

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

Part (1)	5
Introduction.....	5
Regular expression to NFA:.....	7
Algorithm for the conversion of Regular Expression to NFA...	7
The programming language used.....	8
Problems and solutions	8
Our implementation:.....	8
Code:	10
Preprocessor():.....	10
Postfix():.....	11
reg_nfa():	12
NFA to DFA:	13
Our implementation:.....	13
Code:	15
check_init():.....	15
check_fin():.....	15
nfa_dfa()......	15
Regular expression to DFA:.....	17
Our implementation:.....	18
code:	20
treebu():.....	20
DFA	21
Run:.....	23
Part1 References	26
Part (2)	27

TABLE OF CONTENT

Figure 1:finite automata	6
Figure 2: the conversion of Regular Expression to NFA	8
Figure 3:illustrative Diagram of the conversion RE to NFA.....	9
Figure 4: Preprocessor method.....	10
Figure 5:Postfix method code	11
Figure 6:Regular Expression to NFA method code.....	12
Figure 7:illustrative Diagram of the conversion NFA to DFA.....	14
Figure 8: check the initial state and check the final state methods	15
Figure 9:conversion NFA to DFA method code	15
Figure 10:conversion NFA to DFA method code	16
Figure 11:conversion NFA to DFA method code	16
Figure 12:Rules for computing nullable, firstpos and lastpos	17
Figure 13:conversion the RE to DFA	19
Figure 14: Build tree method	20
Figure 15:conversion the Regular Expression to DFA	21
Figure 16: Main method.....	22
Figure 17:Main method.....	22
Figure 18: part1-Screenshot of the code run.....	23
Figure 19: RE To NFA transiton table and NFA to DFA transiton table Screenshot	24
Figure 20: RE to DFA transition table Screenshot	25
Figure 21: Parser type	28
Figure 22:illustrative Diagram of LL(1) parser	35
Figure 23: LL(1) class code.....	36
Figure 24: LL(1) class code.....	37
Figure 25: LL(1) class code	39
Figure 26: LL(1) class code	39
Figure 27: LL(1) class code	40
Figure 28: Grammar class code	40
Figure 29:grammar class code	41
Figure 30: grammar class code.....	41
Figure 32:the first and the follow of each production	45

Table 1: The first set of each production.....	31
Table 2: The follow sets of each production	32
Table 3: Parser table.....	33
Table 4:Test the correct input.....	34

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 $(Q, \Sigma, \delta, q_0, F)$, where:

Q : a finite set of states

Σ : a finite set of the input symbol

q_0 : 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 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)

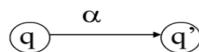
- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
- The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated 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 reads 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 q_0 for input a , there is only one path that goes to q_1 . Similarly, from q_0 , there is only one path for input b going to q_2 .

NFA vs DFA

DFA: For every state q in S and every character α in Σ , one and only one transition of the following form occurs:



NFA: For every state q in S and every character α in $\Sigma \cup \{\epsilon\}$, one (or both) of the following will happen:

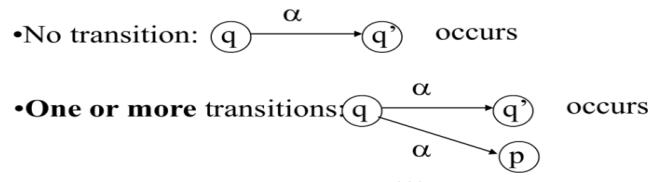


Figure 1:finite automata

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]

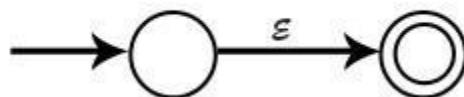
Algorithm for the conversion of Regular Expression to NFA

Input – A Regular Expression R

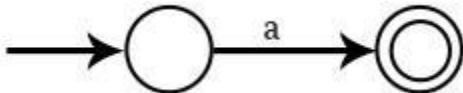
Output – NFA accepting language denoted by R

Method

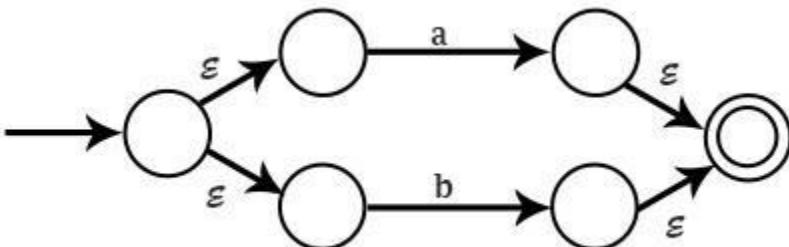
For ϵ , NFA is



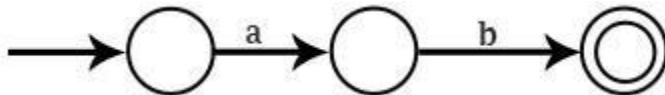
For a NFA is



For $a + b$, or $a | b$ NFA is



For ab , NFA is



For a^* , NFA is

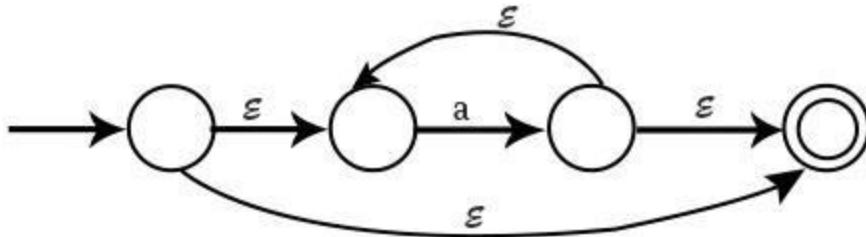


Figure 2: the conversion of Regular Expression to NFA

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 runtime.

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 preprocess() method, which returns string.

We added to this string the '#' symbol and sent to the preprocess() 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 states 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.

