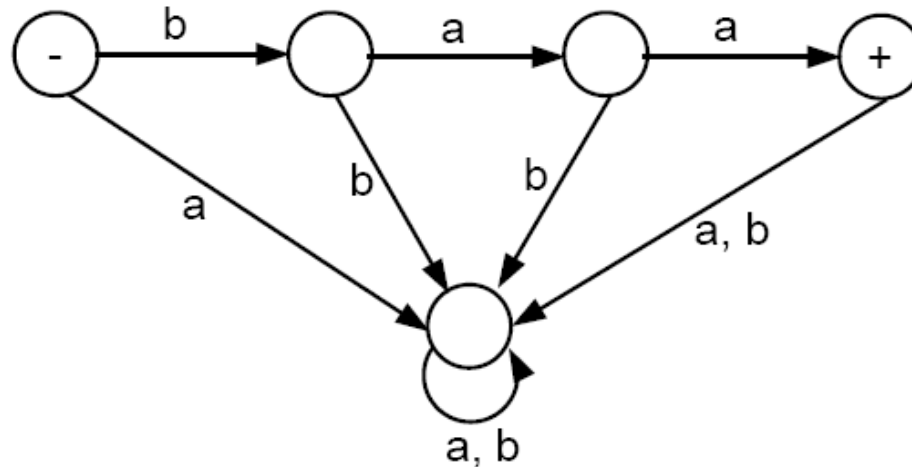


# **Transition Graphs (TG)**

Shakir Ullah Shah

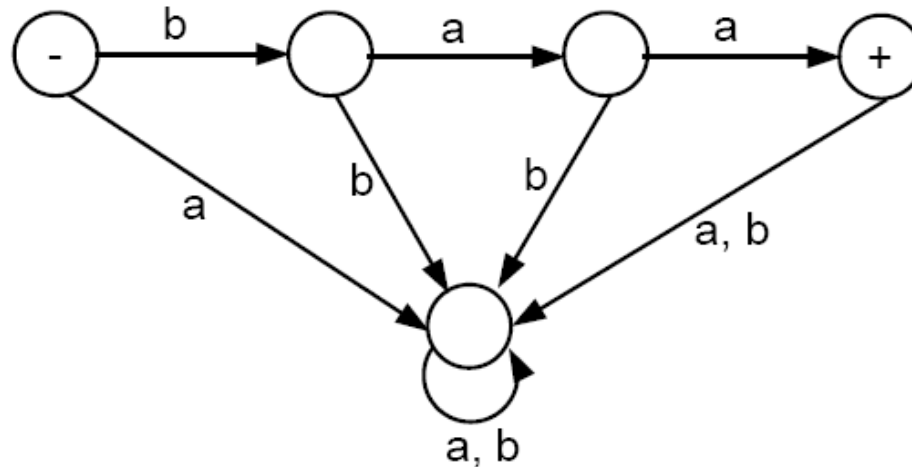
## **Lecture 6**

# Relaxing the Restriction on Inputs



- The language accepted by this FA is  $L = \{baa\}$

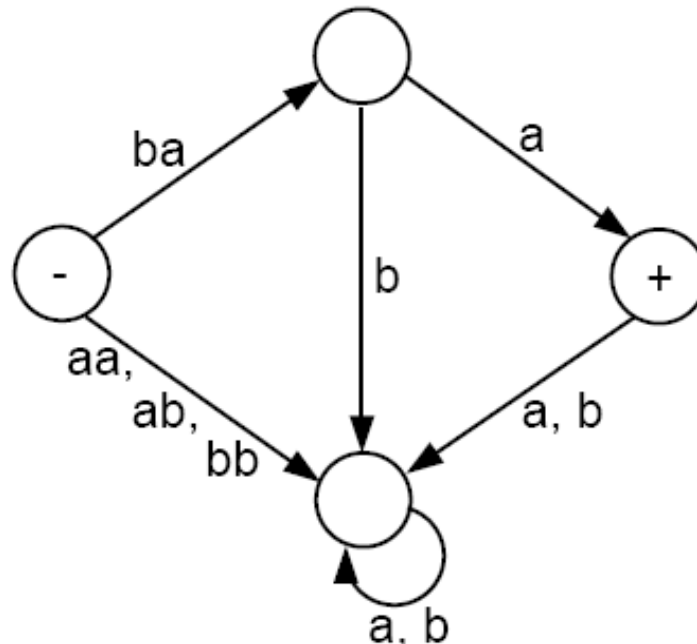
# Relaxing the Restriction on Inputs



- Even the string *baabb* will fail.

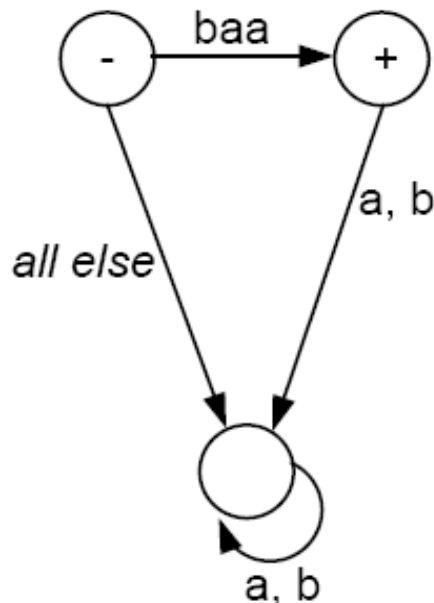
# Relaxing the Restriction on Inputs

- Suppose we now allow a machine to read *either one or two letters* of the input string at a time. Then we may design a machine that accepts only the word *baa* with fewer states as the one below:



