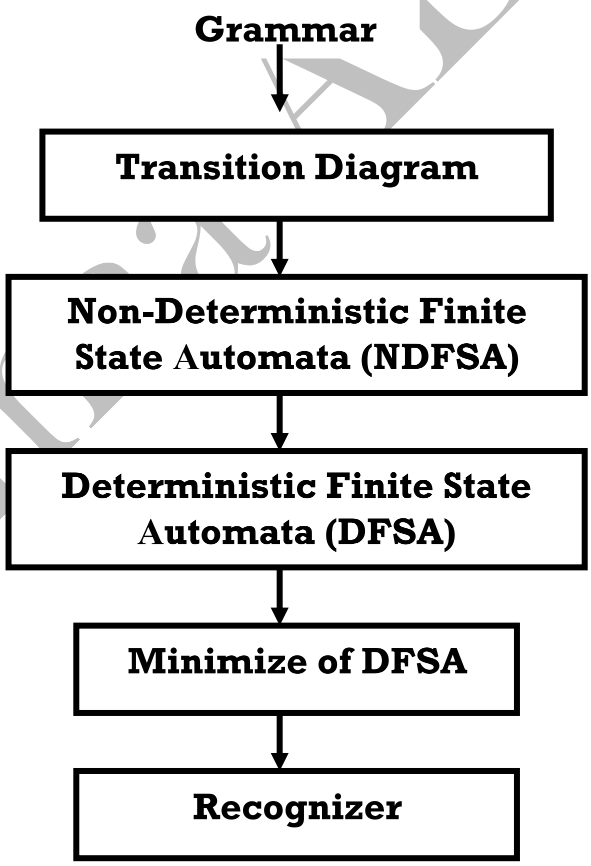
University of Baghdad
College of Education for Pure Science
Ibn-AL-Haithem/ Dep. Of Computer Science
M.Sc. Shaimaa Al-Obaidy

Lexical Analyzer Design

Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

The main sub-phases of the Lexical analyzer phase are shown below in the following figure:-



- **The grammar will converted to a Transition Diagram using special algorithm.**
- **The converted Transition Diagram must be checked whether if it is in NDFSA form or not; if so, the grammar must converted to DFSA using algorithm which will be described in this chapter.**
- **The resulted grammar will be in DFSA form which must be minimized to reduce the number of nodes depending on algorithm designed for this purpose (fast searching and minimum memory storage).**
- **The final sub-phase in lexical analyzer phase is to recognize if the input string or statement is accepted or not depending on a specific grammar.**

***Finite State Automata* (FSA):-**

Is a mathematical model consists of:-

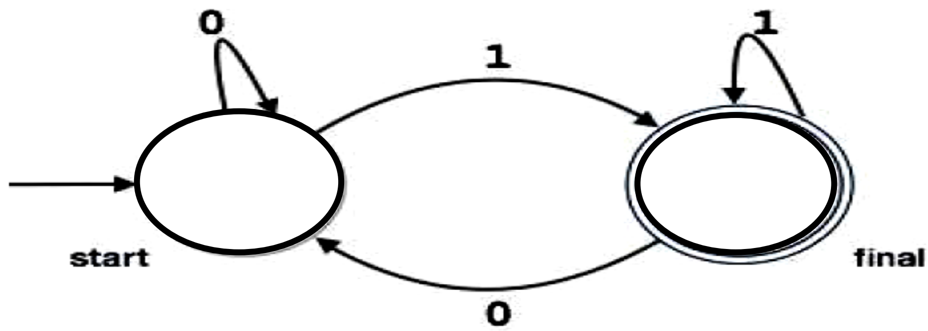
1. **A set of terminal symbols**
2. **Transition functions**
3. **One-Initial state (Start state)**
4. **One or Set of Final states**
5. **Finite set of elements called states**

States : States of FSA are represented by circles. State names or numbers are written inside circles.

Start state : The state from where the FSA starts, is known as the start state. Start state has an arrow pointed towards it.

Final State :- If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles, it is also called the Accepting State.

A transition :- Is denoted by an arrow connecting two states, the arrow is labeled by the symbol (possibly e). The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state.

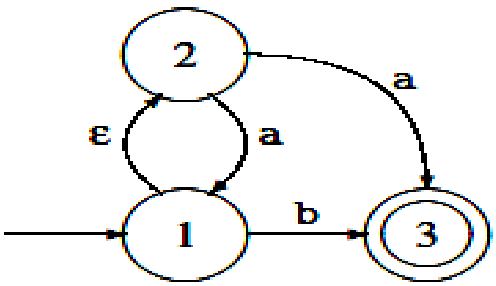
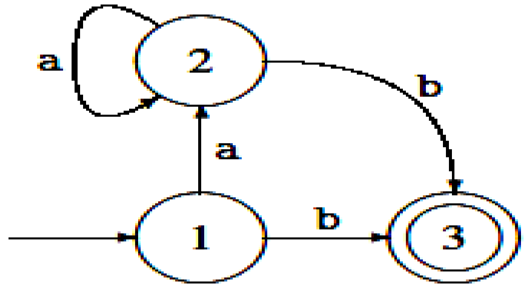


Two types of *FSA* :-

- **Non-Deterministic Finite State Automata (NDFSA)**
- **Deterministic Finite State Automata (DFSA)**

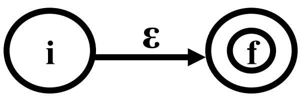
***FSA* is of NDFSA if one of these two conditions is satisfied:-**

1. **There are more than one transition have the same label from that state to another states.**
2. **There is a Ɛ - transition.**

A transition, represent FSA of type NDFSA.	A transition, represent FSA of type DFSA.
	

