We will search in internet in many resources to solve all these problems

Grammar:

PROGRAM → STMTS

STMTS → STMT| STMT ; STMTS

STMT → id = EXPR

EXPR → EXPR + TERM | EXPR - TERM | TERM

TERM → TERM * FACTOR | TERM / FACTOR | FACTOR

FACTOR → ( EXPR ) | id | integer

- PROGRAM =p ,STMTS =C ,STMT =s ,EXPR =E ,TERM =T , FACTOR = F , power =B

**Modfiy Production Rules :**

p -> c

c -> s| s ; c

s -> id = E

E -> E + T | E - T| T

T-> T * B| T / B | B

B -> B ^ F | F

F -> [+] F | [-] F | ( E ) | id | integer

**After elimination of Left Recursion & Left Factoring :**

p -> c

c -> s c'

c' -> # | ; s

s -> id = E

E -> T E'

E' -> + T E' | - T E' | #

T-> B T'

T' -> * B T' | / B T' | #

F -> [+] F | [-] F | ( E ) | id | integr

B -> F B'

B' -> ^ F B' | #

**Find First Set :**

| **Non-Terminal Symbol** | **First Set** |
|---|---|
| **P** | id |
| **C** | Id |
| **C'** | ; , # |
| **S** | Id |
| **E** | ( ,id , integer ,[+] ,[-] |
| **E'** | + ,- ,# |
| **T** | ( ,id , integer ,[+] ,[-] |
| **T'** | * , / ,# |
| **F** | ( ,id , integer ,[+] ,[-] |
| **B** | ( ,id , integer ,[+] ,[-] |
| **B'** | ^ , # |

***Table 1: The first set of each producation***

**Find Follow Set :**

| **Non-Terminal Symbol** | **Follow Set** |
|---|---|
| **P** | $ |
| **C** | $ |
| **C'** | $ |
| **S** | $ , ; |

| | |
|---|---|
| **E** | $ , ; , ) |
| **E’** | $ , ; , ) |
| **T** | $ , ; , ) ,+ ,- |
| **T’** | $ , ; , ) ,+ ,- |
| **F** | + ,- ,* ,/ ,$ ,; ,) ,^ |
| **B** | + ,- ,* ,/ ,$ ,; ,) |
| **B’** | + ,- ,* ,/ ,$ ,; ,) |

***Table 2: The follow sets of each production***

**Crate LL1 Parser :**

| | = | ; | id | + | - | * | / | ( | ) | integer | ^ | [+] | [-] | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **P** | | | P->C | | | | | | | | | | | |
| **C** | | | C->S C’ | | | | | | | | | | | |
| **C’** | | C’->;S | | | | | | | | | | | | C’-># |
| **S** | | | S->id =E | | | | | | | | | | | |
| **E** | | | E->T E’ | | | | | E->T E’ | | E->TE’ | | E->T E’ | E->T E’ | |
| **E’** | | E' →# | | E' →+ T E' | E' → - T E' | | | | E' →# | | | | | E' →# |
| **T** | | | T → B T' | | | | | T → B T' | | T → B T' | | E → T E' | E → T E' | |
| **T’** | | T’-># | | T’-># | T’-># | T' → * B T' | T' → / B T' | | T’-># | | | | | T’-># |
| **F** | | | F → id | | | | | F → ( E ) | | F → integr | | F → [+] F | F → [-] F | |
| **B** | | | B → F B' | | | | | B → F B' | | B → F B' | | B → F B' | B → F B' | |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **B’** | | B’->#| | B’->#| B’->#| B’->#| B’->#| | B’->#| | B' → ^ F B' | | | B’->#|

*Table 3: Parser table*

**Test correct input**

| Stack | Input | Output |
|---|---|---|
| P $ \| | id eq id + id * id $ \| | output P → C |
| C $ \| | id eq id + id * id $ \| | output C → S C' |
| S C' $ \| | id eq id + id * id $ \| | output S → id eq E |
| id eq E C' $ \| | id eq id + id * id $ \| | match id |
| eq E C' $ \| | eq id + id * id $ \| | match eq |
| E C' $ \| | id + id * id $ \| | output E → T E' |
| T E' C' $ \| | id + id * id $ \| | output T → B T' |
| B' T' E' C' $ \| | id + id * id $ \| | output B → F B' |
| F B' T' E' C' $ \| | id + id * id $ \| | output F → id |
| id B' T' E' C' $ \| | id + id * id $ \| | match id |
| B' T' E' C' $ \| | + id * id $ \| | output B' → @ |
| T' E' C' $ \| | + id * id $ \| | output T' → @ |
| E' C' $ \| | + id * id $ \| | output E' → + T E' |
| + T E' C' $ \| | + id * id $ \| | match + |
| T E' C' $ \| | id * id $ \| | output T → B T' |
| B T' E' C' $ \| | id * id $ \| | output B → F B' |
| F B' T' E' C' $ \| | id * id $ \| | output F → id |
| id B' T' E' C' $ \| | id * id $ \| | match id |
| B' T' E' C' $ \| | * id $ \| | output B' → @ |
| T' E' C' $ \| | * id $ \| | output T' → * B T' |
| * B T' E' C' $ \| | * id $ \| | match * |
| B T' E' C' $ \| | id $ \| | output B → F B' |

| | | |
|---|---|---|
| F B' T' E' C' $ \| | id $ \| | output F → id |
| id B' T' E' C' $ \| | id $ \| | match id |
| B' T' E' C' $ \| | $ \| | output B' → @ |
| T' E' C' $ \| | $ \| | output T' → @ |
| E' C' $ \| | $ \| | output E' → @ |
| C' $ \| | $ \| | output C' → @ |