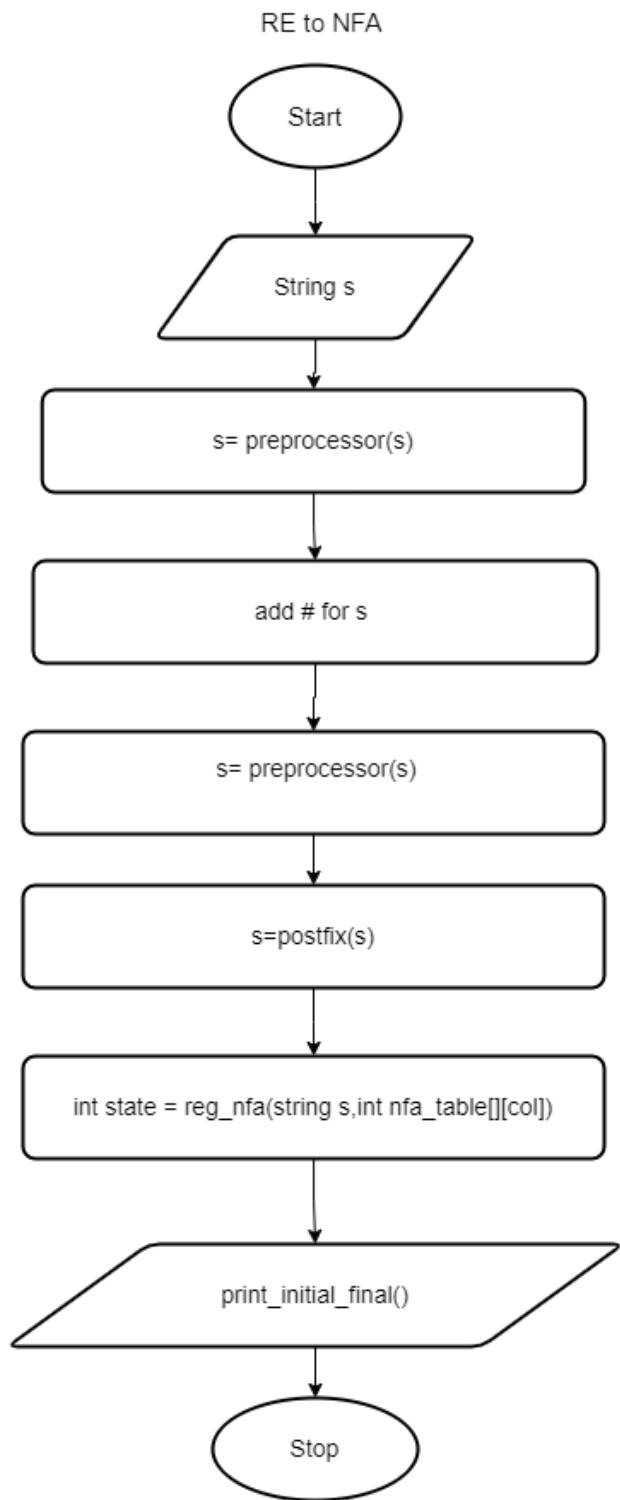


Figure 3:illustrative Diagram of the conversion RE to NFA

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

```
520: string preprocessor(string s){
521:     char x[500];
522:     int l=s.length();
523:     int j=0;
524:     x[j]='(';
525:     j += 1;
526:     for(int i=0;i<l;i++){
527:         x[j]=s[i];
528:         j += 1;
529:         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 a dot and close parenthesis*/
530:             x[j]='.';
531:             j += 1;
532:         }else if(s[i]==')'&&s[i+1]=='('){
533:             x[j]='.';
534:             j += 1;
535:             if(((s[i]>=97&&s[i]<=122)||s[i]==35)&&s[i+1]==')'){
536:                 x[j]='.';
537:                 j += 1;
538:             }else if((s[i]==')'&&(s[i+1]>=97&&s[i+1]<=122||s[i+1]==35)){
539:                 x[j]='.';
540:                 j += 1;
541:             }else if(s[i]=='*'||(s[i+1]>=97&&s[i+1]<=122)||s[i+1]==35)){
542:                 x[j]='.';
543:                 j += 1;
544:             }
545:         }
546:         x[j] = ')';
547:         j += 1;
548:     string p;
549:     for(int i=0;i<j;i++)
550:         p += x[i];
551:     return p;
552: }
```

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.

```
559 string postfix(string s){
560     int l=s.length();
561     char a[5000];
562     stack<char> ch;
563     int j=0;
564     for(int i=0;i<l;i++){
565         char x=a[i];
566         switch(x){
567             case 'a': a[j]='a';
568                         j+=1;
569                         break;
570             case 'b': a[j]='b';
571                         j+=1;
572                         break;
573             case '#': a[j]='#';
574                         j+=1;
575                         break;
576             case '(': ch.push('(');
577                         break;
578             case ')': while(!ch.empty()){
579                 if(ch.top()=='('){
580                     ch.pop();
581                     break;
582                 }else{
583                     a[j]=ch.top();
584                     ch.pop();
585                     j+=1;
586                 }
587             }
588             case '.': if(ch.empty()){
589                 ch.push('.');
590             }else{
591                 char temp = ch.top();
592                 if(temp=='(')
593                     ch.push('.');
594                 else if(temp=='*'){
595                     a[j]=ch.top();
596                     ch.pop();
597                     j+=1;
598                     if(ch.top()=='.' ){
599                         a[j] = '.';
600                         j+=1;
601                     }else if(ch.top()=='.' ){
602                         a[j] = '.';
603                         j+=1;
604                     }else{
605                         ch.push('.');
606                     }
607                 }else if(temp=='.'){
608                     a[j]=ch.top();
609                     ch.pop();
610                     j+=1;
611                     ch.push('.');
612                 }else if(temp=='|'){
613                     ch.push('.');
614                 }
615             }
616             break;
617         case '|': if(ch.empty()){
618                 ch.push('|');
619             }else{
620                 char temp = ch.top();
621                 if(temp == '(')
622                     ch.push('|');
623                 else if(temp == '**'){
624                     a[j] = ch.top();
625                     ch.pop();
626                     j+=1;
627                     ch.push('|');
628                 }else if(temp == '.'){
629                     a[j] = ch.top();
630                     j+=1;
631                     ch.pop();
632                     ch.push('|');
633                 }
634             }
635             case '**': if(ch.empty()){
636                 ch.push(**);
637             }else{
638                 char temp = ch.top();
639                 if(temp == '(' || temp == '.' || temp == '|' )
640                     ch.push(**);
641                 else{
642                     a[j] = ch.top();
643                     ch.pop();
644                     j+=1;
645                     ch.push(**);
646                 }
647             }
648         }
649     }
650     break;
651 }
652 string p;
653 for(int i=0;i<j;i++){
654     p += a[i];
655 }
656 return p;
657 }
```

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

```
744 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 ,epsilon
745     int l = s.length();
746     int states = 1;
747     int m,n,j,counter;
748     for(int i=0;i<l;i++){
749         char x = s[i]; // traverse over postfix string char by car
750         switch(x){
751             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 which
752                         init[a] = states;
753                         a += 1;
754                         states += 1;
755                         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
756                         fin[b] = states;
757                         b += 1;
758                         nfa_table[states][0] = states; // initiate the new state(state=2) to the nfa_table
759                         states +=1;
760                         break;
761             case 'b': nfa_table[states][0] = states; // the same process as a
762                         init[a] = states;
763                         a += 1;
764                         states += 1;
765                         nfa_table[states-1][2] = states;
766                         fin[b] = states;
767                         b += 1;
768                         nfa_table[states][0] = states;
769                         states +=1;
770                         break;
771             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
772                         n = init[a-1]; //and the second initial state in array init
773                         nfa_table[m][3]=n; // add the n to epsilon col in the m'th raw
774                         reduce_fin(b-2); //reduces final state , or delete the m since we Link it with another state in the table
775                         a -= 1; //reduces initial states, or delete the n since we Link it with another state in the table
776                         break;
777             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
778                         m = init[j-counter];
779                         nfa_table[states][3+counter]=m;// with epsilon as input for each of them
780                         }
781                         a=a-2; // after we connected the initial states now it's time to delete them from the init array
782                         init[a]=states; // and add the new initial state to the array
783                         a -=1;
784                         nfa_table[states][0] = states; // add the new state to the table
785                         states += 1;
786                         //We now connect the final states that were previously linked
787                         // to the initial states with the new states to become the current final states
788                         for(j=b-1,counter=0;counter<2;counter++){
789                             m = fin[j-counter];
790                             nfa_table[m][3]=states;
791                         }
792                         b=b-2; // after we connected the final states now it's time to delete them from the fin array
793                         fin[b]=states; // and add the new final state to array
794                         b -= 1;
795                         nfa_table[states][0] = states; // add the new state to the table
796                         states += 1;
797                         break;
798             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
799                         nfa_table[states][3] = m; // initial state connected to the final with epsilon
800                         nfa_table[states][0] = states; // add the current final state to the table
801                         init[a-1] = states; //after we connected the initial states now it's time to delete them from the init array
802                         states += 1;
803                         n = fin[b-1]; // now we take the current final and connecting it in reverse dirction to current initial state
804                         nfa_table[n][3]=m; // with epsilon
805                         nfa_table[n][4]=states; // then the current final state is connect to Last state which is the final now
806                         nfa_table[states-1][4]=states;
807                         fin[b-1]=states;
808                         nfa_table[states][0] = states;
809                         states += 1;
810                         break;
811                     }
812                 }
813             }
```

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 $N < Q, \Sigma, q_0, \delta, F >$ which recognizes a language L. Then the DFA $D < Q', \Sigma, q_0, \delta', F' >$ can be constructed for language L as:

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

Step 2: Add q_0 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.