Formal method for converting R.E. to NDFSA :-

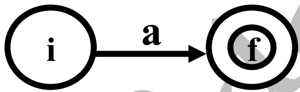
① If we have an R.E.= ε then the NDFSA will be as follows:-



where i = initial state , f =final state

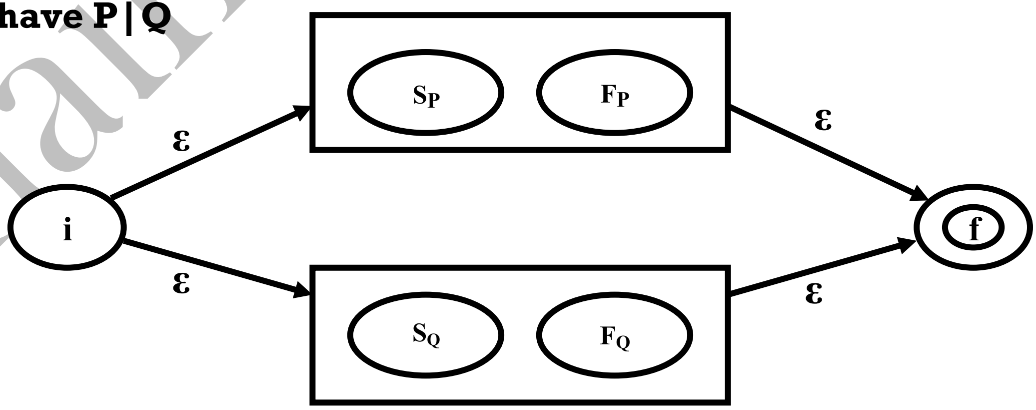
....

② If we find a terminal symbol like a, then the NDFSA will be as follows:-

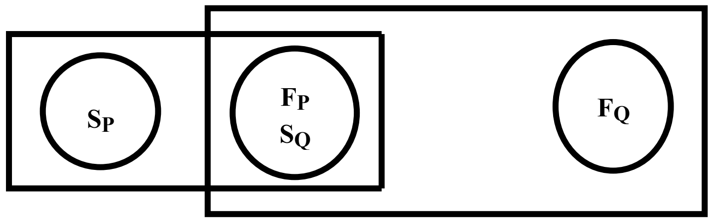


....