***Table 4:Test the correct input***

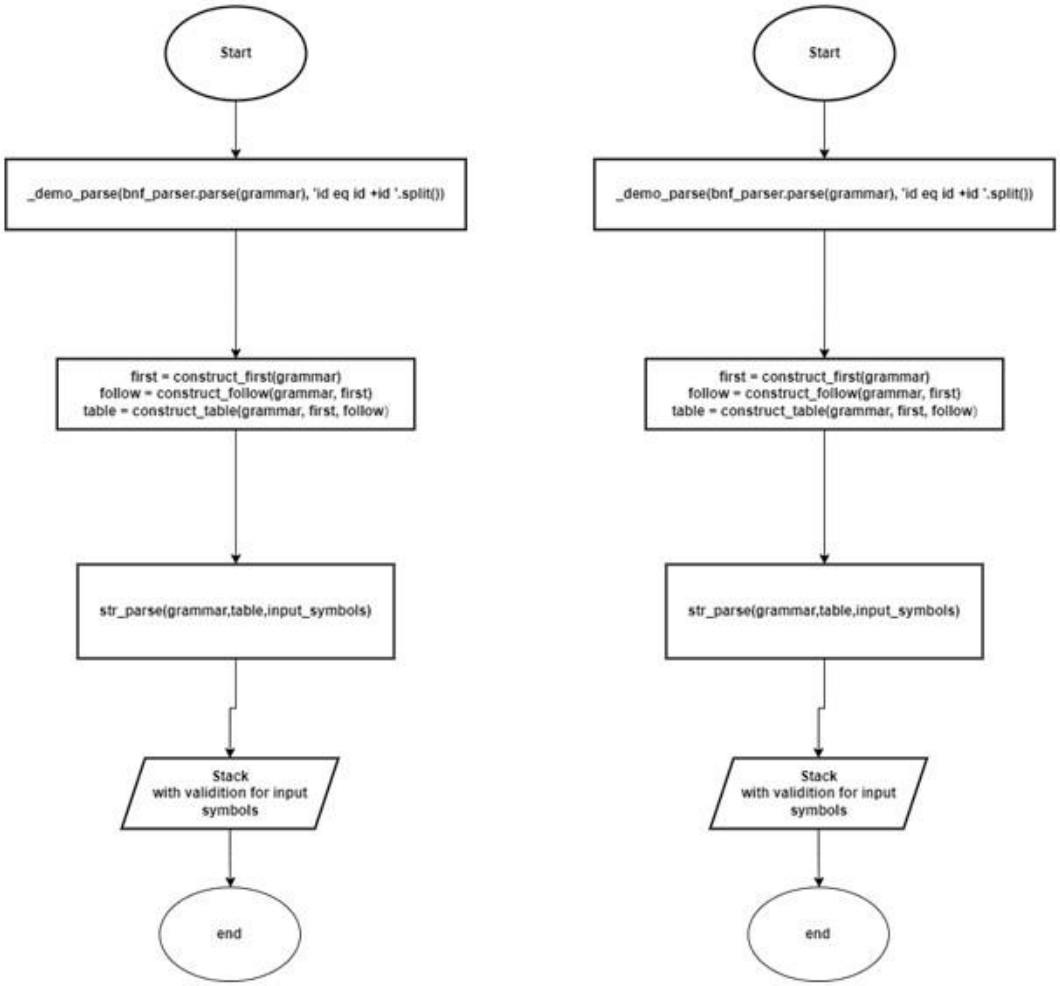
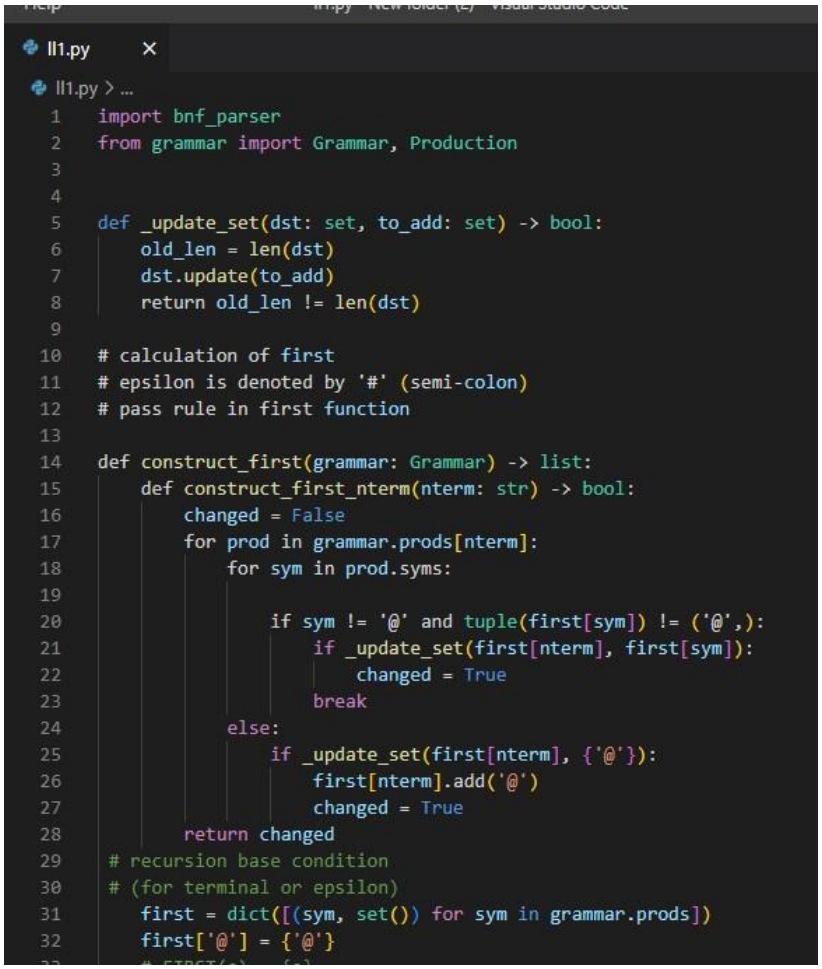



