Kleene’s Theorem

Shakir Ullah Shah

Lecture 7

FAST National University of Computer and Emerging Sciences, Peshawar

Proof of Part 3: Converting Regular Expressions into FAs

We prove this part by recursive definition and constructive algorithm at the same time.

The set of regular expressions is defined by the following rules:

- Rule 1: Every letter of the alphabet ∑ is a regular expression, Λ itself is a regular expression.
- Rule 2: If r_1 and r_2 are regular expressions, then so are:

  (i) (r_1)

  (ii) $r_1 + r_2$

  (iii) $r_1 r_2$

  (iv) r_1^*
- Rule 3: Nothing else is a regular expression.

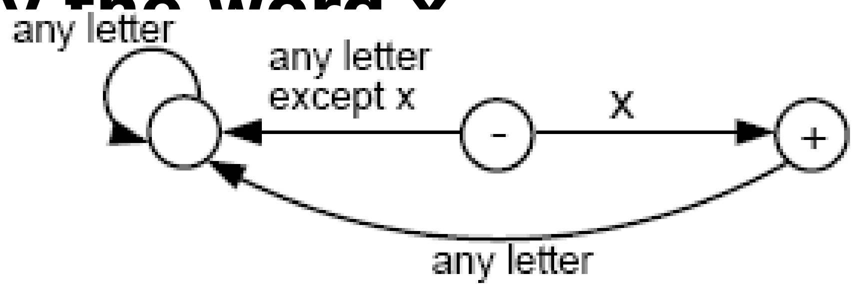
Rule 1

There is an FA that accepts any particular letter of the alphabet.

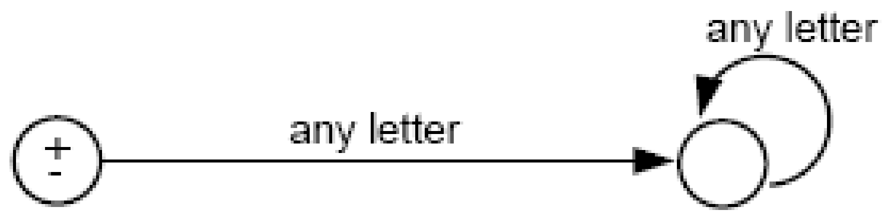
There is an FA that accepts only the word Λ.

Proof of rule 1

If letter x is in ∑, then the following FA accepts only the word x



The following FA accepts only : Λ



Rule 2

If there is an FA called FA_1 that accepts the language defined by the regular expression r_1, i.e. $L(FA_1)= r_1$ and

there is an FA called FA_2 that accepts the language defined by the regular expression r_2, i.e. $L(FA_2)=r_2$

then there is an FA that we shall call FA_3 that accepts the language defined by the regular expression $(r_1 + r_2)$ i.e. $L(FA_3)= (r_1 + r_2)$

Proof of Rule 2

We shall show that FA_3 exists by presenting an algorithm showing how to construct FA_3.

Algorithm:

- Starting with two machines, FA_1 with states $x_1; x_2; x_3; \ldots$, and FA_2 with states $y_1; y_2; y_3; \ldots$, we construct a new machine FA_3 with states $z_1; z_2; z_3; \ldots$ where each z_i is of the form $x_{something}$ or $y_{something}$.
- The combination state x_{start} or y_{start} is the start state of the new machine FA_3.
- If either the x part or the y part is a final state, then the corresponding z is a final state.

Algorithm (cont.)

– To go from one state z to another by reading a letter from the input string, we observe what happens to the x part and what happens to the y part and go to the new state z accordingly. We could write this as a formula:

z_{new} after reading letter p = (x_{new} after reading letter p on FA_1) or (y_{new} after reading letter p on FA_2)

Remarks

The new machine FA_3 constructed by the above algorithm will simultaneously keep track of where the input would be if it were running on FA_1 alone, and where the input would be if it were running on FA_2 alone.

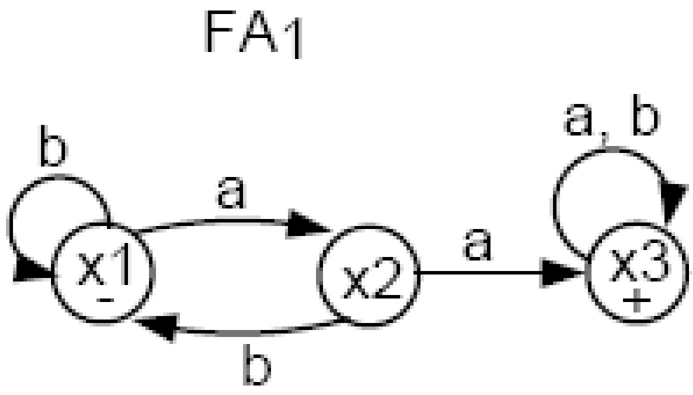
If a string traces through the new machine FA_3 and ends up at a final state, it means that it would also end at a final state either on machine FA_1 or on machine FA_2. Also, any string accepted by either FA_1 or FA_2 will be accepted by this FA_3. So, the language FA_3 accepts is the union of the languages accepted by FA_1 and FA_2, respectively.

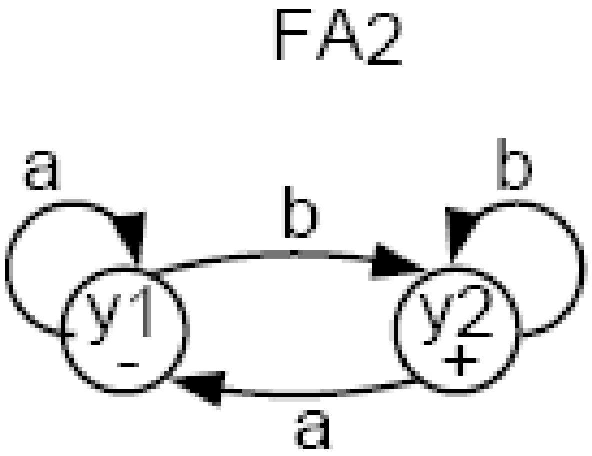
Note that since there are only finitely many states *x's* and finitely many states *y's*, there can be only finitely many possible states *z's*.

Let us look at an example illustrating how the algorithm

Example

Consider the following two FAs:





FA_1 accepts all words with a double a in them.

FA_2 accepts all words ending with b.

Let's follow the algorithm to build *FA_3* that accepts the union of the two languages.

Combining the FAs

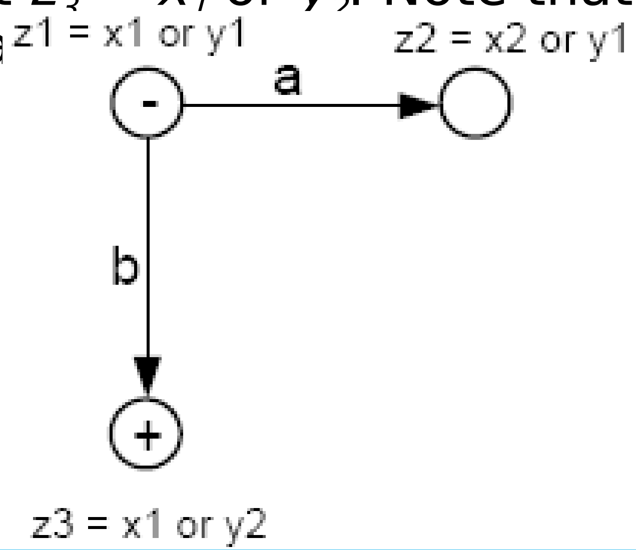
The start (-) state of FA_3 is z_1 = x_1 or y_1.

In z_1, if we read an *a*, we go to x_2 (observing FA_1), or we go to y_1 (observing FA_2).

Let z_2 = x_2 or y_1.

In z_1, if we read a *b*, we go to x_1 (observing FA_1), or to y_2 (observing FA_2).

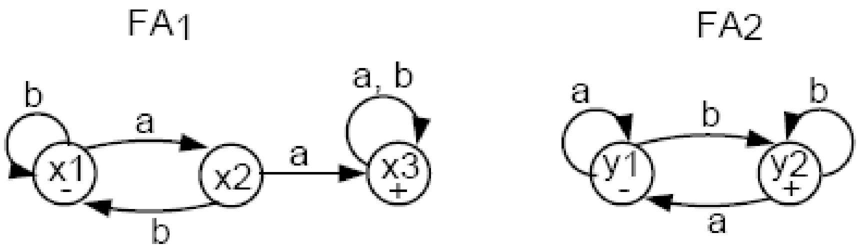
Let z_3 = x_1 or y_2. Note that z_3 must be a final state since y_2 is a fina

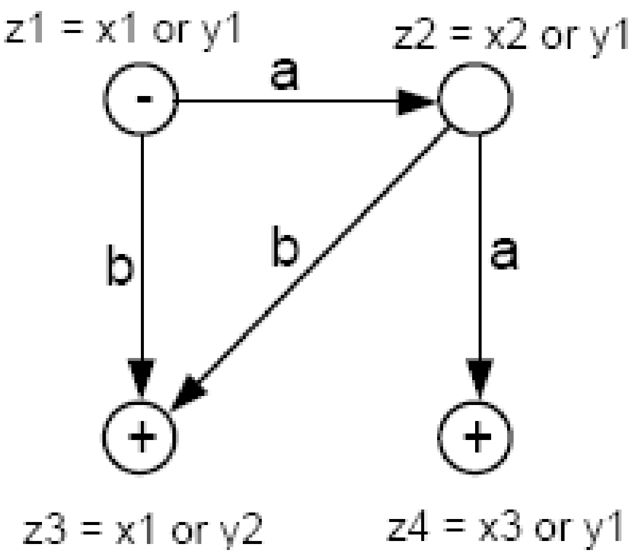


Old states	New states after reading	
	a	b
z_1– € (x_1,y_1)	**(x_2,y_1) € z_2**	**(x_1,y_2) € z_3**

In z_2, if we read an *a*, we go to x_3 or y_1. Let $z_4 = x_3$ or y_1. z_4 is a final state because x_3 is.

In z_2, if we read a *b*, we go to x_1 or y_2, which is z_3.

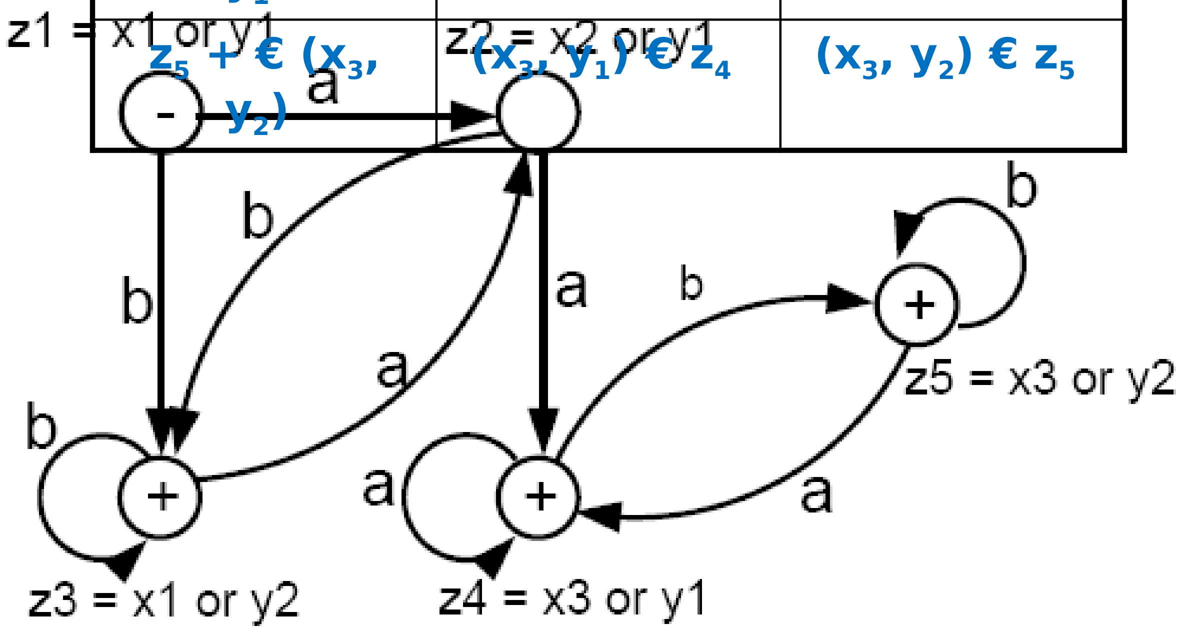




Old states	New states after reading	
	a	b
z_1- € (x_1,y_1)	**(x_2,y_1) € z_2**	**(x_1,y_2) € z_3**
z_2 € (x_2, y_1)	**(x_3, y_1) € z_4**	**(x_1, y_2) € z_3**