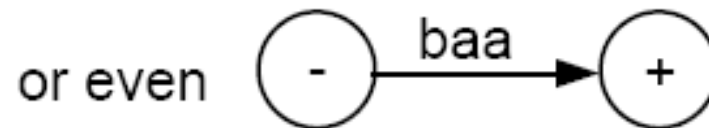
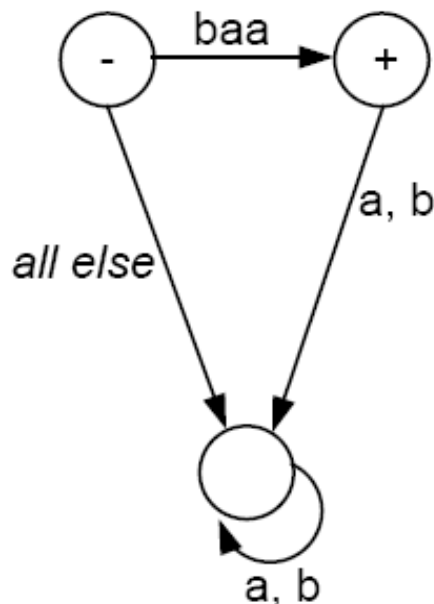
# Relaxing the Restriction on Inputs

- If we go further to allow a machine to read *up to three letters* of the input string at a time, then we may design the machine accepting only the word baa with even fewer states as follows:

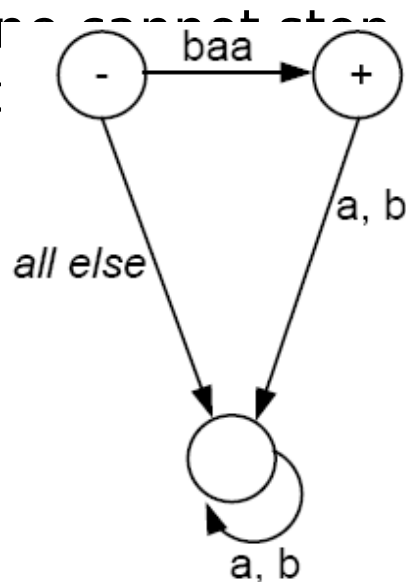


# Relaxing the Restriction on Inputs

- The picture on the right introduces some problems:
  - If we begin in the start (-) state and the first letter of the input is an a, we have no direction as to what to do.



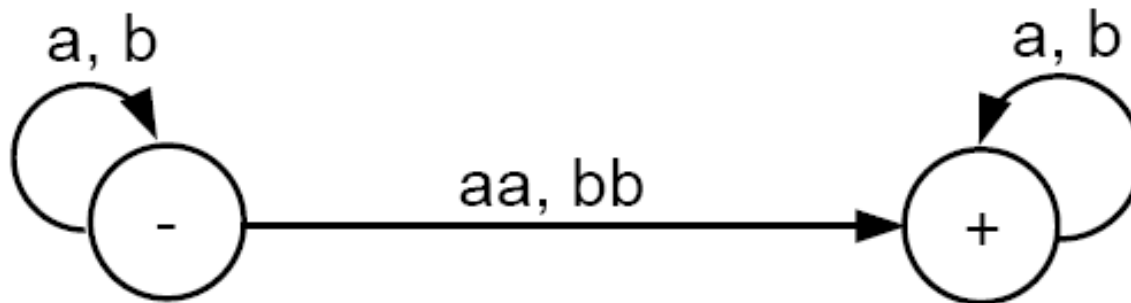
- The picture on the right introduces some problems:
  - We also have problem even with the input baaa. The first three letters take us to the accept state, but then the picture does not tell us where to go when the last a is read. (Remember that according to the rules of FAs, one must keep reading input letters until the input runs out.)



- There are now two different ways that an input can be rejected:
  - (i) It could peacefully trace a path ending in a non-final state, or
  - (ii) it could crash while being processed.

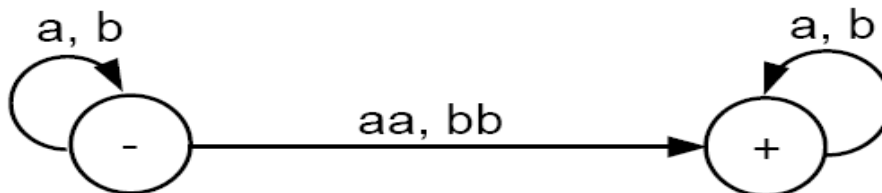


- all words containing a double letter (aa or bb)



all words containing a double letter (aa or bb)

- Let the input is baa.
- If we first read b and then read aa we will go to the final state. Hence, the string is accepted.
- If we first read b, then read a, and then read a, we will loop back and be stuck at the start state. Hence, the string is rejected in this case.
- If we first read two letters ba at once, then there is no edge to tell us where to go. So, the machine crashes and the input string is rejected.
- What shall we say? Is this input string a word in the language of this machine or not?