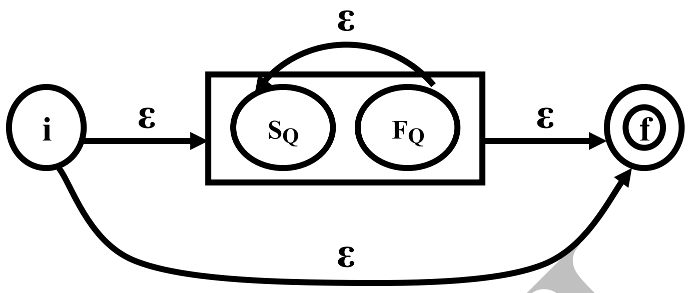
③ If we have P | Q



④ **If we have P.Q**

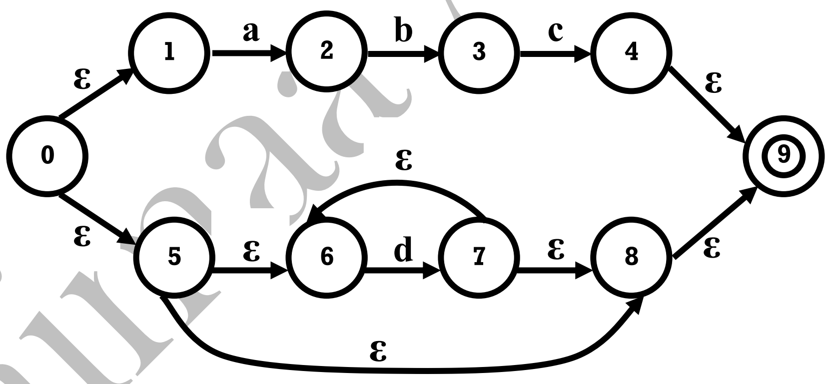


⑤ **If we have Q***



Example **:-**

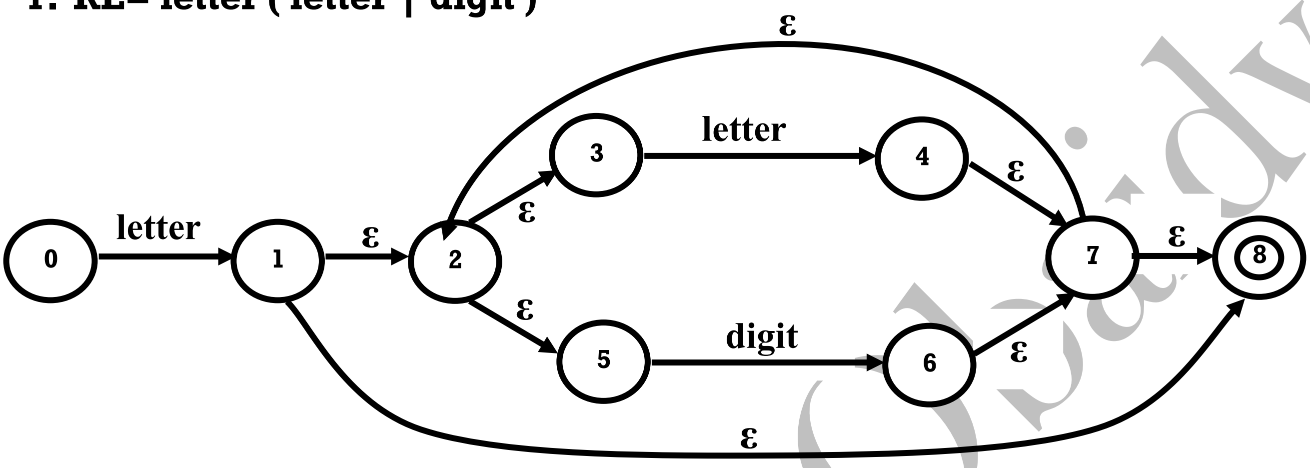
R.E.= abc|d*



……………………………………

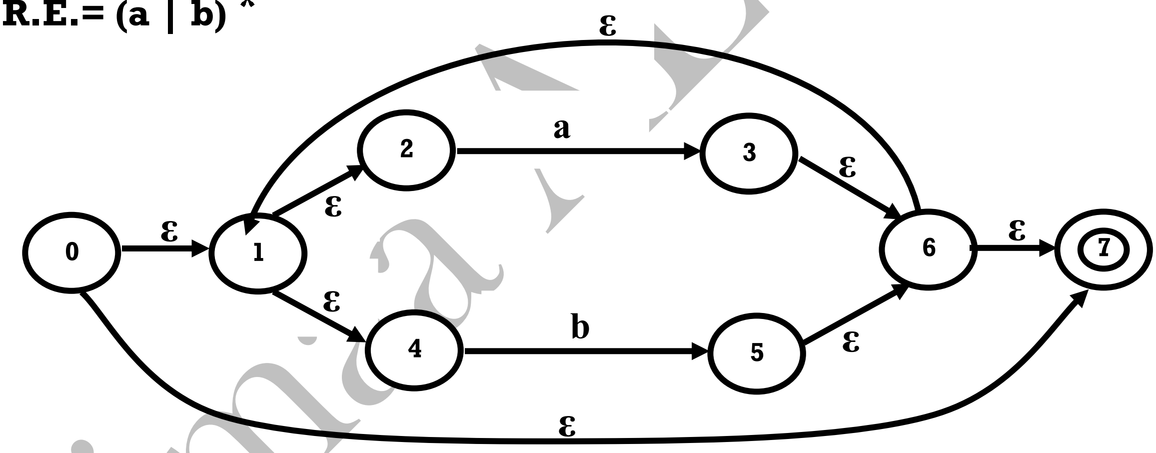
Examples :-

1. **RE= letter (letter | digit)***



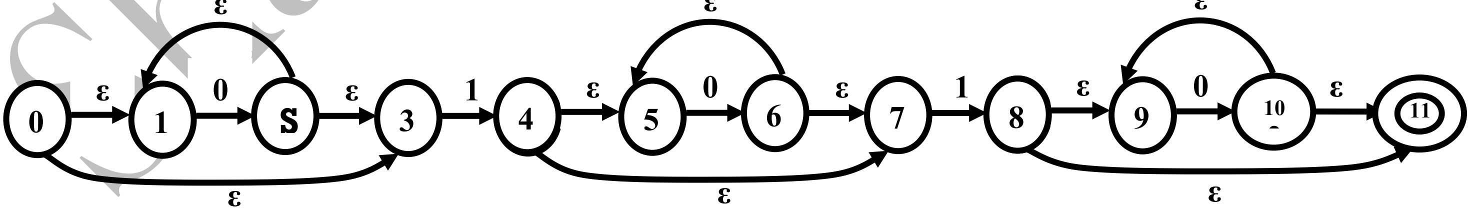
....

2. **R.E.= (a | b) ***

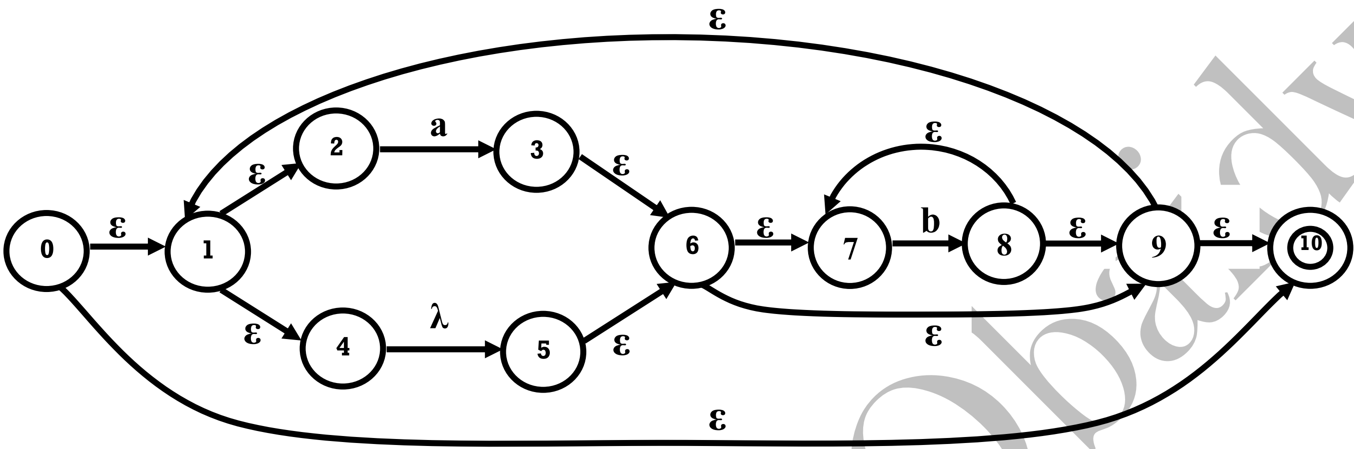


....

3. **R.E.= $0^*1\ 0^*1\ 0^*$**



4. R.E.= $((\lambda \mid a)\, b^*)^*$



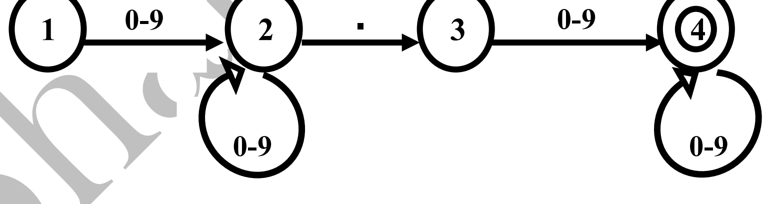
Data structure representation of FSA :-

① *Transition Matrix*

We must have a matrix with the number of its rows equal to the number of the FSA states in the diagram while the number of its columns in this matrix equal to the number of its inputs (labels).

This type of representation has a disadvantage that it contains many blank spaces, while the advantage of this type is that the indexing is fast.

For example:-

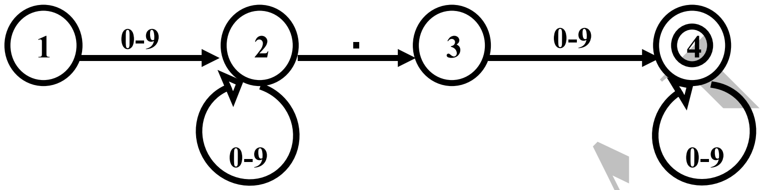


	0-9	.
1	2	#
2	2	3
3	4	#
4	4	#

② ***Graph Representation***

In this representation we have a fixed number of columns which is equal to 2 and the labels of these two columns are *Input Symbol* & *Next State* while the number of rows differs from one transition diagram to another and these rows are labeled by the number of states. The disadvantage of this representation is that it takes a long time for searching (search slow) while the advantage of this representation is that it is compact.