Old states	New states after reading	
	a	b
z_1– € (x_1,y_1)	**(x_2,y_1) € z_2**	**(x_1,y_2) € z_3**
z_2 € (x_2, y_1)	**(x_3, y_1) € z_4**	**(x_1, y_2) € z_3**
z_3 + € (x_1, y_2)	**(x_2, y_1) € z_2**	**(x_1, y_2) € z_3**
z_4 + € (x_3, y_1)	**(x_3, y_1) € z_4**	**(x_3, y_2) € z_5**
z_5 + € (x_3, y_2)	**(x_3, y_1) € z_4**	**(x_3, y_2) € z_5**

Old states	New states after reading	
	a	b
z_1- € (x_1,y_1)	(x_2,y_1) € z_2	(x_1,y_2) € z_3
z_2 € (x_2, y_1)	(x_3, y_1) € z_4	(x_1, y_2) € z_3
z_3 + € (x_1, y_2)	(x_2, y_1) € z_2	(x_1, y_2) € z_3
z_4 + € (x_3, y_1)	(x_3, y_1) € z_4	(x_3, y_2) € z_5
z_5 + € (x_3, y_2)	(x_3, y_1) € z_4	(x_3, y_2) € z_5

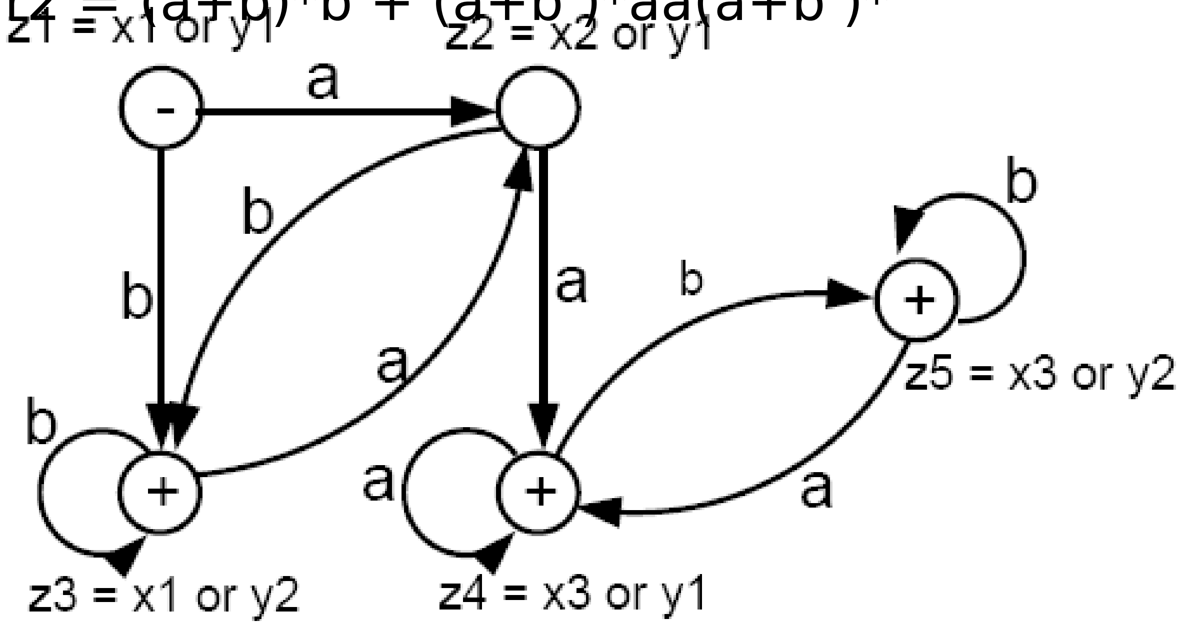


This machine accepts all words that have a double *a* or that end with *b*.

The labels $z_1 = x_1$ or y_1, $z_2 = x_2$ or y_1, etc. can be removed if you want.

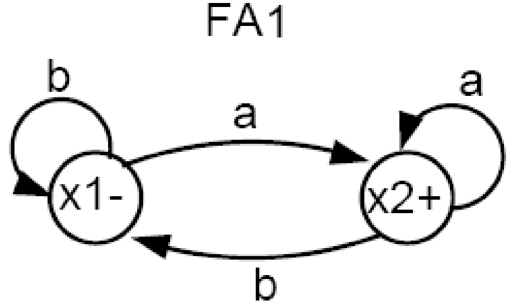
RE corresponding to the above FA may be

r1+r2 = (a+b)*b + (a+b)*aa(a+b)*



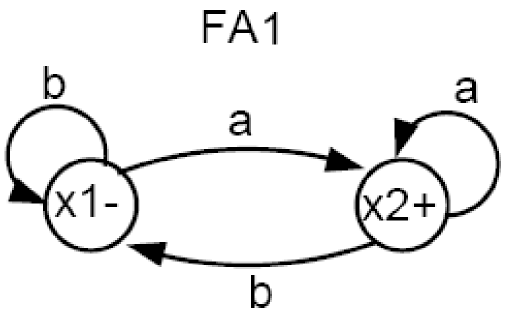
Example

Let r_1=(a+b)*a (**words that end in *a)*** and the corresponding FA_1 be

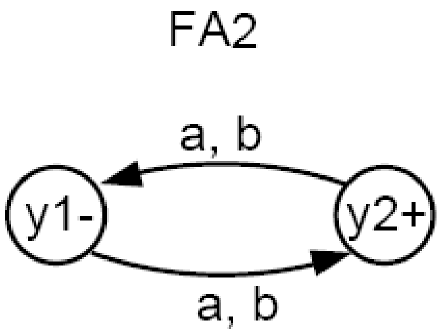


Example

Let r_1=(a+b)*a (**words that end in *a)*** and the corresponding FA_1 be

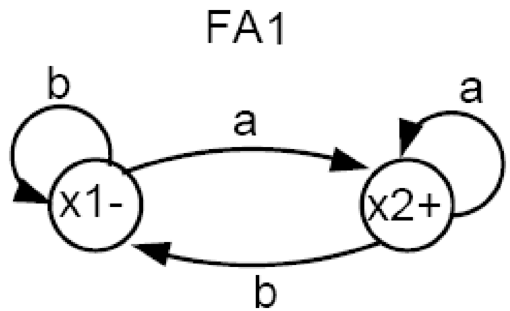


Let r_2 = (a+b)((a+b)(a+b))* or (a+b)(a+b))*(a+b) (**words of odd length)**

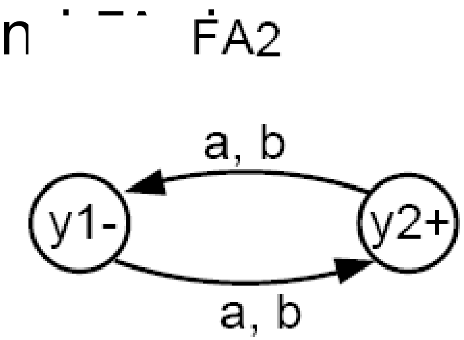


Example

Let r_1=(a+b)*a (**words that end in *a)*** and the corresponding FA_1 be



Let r_2 = (a+b)((a+b)(a+b))* or (a+b)(a+b))*(a+b) (**words of odd length)** an



Task:

Generate Union of FAs corresponding to r1 and r2 i.e.

r1+ r2

Rule 3

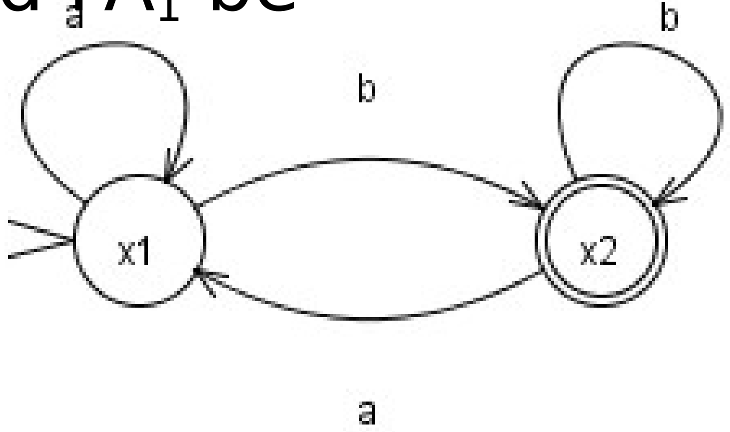
If there is an FA_1 that accepts the language defined by the regular expression r_1, and

there is an FA_2 that accepts the language defined by the regular expression r_2,

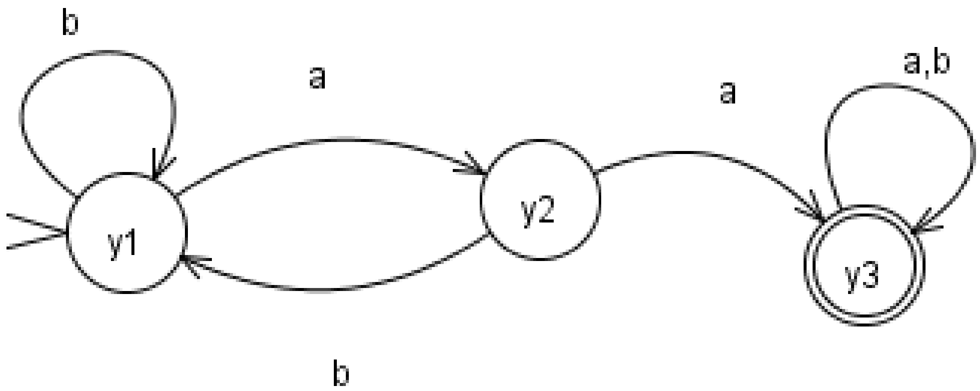
then there is an FA_3 that accepts the language defined by the (concatenation) regular expression (r_1r_2), i.e. the product language.

Example

Let r_1 =(a+b)*b defines L_1 and FA_1 be



and r_2 = (a+b)*aa (a+b)* defines L_2 and FA_2 be



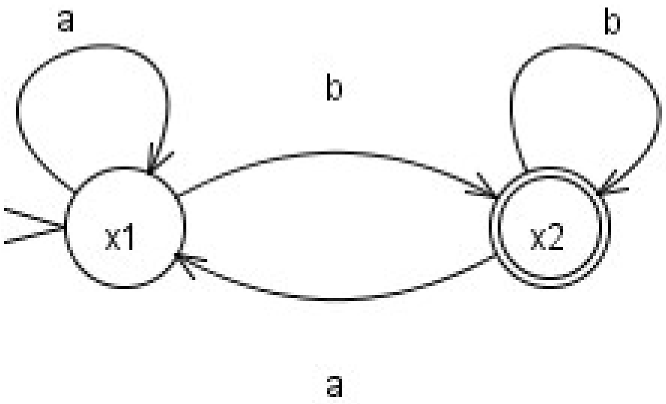
Concatenation of two FAs Continued …

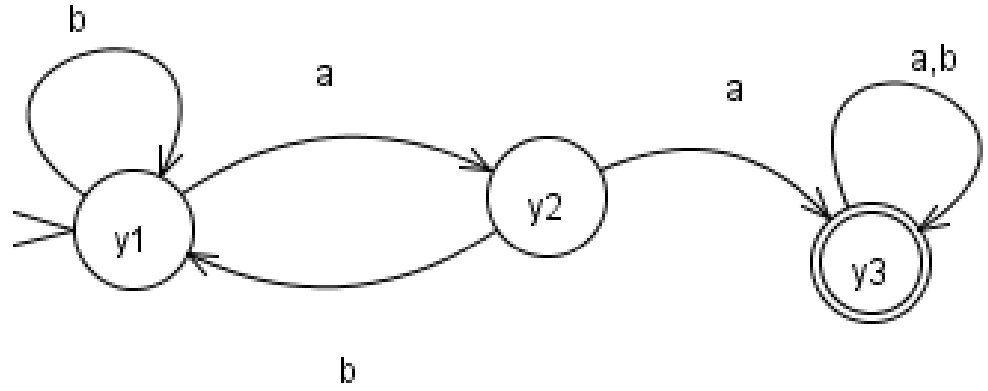
Let FA_3 be an FA corresponding to r_1r_2 , then the initial state of FA_3 must correspond to the initial state of FA_1 and the final state of FA_3 must correspond to the final state of FA_2 . Since the language corresponding to r_1r_2 is the concatenation of corresponding languages L_1 and L_2 , consists of the strings obtained, concatenating the strings of L_1 to those of L_2 , therefore *the moment a final state of* <u>*FA_1*</u> *is entered, the possibility of the initial state of* <u>*FA_2*</u>*will be included as well*.