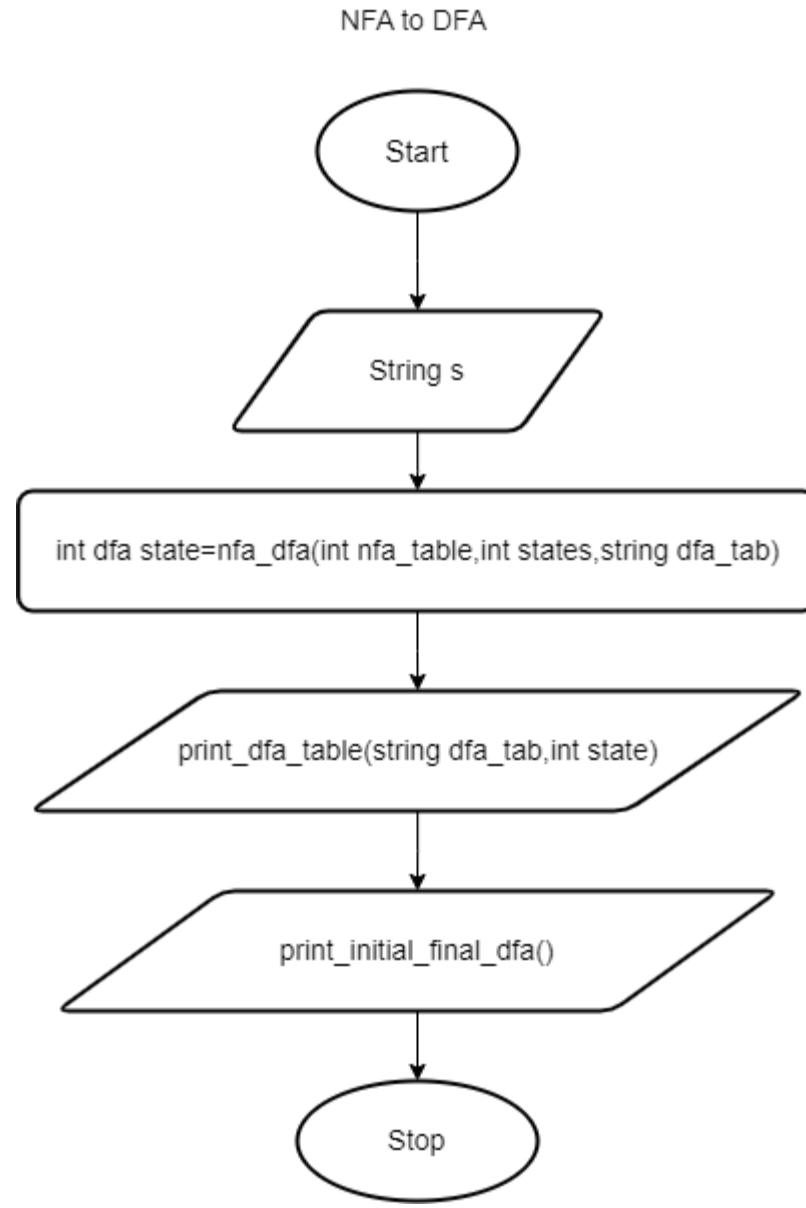


Figure 7:illustrative Diagram of the conversion NFA to DFA

Code:

check_init():

find the initial state for DFA

check_fin():

find final state for DFA

nfa_dfa():

```
903 int nfa_dfa(int nfa_table[][col],int states,string dfa_tab[][]){
904     bool flag[states];
905     ...
906     void check_init(vector<int> v, string s){
907         for(int i=0;i<v.size();i++){
908             if(v[i] == init){
```

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

```
93 void check_fin(vector<int> v, string s){
94     for(int i=0;i<v.size();i++){
95         if(v[i] == fin[i]){
96             fin_dfa[_b] = s;
97             _b += 1;
98         }
99     }
100 }
101 // transition of e-closure to over input symbol a
102 int counter = 0;
103 while(true){
104     while(!st.empty()){
105         vector<int> v;
106         v = st.top();
107         st.pop();
108         counter += 1;
109         dfa_tab[state][0] = map_state[v];    //find transition of a state over input symbol 'a' and 'b'
110         for(int i=0;i<v.size();i++){
111             flag[v[i]] = false;
112             int temp = nfa_table[v[i]][1];    //over input symbol a
113             int temp1 = nfa_table[v[i]][2];   //over input symbol b
114             if(temp>0)
115                 v3.push_back(temp);
116             if(temp1>0)
117                 v3.push_back(temp1);
118         }
119     }
120 }
```

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

```

Figure 10:conversion NFA to DFA method code

```

936
937     for(it = map_temp1.begin(); it != map_temp1.end(); it++){
938         v4.push_back(it->first);
939         flag[it->first] = false;
940     }

```

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.

```

970     if(v2.empty()){
971         dfa_tab[state][i] = "";
972     } else {
973         string t = map_state[v2];
974         char flag1 = t[0];
975         if( flag1 == 'q'){
976             if( dfa_tab[state][i] == map_state[v2]; //checking v2 has already been mapped or not
977                 {
978                     dfa_tab[state][i] = state_name[j];
979                     map_state[v2] = dfa_tab[state][i];
980                     check_init(v2,map_state[v2]);
981                     check_fin(v2,map_state[v2]);
982                     cout<<endl<<map_state[v2]<<" represents :- ";
983                     for(int i=0;i<v2.size();i++)
984                         cout<<v2[i]<<" ";
985                     cout<<endl;
986                     st.push(v2); //not mapped state will be pushed into stack
987                 }
988             }
989         }
990     }
991
992     if(v4.empty()){
993         dfa_tab[state][i] = "";
994     } else {
995         string t = map_state[v4];
996         char flag1 = t[0];
997         if( flag1 == 'q'){
998             if( dfa_tab[state][i] == map_state[v4];
999                 {
1000                     dfa_tab[state][i] = map_state[v4];
1001                 }
1002             }
1003         }

```

Figure 11:conversion NFA to DFA method code

```

1008
1009     for(int i=0;i<v4.size();i++)
1010         cout<<v4[i]<<" ";
1011     cout<<endl;
1012     st.push(v4);
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 $\text{nullable}(n)$, $\text{firstpos}(n)$, $\text{lastpos}(n)$ and $\text{followpos}(i)$ from the syntax tree.
- $\text{nullable}(n)$: Is true for $*$ node and node labeled with ϵ . For other nodes it is false.
- $\text{firstpos}(n)$: Set of positions at node n that corresponds to the first symbol of the sub-expression rooted at n .
- $\text{lastpos}(n)$: Set of positions at node n that corresponds to the last symbol of the sub-expression rooted at n .
- $\text{followpos}(i)$: 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 ϵ	True	\emptyset	\emptyset
A leaf with position i	False	{i}	{i}
An or node $n = c_1 c_2$	Nullable (c_1) or Nullable (c_2)	firstpos (c_1) U firstpos (c_2)	lastpos (c_1) U lastpos (c_2)
A cat node $n = c_1 c_2$	Nullable (c_1) and Nullable (c_2)	If (Nullable (c_1)) firstpos (c_1) U firstpos (c_2) else firstpos (c_1)	If (Nullable (c_2)) lastpos (c_1) U lastpos (c_2) else lastpos (c_1)
A star node $n = c_1^*$	True	firstpos (c_1)	lastpos (c_1)

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 followpos 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 preprocess() method, which returns a string.