***Figure 22:illustrative Diagram of LL(1) parser***

**CODE :**

In class LL 1, it begins receiving production rule after left Recarsion solution and left factoring application begins calculating first function and following function, plots LL1 table, receives input, and determines whether it is acceptable or not.



ll1.py

ll1.py > ...

```python
import bnf_parser
from grammar import Grammar, Production


def _update_set(dst: set, to_add: set) -> bool:
    old_len = len(dst)
    dst.update(to_add)
    return old_len != len(dst)

# calculation of first
# epsilon is denoted by '#' (semi-colon)
# pass rule in first function

def construct_first(grammar: Grammar) -> list:
    def construct_first_nterm(nterm: str) -> bool:
        changed = False
        for prod in grammar.prods[nterm]:
            for sym in prod.syms:

                if sym != '@' and tuple(first[sym]) != ('@',):
                    if _update_set(first[nterm], first[sym]):
                        changed = True
                    break
            else:
                if _update_set(first[nterm], {'@'}):
                    first[nterm].add('@')
                    changed = True
        return changed
 # recursion base condition
 # (for terminal or epsilon)
    first = dict([(sym, set()) for sym in grammar.prods])
    first['@'] = {'@'}
```

***Figure 23: LL(1) class code***

```
                    changed = True
            return changed
    # recursion base condition
    # (for terminal or epsilon)
        first = dict([(sym, set()) for sym in grammar.prods])
        first['@'] = {'@'}
        # FIRST(a) = {a}
        for term in grammar.terms:
            first[term] = {term}

        changed = True
        while changed:
            changed = False
            # condition for Non-Terminals
            for nterm in grammar.prods:
                if construct_first_nterm(nterm):
                    changed = True
        return first


# returns {'@'} on empty syms
def get_first_from_syms(first: list, syms: list) -> set:
    # First, assume that eps is already in FIRST
    result = {'@'}
    for sym in syms:
        # save result in 'result' list
        result.update(first[sym])
        if '@' not in first[sym]:
            # If the eps transition link stops here, remove eps
            return result - {'@'}
    # Eps transition link doesn't stop till end, keep it
    return result
```

***Figure 24: LL(1) class code***

ll1.py > ...

```python
# calculation of follow
def construct_follow(grammar: Grammar, first: list) -> list:
    def construct_follow_nterm(prod: Production) -> bool:
        changed = False
        i = 0
        while i < len(prod.syms):
            nterm = prod.syms[i]
            # Only process nonterminals
            if not grammar.is_nonterminal(nterm):
                i += 1
                continue
            i += 1

            remaining_syms = prod.syms[i:]
            remaining_first = get_first_from_syms(first, remaining_syms)
            if _update_set(follow[nterm], remaining_first - {'@'}):
                changed = True
            if '@' in remaining_first:
                if _update_set(follow[nterm], follow[prod.nterm]):
                    changed = True
        return changed

    follow = dict([(sym, set()) for sym in grammar.prods])
    # follow start symbol =$
    follow[grammar.start] = {'$'}
    changed = True
    while changed:
        changed = False
        for prodlist in grammar.prods.values():
            for prod in prodlist:
                if construct_follow_nterm(prod):
                    changed = True
                else:
                    table[nterm].append((term, prod))
    return table


def construct_conflicts(table: dict) -> list:
    # result[index] = tuple(nonterm, term, list(Productions))
    result = list()
    for nterm, pairs in table.items():
        terms = set([pair[0] for pair in pairs])
        for term in terms:
            prods = list(filter(lambda pair, term=term: pair[0] == term, pairs))
            if len(prods) != 1:
                result.append((nterm, term, [pair[1] for pair in prods]))
    return result

# writing first ,follow
def _str_first_or_follow(grammar: Grammar, first: list, title) -> str:
    result = '   ' + title + ':'
    for sym in grammar.prods:
        result += '\n    {}: {}'.format(sym, ' '.join(first[sym]))
    return result

#writing follow in list
def str_follow(grammar: Grammar, follow: list) -> str:
    return _str_first_or_follow(grammar, follow, 'FOLLOW')

#writing first in list
def str_first(grammar: Grammar, first: list) -> str:
    return _str_first_or_follow(grammar, first, 'FIRST')
```