- **We shall say that a string is accepted if there is **some way** it could be processed so as to arrive at a final state.**
- Due to many difficulties inherent in expanding our definition of machine to reading more than one letter of input at a time, we shall leave the definition of finite automaton alone and call these new machines **transition graphs**.
- Transition graphs were invented by John Myhill in 1957

# Transition Graph (Definition)

- Method 5 (**Transition Graph**)

**Definition:** A Transition graph (TG), is a collection of the followings

1. Finite number of states, at least one of which is start state and some (maybe none) final states.

# Transition Graph (Definition)

- Method 5 (**Transition Graph**)

**Definition:** A Transition graph (TG), is a collection of the followings

1. Finite number of states, at least one of which is start state and some (maybe none) final states.
2. Finite set of input letters ( $\Sigma$ ) from which input strings are formed.

# Transition Graph (Definition)

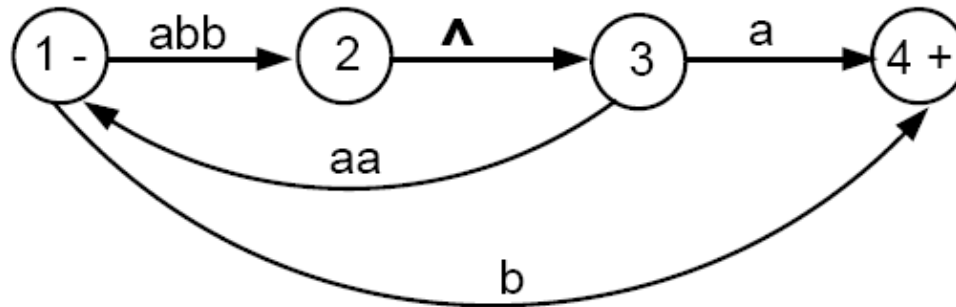
- Method 5 (**Transition Graph**)

**Definition:** A Transition graph (TG), is a collection of the followings

1. Finite number of states, at least one of which is start state and some (maybe none) final states.
2. Finite set of input letters ( $\Sigma$ ) from which input strings are formed.
3. Finite set of transitions that show how to go from one state to another based on reading specified substrings of input letters, possibly even the null string  $\Lambda$ .

- It is to be noted that in TG there **may exist more than one paths** for certain **string**, while there may not exist any path for certain string as well.
- If there exists **at least one path** for a certain string, starting from initial state and ending in a final state, the string is supposed to be accepted by the TG, otherwise the string is supposed to be rejected. Obviously collection of accepted strings is the language accepted by the TG.

- consider the following TG:

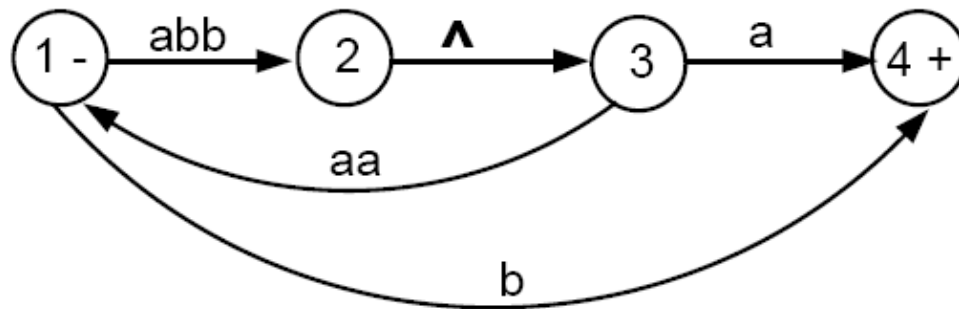


- The path from state 1 to state 2 to state 3 back to state 1 and then to the final state 4 corresponds to the string  $(abb)(\Lambda)(aa)(b) = abbaab$ .
- Some other accepted words are *abba*, *abbaaabba*, and *b*.
- When an edge is labeled with  $\Lambda$ , it means that we can take the ride it offers for free (without consuming any letter from the input string).

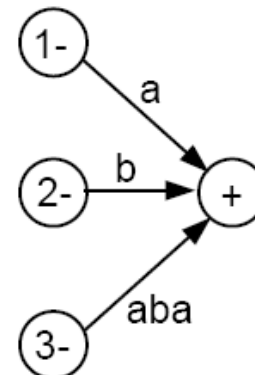
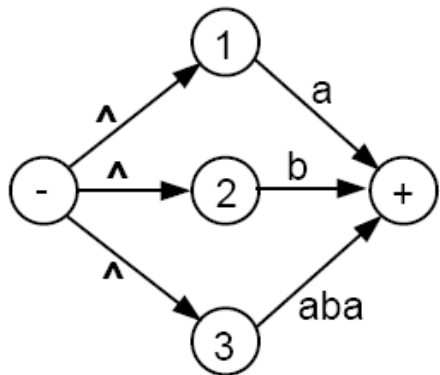


- If we are presented with a particular string to run on a given TG, we must decide how to break the string into substrings that may correspond to the labels of edges in the TG.

- Let's run the input string `abbab` on the machine:
  - The substring `abb` takes us from state 1 to state 2.
  - We move to state 3 along the  $\Lambda$ -edge without any substring being consumed.
  - We are now in state 3 and what is left of the input string is the substring `ab`. We cannot read `aa`, so we must read only `a` and go to state 4.
  - At state 4, we have `b` left in the input string but no edge to follow, so we must crash and reject the input string `abbab`.