***For the previous example*:-**



	Input Symbol	Next State
1	0-9	2
2	0-9	2
2	.	3
3	0-9	4
4	0-9	4

Transformation of NDFSA to DFSA:-

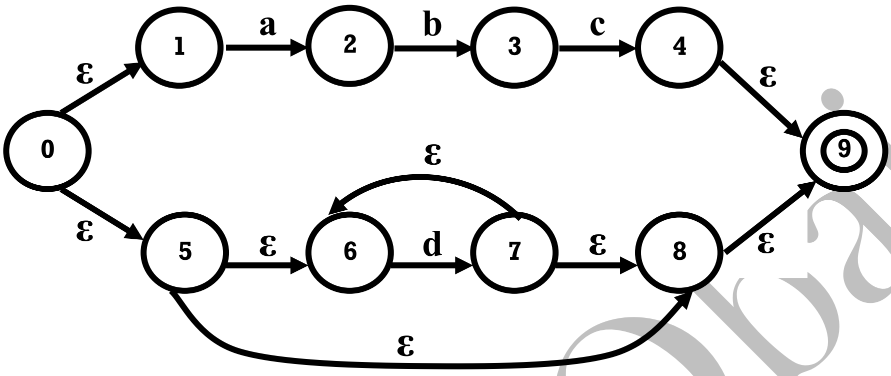
Before we use an algorithm to convert the grammar which is NDFSA form to DFSA form, we must deal with a special function known as Ɛ-*Closure Function*, which can be explained using the following procedure:-

***Function Ɛ-Closure* (M) :-**

```
Begin
  Push all states in M  into stack;
  Initialize Ɛ-Closure (M) to M;
  While stack is not empty do
     Begin
       Pop S;
       For each state X with an edge labeled Ɛ from S to X do
       If X is not in Ɛ-Closure (M) then
          Begin
            Push X;
            Add  X to Ɛ-Closure (M);
          End;
     End;
End;
```

Example **:-**

R.E.= abc|d*



To compute randomly the ε-Closure for the following states:-

ε-Closure ({0}) = {0, 1, 5, 6, 8, 9}

ε-Closure ({1}) = {1}

ε-Closure ({7, 8}) = {7, 8, 9, 6}

ε-Closure ({2, 3, 4})={2, 3, 4, 9}

Algorithm for transforming NDFSA to DFSA:-

Initially let x= $\mathcal{E}$-Closure ({S_0}) marked as the start state of DFSA, S_0 is the start state of NDFSA;

While there is unmarked states X = {S_1, S_2, … ,S_n} of DFSA do

Begin

For each terminal symbol (a $\in$ Σ) do

Begin

Let M be the set of states to which there is transition on *a* from some states S_i in X ;

Y = $\mathcal{E}$-Closure ({ M });

If Y has not yet been added to the set of states of DFSA then make Y an unmarked state of DFSA;

Create an edge by adding a transition from X to Y labeled *a* if not present;

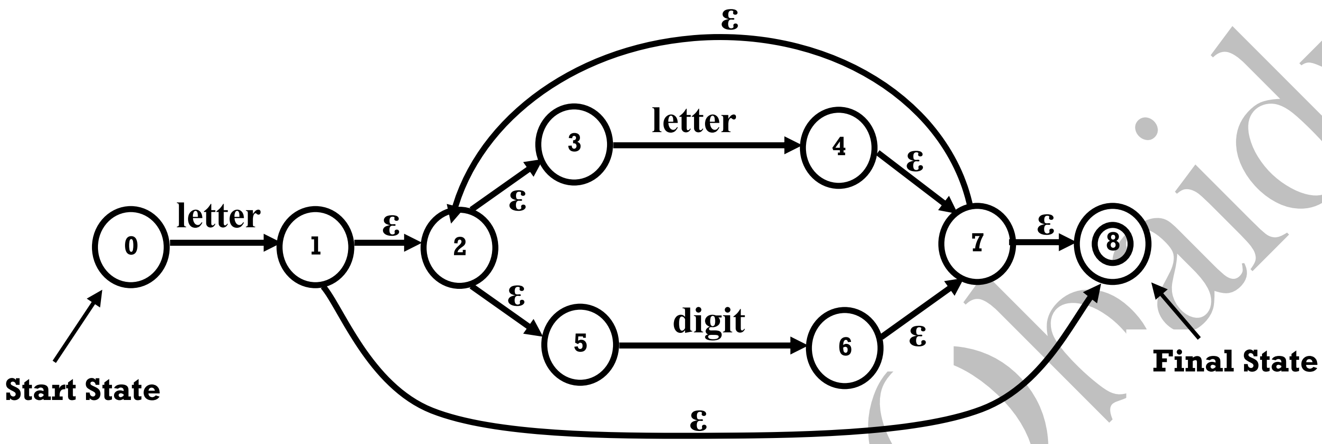
End;

End;

End {algorithm}

Examples:-

① **R.E. = Letter (letter | digit)***



ε-Closure ({ 0 }) = {0} ◂······ Create a new node called for example A

A → **letter** ; M={1}; **ε-Closure ({1})={1,2,3,5,8} ◂······ Create a new node called for example B** (must be a final node because of node 8).

A → **digit** ; M=∅;

B → **letter** ; M={4}; **ε-Closure ({4})={4,7,8,2,3,5} ◂······ Create a new node called for example C** (must be a final node because of node 8).

B → **digit** ; M={6}; **ε-Closure ({6})={6,7,8,2,3,5} ◂······ Create a new node called for example D** (must be a final node because of node 8).