*Figure 25: LL(1) class code*

```python
#writing table
def str_table(table: dict) -> str:
    result = '  Table:'
    for nterm, pairs in table.items():
        result += '\n    {}:'.format(nterm)
        for term, prod in sorted(pairs, key=lambda x: x[0]):
            result += '\n      {}: {}'.format(term, prod)
    return result


def str_conflicts(conflicts: list) -> str:
    result = ''
    for nterm, term, prods in conflicts:
        result += '\n    {}, {}:'.format(nterm, term)
        for prod in prods:
            result += '\n      ' + str(prod)
    if result:
        return '  Conflicts:' + result
    else:
        return '  Conflicts: None'

#  writing ll1 table
def str_ll1(grammar: Grammar) -> str:
    first = construct_first(grammar)
    follow = construct_follow(grammar, first)
    table = construct_table(grammar, first, follow)
    conflicts = construct_conflicts(table)

    result = 'LL(1):'
    result += '\n' + str_first(grammar, first)
    result += '\n' + str_follow(grammar, follow)
    result += '\n' + str_table(table)
```

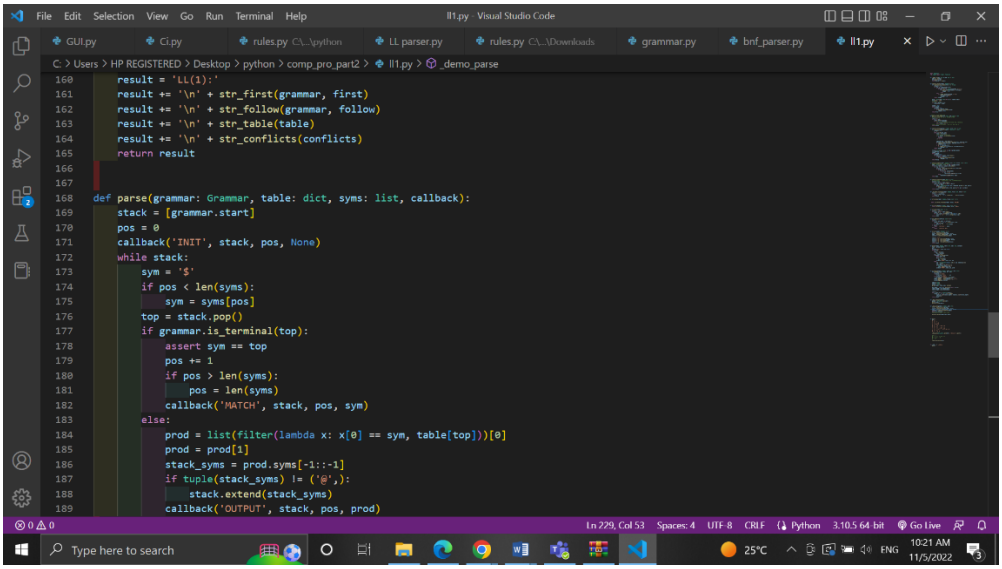



*Figure 26: LL(1) class code*

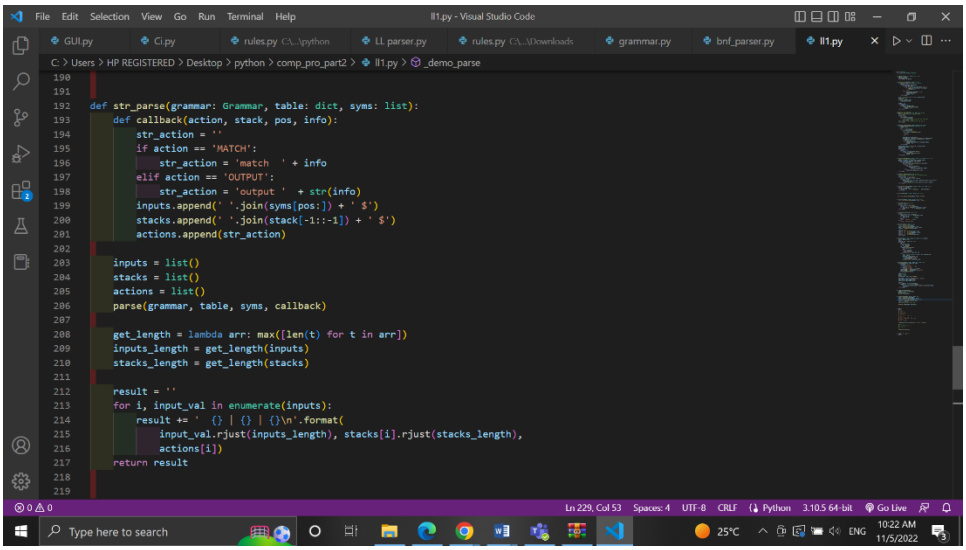



***Figure 27: LL(1) class code***

In class grammar we focus in production in firstly we create a new list an empty then add all productions of rules in this list, then searched for non-terminal and terminal symbols in production. If get non terminal we add in prods and if get terminal we add in term. Then create methods to to found all symbols with dash. And create method to return start symbol. And create method to find duplicates of symbols in production.