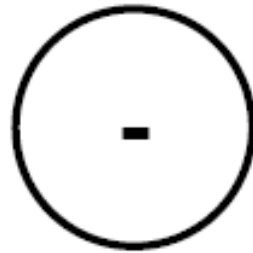


- Because we allow some edges to be traversed for free, it is logical to allow for the possibility of more than one start state, as illustrated below:



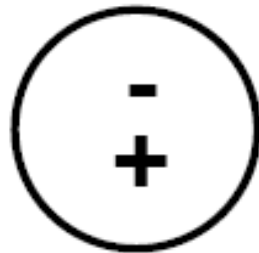
- These two machines are equivalent, in the sense that all the strings accepted by the first are accepted by the second and vice versa.
- Important Note: **Every FA is also a TG. However, NOT every TG satisfies the definition of an FA.**

# Example



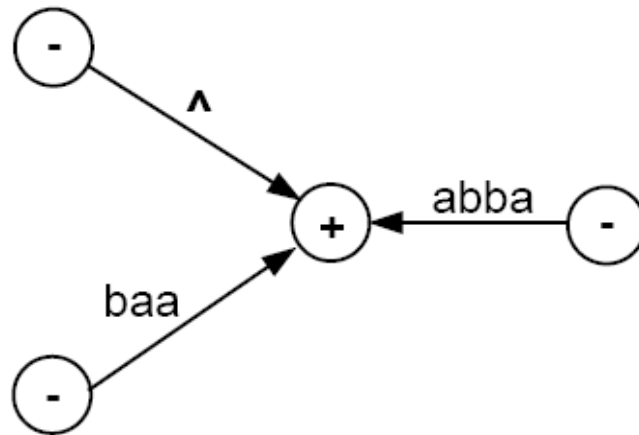
- This TG accepts nothing, not even the null string.
- To be able to accept anything, it must have a final state.

# Example



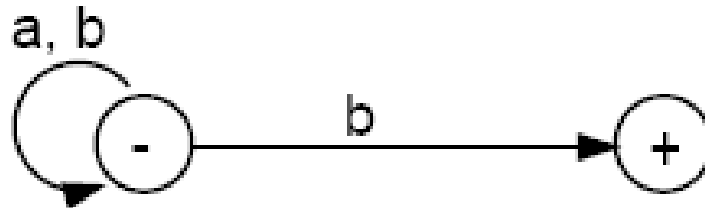
- This TG accepts only the null string  $\Lambda$ .
- Any other string cannot have a successful path to the final state through labels of edges because there are no edges (and hence no labels).
- Any TG in which some start state is also a final state will always accept the null string  $\Lambda$ . This is also true of FAs.

# Example



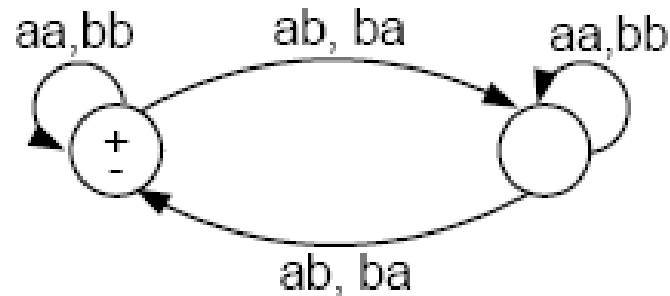
- This machine accepts only the words  $\Lambda$ , baa, and abba.
- Anything read while in the final state will cause a crash, because the final state has no outgoing edges.

# Example



- We can read all the input letters, one at a time, and stay in the left-side state. When we read a b, if it is the very last letter of the input string, we can use it to go to the final state. Note that this b must be the very last letter, because once we are in the right-side state, if we try to read another letter, we will crash.
- It is possible for an input string ending with a b to follow an unsuccessful path that does not lead to acceptance (e.g., following the b-edge too soon and crash, or looping back to the - state when reading the last b).
- However, all words ending with a b can be accepted by some path. Hence, the language accepted by this TG is  $(a + b)^*b$ .

# Example



- In this TG, every edge is labeled with a pair of letters. Thus, for a string to be accepted, it must have an even number of letters.
- This TG accepts exactly the language EVEN - EVEN.
- Recall that EVEN - EVEN is the language of all words containing an even number of a's and an even number of b's, including the null string  $\Lambda$ .



# **Generalized Transition Graphs (GTG)**

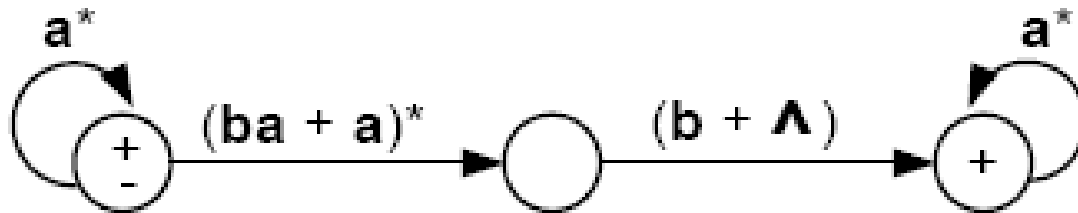
# Definition

A **generalized transition graph (GTG)** is a collection of three things:

1. A finite set of states, of which at least one is a start state and some (maybe none) are final states.
2. An alphabet  $\Sigma$  of input letters.
3. Directed edges connecting some pairs of states, each labeled with a regular expression.