Concatenation of two FAs Continued …

Since, in general, FA_3 will be different from both FA_1 and FA_2 , so the labels of the states of FA_3 may be supposed to be z_1,z_2, z_3, …, where z_1 stands for the initial state. Since z_1 corresponds to the states x_1 , so there will be two transitions separately for each letter read at z_1 . It will give two possibilities of states which correspond to either z_1 or different from z_1 . This process may be expressed in the following transition table for all possible states of FA_3

Example continued …

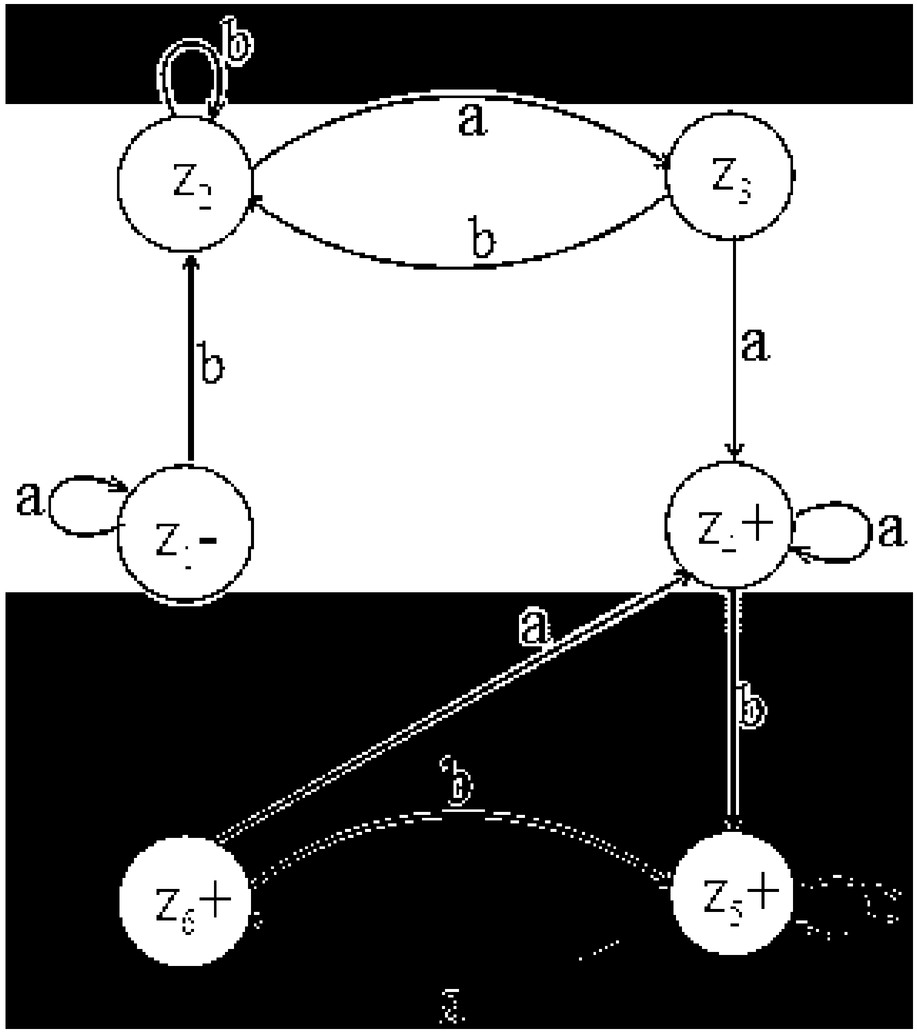




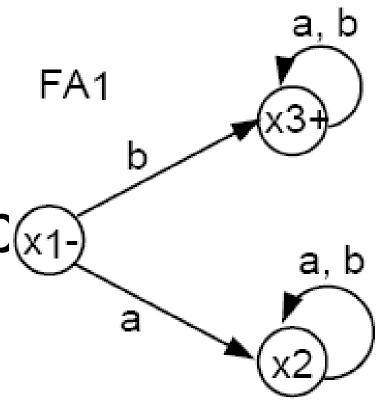
Old states	New states after reading	
	a	b
z_1– € x_1	x_1 € z_1	(x_2,y_1) € z_2

Example continued ...

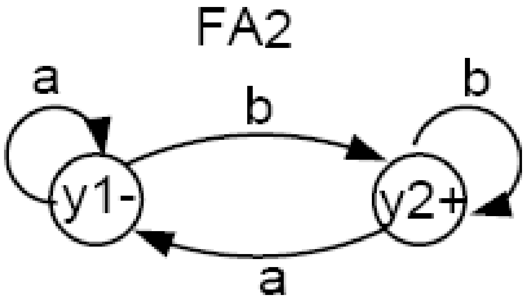
Old states	New states after reading	
	a	b
z_1– € x_1	x_1 € z_1	(x_2,y_1) € z_2
z_2 € (x_2, y_1)	(x_1, y_2) € z_3	(x_2, y_1) € z_2
z_3 € (x_1, y_2)	(x_1, y_3) € z_4	(x_2, y_1) € z_2
z_4 + € (x_1, y_3)	(x_1, y_3) € z_4	(x_2, y_1, y_3) € z_5
z_5 + € (x_2, y_1, y_3)	(x_1, y_2, y_3) € z_6	(x_2, y_1, y_3) € z_5
z_6 + € (x_1, y_2, y_3)	(x_1, y_3) € z_4	(x_2, y_1, y_3) € z_5


b
a
b
b
a
a
a
a
b
b

Example



Let r_1 =b(a+b)* , the corresponding FA_1 b



also r_2 = (a+b)*b

Task:

Generate the FA representing r1 r2 using Concatenation Algorithm

KLEENE'S THEOREM PART 3 …

Closure of an FA

If *r* is a regular expression and FA_1 is a finite automaton that accepts exactly the language defined by *r*, then there is an FA, called FA_2, that will accepts exactly the language defined by r^*.

KLEENE'S THEOREM PART 3 …

<u>Closure of an FA</u>

Closure of an FA, is same as concatenation of an FA with itself, except that

- the initial state of the required FA is a final state as well (because Λ is also accepted in closure).
- non final state of the required FA as well.
  - Means initial state of given FA will correspond to two sates in required FA.

EXAMPLE

Consider the regular expression $r = aa^*bb^*$.

EXAMPLE

Consider the regular expression r = aa*bb*.

This defines the language where all the a’s come before all the b’s.

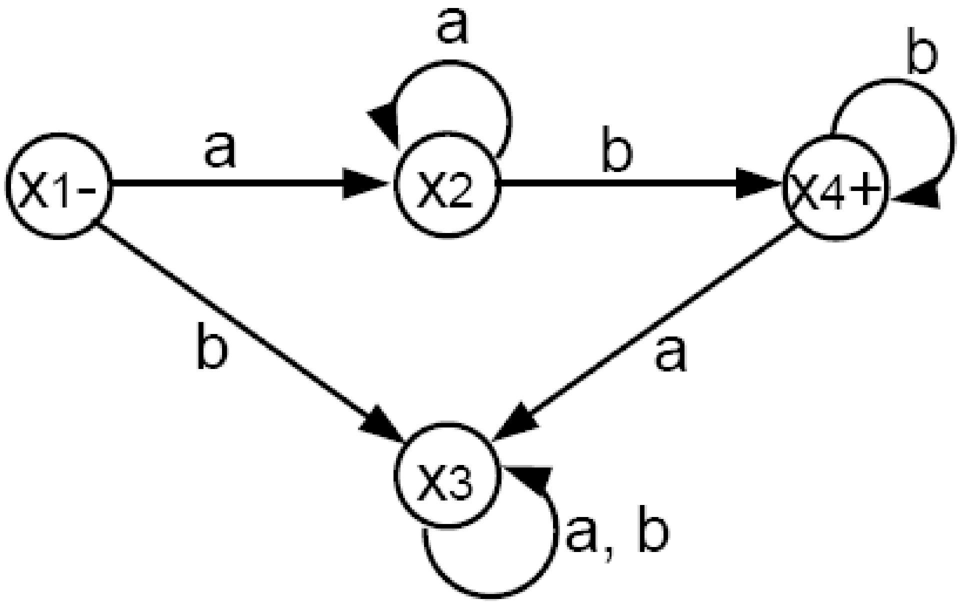
The FA that accepts this language is:

EXAMPLE

Consider the regular expression r = aa*bb*.

This defines the language where all the a’s come before all the b’s.

The FA that accepts this language is:



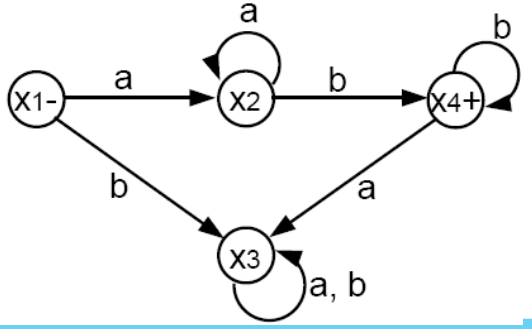
EXAMPLE

Let us now build FA_2 that accepts r* = (aa*bb*)*.

We begin with the start state $z_1 = x_1$.

In z_1, reading an a takes us to $x_2 = z_2$. Reading a b takes us to $x_3 = z_3$.

Old states	New states after reading	
	a	b
z_1- +=x_1	**X_2= z_2**	**x_3= z_3**



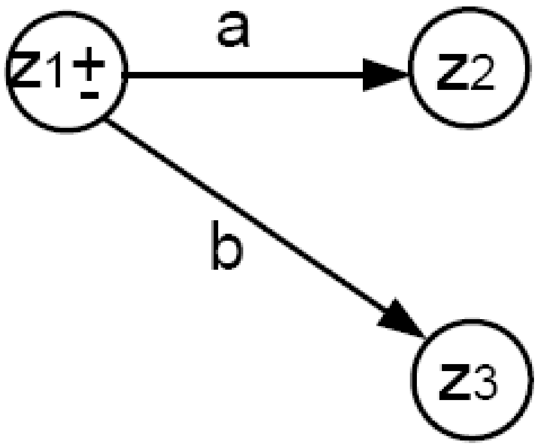
EXAMPLE

Let us now build FA_2 that accepts r* = (aa*bb*)*.

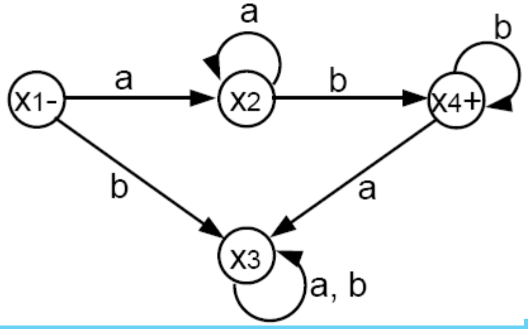
We begin with the start state $z_1 = x_1$.

In z_1, reading an a takes us to $x_2 = z_2$. Reading a b takes us to $x_3 = z_3$.