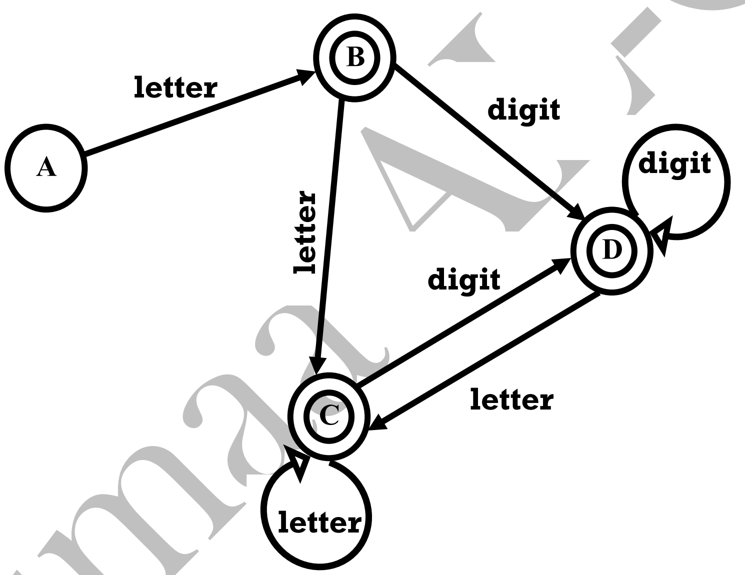
C → **letter** ; M={4}; **No need to create a new node because ε-Closure ({4}) has been computed and by which we have node C.**

C → **digit** ; M={6}; **No need to create a new node because ε-Closure ({6}) has been computed and by which we have node D.**

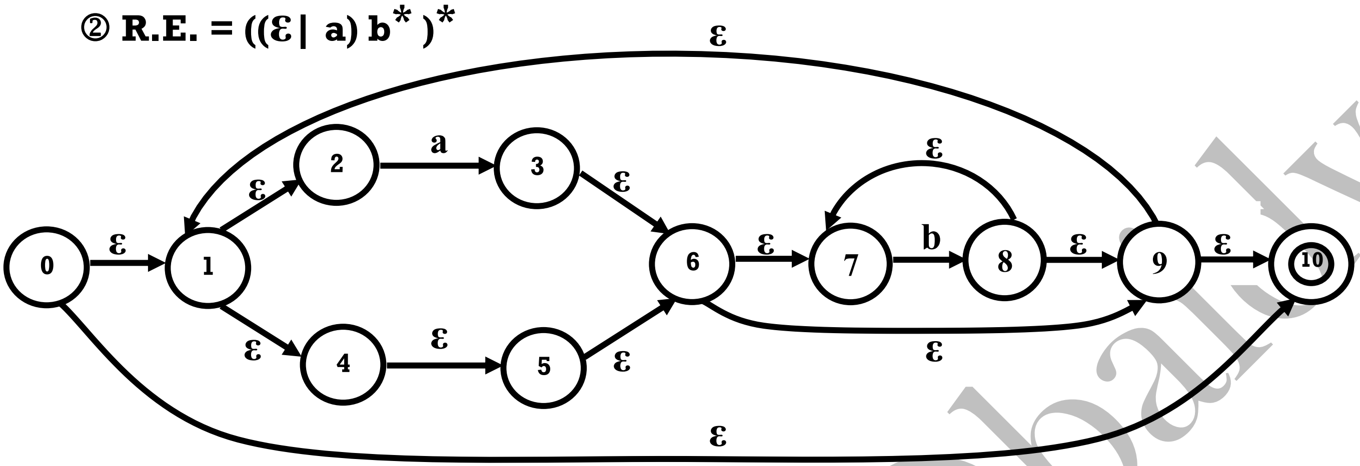
D → **letter** ; M={4}; **No need to create a new node because ε-Closure ({4}) has been computed and by which we have node <u>C</u>.**

→ **digit** ; M={6}; **No need to create a new node because ε-Closure ({6}) has been computed and by which we have node <u>D</u>.**

Since of no nodes will be created and all the created nodes have been manipulated, we will reach to the final step by which we have the DFSA, this step will convert all the above work into a graph as follows:-



② **R.E. = $((\varepsilon \mid a)\, b^{*})^{*}$**



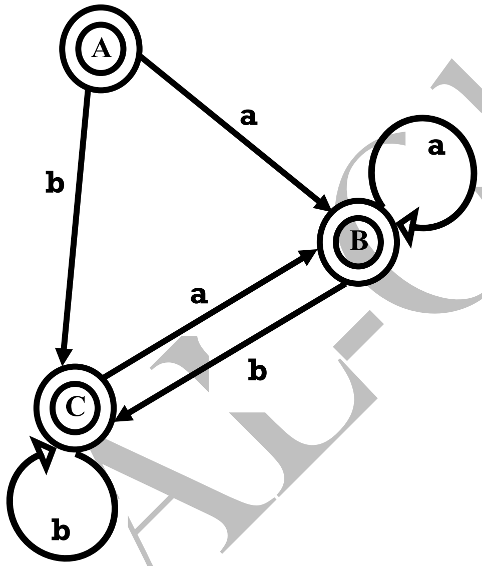
ε-Closure ({0}) = {0,1,2,4,5,6,7,9,<u>10</u>} ◂······· **Create a new node called for example <u>A</u>**(must be a final node because of node 10).

A
- **a** ; M={3}; **ε-Closure ({3})={3,6,7,9,<u>10</u>,1,2,4,5}** ◂······· **Create a new node called for example <u>B</u>** (must be a final node because of node 10).
- **b** ; M={8}; **ε-Closure ({8})={8,7,9,<u>10</u>,1,2,4,5,6}** ◂······· **Create a new node called for example <u>C</u>** (must be a final node because of node 10).

B
- **a** ; M={3}; **No need to create a new node because ε-Closure ({3}) has been computed and by which we have node <u>B</u>.**
- **b** ; M={8}; **No need to create a new node because ε-Closure ({8}) has been computed and by which we have node <u>C</u>.**

C
- **a** ; M={3}; **No need to create a new node because ε-Closure ({3}) has been computed and by which we have node <u>B</u>.**
- **b** ; M={8}; **No need to create a new node because ε-Closure ({8}) has been computed and by which we have node <u>C</u>.**

Since of no nodes will be created and all the created nodes have been manipulated, we will reach to the final step by which we have the DFSA, this step will convert all the above work into a graph as follows:-



③ R.E. = (a | b)*abb

University of Baghdad
College of Education for Pure Science
Ibn-AL-Haithem/ Dep. Of Computer Science
M.Sc. Shaimaa Al-Obaidy

Minimizing of DFSA:-

The purposes of minimization are:-

1. **Efficiency.**
2. **Optimal DFSA.**

Algorithm:-

1. **Construct an initial partition Л of the set of states with two groups: the accepting states F and the non accepting states S-F; where S is the set of all states of DFSA.**
2. **for each group G of Л do**