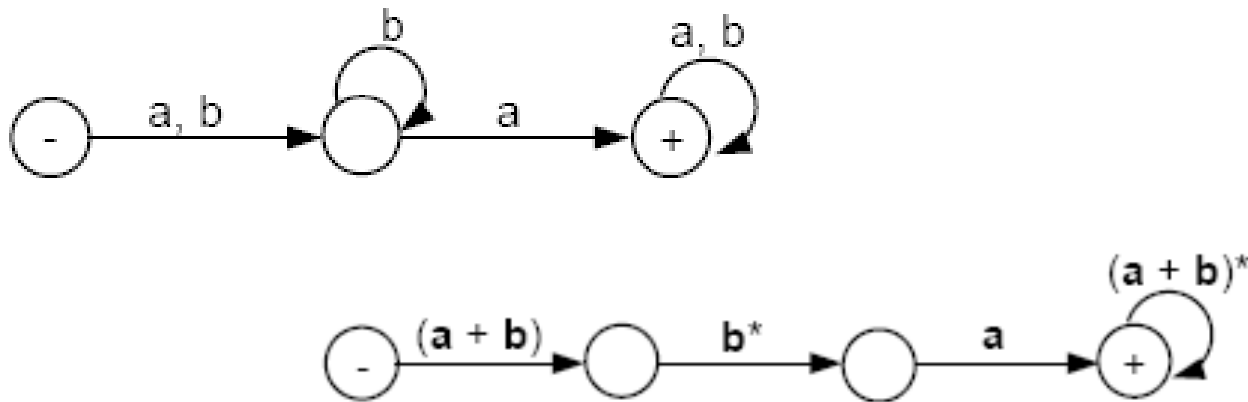
# Example

- Consider this GTG:



- This GTG accepts all strings without a double b.
- Note that the word containing the single letter b can take the free ride along the  $\Lambda$ -edge from start to middle, and then have letter b read to go to the final state.

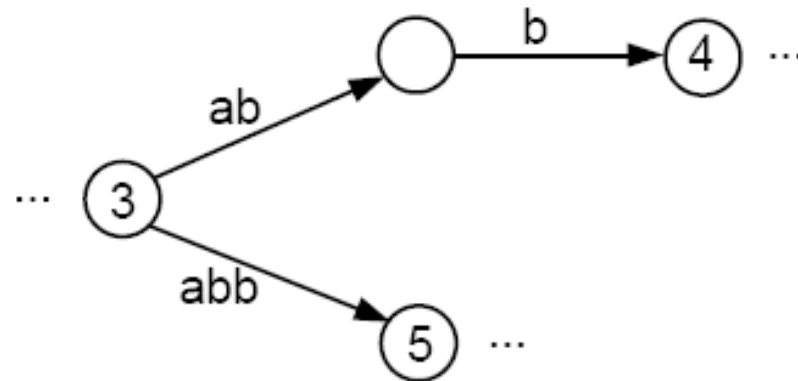
- Note that there is no difference between the Kleene star closure for regular expressions and a loop in transition graphs, as illustrated in the following figure:



- In the first picture we may loop in the middle state or go to the third state. To not loop corresponds to taking the  $\wedge$  choice from the  $b^*$ -edge in the second picture.

# NonDeterminisim

- We have already seen that in a TG, a particular string of input letters may trace through the machine on different paths, depending on our choice of grouping.



- This figure shows part of some TG.
- The input string *abb* can go from state 3 to state 4, or to state 5, depending on whether we read the letters two and one, or all three at once.

# NonDeterminisim

- The ultimate path through the machine is NOT determined by the input alone. Human choice becomes a factor in selecting the path. The machine does not make all its own determination.
- Therefore, we say that TGs are **nondeterministic**.

# Some ways to define a language

- A language can be defined by:
  - (i) **Regular Expression (RE),**
  - (ii) **Finite Automaton (FA),**
  - (iii) **Transition Graph (TG).**

# Kleene's Theorem

If a language can be expressed by

1. FA or
2. TG or
3. RE

then it can also be expressed by other two as well.

- In other words, Kleene proved that **all three of these methods of defining languages are *equivalent*.**



# Kleene's Theorem (contd.)

- To prove Kleene's theorem, we need to prove 3 parts:
- **Part 1:** Every language that can be defined by a finite automaton can also be defined by a transition graph.
- **Part 2:** Every language that can be defined by a transition graph can also be defined by a regular expression.
- **Part 3:** Every language that can be defined by a regular expression can also be defined by a finite automaton.

# Proof(Kleene's Theorem Part I)

- This is the easiest part.

# Proof(Kleene's Theorem Part I)

- This is the easiest part.
- Since every FA can be considered to be a TG as well, *therefore there is nothing to prove.*

# Proof of Part 2: Turning TGs into Regular Expressions

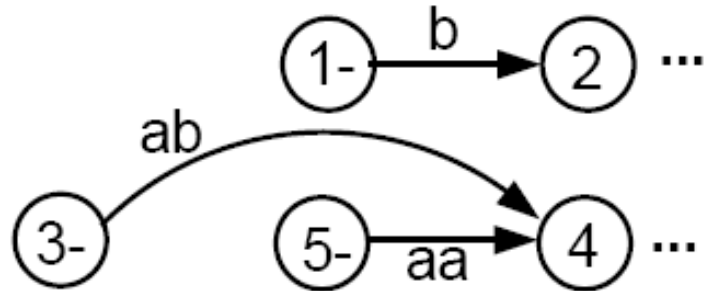
- We prove this part by providing a **constructive algorithm**:
  - We present a algorithm that starts out with a transition graph and ends up with a regular expression that defines the same language.