Old states	New states after reading	
	a	b
z_1- +=x_1	**X_2= z_2**	**x_3= z_3**



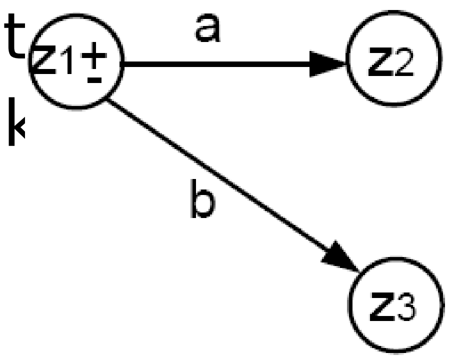
...



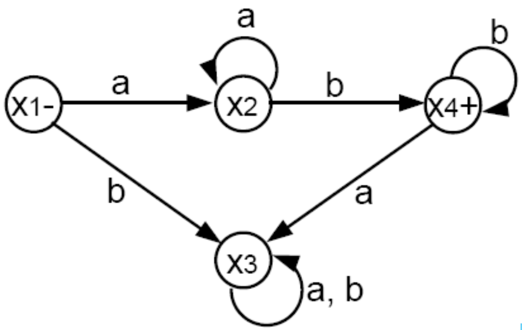
EXAMPLE

In z_2, if we read an *a* we go back to z_2. If we read a *b*, we go to x_4 , or we have the option of jumping to the start state x_1 (since x_4 is a final state). Hence, let $z_4 += x_4$ or x_1.

In z_3, just … either an *a* or a *b*, we loop back …



…

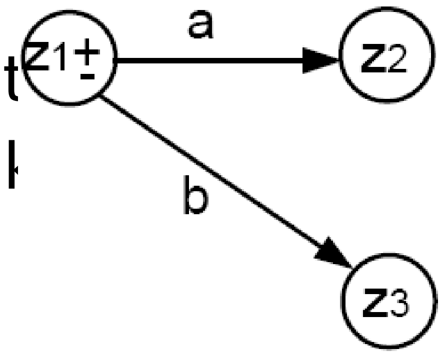


Old states	New states after reading	
	a	b
$z_1-\ +=x_1$	$x_2= z_2$	$x_3= z_3$
$z_2=x_2$	$x_2= z_2$	$X_{1=}z_4+$ $X_{4=}z_4+$
$z_3=x_3$	$x_3= z_3$	$x_3= z_3$

EXAMPLE

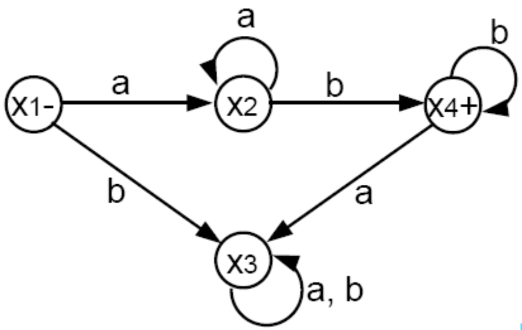
In z_2, if we read an *a* we go back to z_2. If we read a *b*, we go to x_4 , or we have the option of jumping to the start state x_1 (since x_4 is a final state). Hence, let $z_4+= x_4$ or x_1.

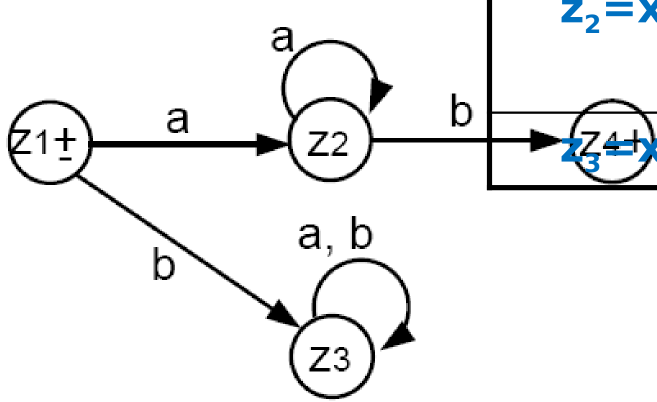
In z_3, just ... either an *a* or a *b*, we loop back ...



Old states	New states after reading	
	a	b
z_1- $+=x_1$	$x_2= z_2$	$x_3= z_3$
$z_2=x_2$	$x_2= z_2$	$X_{1=}z_4+$ $X_{4=}z_4+$
$z_3=x_3$	$x_3= z_3$	$x_3= z_3$

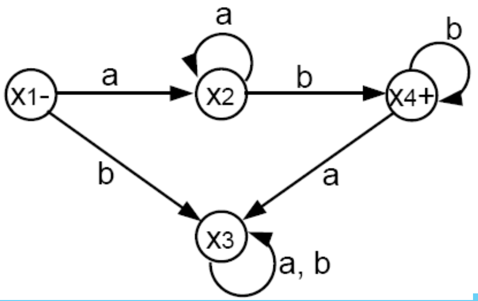
...





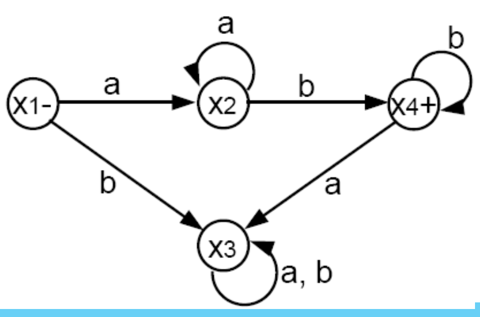
EXAMPLE

Old states	New states after reading	
	a	b
z_1- +=x_1	$x_2 = z_2$	$x_3 = z_3$
$z_2 = x_2$	$x_2 = z_2$	$X_{1} = z_4+$ $X_{4} = z_4+$
$z_3 = x_3$	$x_3 = z_3$	$x_3 = z_3$



EXAMPLE

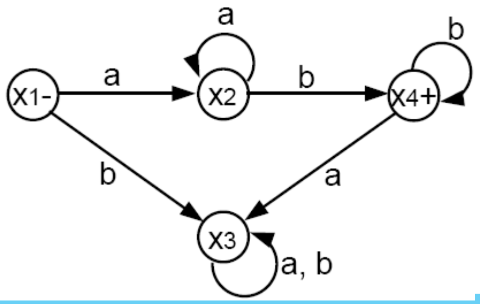
Old states	New states after reading	
	a	b
$z_1- += x_1$	$x_2 = z_2$	$x_3 = z_3$
$z_2 = x_2$	$x_2 = z_2$	$X_{1=}z_4+$ $X_{4=}z_4+$
$z_3 = x_3$	$x_3 = z_3$	$x_3 = z_3$
$z_4 + = x_1$ $z_4 + = x_4$	$x_2 = z_2$ $x_3 = z_3$ $(x_2, x_3) = z_5$	$x_3 = z_3$ $x_4 = z_4$ $x_1 = z_1$ $(x_3, x_{4,} x_1) = z_6$



- In z_4, what happens if we read an a? If $z_4 = x_1$, we go to x_2. If $z_4 = x_4$, we go to x_3. Hence, we will be in x_2 or x_3. So, let $z_5 = x_2$ or x_3.
- In z_4, if we read a *b*? If z_4 means x_1, we go x_3. If z_4 means x_4, we go to x_4 or jump to x_1 (due to final x_4). Thus, let $z_6 = x_1$ or x_3 or x_4. z_6 must be a final state since x_4 is.

EXAMPLE

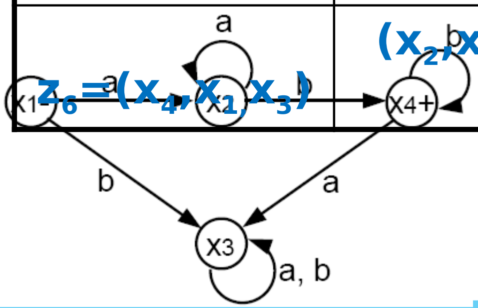
Old states	New states after reading	
	a	b
z_1- +=x_1	x_2= z_2	x_3= z_3
z_2=x_2	x_2= z_2	$X_{1=}z_4$+ $X_{4=}z_4$+
z_3=x_3	x_3= z_3	x_3= z_3
z_4+=x_1 z_4+=x_4	x_2= z_2 x_3= z_3 (x_2,x_3)= z_5	x_3= z_3 x_4= z_4 x_1= z_1 $(x_3,x_{4,}x_1)$= z_6
z_5=(x_2,x_3)	(x_2,x_3)= z_5	$(x_4,x_{1,}x_3)$= z_6



- In z5, reading an a takes us to x2 or x3, which is still z5. So, we have an a-loop at z5.
- In z5, reading a b takes us to x4 or x1, or x3, which is z6.

EXAMPLE

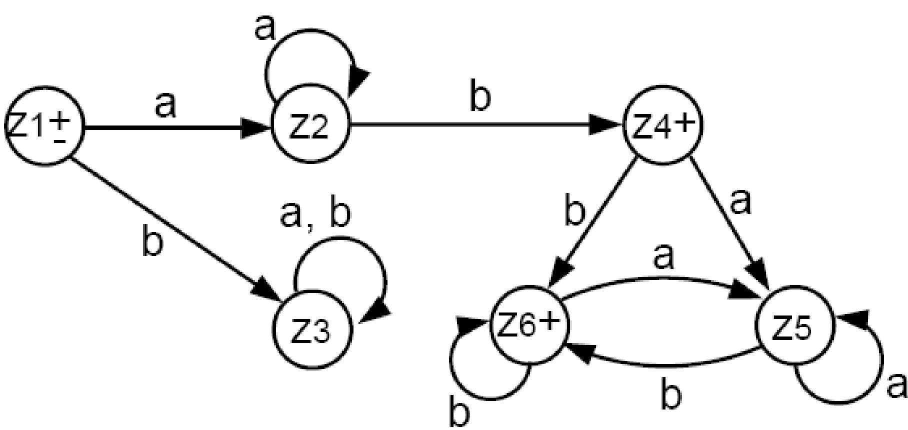
Old states	New states after reading a	New states after reading b
$z_1 - + = x_1$	$x_2 = z_2$	$x_3 = z_3$
$z_2 = x_2$	$x_2 = z_2$	$X_{1=} z_4 +$ $X_{4=} z_4 +$
$z_3 = x_3$	$x_3 = z_3$	$x_3 = z_3$
$z_4 + = x_1$ $z_4 + = x_4$	$x_2 = z_2$ $x_3 = z_3$ $(x_2, x_3) = z_5$	$x_3 = z_3$ $x_4 = z_4$ $x_1 = z_1$ $(x_3, x_{4,} x_1) = z_6$
$z_5 = (x_2, x_3)$	$(x_2, x_3) = z_5$	$(x_4, x_{1,} x_3) = z_6$
$z_6 = (x_4, x_{1,} x_3)$	$(x_2, x_3) = z_5$	$(x_4, x_{1,} x_3) = z_6$

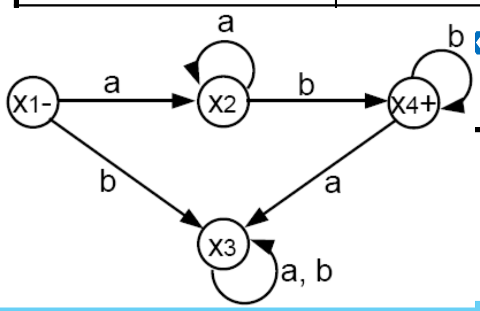


- In z6, reading an a, take us to x2 or x3, which is z5.
- In z6, reading a b takes us to x3, x4, or x1, which is still z6. So, we have a b-loop at z6.