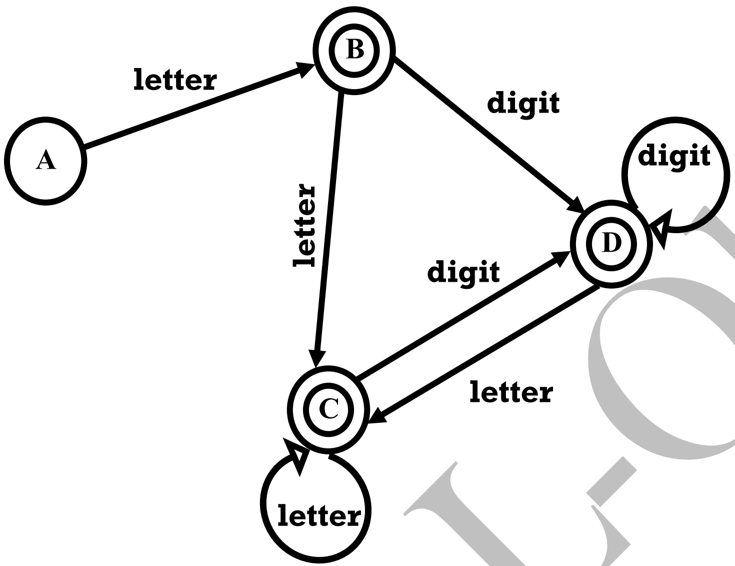
   Begin

   partition G into subgroups such that two states S and T of G are in the same subgroup if and only if for all input symbols *a*, and states S and T have transitions on *a* to states in the same group of Л, replace G in Л_{new} by the set of all subgroups formed .

   End
3. **If Л_{new} = Л, let Л_{final} = Л and continue with step (4), otherwise repeat step (2) with Л := Л_{new}**
4. **Choose one state in each group of the partition Л_{final} as the representative for that group.**

Example **:-**

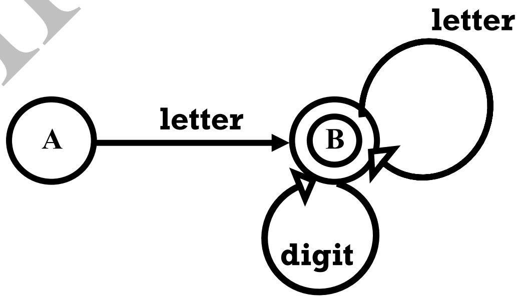
The DFSA for the R.E. = Letter (letter | digit)* is as follows:-



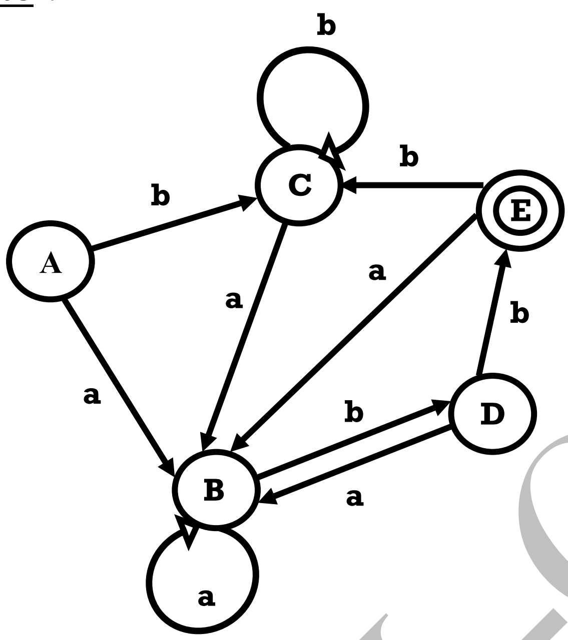
Group$_1$= {A} which represents the set of not final nodes while Group$_2$ = {B,C,D} which represents the set of final nodes.

Always minimization acts on the nodes of the same type (on the nodes of one group)

After applying the previous algorithm, the minimization figure will be as follows:-



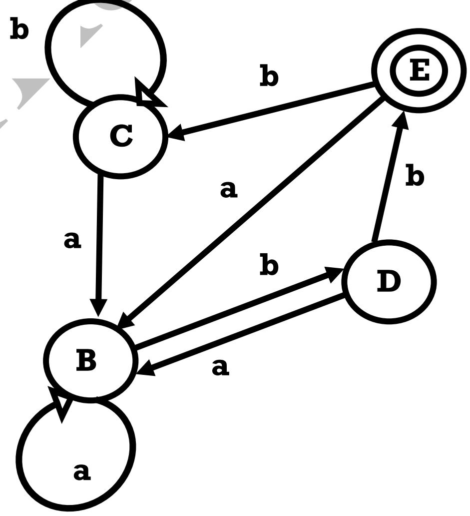
Another example **:-**



Group$_1$= {A,B,C,D} which represents the set of not final nodes while Group$_2$ = {E} which represents the set of final nodes.

Always minimization acts on the nodes of the same type (on the nodes of one group)

After applying the previous algorithm, the minimization figure will be as follows:-



***FSA Accepter (Recognizer)*:-**

This will represents the final sub-phase for the lexical analyzer ,by using a specific algorithm shown below we can specify the input string or statement is accepted or not depending on a given grammar.

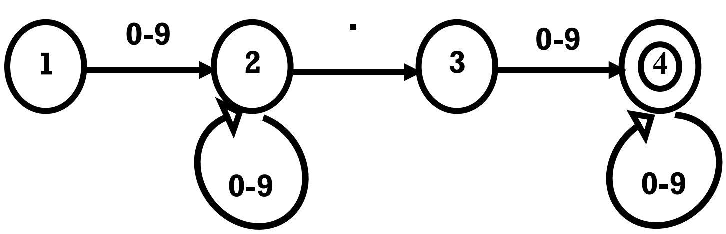
Never can apply the algorithm unless the grammar will be in *minimized form*.

First, a transition matrix must be created for a given FSA, then doing a table having two columns, the first represents the number of states while the other represents the symbols for a given input string.