- **Step 1: Creating A Unique Start State**

If a TG has more than one start states, then introduce a new start state connecting the new state to the old start states by the transitions labeled by  $\Lambda$  and make the old start states the non-start states. This step can be shown by the following example

# Step 1: Creating A Unique Start State

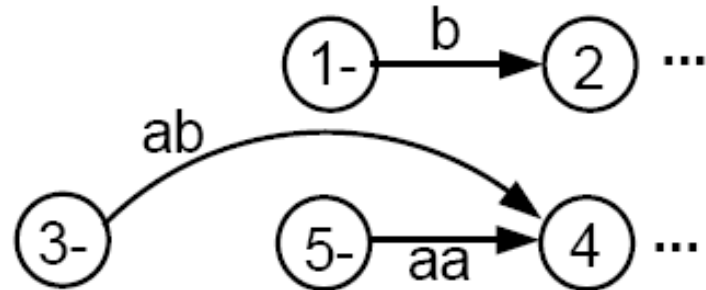
- Consider a fragment of T:



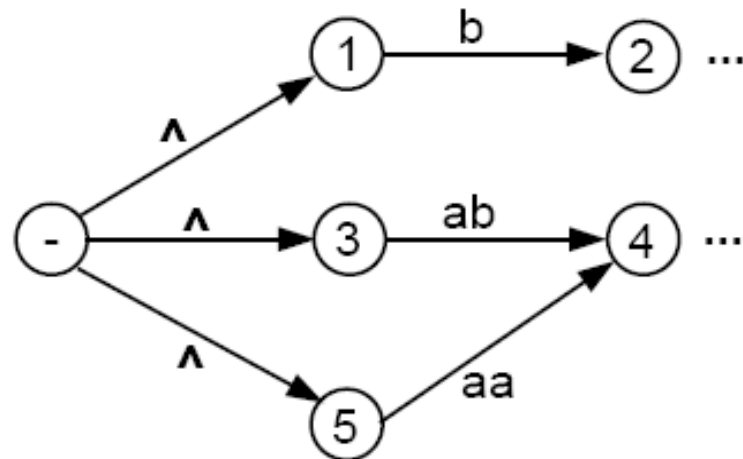
- The above fragment of T can be replaced by

# Step 1: Creating A Unique Start State

- Consider a fragment of T:



- The above fragment of T can be replaced



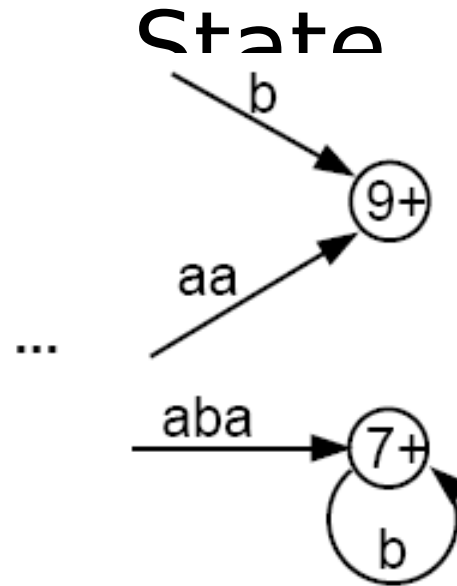
## **Step 2:** Creating a Unique Final State

If  $T$  had no final state, then it accepts no strings at all and has no language. So, we need to produce no regular expression other than the null, or empty, expression  $\Phi$

If a TG has more than one final states, then introduce a new final state, connecting the old final states to the new final state by the transitions labeled by  $\Lambda$  as depicted in the next slide

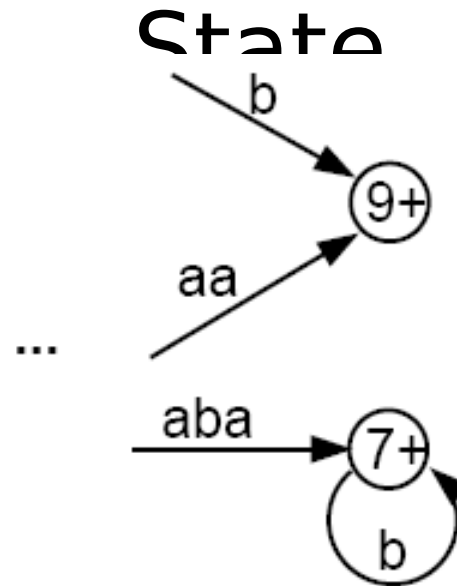


## Step 2: Creating a Unique Final

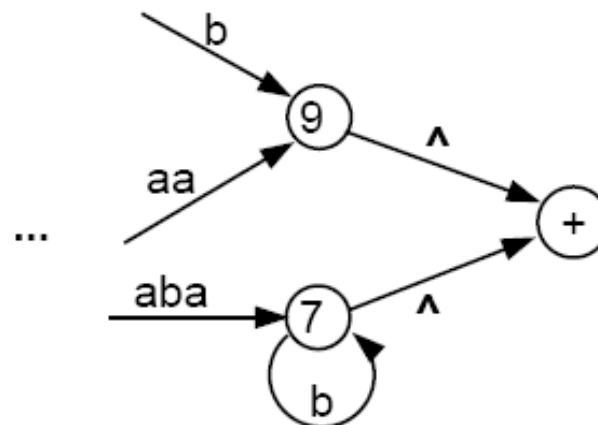


becomes

# Step 2: Creating a Unique Final




becomes



## Step 2: Creating a Unique Final State

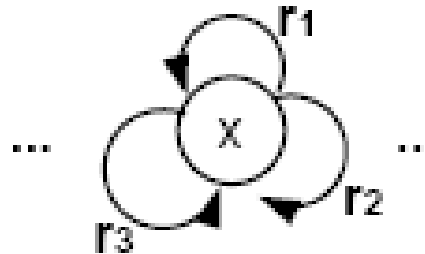
- We shall require that the unique final state be a different state from the unique start state. If an old state used to have  $\pm$ , then both signs are removed from the old state to newly created states, using the processes described above.
- It should be clear that the addition of these two new states does not affect the language that  $T$  accepts.