We added to this string the '#' symbol and sent to the preprocess() 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

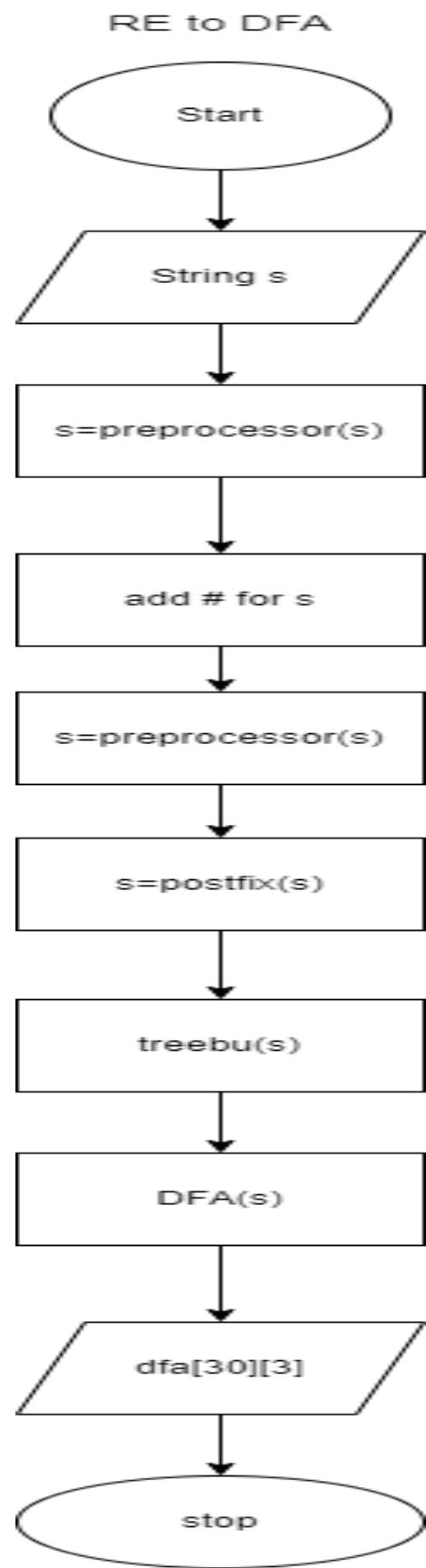


Figure 13:conversion the RE to DFA

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             tree[i]=new Node(s[i]);
513             if(s[i-1]=='*'&&(s[i-2]=='a'||s[i-2]=='b')){
514                 tree[i-3]=new Node(s[i-3]); //Left
515                 tree[i-1]=new Node(s[i-1]); //right
516             }else if(s[i-1]=='*'&&s[i-2]=='|'){
517                 tree[i-5]=new Node(s[i-5]); //Left
518                 tree[i-1]=new Node(s[i-1]); //right
519             }else{
520                 tree[i-2]=new Node(s[i-2]); //left
521                 tree[i-1]=new Node(s[i-1]); //right
522             }
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         tree[i]=new Node(s[i]);
532         tree[i-1]=new Node(s[i-1]); //left
533     }
534 }
535 }
536 node_pos();
537 node_null();
538 node_firstpos();
539 node_Laststpos();
540 node_followPos();
541 }
542 }
```

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

```

353. int DFA(string s){
354.     initdfa();
355.
356.     string arr[30][2];
357.     for(int m =0; m<30; m++){
358.         arr[m][0] = "null";
359.         arr[m][1] = "null";
360.     }
361.     string set=followpos[1];
362.     arr[0][0] = "states";
363.     arr[0][1] = "sets";
364.     arr[1][0] = "A";
365.     arr[1][1] = "set";
366.     int i=1;
367.
368.     int counter=1;
369.     while(arr[i][1]!="null"){
370.         string new_set_a;
371.         string new_set_b;
372.         string ex_a="null";
373.         string ex_b="null";
374.         new_set_a = generate_set_a(arr[i][1]);
375.         new_set_b = generate_set_b(arr[i][1]);
376.         for(int a=0; a<30; a++){
377.             if(new_set_a==arr[a][1]){
378.                 ex_a=arr[a][0];
379.             }if(new_set_b==arr[a][1]){
380.                 ex_b=arr[a][0];
381.             }
382.         }
383.         if(ex_a!="null"){
384.             dfa[i][0]=arr[i][0][0];
385.             dfa[i][1]=ex_a[0];
386.         }
387.         else {
388.             counter++;
389.             arr[counter][0]=arr[counter-1][0][0]+1;
390.             arr[counter][1]=new_set_a;
391.             dfa[i][0]=arr[i][0][0];
392.             dfa[i][1]=arr[counter][0][0];
393.         }
394.         if(ex_b!="null"){
395.             dfa[i][0]=arr[i][0][0];
396.             dfa[i][2]=ex_b[0];
397.         }
398.         else{
399.             counter++;
400.             arr[counter][0]=arr[counter-1][0][0]+1;
401.             arr[counter][1]=new_set_b;
402.             dfa[i][0]=arr[i][0][0];
403.             dfa[i][2]=arr[counter][0][0];
404.         }
405.         i++;
406.
407.         dfa[3][0] = 'A';
408.         cout << endl << endl;
409.         for(int i=0;i<40;i++){
410.             cout << "=";
411.             cout << endl << endl;
412.             cout << setw(38) << "TRANSITION TABLE FOR DFA" << endl << endl;
413.             cout << setw(10) << "States" << setw(10) << "a" << setw(10) << "b" << endl;
414.             for(int i=0;i<48;i++){
415.                 cout << "=";
416.                 cout << endl;
417.                 cout << endl;
418.                 int n=1;
419.                 while(dfa[n][2]!='0'){
420.                     for(int j=0;j<3;j++){
421.                         cout << setw(10) << dfa[n][j];
422.                     }
423.                     cout << endl;
424.                     n++;
425.                 }
426.             }
427.
428.         return 0;
429.     }
430.
```

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.

```
1851- int main(){
1852     cout<<"Welcome :)"<<endl;
1853     cout<<"You have two choices :-"<<endl;
1854     cout<<"From regular expression to NFA to DFA enter 1"<<endl;
1855     cout<<"From regular expression to DFA enter 2"<<endl;
1856     cout<<"To exit enter 3"<<endl;
1857     int x;
1858     cin>>x;
1859
1860     if(x==1){
1861         cout<<"Enter a regular expression :- "<<endl<<"e.g. (a|b)*abb"<<endl;
1862         string s;
1863         cin>>s;
1864
1865         s=preprocessor(s);
1866         cout<<"After preprocessed "<<s;
1867         cout<<endl;
1868
1869         s=postfix(s);
1870         cout<<"Postfix "<<s<<endl;
1871
1872         int nfa_table[1000][col];
1873         initialise(nfa_table);
1874         int states=0;
1875
1876         states = reg_nfa(s,nfa_table);
1877         print_nfa_table(nfa_table,states);
1878
1879         string dfa_tab[states][col];
1880         int dfa_state = 0;
1881         dfa_state = nfa_dfa(nfa_table,states,dfa_tab);
1882     }
1883 }
```

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.