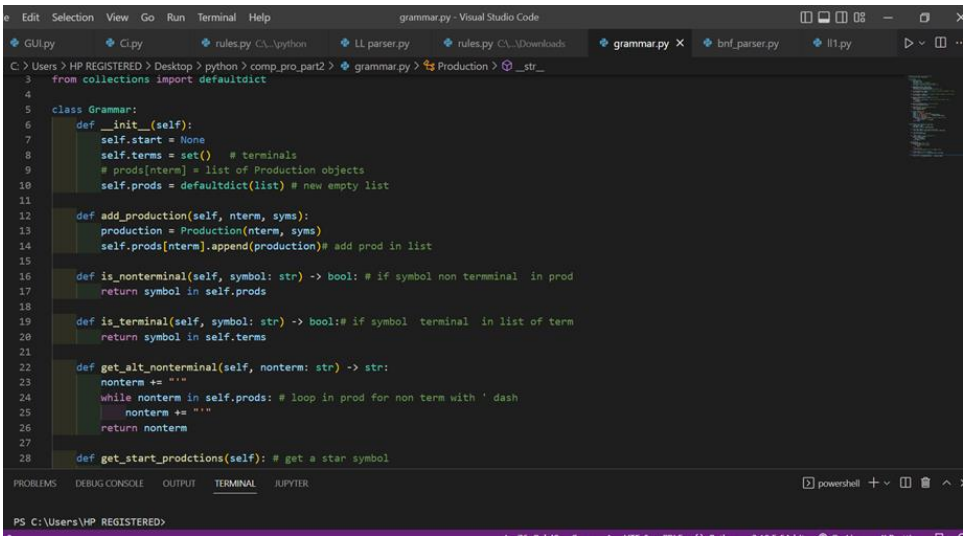



***Figure 28: Grammar class code***

In this we print a all-contents production, start symbol, non-terminal symbol and terminal symbol .In class production firstly, remove epsilon from rules and return rules without epsilon. And create the we add all productions. And create a method to return productions.