- The machine ing shape:

where there are no other - or + states.

## Step 3 : Combining Edges

- If  $T$  has some internal state  $x$  (not the - or the + state) that has more than one loop circling back to itself:

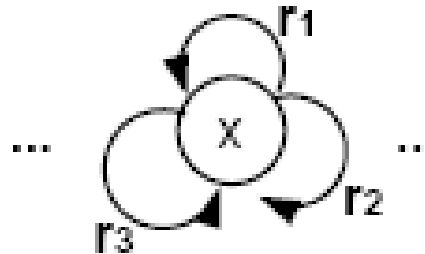


where  $r_1$ ,  $r_2$ , and  $r_3$  are all regular expressions or simple strings.

- We can replace the three loops by one loop labeled with a regular expression:

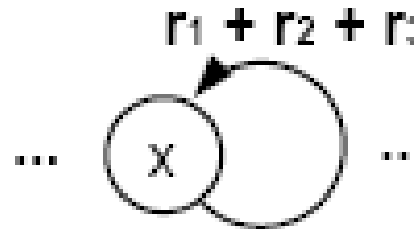
## Step 3 : Combining Edges

- If  $T$  has some internal state  $x$  (not the - or the + state) that has more than one loop circling back to itself:



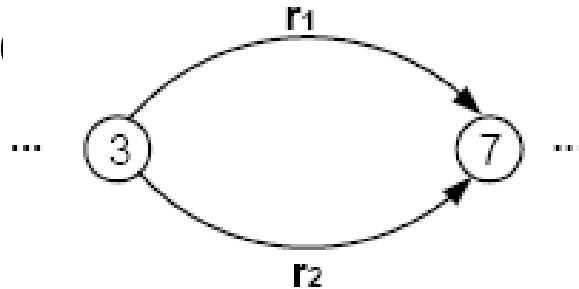
where  $r_1$ ,  $r_2$ , and  $r_3$  are all regular expressions or simple strings.

- We can replace the three loops by one loop labeled with a regul



## Step 3 : Combining Edges

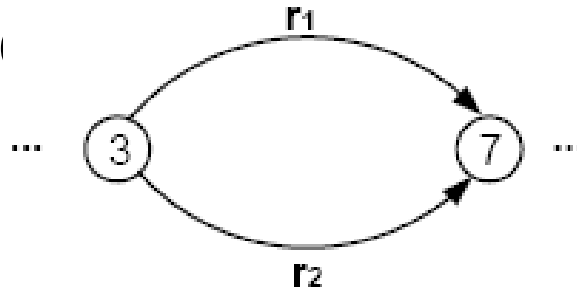
- Similarly, if two states are connected by **more than one edge** going in the same direction



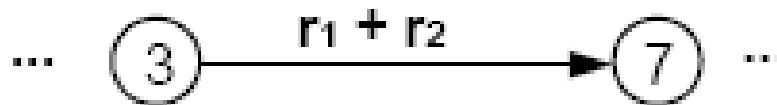
- We can replace this with a single edge labeled with a regular expression:

## Step 3 : Combining Edges

- Similarly, if two states are connected by **more than one edge** going in the same direction

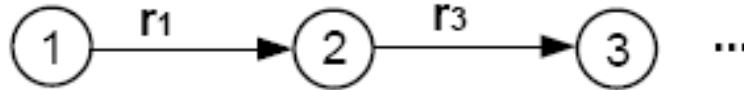


- We can replace this with a single edge labeled with a regular expression:



## Step 4: Bypass and State Elimination

- If T has 3 states in a row connected by edges labeled with regular expressions or simple strings, we can eliminate the middle state, as in the following example...

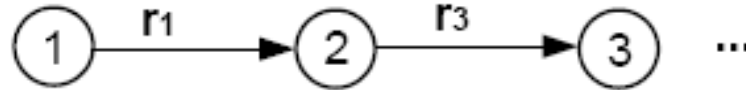


Can be replaced with



## Step 4: Bypass and State Elimination

- If T has 3 states in a row connected by edges labeled with regular expressions or simple strings, we can eliminate the middle state, as in the following example...