```
1014     v1.clear();
1015     v2.clear();
1016     v3.clear();
1017     v4.clear();
1018     state += 1;
1019
1020
1021     int k = 1;
1022     for(k<1;k<states;k++){
1023         if(flag[k]){
1024             v = closure(nfa_table,k);
1025             map_state[v] = state_name(j++);
1026             check_init(v,map_state[v]);
1027             check_fin(v,map_state[v]);
1028             cout<<v<<" represents :- ";
1029             for(i=0;i<map_state[v];i++)
1030                 cout<<v[i]<<"#";
1031             cout<<endl;
1032             st.push(v);
1033             break;
1034         }
1035     }
1036
1037     if(k == states)
1038         break;
1039
1040     }
1041
1042     print_dfa_table(dfa_tab,state); //function to print dfa table
1043
1044     return state;
1045
1046 }
```

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

```
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  
1  
Enter a regular expression :-  
e.g. (a|b)*abb
```

Figure 18: part1-Screenshot of the code run

```

Enter a regular expression :-
e.g. (a|b)*abb
(a|b)*abb
after preprocessed ((a|b)*.a.b.b)
postfix ab|*a.b.b.

```

TRANSITION TABLE FOR NFA

States	a	b	e	e
1	2	--	--	--
2	--	--	6	--
3	--	4	--	--
4	--	--	6	--
5	--	--	3	1
6	--	--	5	8
7	--	--	5	8
8	--	--	9	--
9	10	--	--	--
10	--	--	11	--
11	--	12	--	--
12	--	--	13	--
13	--	14	--	--
14	--	--	--	--

```

*****  
initial state/s is/are :- 7  
final state/s is/are :- 14

```

TRANSITION TABLE FOR DFA

States	a	b
q0	q1	q2
q2	q1	q2
q1	q1	q3
q3	q1	q4
q4	q1	q2

```

*****  
initial state/s is/are :- q0  
final state/s is/are :- q4

```

```

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

```

Figure 19: RE To NFA transition table and NFA to DFA transition 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/1ZMTr44Ydz4uZPu7hInJGKhuQ5ZHI4bbK/view>
- [8] [https://www.geeksforgeeks.org/compiler-design-ll1-parser-in-python/#:~:text=LL\(1\)%20grammar,follows%20Top%2Ddown%20parsing%20method](https://www.geeksforgeeks.org/compiler-design-ll1-parser-in-python/#:~:text=LL(1)%20grammar,follows%20Top%2Ddown%20parsing%20method)

Part (2)

Introduction	
Syntax Analyzer	
Parsers	
LL (1)	
Modify the Grammar.....	
LL (first, follow, table, tree)	
Project code.....	
Output.....	

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

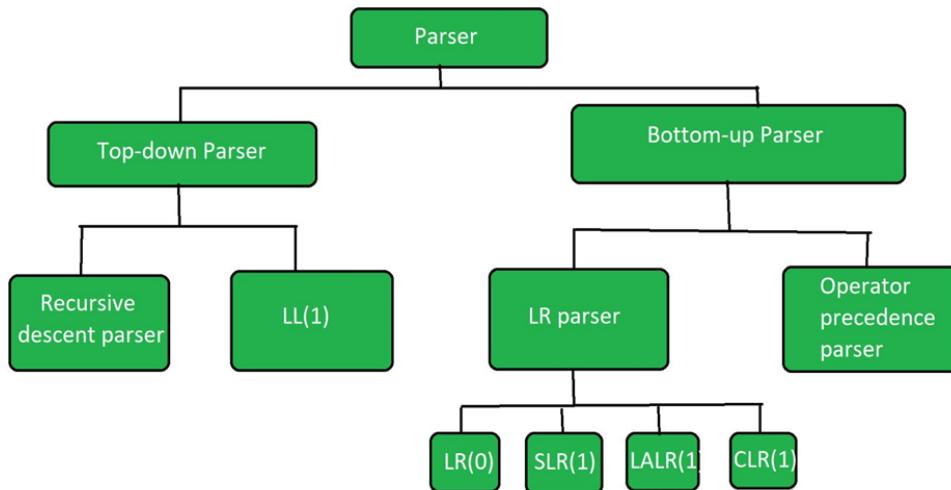


Figure 21: Parser type

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 $A \rightarrow \alpha$. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry $A \rightarrow \alpha$ in the table.

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

If the First(α) contains ϵ and Follow(A) contains \$ as terminal, then make entry $A \rightarrow \epsilon$ 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 how 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 :

$p \rightarrow c$
 $c \rightarrow s \ c'$
 $c' \rightarrow \# \mid ; \ s$
 $s \rightarrow id = E$
 $E \rightarrow T \ E'$
 $E' \rightarrow + \ T \ E' \mid - \ T \ E' \mid \#$
 $T \rightarrow B \ T'$
 $T' \rightarrow * \ B \ T' \mid / \ B \ T' \mid \#$
 $F \rightarrow [+] \ F \mid [-] \ F \mid (\ E) \mid id \mid integr$
 $B \rightarrow F \ B'$
 $B' \rightarrow ^\wedge F \ B' \mid \#$

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 production

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

Create 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-> integer		F-> [+] F	F-> [-] F	
B			B-> FB'					B-> FB'		B->F B'		B-> FB' B'	B-> F B'	

B'		$B' -> \#$		$B' -> \#$	$B' -> \#$	$B' -> \#$	$B' -> \#$		$B' -> \#$		$B' \rightarrow ^F B'$			$B' -> \#$
------	--	------------	--	------------	------------	------------	------------	--	------------	--	------------------------	--	--	------------

Table 3: Parser table

Test correct input

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

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

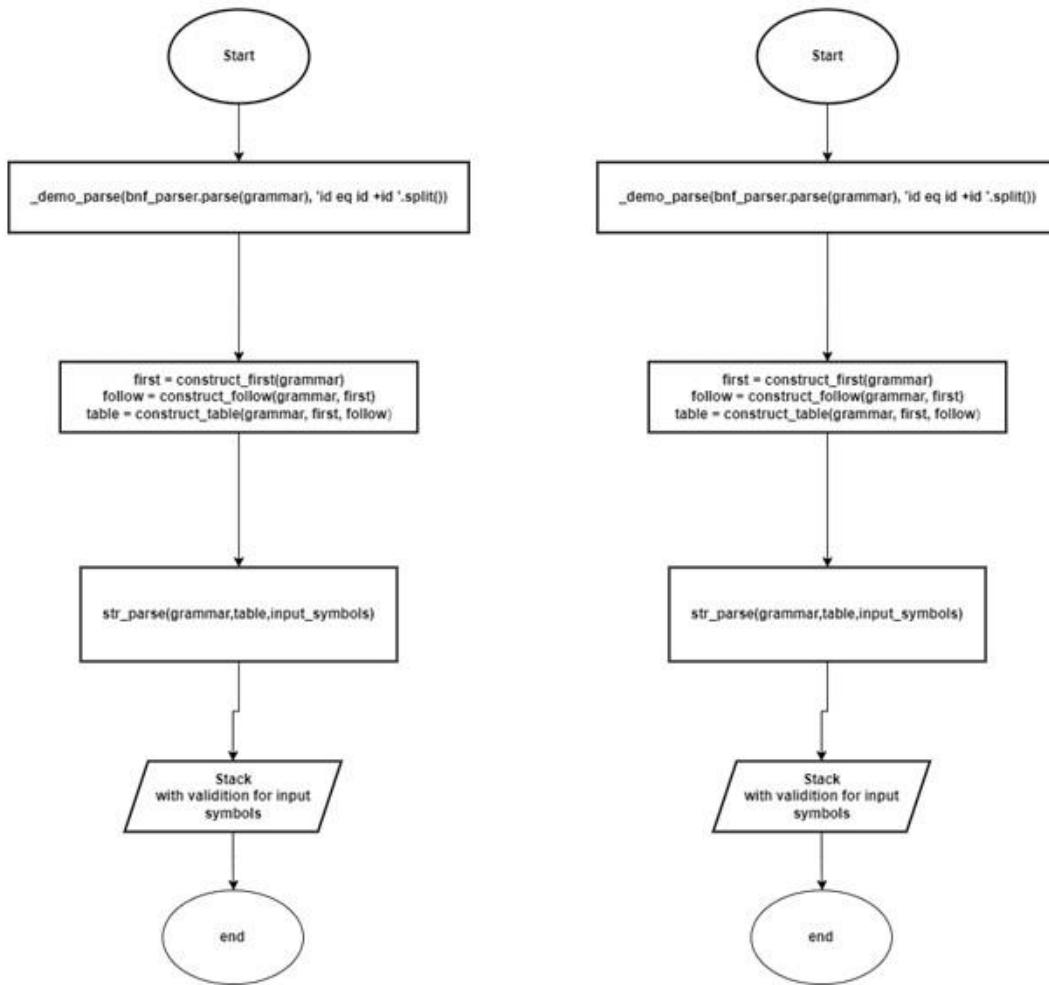


Figure 22:illustrative Diagram of LL(1) parser

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.