***Algorithm* :-**

```
Begin
    State = Start State of the FSA;
    Symbol = First Input Symbol;
      If Matrix [State, Symbol] ≠ Error Indication then
          Begin
            State = Matrix [State, Symbol];
            Symbol = Next Input Symbol;
          End
      Else Input is not accepted
    If State is a Final State of FSA then Input is accepted
      Else Input is not accepted
End;
```

Example **:- Having the following FSA representation shown below:-**



Depending on the above representation, for <u>1.3$</u> and <u>37$</u> ,you asked to recognize which one is accepted and which one is not accepted?

***Solution*:-**

The *Transition Matrix* for the above FSA:-

	0-9	.
1	2	#
2	2	3
3	4	#
4	4	#

For the String = <u>1.3 $</u>

State	Input symbol
1	1
2	.
3	3
4	$

It is accepted because state number 4 is a final State

For the String = <u>37 $</u>

State	Input symbol
1	3
2	7
2	$

It is not accepted because state number 2 is not a final state and the expression is finished

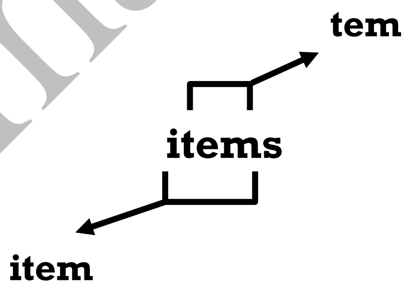
This algorithm was slow and overlapping token, so a new algorithm can be used to recognize the overlapping token.

***For example*:-**

Suppose that we have this language:

{"bit" , "byte" , "item" , "tem"}

Now if we take the word *items*, we will find two words overlapping with each other, these words are: *item* and *tem*

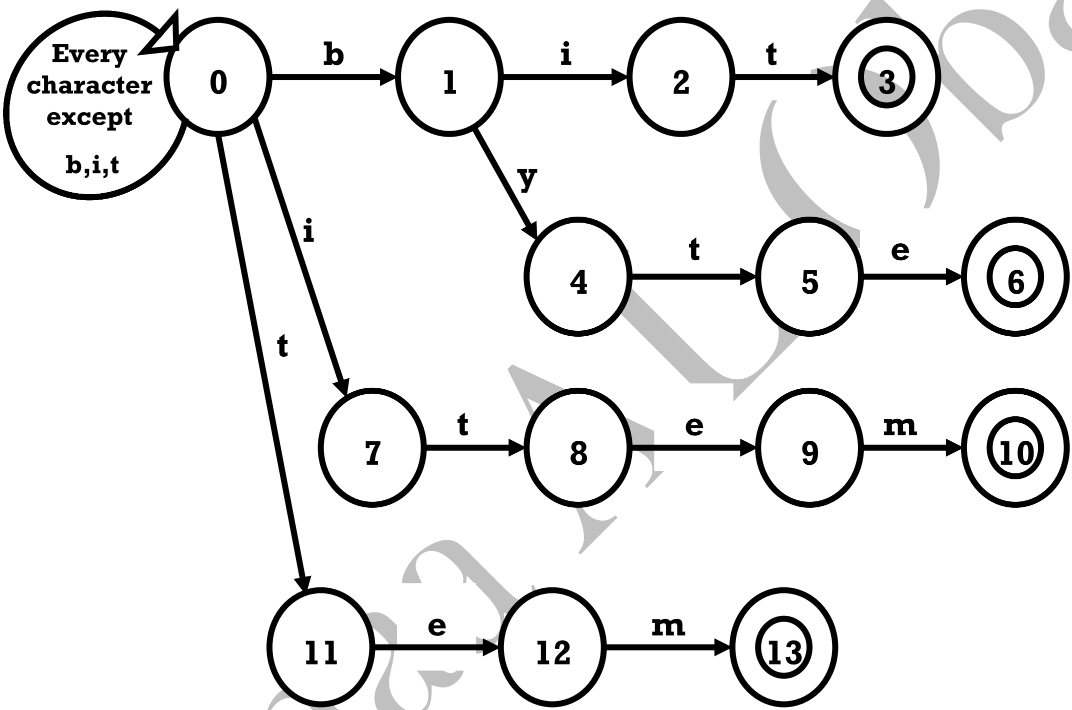


The new algorithm is known as AHO Algorithm and depends on the following steps:-

(For the above example)

Step 1:- Constructing Tree-Structured DFSA.

(Always the input for the first node is all letters except the letters that are outputted from it).



Step 2:- Determine fall back function f (Q) =R which is calculated as follows:-

- **Find largest route α which lead to Q <u>from a state that is not the start state</u>.**
- **Find the route α but this time <u>from the start state and finished in R</u>.**
- **F(Q)=R.**

Q	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F(Q)	0	0	7	8	0	11	12	0	11	12	13	0	0	0

Step 3:- Construct the Matrix Representation for the DFSA, the number of rows in it equal to the number ob nodes found in DFSA, while the number of columns equal to the number of characters that form the input language.

	b	i	t	m	y	e
0	1	7	11	0	0	0
1	#	2	#	#	4	#
2	#	#	3	#	#	#
3	#	#	#	#	#	#
4	#	#	5	#	#	#
5	#	#	#	#	#	6
6	#	#	#	#	#	#
7	#	#	8	#	#	#
8	#	#	#	#	#	9
9	#	#	#	10	#	#
10	#	#	#	#	#	#
11	#	#	#	#	#	12
12	#	#	#	13	#	#
13	#	#	#	#	#	#