EXAMPLE

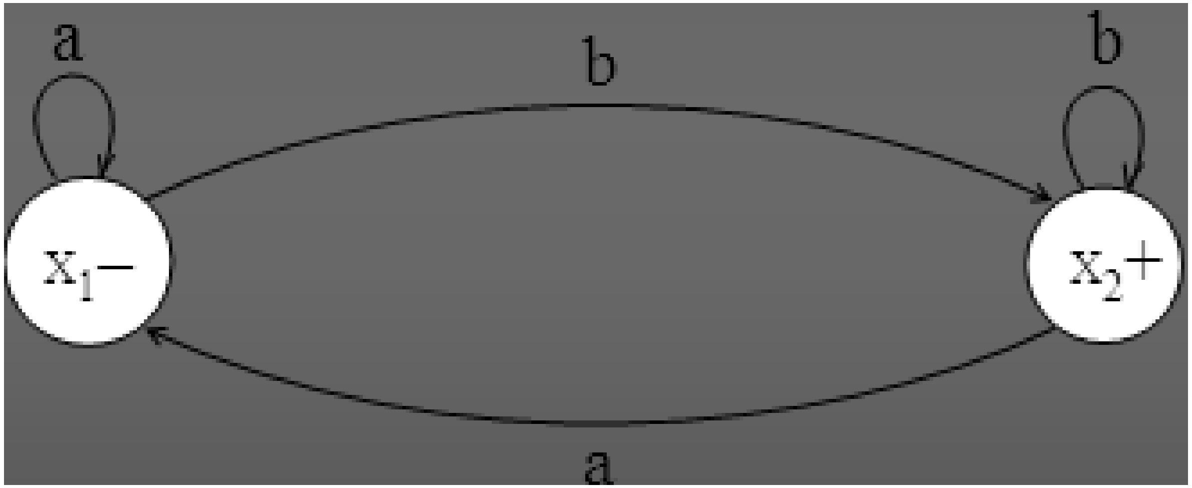
Old states	New states after reading	
	a	b
z_1- +$=x_1$	$x_2= z_2$	$x_3= z_3$
$z_2=x_2$	$x_2= z_2$	$X_{1=}z_4+$ $X_{4=}z_4+$
$z_3=x_3$	$x_3= z_3$	$x_3= z_3$
$z_4+=x_1$ $z_4+=x_4$	$x_2= z_2$ $x_3= z_3$ $(x_2,x_3)= z_5$	$x_3= z_3$ $x_4= z_4$ $x_1= z_1$ $(x_3,x_{4,}x_1)= z_6$
$z_5=(x_2,x_3)$	$(x_2,x_3)= z_5$	$(x_4,x_{1,}x_3)= z_6$
	$\ldots_3)= z_5$	$(x_4,x_{1,}x_3)= z_6$





EXAMPLE

Let r=(a+b)*b and the corresponding FA be



then the FA corresponding to r* may be determined as under

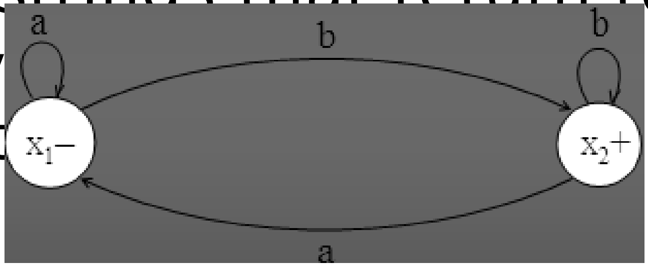
EXAMPLE

In this case we need to represent x_1 as two separate z-states in FA_2,

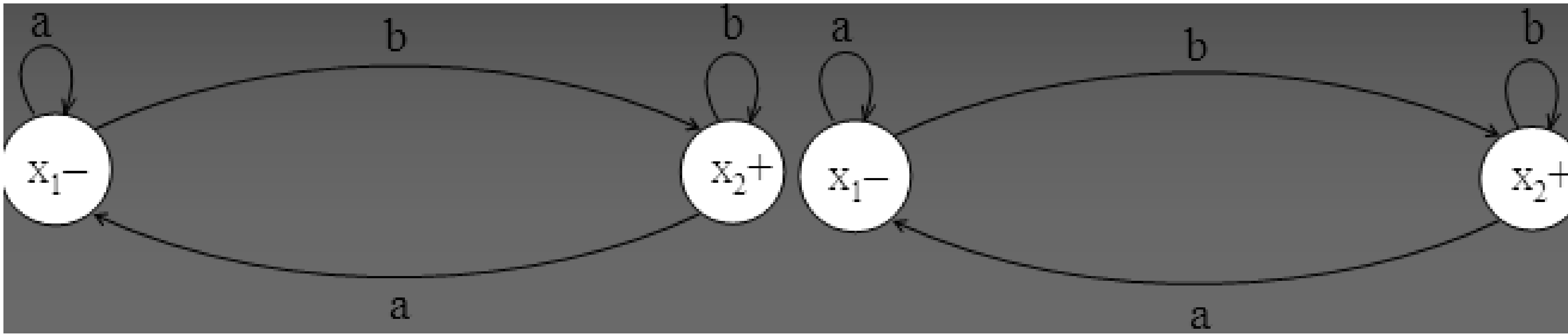
- one as a start and final state ± (z1 ± = x1), and the other as
- the non-final start state (z2 = x1) .

The ± state is necessary for FA_2 to accept .

The non-final start state is necessary for FA_2 to operate correctly, since some strings that return to the start state x_1 may not be v
therefore should not be accept

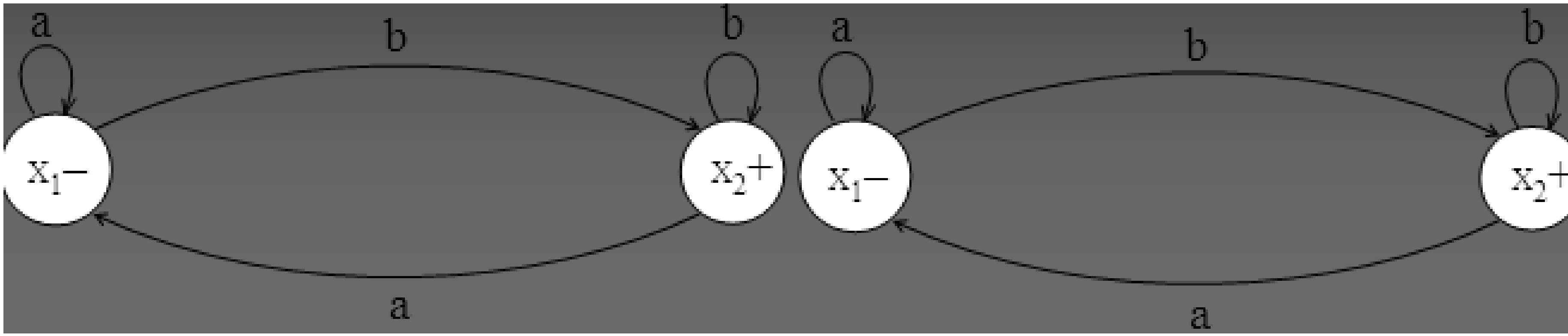


EXAMPLE



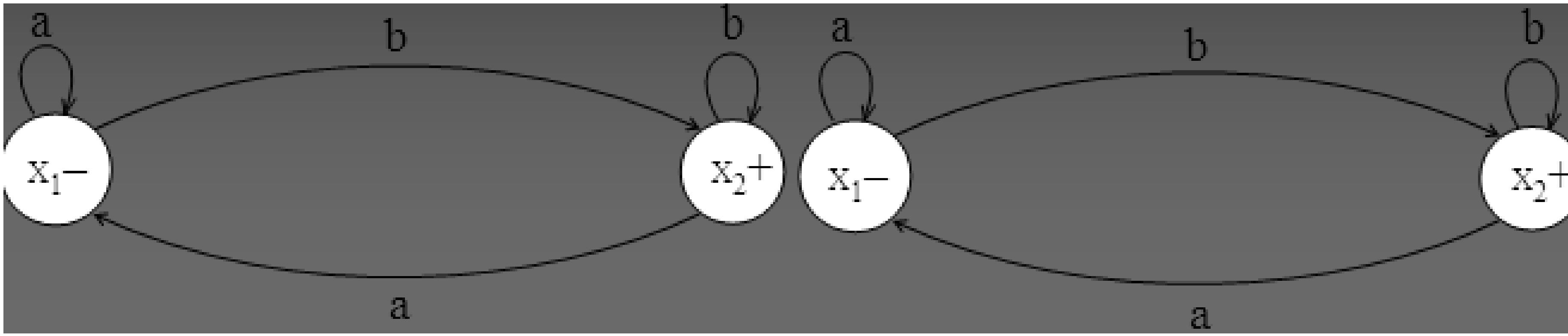
Old states	New states after reading	
	a	b
z_1 - + x_1	Non-final x_1 z_2	(x_2, x_1) z_3

EXAMPLE



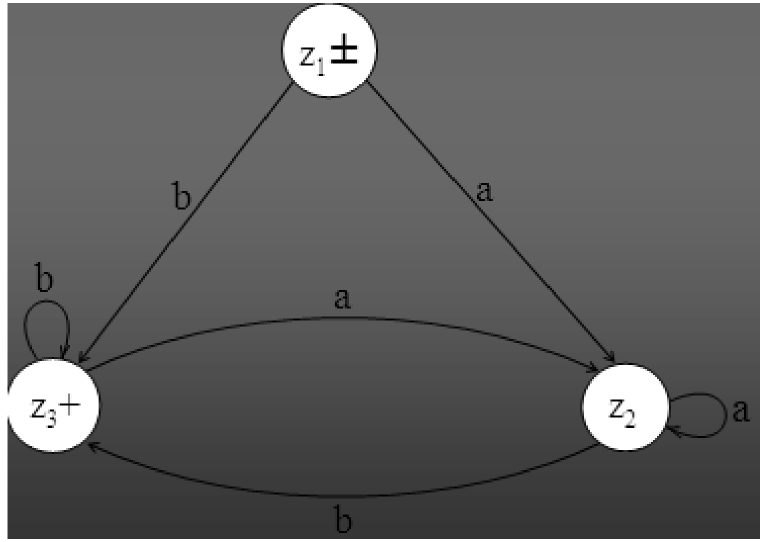
Old states	New states after reading	
	a	b
z_1 - + x_1	Non-final x_1 z_2	(x_2, x_1) z_3
Non-final z_2	x_1 z_2	(x_2, x_1) z_3

EXAMPLE



Old states	New states after reading	
	a	b
z_1 - + x_1	Non-final x_1 z_2	(x_2, x_1) z_3
Non-final z_2	x_1 z_2	(x_2, x_1) z_3

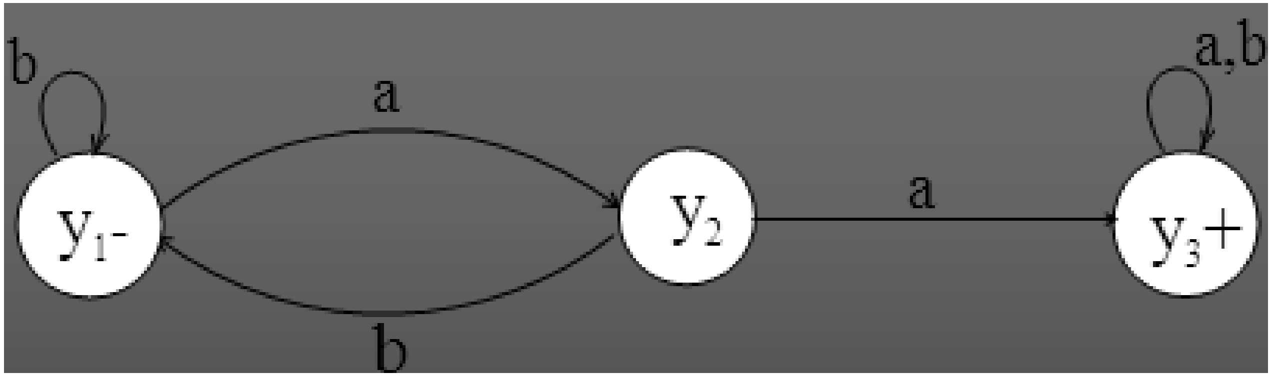
EXAMPLE



Old states	New states after reading	
	a	b
z_1 - + x_1	Non-final x_1 z_2	(x_2, x_1) z_3
Non-final z_2	x_1 z_2	(x_2, x_1) z_3