```
Help      ll1.py  New folder (2)  Visual Studio Code
ll1.py  X
ll1.py > ...
1  import bnf_parser
2  from grammar import Grammar, Production
3
4
5  def _update_set(dst: set, to_add: set) -> bool:
6      old_len = len(dst)
7      dst.update(to_add)
8      return old_len != len(dst)
9
10 # calculation of first
11 # epsilon is denoted by '#' (semi-colon)
12 # pass rule in first function
13
14 def construct_first(grammar: Grammar) -> list:
15     def construct_first_nterm(nterm: str) -> bool:
16         changed = False
17         for prod in grammar.prods[nterm]:
18             for sym in prod.syms:
19
20                 if sym != '@' and tuple(first[sym]) != ('@',):
21                     if _update_set(first[nterm], first[sym]):
22                         changed = True
23                     break
24                 else:
25                     if _update_set(first[nterm], {'@'}):
26                         first[nterm].add('@')
27                         changed = True
28             return changed
29     # recursion base condition
30     # (for terminal or epsilon)
31     first = dict([(sym, set()) for sym in grammar.prods])
32     first['@'] = {'@'}
33     # EXTRACT()
```

Figure 23: LL(1) class code

```

27     |     |     |     changed = True
28     |     |     |
29     # recursion base condition
30     # (for terminal or epsilon)
31     first = dict([(sym, set()) for sym in grammar.prods])
32     first['@'] = {'@'}
33     # FIRST(a) = {a}
34     for term in grammar.terms:
35         first[term] = {term}
36
37     changed = True
38     while changed:
39         changed = False
40         # condition for Non-Terminals
41         for nterm in grammar.prods:
42             if construct_first_nterm(nterm):
43                 changed = True
44     return first
45
46
47     # returns {'@'} on empty syms
48     def get_first_from_syms(first: list, syms: list) -> set:
49         # First, assume that eps is already in FIRST
50         result = {'@'}
51         for sym in syms:
52             # save result in 'result' list
53             result.update(first[sym])
54             if '@' not in first[sym]:
55                 # If the eps transition link stops here, remove eps
56                 return result - {'@'}
57         # Eps transition link doesn't stop till end, keep it
58         return result
59

```

Figure 24: LL(1) class code

```

❷ ll1.py > ...
59
60     # calculation of follow
61     def construct_follow(grammar: Grammar, first: list) -> list:
62         def construct_follow_nterm(prod: Production) -> bool:
63             changed = False
64             i = 0
65             while i < len(prod.syms):
66                 nterm = prod.syms[i]
67                 # Only process nonterminals
68                 if not grammar.is_nonterminal(nterm):
69                     i += 1
70                     continue
71                 i += 1
72
73                 remaining_syms = prod.syms[i:]
74                 remaining_first = get_first_from_syms(first, remaining_syms)
75                 if _update_set(follow[nterm], remaining_first - {'@'}):
76                     changed = True
77                 if '@' in remaining_first:
78                     if _update_set(follow[nterm], follow[prod.nterm]):
79                         changed = True
80             return changed
81
82         follow = dict([(sym, set()) for sym in grammar.prods])
83         # follow start symbol =>
84         follow[grammar.start] = {'$'}
85         changed = True
86         while changed:
87             changed = False
88             for prolist in grammar.prods.values():
89                 for prod in prolist:
90                     if construct_follow_nterm(prod):
91                         changed = True
92
93             else:
94                 table[nterm].append((term, prod))
95
96
97     return table
98
99
100
101    def construct_conflicts(table: dict) -> list:
102        # result[index] = tuple(nonterm, term, list(Productions))
103        result = list()
104        for nterm, pairs in table.items():
105            terms = set([pair[0] for pair in pairs])
106            for term in terms:
107                prods = list(filter(lambda pair, term=term: pair[0] == term, pairs))
108                if len(prods) != 1:
109                    result.append((nterm, term, [pair[1] for pair in prods]))
110
111    return result
112
113
114    # writing first ,follow
115    def _str_first_or_follow(grammar: Grammar, first: list, title) -> str:
116        result = ' ' + title + ':'
117        for sym in grammar.prods:
118            result += '\n    {}: {}'.format(sym, ' '.join(first[sym]))
119
120        return result
121
122
123    #writing follow in list
124    def str_follow(grammar: Grammar, follow: list) -> str:
125        return _str_first_or_follow(grammar, follow, 'FOLLOW')
126
127
128    #writing first in list
129    def str_first(grammar: Grammar, first: list) -> str:
130        return _str_first_or_follow(grammar, first, 'FIRST')
131
132
133
```

Figure 25: LL(1) class code

```
ll1.py > ...
137     #writing table
138     def str_table(table: dict) -> str:
139         result = ' Table:'
140         for nterm, pairs in table.items():
141             result += '\n      {}:{'.format(nterm)
142             for term, prod in sorted(pairs, key=lambda x: x[0]):
143                 result += '\n          {}:{} {}'.format(term, prod)
144         return result
145
146
147     def str_conflicts(conflicts: list) -> str:
148         result = ''
149         for nterm, term, prods in conflicts:
150             result += '\n      {}, {}:{'.format(nterm, term)
151             for prod in prods:
152                 result += '\n          ' + str(prod)
153         if result:
154             return ' Conflicts:' + result
155         else:
156             return ' Conflicts: None'
157
158     # writing ll1 table
159     def str_ll1(grammar: Grammar) -> str:
160         first = construct_first(grammar)
161         follow = construct_follow(grammar, first)
162         table = construct_table(grammar, first, follow)
163         conflicts = construct_conflicts(table)
164
165         result = 'LL(1):'
166         result += '\n' + str(first)
167         result += '\n' + str(follow)
168         result += '\n' + str_table(table)
```

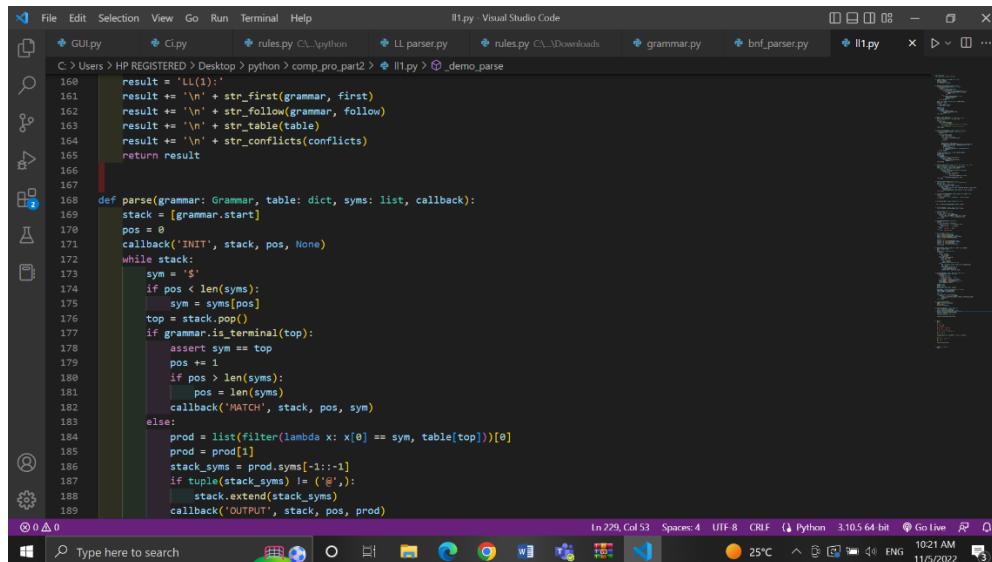


Figure 26: LL(1) class code

```

190
191
192 def str_parse(grammar: Grammar, table: dict, syms: list):
193     def callback(action, stack, pos, info):
194         str_action = ''
195         if action == 'MATCH':
196             str_action = 'match ' + info
197         elif action == 'OUTPUT':
198             str_action = 'output ' + str(info)
199         inputs.append(''.join(syms[pos:]))
200         stacks.append(''.join(stack[-i:-1]) + '$')
201         actions.append(str_action)
202
203     inputs = list()
204     stacks = list()
205     actions = list()
206     parse(grammar, table, syms, callback)
207
208     get_length = lambda arr: max([len(t) for t in arr])
209     inputs_length = get_length(inputs)
210     stacks_length = get_length(stacks)
211
212     result = ''
213     for i, input_val in enumerate(inputs):
214         result += '{0} | {1}\n'.format(
215             input_val.rjust(inputs_length), stacks[i].rjust(stacks_length),
216             actions[i])
217
218     return result

```

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.