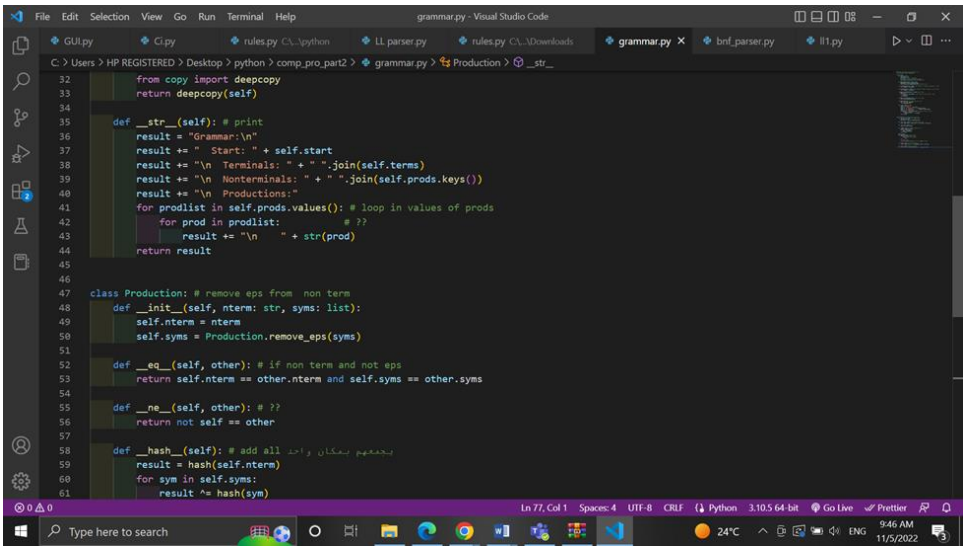



***Figure 29:grammar class code***

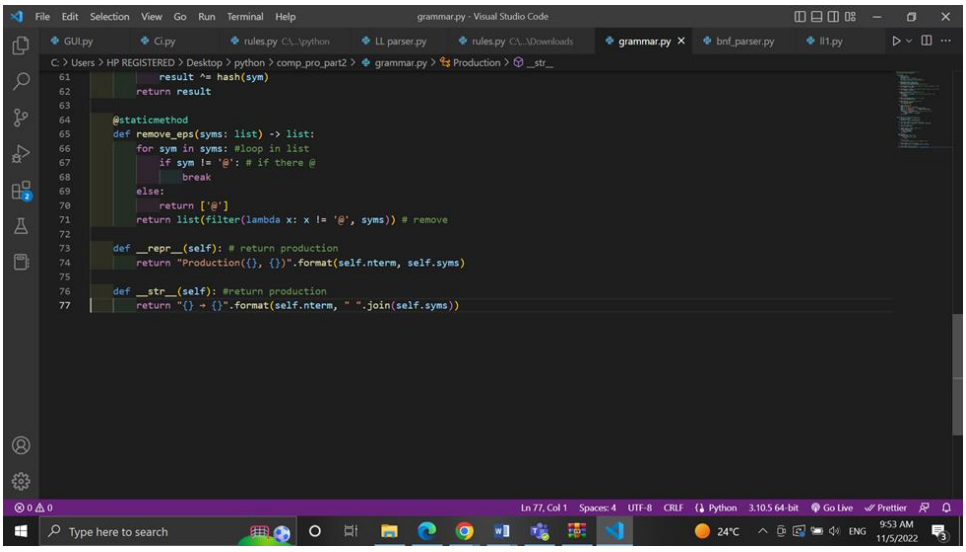



***Figure 30: grammar class code***

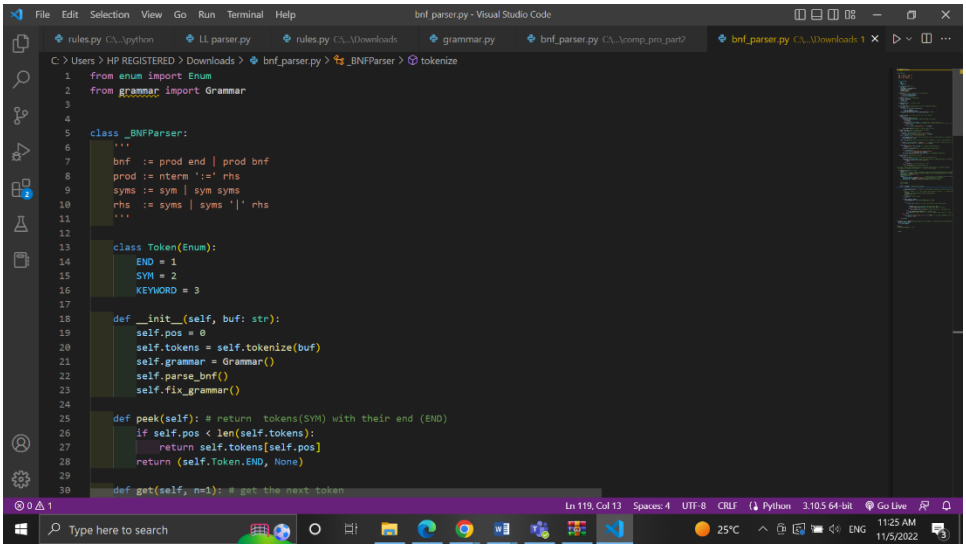



***Figure 31: bnf class code***

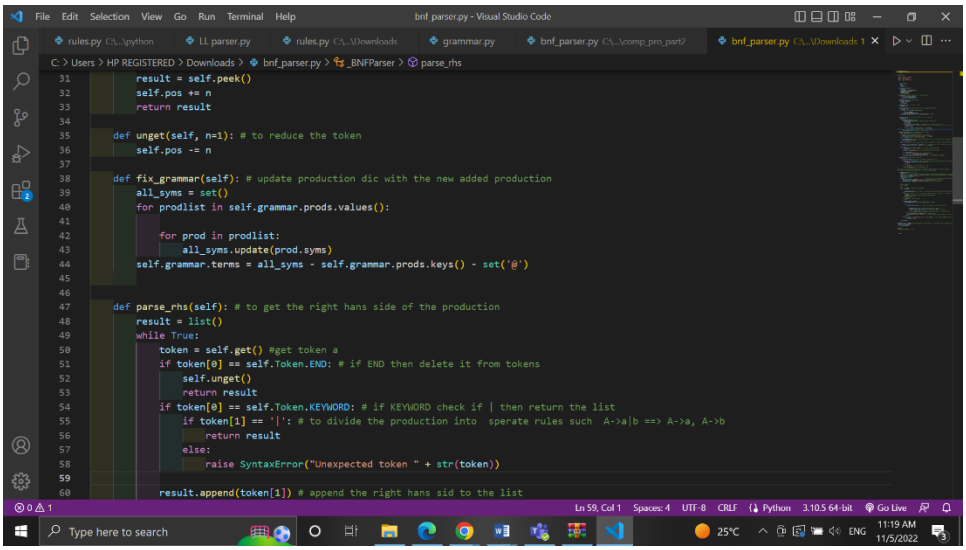