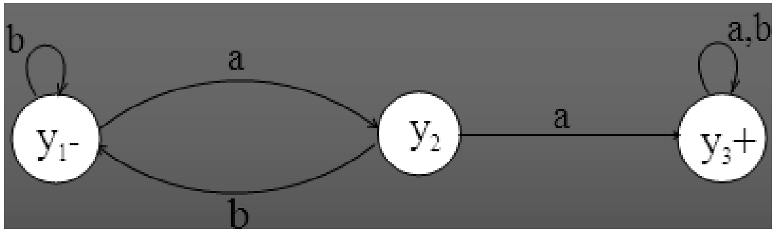
EXAMPLE

Let r=(a+b)*aa(a+b)* and the corresponding FA be



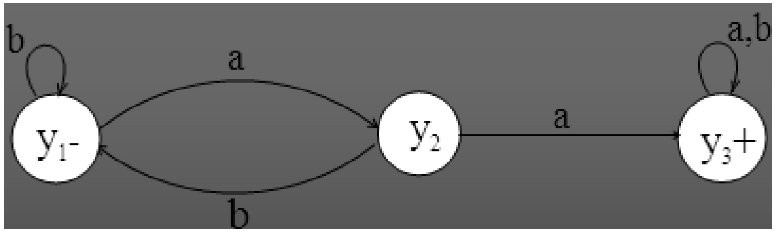
EXAMPLE

Old states	New states after reading	
	a	b
z_1 - + y_1	y_2 z_3	y_1 z_2



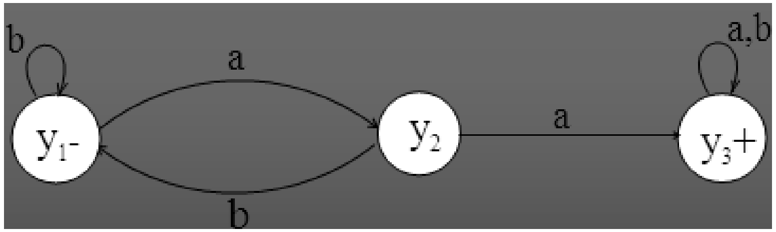
EXAMPLE

Old states	New states after reading	
	a	b
z_1 - + y_1	y_2 z_3	y_1 z_2
z_2 y_1	y_2 z_3	y_1 z_2



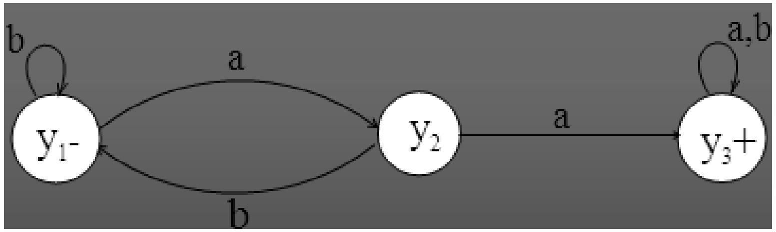
EXAMPLE

Old states	New states after reading	
	a	b
z_1 - + y_1	y_2 z_3	y_1 z_2
z_2 y_1	y_2 z_3	y_1 z_2
z_3 y_2	(y_3,y_1) z_4	y_1 z_2



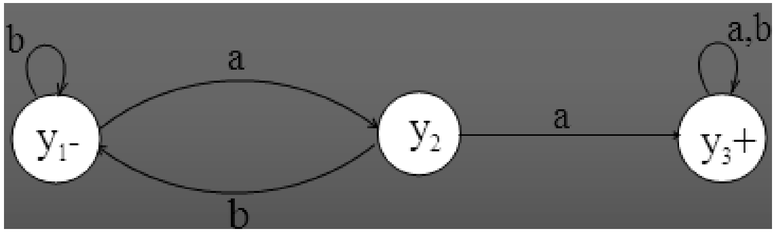
EXAMPLE

Old states	New states after reading	
	a	b
z_1 - + y_1	y_2 z_3	y_1 z_2
z_2 y_1	y_2 z_3	y_1 z_2
z_3 y_2	(y_3,y_1) z_4	y_1 z_2
z_4^+ (y_3, y_1)	(y_3, y_1, y_2) z_5	(y_3, y_1) z_4

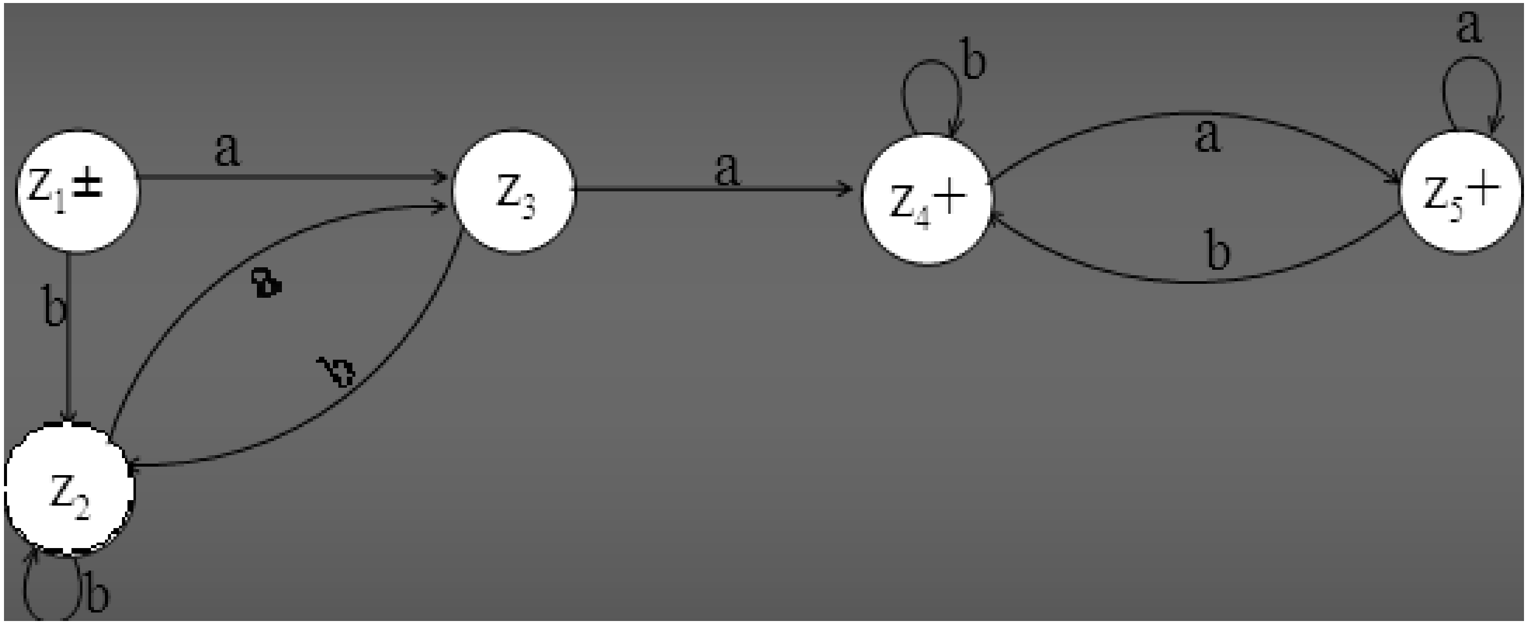


EXAMPLE

Old states	New states after reading	
	a	b
z_1 - + y_1	y_2 z_3	y_1 z_2
z_2 y_1	y_2 z_3	y_1 z_2
z_3 y_2	(y_3,y_1) z_4	y_1 z_2
z_4^+ (y_3, y_1)	(y_3 , y_1, y_2) z_5	(y_3, y_1) z_4
z_5^+ (y_3, y_1, y_2)	(y_3, y_1 , y_2) z_5	(y_3, y_1) z_4



EXAMPLE



PROOF

We have finished the proof of part 3 of Kleene’s theorem.

PROOF

We have finished the proof of part 3 of Kleene's theorem.

Because of Rules 1, 2, 3, and 4, we know that all regular expressions have corresponding finite automata that define the same language.

PROOF

We have finished the proof of part 3 of Kleene's theorem.

Because of Rules 1, 2, 3, and 4, we know that all regular expressions have corresponding finite automata that define the same language.

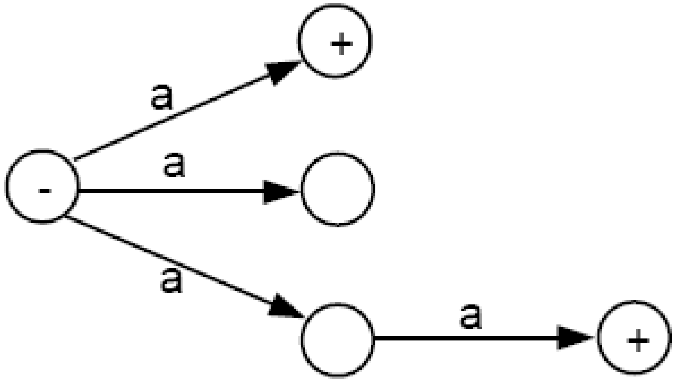
This is because while we are constructing the regular expression from elementary building blocks using recursive definition, we can simultaneously be constructing the corresponding FA using the algorithms

Nondeterministic Finite Automaton (NFA)

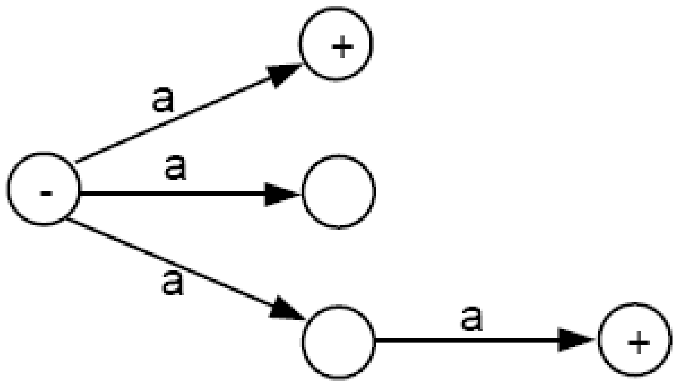
NONDETERMINISTIC FINITE AUTOMATON (NFA)

- **Definition:** An NFA is a TG with a unique start state and a property of having single letter as label of transitions. An NFA is a collection of three things
- 1) Finite many states with *one initial* and some final states
- 2) Finite set of input letters, say, Σ ={a, b, c}
- 3) Finite set of transitions, showing where to move if a letter is input at certain state (Λ is not a valid transition), there may be more than one transition for certain letters and there may not be any transition for certain letters.

EXAMPLES OF NFA

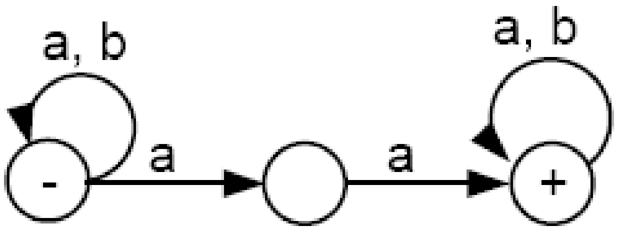


EXAMPLES OF NFA



It is to be noted that the above NFA accepts the language consisting of a and aa.

EXAMPLES OF NFA



It is to be noted that the above NFA accepts the language of strings, defined over $\Sigma = \{a, b\}$, containing aa.

THEOREM 7

- **for every NFA, there is some FA that accepts exactly the same language.**

THEOREM 7

- **for every NFA, there is some FA that accepts exactly the same language.**
- **Proof 1**

THEOREM 7

- **for every NFA, there is some FA that accepts exactly the same language.**
- **Proof 1**

- By the proof of part 2 of Kleene's theorem, we can convert an NFA into a regular expression, since an NFA is a TG.

THEOREM 7

- **for every NFA, there is some FA that accepts exactly the same language.**
- **Proof 1**

- By the proof of part 2 of Kleene’s theorem, we can convert an NFA into a regular expression, since an NFA is a TG.

- By the proof of part 3 of Kleene’s theorem, we can construct an FA that accepts the same language as the regular expression. Hence, for every

THEOREM 7

- **for every NFA, there is some FA that accepts exactly the same language.**
- **Proof 1**

- By the proof of part 2 of Kleene's theorem, we can convert an NFA into a regular expression, since an NFA is a TG.