```

3   from collections import defaultdict
4
5   class Grammar:
6       def __init__(self):
7           self.start = None
8           self.terms = set() # terminals
9           # prods[nterm] = list of Production objects
10          self.prods = defaultdict(list) # new empty list
11
12      def add_production(self, nterm, syms):
13          production = Production(nterm, syms)
14          self.prods[nterm].append(production) # add prod in list
15
16      def is_nonterminal(self, symbol: str) -> bool: # if symbol non terminal in prod
17          return symbol in self.prods
18
19      def is_terminal(self, symbol: str) -> bool: # if symbol terminal in list of term
20          return symbol in self.terms
21
22      def get_alt_nonterminal(self, nonterm: str) -> str:
23          nonterm += '-'
24          while nonterm in self.prods: # loop in prod for non term with ' dash
25              nonterm += '-'
26          return nonterm
27
28      def get_start_prodictions(self): # get a star symbol

```

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.

```

File Edit Selection View Go Run Terminal Help grammar.py - Visual Studio Code
C:\> Users > HP REGISTERED > Desktop > python > comp_pro_part2 > grammar.py > $ Production > __str__
32     from copy import deepcopy
33     return deepcopy(self)
34
35     def __str__(self): # print
36         result = "Grammar:\n"
37         result += " Start: " + self.start
38         result += " Terminals: " + " ".join(self.term)
39         result += " NonTerminals: " + " ".join(self.prods.keys())
40         result += " Productions:"
41         for prodlist in self.prods.values(): # loop in values of prods
42             for prod in prodlist: # ??
43                 result += "\n    " + str(prod)
44
45
46     class Production: # remove eps from non term
47         def __init__(self, nterm: str, syms: list):
48             self.nterm = nterm
49             self.syms = Production.remove_eps(syms)
50
51         def __eq__(self, other): # if non term and not eps
52             return self.nterm == other.nterm and self.syms == other.syms
53
54         def __ne__(self, other): # ??
55             return not self == other
56
57         def __hash__(self): # add all symbols to hash
58             result = hash(self.nterm)
59             for sym in self.syms:
60                 result ^= hash(sym)
61
62     result += "\n"
63     return result
64
65     @staticmethod
66     def remove_eps(syms: list) -> list:
67         for sym in syms: #loop in list
68             if sym == '@': # if there @
69                 break
70             else:
71                 return ['@']
72         return list(filter(lambda x: x != '@', syms)) # remove
73
74     def __repr__(self): # return production
75         return "Production({},{})".format(self.nterm, self.syms)
76
77     def __str__(self): #return production
78         return "{} + {}".format(self.nterm, " ".join(self.syms))

```

Figure 29:grammar class code

```

File Edit Selection View Go Run Terminal Help grammar.py - Visual Studio Code
C:\> Users > HP REGISTERED > Desktop > python > comp_pro_part2 > grammar.py > $ Production > __str__
61         result += "\n"
62         return result
63
64     @staticmethod
65     def remove_eps(syms: list) -> list:
66         for sym in syms: #loop in list
67             if sym == '@': # if there @
68                 break
69             else:
70                 return ['@']
71         return list(filter(lambda x: x != '@', syms)) # remove
72
73     def __repr__(self): # return production
74         return "Production({},{})".format(self.nterm, self.syms)
75
76     def __str__(self): #return production
77         return "{} + {}".format(self.nterm, " ".join(self.syms))

```

Figure 30: grammar class code

```
C:\> Users > HP REGISTERED > Downloads > bnf_parser.py > _BNFParser > tokenize
1 from enum import Enum
2 from grammar import Grammar
3
4
5 class _BNFParser:
6     ...
7     bnf := prod end | prod bnf
8     prod := nterm ':=' rhs
9     sym := sym | sym sym
10    rhs := sym | sym sym '|' rhs
11    ...
12
13    class Token(Enum):
14        END = 1
15        SYM = 2
16        KEYWORD = 3
17
18        def __init__(self, buf: str):
19            self.pos = 0
20            self.tokens = self.tokenize(buf)
21            self.grammar = Grammar()
22            self.parse_bnf()
23            self.fix_grammar()
24
25        def peek(self): # return tokens(SYM) with their end (END)
26            if self.pos < len(self.tokens):
27                return self.tokens[self.pos]
28            return (self.Token.END, None)
29
30        def get(self, n=1): # get the next token
```

Figure 31: bnf class code

```
C:\> Users > HP REGISTERED > Downloads > bnf_parser.py > _BNFParser > parse_rhs
31         result = self.peek()
32         self.pos += n
33         return result
34
35     def unget(self, n=1): # to reduce the token
36         self.pos -= n
37
38     def fix_grammar(self): # update production dic with the new added production
39         all_syms = set()
40         for prodict in self.grammar.prods.values():
41
42             for prod in prodict:
43                 all_syms.update(prod.syms)
44         self.grammar.terms = all_syms - self.grammar.prods.keys() - set('@')
45
46
47     def parse_rhs(self): # to get the right hans side of the production
48         result = list()
49         while True:
50             token = self.get() #get token a
51             if token[0] == self.Token.END: # if END then delete it from tokens
52                 self.unget()
53                 return result
54             if token[0] == self.Token.KEYWORD: # if KEYWORD check if | then return the list
55                 if token[1] == '|': # to divide the production into separate rules such A->a|b ==> A->a, A->b
56                 return result
57             else:
58                 raise SyntaxError("Unexpected token " + str(token))
59
60             result.append(token[1]) # append the right hans sid to the list
```

Figure 32: bnf class code

```
File Edit Selection View Go Run Terminal Help bnf_parser.py - Visual Studio Code
C:\Users> HP REGISTERED > Downloads > bnf_parser.py > _BNFParser > token
61
62     def parse_prod(self): # get all production in the our grammar
63         token = self.get() # get the non-terminals
64
65         if token[0] != self.Token.SYM: # ensure it's a non-terminal
66             raise SyntaxError("Nonterminal expected, got " + str(token))
67         nterm = token[1] # store the non-terminal such P , B
68
69         if not self.grammar.start: # assian the fist non-terminal to start and add condition to not changed every iteration
70             self.grammar.start = nterm
71
72         token = self.get() # get move to the next token which is the keyword := to get the right hand side by parse_rhs()
73
74         if token[0] != self.Token.KEYWORD or token[1] != '=': # check the next token is keyword to reach the hand right side
75             raise SyntaxError("Keyword := expected, got " + str(token))
76
77         while True:
78             if self.peek()[0] == self.Token.END: # if the end of production then stop
79                 return
80             prod_list = self.parse_rhs() # get the right hand side of the non-terminal
81
82             if not len(prod_list):
83                 raise SyntaxError("Empty right hand side of production "
84                     "for nonterminal " + nterm)
85             self.grammar.add_production(nterm, prod_list) # add it to the grammmer as production
86
87     def parse_bnf(self): #extract the non-terminal
88         while True:
89             token = self.peek() # traverse over non-terminal token and thier end
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
```

Figure 33: bnf class code