Step 4:- Apply the steps of AHO Algorithm which is shown below:-

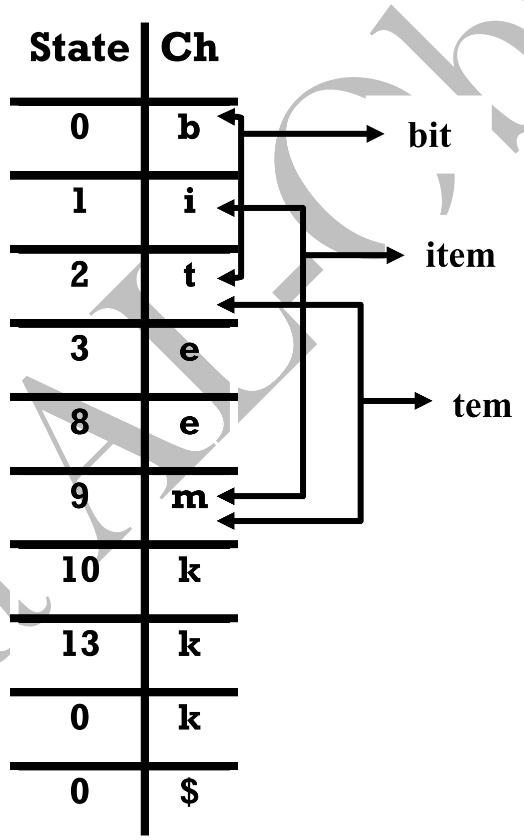
***Algorithm* :-**

```
Begin
    State = Start State;
    Ch = First Character of Input;
    While input symbols are not already exhausted do
      If Matrix [State, Ch] ≠ error indication then
       Begin
          State = Matrix [State, Ch];
          Ch = next Character;
       End
     Else begin
    If State is a Final State then Signal;
    If State = 0  then   Ch= Next Character & State = Same State
       Else   State= f (State) & Ch=Same Character
      End;
End;
```

Example **:-**

Input String = **<u>bitemk$</u>** **for the same language {"bit" , "byte" , "item" , "tem"}**

After **constructing Tree-Structured DFSA, and create a Transition Matrix for it with computing the value of the fall back function**

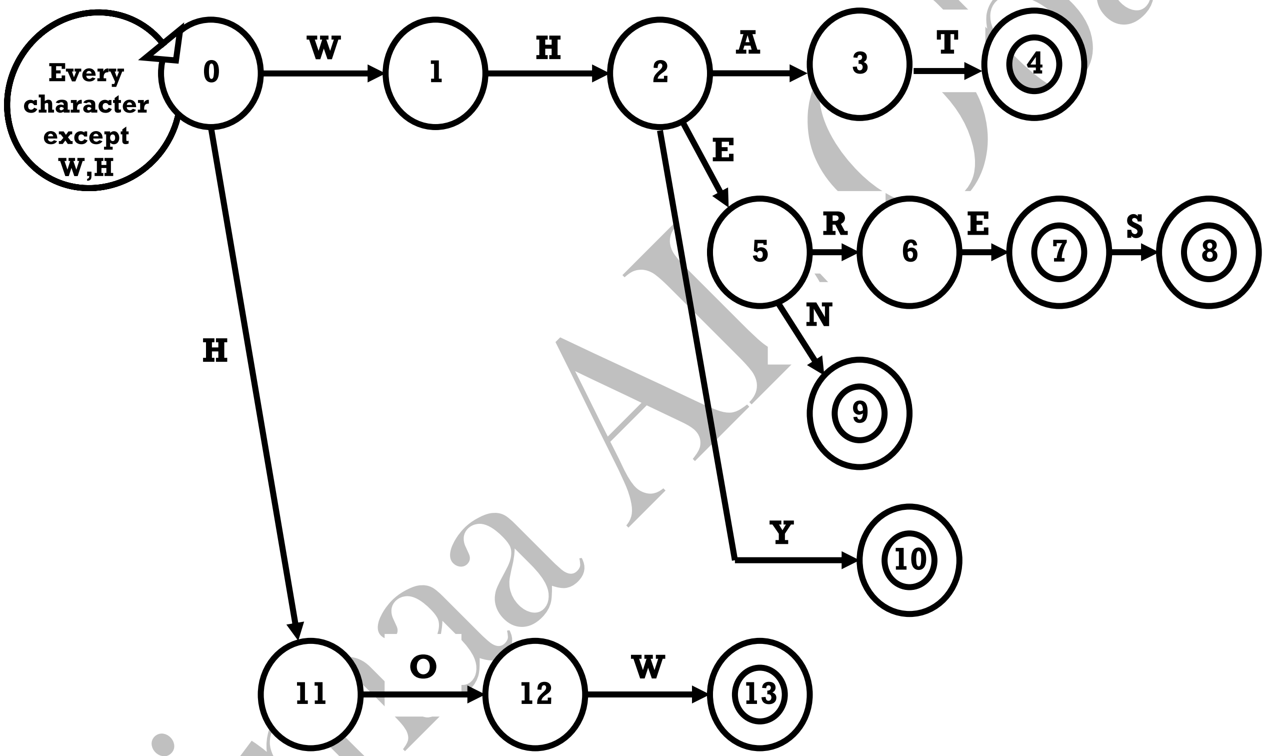


Example **:-**

If you have the following language:-

{"WHAT", "WHERE"," WHEN"," WHERES","HOW"," WHY"} and you asked to apply AHO algorithm on it to specify the words that are overlapped with each other in this string:- (WHYOWNSE$)

Step 1:- Constructing Tree-Structured DFSA.



Step 2:- Compute fall back function f (Q) as follows:-

Q	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F(Q)	0	0	11	0	0	0	0	0	0	0	0	0	0	1

Step 3:- Construct the Matrix Representation for the DFSA, the number of rows in it equal to the number ob nodes found in DFSA, while the number of columns equal to the number of characters that form the input language.

	W	H	A	E	Y	N	O	S	R
0	1	11	0	0	0	0	0	0	0
1	#	2	#	#	#	#	#	#	#
2	#	#	3	5	10	#	#	#	#
3	#	#	#	#	#	#	#	#	#
4	#	#	#	#	#	#	#	#	#
5	#	#	#	#	#	9	#	#	6
6	#	#	#	7	#	#	#	#	#
7	#	#	#	#	#	#	#	8	#
8	#	#	#	#	#	#	#	#	#
9	#	#	#	#	#	#	#	#	#
10	#	#	#	#	#	#	#	#	#
11	#	#	#	#	#	#	12	#	#
12	13	#	#	#	#	#	#	#	#
13	#	#	#	#	#	#	#	#	#

Step 4:- Apply the steps of <u>AHO Algorithm</u> on the string :- (<u>WHYOWNSE$</u>).

State	Ch	
0	W	WHY
1	H	
2	Y	
10	O	Matrix [State,Ch]=#
0	O	
0	W	
1	N	Matrix [State,Ch]=#
0	N	
0	S	
0	E	
0	$	