- By the proof of part 3 of Kleene's theorem, we can construct an FA that accepts the same language as the regular expression. Hence, for every

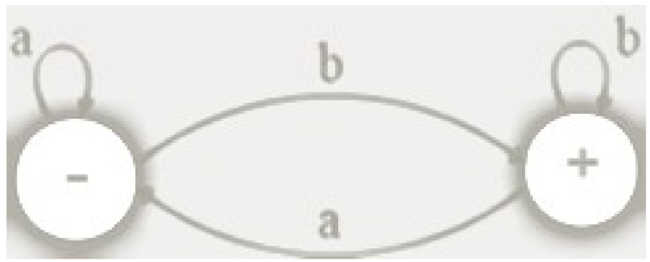
- NFA, there is a corresponding FA.

NOTE

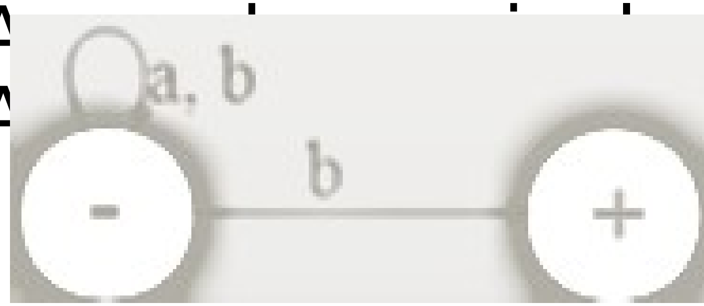
- Theorem 7 means that all NFAs can be converted into FAs.
- Clearly, all FAs can be considered as NFAs that do not make use of the option of extra freedom of edge production.
- Hence, as language acceptors, NFA = FA.

EXAMPLE

Consider the following FA corresponding to (a+b)*b



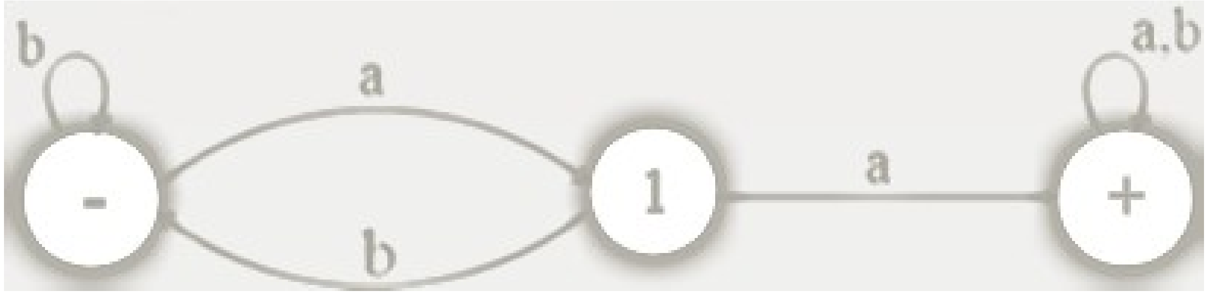
The above FA ... nt to the following NFA



Can the structure of above NFA be compared with the corresponding RE ?

EXAMPLE

Consider the following FA



The above FA may be equivalent to the following NFA

Can the structure of above NFA be compared with the corresponding RE ?

NOTE

- It is to be noted that every FA can be considered to be an NFA as well , but the converse may not true.
- It may also be noted that every NFA can be considered to be a TG as well, but the converse may not true.
- It may be observed that if the transition of null string is also allowed at any state of an NFA then what will be the behavior in the new structure. This structure is defined in the following

NFA WITH NULL STRING

<u>Definition</u>: If in an NFA, Λ is allowed to be a label of an edge then the NFA is called NFA with Λ (NFA- Λ). An NFA- Λ is a collection of three things

(1) Finite many states with one initial and some final states.

(2) Finite set of input letters, say, Σ ={a, b, c}.

(3) Finite set of transitions, showing where to move if a letter is input at certain state. There may be more than one transitions for certain letter and there may not be any transition for a certain letter. The transition of Λ is also

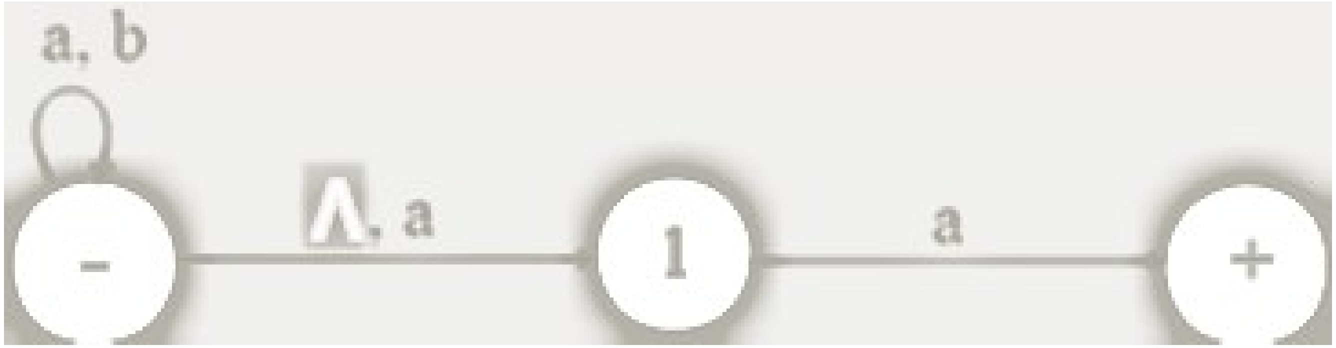
EXAMPLE

Consider the following NFA with Null string

The above NFA with Null string accepts the language of strings, defined over Σ = {a, b}, **ending in b**.

EXAMPLE

Consider the following NFA with Null string



The above NFA with Null string accepts the language of strings, defined over Σ = {a, b}, **ending in a**.

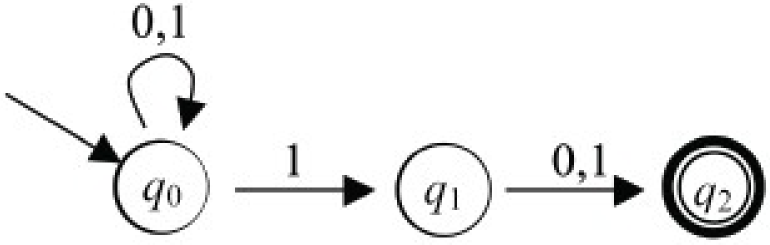
EXAMPLE

It is to be noted that every FA may be considered to be an NFA- Λ as well, but the converse may not true.

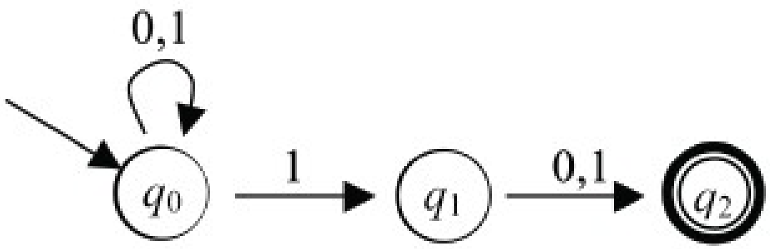
Similarly every NFA- Λ may be considered to be a TG as well, but the converse may not true.

NFA TO FA

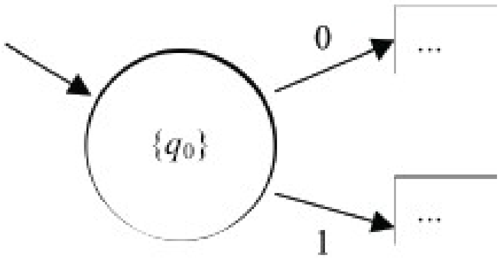
- **For any NFA, there is a DFA that recognizes the same language**
- Proof is by construction: a DFA that keeps track of the set of states the NFA might be in
- This is called the *subset construction*
- First, an example starting from this NFA:



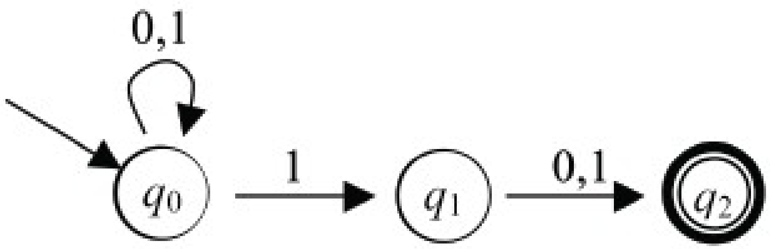
NFA TO FA



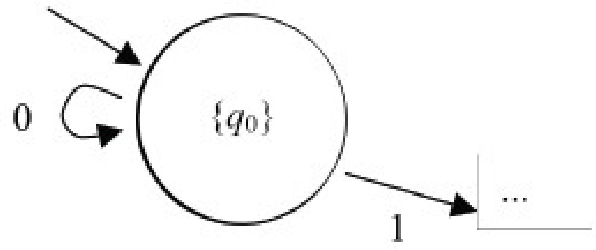
- Initially, the set of states the NFA could be in is just {*q0*}
- So our DFA will keep track of that using a start state labeled {*q0*}*:*



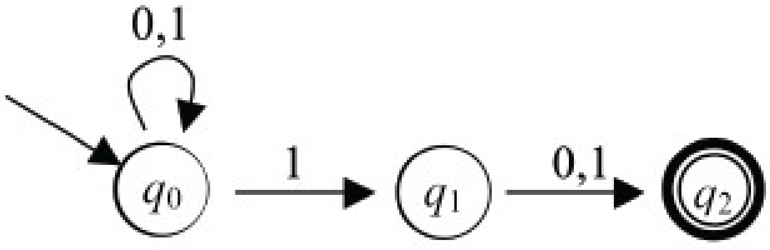
NFA TO FA



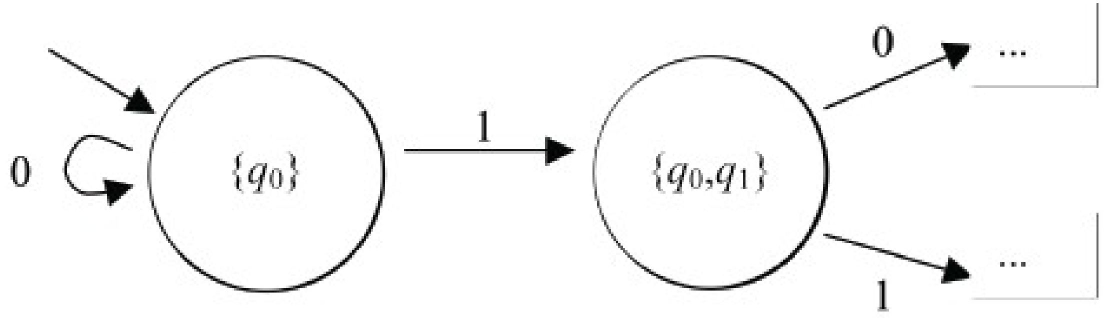
- Now suppose the set of states the NFA could be in is {*q0}, and it reads a 0*
- The set of possible states after reading the 0 is {*q0}, so we can show that transition:*



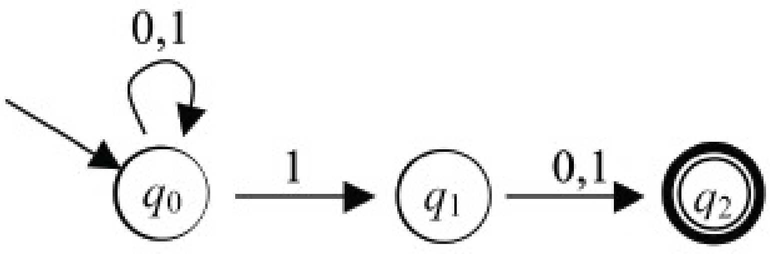
NFA TO FA



- Suppose the set of states the NFA could be in is {*q0}, and it reads a 1*
- The set of possible states after reading the 1 is {*q0,q1}, so we need another state:*