```
File Edit Selection View Go Run Terminal Help bnf_parser.py - Visual Studio Code
C:\Users> HP REGISTERED > Downloads > bnf_parser.py > _BNFParser > tokenize
91         if token[0] == self.Token.END: # if the first token is the end of terminal then check if the second is not the non-terminal tok
92             if token[1] is None: # if the end of grammar out
93                 return
94             else:
95                 self.get() # get the next token and it's END
96             else:
97                 self.parse_prod() # parse the non-terminal only
98
99 # this method is used to define the grammer pattern to identify the the pattern of our regular expression
100 # it recognize the whitespace and the keyword (| , := ) and the input symbols
101
102     def tokenize(self, buf: str) -> list:
103         result = list()
104         import re # import re module which contain the functions needed for handling patterns and regular expressions.
105         #save the common grammer pattern in a variable to compare it later and find the production for the program
106         regexps_space = re.compile("[ \t]+")
107         #print(regexps_space)
108         regexps = (
109             (self.Token.KEYWORD, re.compile(r"\|\|:=|"), # link KEYWORD value with it's pattern
110             (self.Token.SYM, re.compile(r"[ \t\r\n]+"),# link SYM value with it's pattern
111             (self.Token.END, re.compile(r"[ \t\r\n]+"),# link END value with it's pattern
112             )
113
114         #print("regrops \n")
115         #print(regexps)
116
117         i = 0
118         while i < len(buf): # traverse over our grammer
119             # Skip spaces
120
```

Figure 34: bnf class code

```
File Edit Selection View Go Run Terminal Help bnf_parser.py - Visual Studio Code
C:\Users\HP REGISTERED> Downloads > bnf_parser.py > _BNFParser > tokenize
119     # Skip spaces
120
121     m = re.compile_space.match(buf, i) # Check if the given productions matches the pattern that was previously defined at index i if
122     if m:
123         #print("in tokenize if \n") # m= <re.Match object; span=(6, 8), match=' '> it contain locations at which the match st
124         #print(m)
125         i = m.end() # update the index to check the seond production
126         #print("in token i \n")
127         #print(i)
128     if i == len(buf): # end the grammar
129         break
130
131     for token, regexp in regexps:
132         m = regexp.match(buf, i) # find the KEYWORD and SYM and END
133
134         if m:
135             if token == self.Token.SYM: # if token is SYM(terminal and non terminal symbols)
136
137                 if m.group() == r"\": # check if the token name has dash
138                     result.append((self.Token.SYM, ""))
139                 else:
140                     result.append((token, m.group())) # else add token whih it's value without dash
141
142             else:
143                 result.append((token, m.group())) # if KEYWORD and END
144             i = m.end() # save the end of match in i such as m= <re.Match object; span=(6, 8), match=' ',8 is the end of match
145             break
146
147         else:
148             raise SyntaxError("Unknown token at pos {} ({})".format(i, buf[i:i + 18]))
149
150
151
152
153     return result
154
155 def parse(bnf: str) -> Grammar: # senf the grammar to start recognize it's pattern and retrun the grammar after know it's pattern
156     parser = _BNFParser(bnf)
157     return parser.grammar
158
159 def main():
160     #print(parse(_BNFParser.__doc__))
161
162     main()
163
```

Figure 35: bnf class code

```
File Edit Selection View Go Run Terminal Help bnf_parser.py - Visual Studio Code
C:\Users\HP REGISTERED> Downloads > bnf_parser.py > _BNFParser > tokenize
149
150
151
152
153     return result
154
155 def parse(bnf: str) -> Grammar: # senf the grammar to start recognize it's pattern and retrun the grammar after know it's pattern
156     parser = _BNFParser(bnf)
157     return parser.grammar
158
159 def main():
160     #print(parse(_BNFParser.__doc__))
161
162     main()
163
```

Figure 36: bnf class code

Grammar:

```

Grammar:
Start: P
Terminals: - ; * int n / ( + ) eq b id p
Nonterminals: P C C' S E E' T T' F B B'
Productions:
P → C
C → S C'
C' → @
C' → ;
S → id eq E
E → T E'
E' → + T E'
E' → - T E'
E' → @
T → B T'
T' → * B T'
T' → / B T'
T' → @
F → p F
F → n F
F → ( E )
F → id
F → int
B → F B'
B' → b F B'
B' → @

```

Figure 37 screenshot of rules

The First and the follow of each production

```

FIRST:
P: id
C: id
C': id
S: id
E: int n ( id p
E': @ - +
T: int n ( id p
T': * / @
F: id int n ( p
B: int n ( id p
B': b @
*****



FOLLOW:
P: $
C: $
C': $
S: $
E: $ ;
E': $ ;
T: $ - ;
T': $ - ;
F: $ - ; * / +
B: $ - ; * /
B': $ - ; * /

```

Figure 31:the first and the follow of each production

Parser table:

Table:	
P:	$\text{id}: P \rightarrow C$
C:	$\text{id}: C \rightarrow S \ C'$
C':	$\$: C' \rightarrow \emptyset$
	$;: C' \rightarrow ; \ S$
S:	$\text{id}: S \rightarrow \text{id eq E}$
E:	$(: E \rightarrow T \ E'$ $\text{id}: E \rightarrow T \ E'$ $\text{int}: E \rightarrow T \ E'$ $n: E \rightarrow T \ E'$ $p: E \rightarrow T \ E'$ E' : $\$: E' \rightarrow \emptyset$ $;: E' \rightarrow \emptyset$ $+: E' \rightarrow + \ T \ E'$ $-: E' \rightarrow - \ T \ E'$ $\cdot: E' \rightarrow \cdot \ T \ E'$
T:	$(: T \rightarrow B \ T'$ $\text{id}: T \rightarrow B \ T'$ $\text{int}: T \rightarrow B \ T'$ $n: T \rightarrow B \ T'$ $p: T \rightarrow B \ T'$
T':	$\$: T' \rightarrow \emptyset$ $;: T' \rightarrow \emptyset$
T' :	$\$: T' \rightarrow \emptyset$ $;: T' \rightarrow \emptyset$ $*: T' \rightarrow * \ B \ T'$ $+: T' \rightarrow \emptyset$ $-: T' \rightarrow \emptyset$ $/: T' \rightarrow / \ B \ T'$ $\cdot: T' \rightarrow \emptyset$
F:	$(: F \rightarrow (\ E)$ $\text{id}: F \rightarrow \text{id}$ $\text{int}: F \rightarrow \text{int}$ $n: F \rightarrow n \ F$ $p: F \rightarrow p \ F$
B:	$(: B \rightarrow F \ B'$ $\text{id}: B \rightarrow F \ B'$ $\text{int}: B \rightarrow F \ B'$ $n: B \rightarrow F \ B'$ $p: B \rightarrow F \ B'$
B' :	$\$: B' \rightarrow \emptyset$ $;: B' \rightarrow \emptyset$ $*: B' \rightarrow \emptyset$ $+: B' \rightarrow \emptyset$ $-: B' \rightarrow \emptyset$ $/: B' \rightarrow \emptyset$ $\cdot: B' \rightarrow \emptyset$ $b: B' \rightarrow b \ F \ B'$

Figure 38 parser table

Test input valid and invalid:

```

input symbol :
['id', 'eq', 'id', '+', 'id', '*', 'id']

id eq id + id * id $ | P $ | output P -> C
id eq id + id * id $ | C $ | output C -> S 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 $ | output P -> C
id eq id b n $ | C $ | output C -> S 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