***Figure 32: bnf class code***

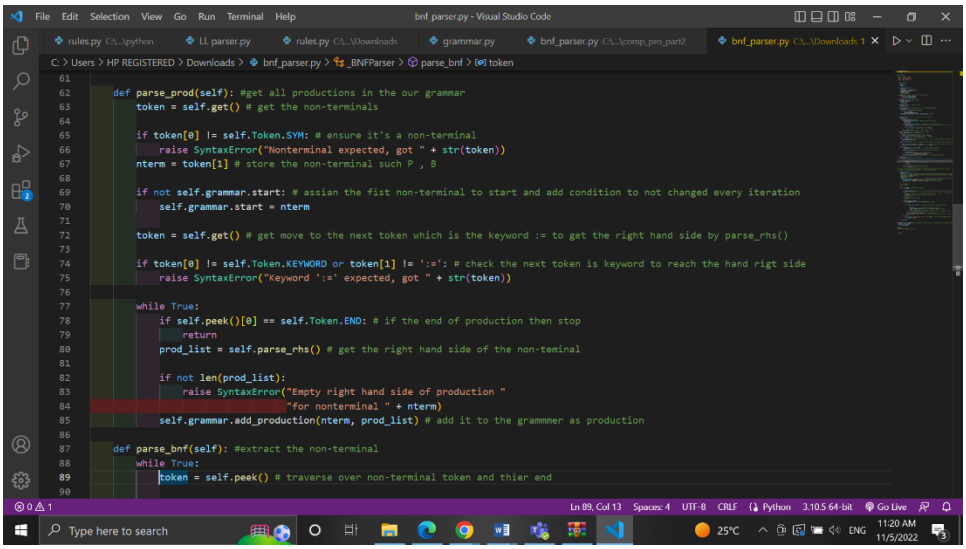



***Figure 33: bnf class code***

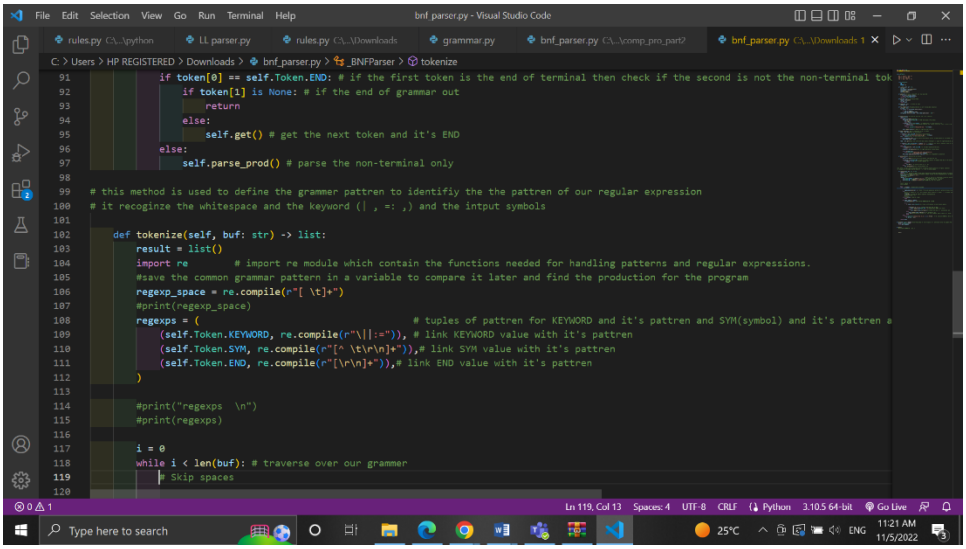



***Figure 34: bnf class code***

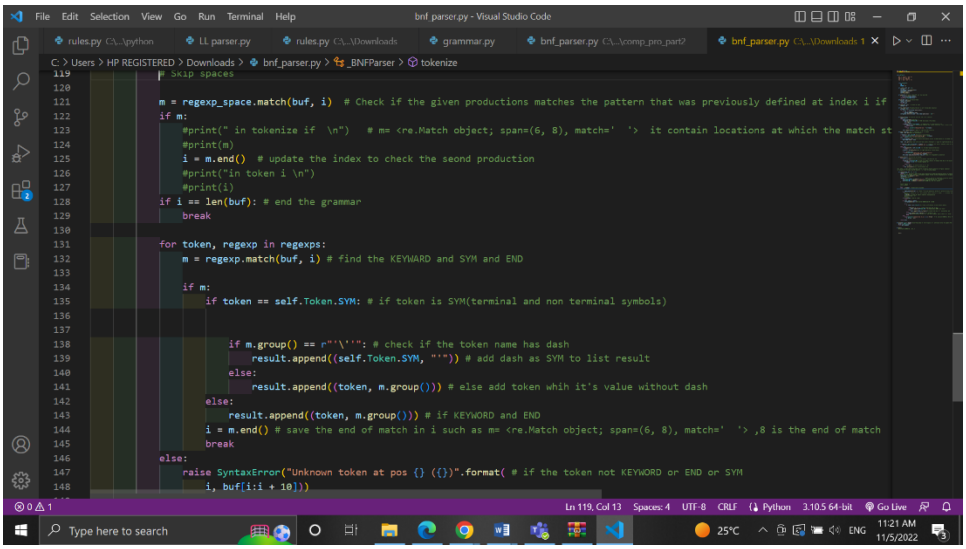



***Figure 35: bnf class code***

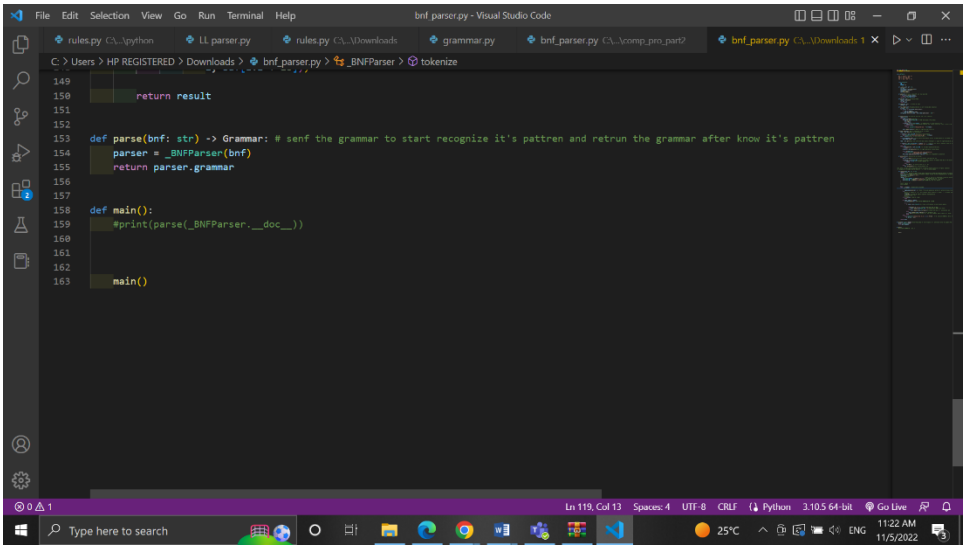



***Figure 36: bnf class code***

Grammar:


***Figure 37 screenshot of rules***

The First and the follow of each production

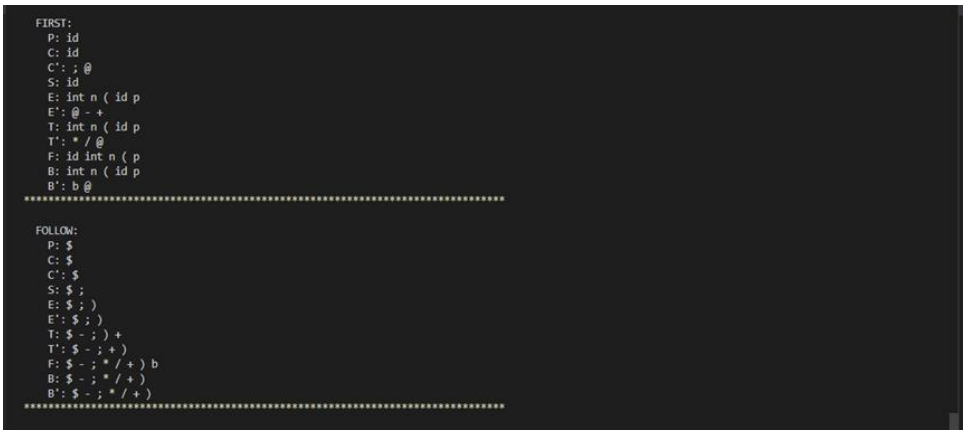



***Figure 31:the first and the follow of each production***

Parser table:

```
Table:
  P:
    id: P → C
  C:
    id: C → S C'
  C':
    $: C' → @
    ;: C' → ; S
  S:
    id: S → id eq E
  E:
    (: E → T E'
    id: E → T E'
    int: E → T E'
    n: E → T E'
    p: E → T E'
  E':
    $: E' → @
    ): E' → @
    +: E' → + T E'
    -: E' → - T E'
    ;: E' → @
  T:
    (: T → B T'
    id: T → B T'
    int: T → B T'
    n: T → B T'
    p: T → B T'
  T':
    $: T' → @
    ): T' → @
```

```
  T':
    $: T' → @
    ): T' → @
    *: T' → * B T'
    +: T' → @
    -: T' → @
    /: T' → / B T'
    ;: T' → @
  F:
    (: F → ( E )
    id: F → id
    int: F → int
    n: F → n F
    p: F → p F
  B:
    (: B → F B'
    id: B → F B'
    int: B → F B'
    n: B → F B'
    p: B → F B'
  B':
    $: B' → @
    ): B' → @
    *: B' → @
    +: B' → @
    -: B' → @
    /: B' → @
    ;: B' → @
    b: B' → b F B'
```

***Figure 38 parser table***

Test input valid and invalid:

```
input symbol :

['id', 'eq', 'id', '+', 'id', '*', 'id']


  id eq id + id * id $ |            P $ |
  id eq id + id * id $ |            C $ | output P → C
  id eq id + id * id $ |          S C' $ | output C → S C'
  id eq id + id * id $ |     id eq E C' $ | output S → id eq E
     eq id + id * id $ |       eq E C' $ | match  id
        id + id * id $ |         E C' $ | match  eq
        id + id * id $ |       T E' C' $ | output E → T E'
        id + id * id $ |     B T' E' C' $ | output T → B T'
        id + id * id $ |  F B' T' E' C' $ | output B → F B'
        id + id * id $ | id B' T' E' C' $ | output F → id
          + id * id $ |    B' T' E' C' $ | match  id
          + id * id $ |       T' E' C' $ | output B' → @
          + id * id $ |         E' C' $ | output T' → @
          + id * id $ |     + T E' C' $ | output E' → + T E'
            id * id $ |       T E' C' $ | match  +
            id * id $ |     B T' E' C' $ | output T → B T'
            id * id $ |  F B' T' E' C' $ | output B → F B'
            id * id $ | id B' T' E' C' $ | output F → id
              * id $ |    B' T' E' C' $ | match  id
              * id $ |       T' E' C' $ | output B' → @
              * id $ |   * B T' E' C' $ | output T' → * B T'
                id $ |     B T' E' C' $ | match  *
                id $ |  F B' T' E' C' $ | output B → F B'
                id $ | id B' T' E' C' $ | output F → id
                  $ |    B' T' E' C' $ | match  id
                  $ |       T' E' C' $ | output B' → @
                  $ |         E' C' $ | output T' → @
                  $ |           C' $ | output E' → @
                  $ |             $ | output C' → @

 the input is valid
```

***Figure 39 valid input***

```
  input symbol :

  ['id', 'eq', 'id', 'b', 'n']


    id eq id b n $ |              P $ |
    id eq id b n $ |              C $ | output P → C
    id eq id b n $ |            S C' $ | output C → S C'
    id eq id b n $ |       id eq E C' $ | output S → id eq E
       eq id b n $ |         eq E C' $ | match  id
          id b n $ |           E C' $ | match  eq
          id b n $ |         T E' C' $ | output E → T E'
          id b n $ |       B T' E' C' $ | output T → B T'
          id b n $ |    F B' T' E' C' $ | output B → F B'
          id b n $ |  id B' T' E' C' $ | output F → id
             b n $ |     B' T' E' C' $ | match  id
             b n $ | b F B' T' E' C' $ | output B' → b F B'
               n $ |    F B' T' E' C' $ | match  b
               n $ | n F B' T' E' C' $ | output F → n F
                 $ |    F B' T' E' C' $ | match  n

  the input is invalid
```

***Figure 40 invalid input***