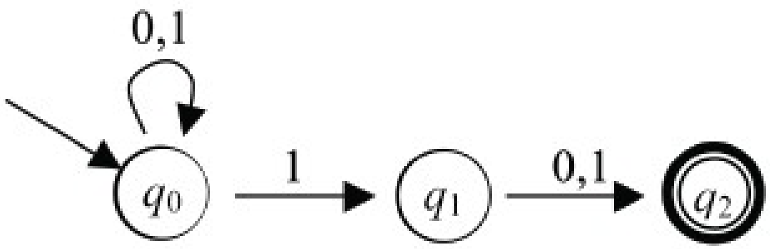


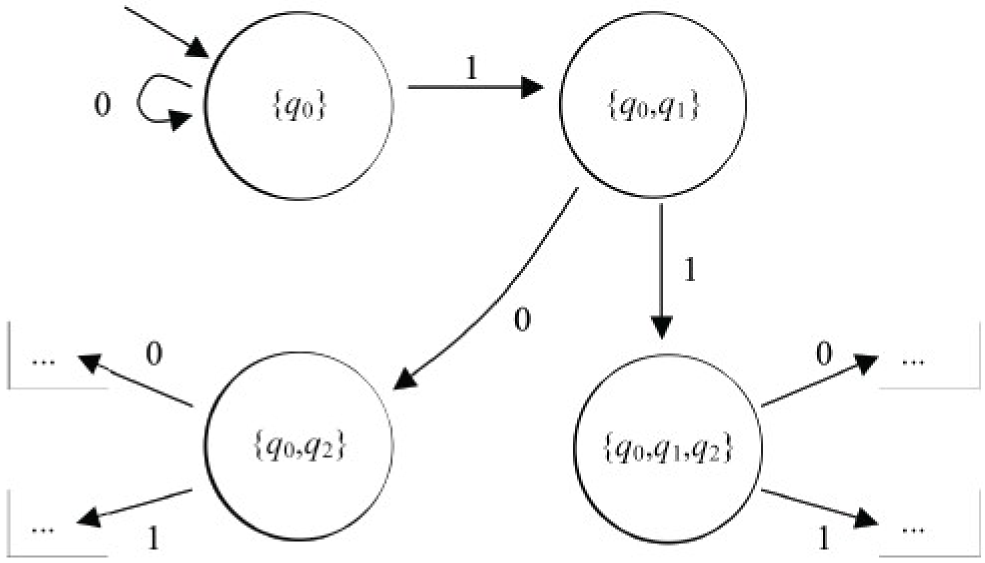
NFA TO FA



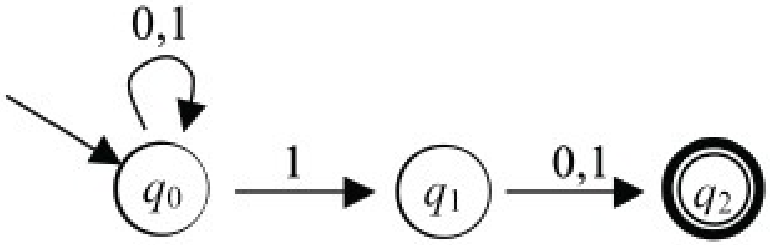
- From {*q0,q1} on a 0, the next set of possible* states is δ(*q0,0) ∪ δ(q1,0) = {q0,q2}*
- From {*q0,q1} on a 1, the next set of possible* states is δ(*q0,1) ∪ δ(q1,1) = {q0,q1,q2}*
- Adding these transitions and states, we get…

NFA TO FA



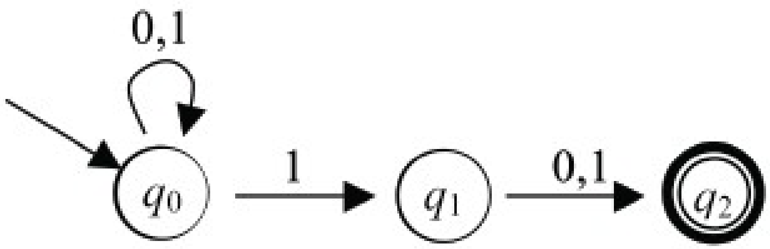


NFA TO FA

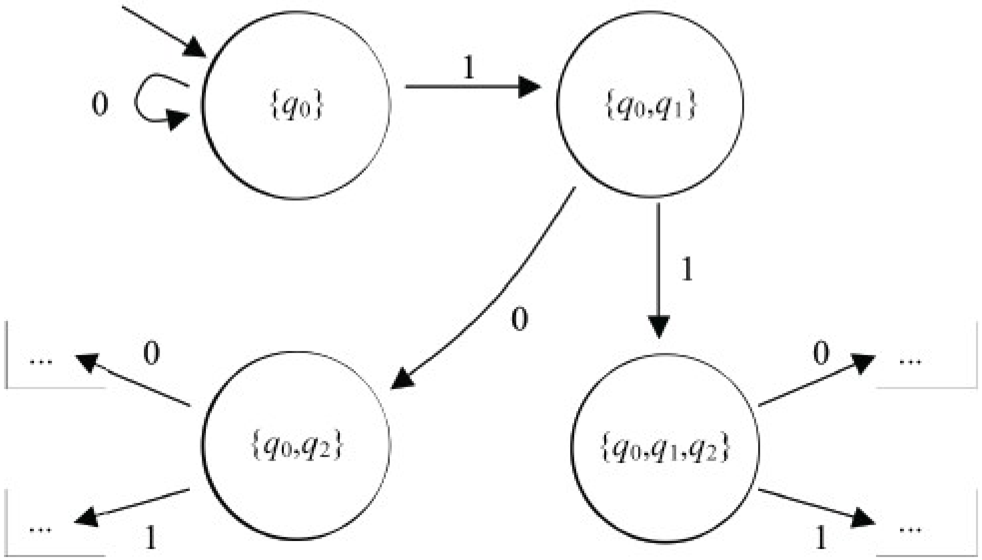


- Eventually, we find that no further states are generated
- That's because there are only finitely many possible sets of states: *P(Q)*
- In our example, we have already found all sets of states reachable from {*q0*}...

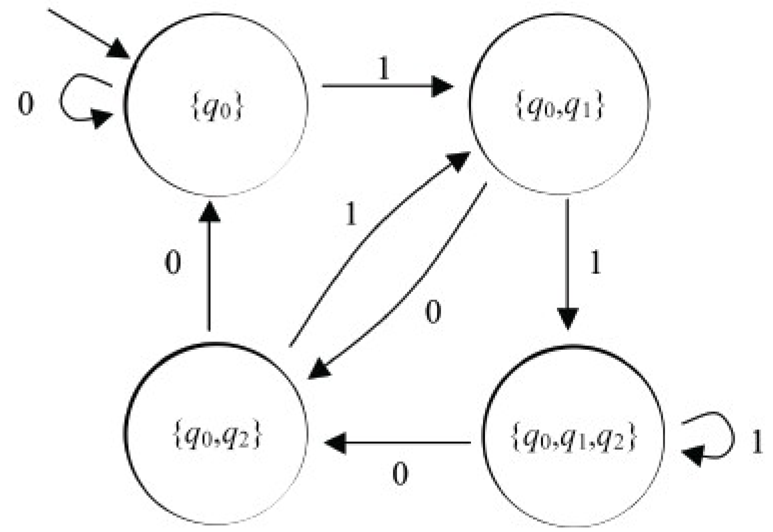
NFA TO FA



0,1
1
0,1
q_0
q_1
q_2

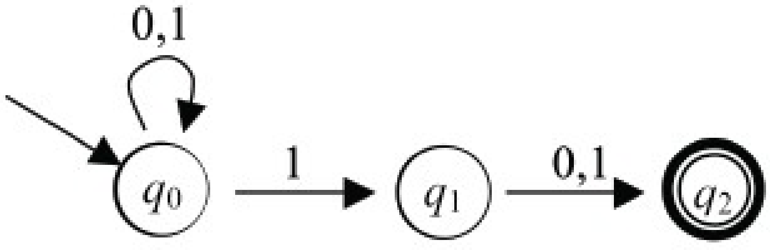


0
1
$\{q_0\}$
$\{q_0,q_1\}$
1
0
0
0
...
...
$\{q_0,q_2\}$
$\{q_0,q_1,q_2\}$
1
1
...
...



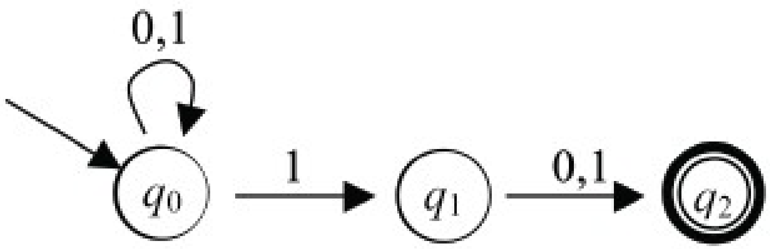
0
1
$\{q_0\}$
$\{q_0,q_1\}$
1
0
1
0
$\{q_0,q_2\}$
$\{q_0,q_1,q_2\}$
0
1

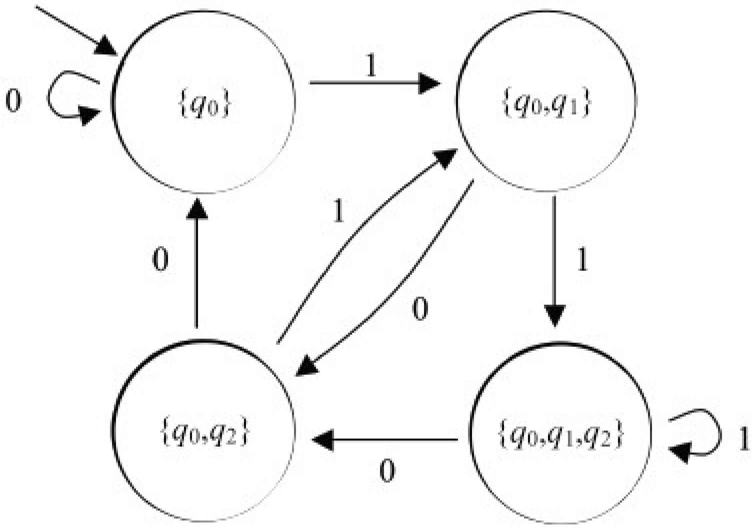
NFA TO FA

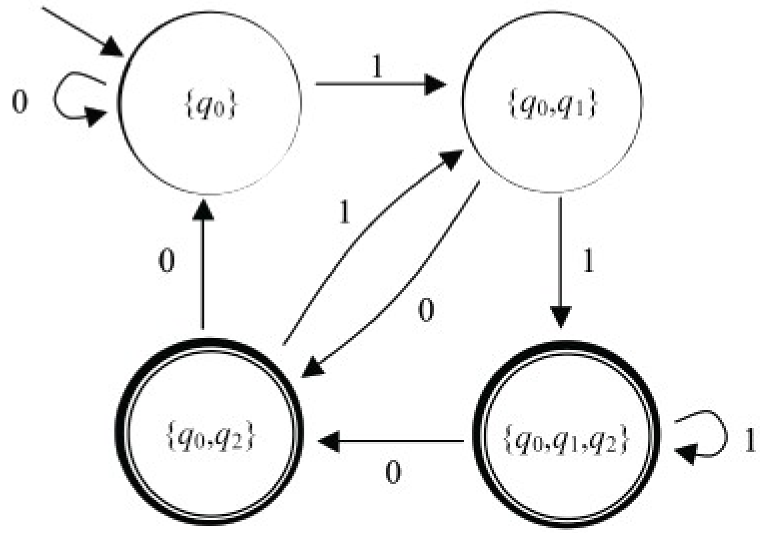


- It only remains to choose the accepting states
- An NFA accepts *x if its set of possible states* after reading *x includes at least one accepting* state
- So our DFA should accept in all sets that contain at least one NFA accepting state

NFA TO FA


0,1
q_0
1
q_1
0,1
q_2


0
{q_0}
1
{q_0,q_1}
1
0
0
1
{q_0,q_2}
0
{q_0,q_1,q_2}
1


0
{q_0}
1
{q_0,q_1}
1
0
0
1
{q_0,q_2}
0
{q_0,q_1,q_2}
1