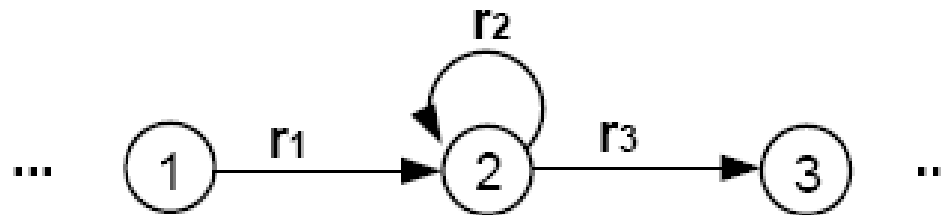


Can be replaced with



## Step 4: Bypass and State Elimination

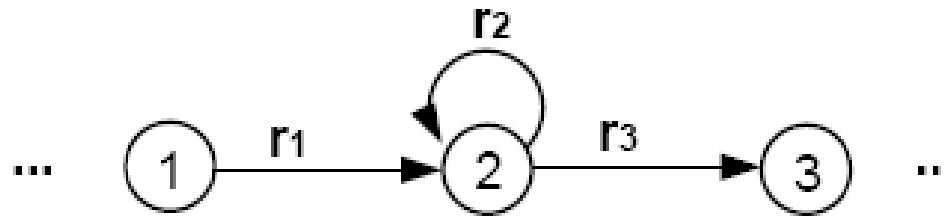
- If the middle state has a loop, we can proceed as follows:



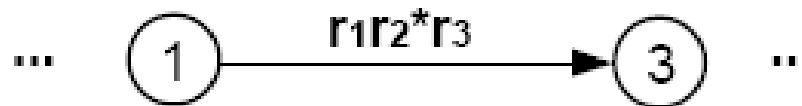
Can be replaced with

## Step 4: Bypass and State Elimination

- If the middle state has a loop, we can proceed as follows:

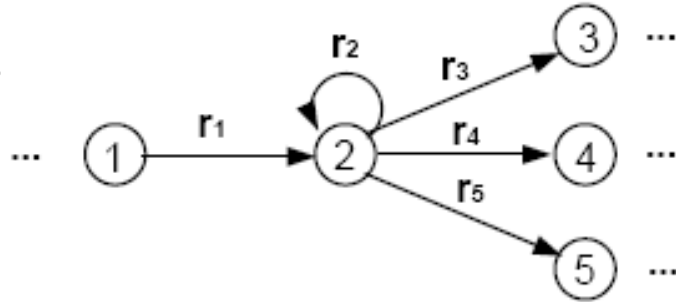


Can be replaced with



## Step 4: Bypass and State Elimination

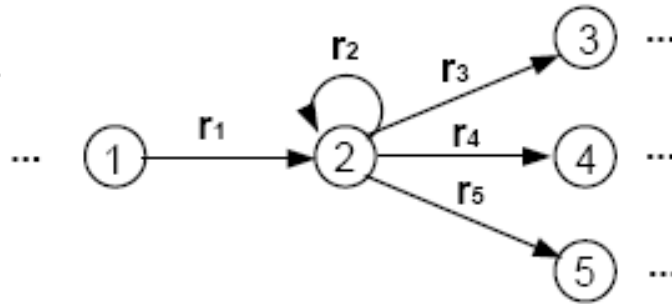
- If the middle state is connected to more than one state, then the bypass and elimination process can be done as follows



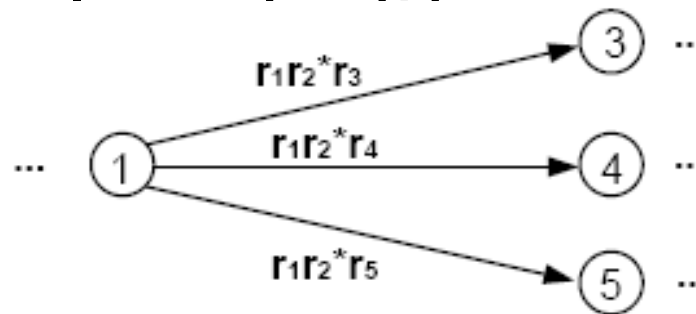
Can be replaced with

## Step 4: Bypass and State Elimination

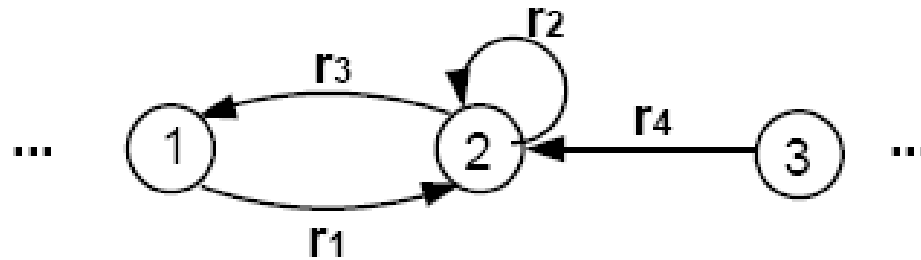
- If the middle state is connected to more than one state, then the bypass and elimination process can be done as follows



Can be re

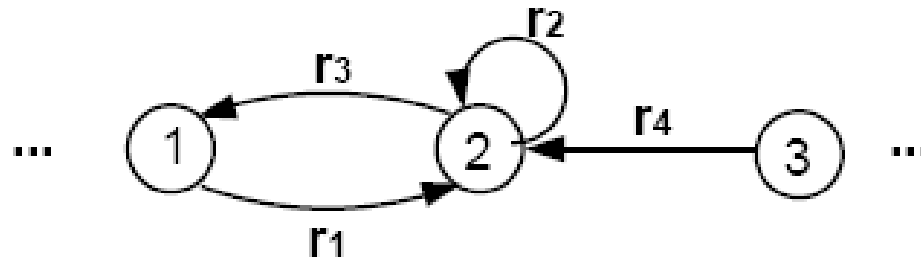


# Special Cases

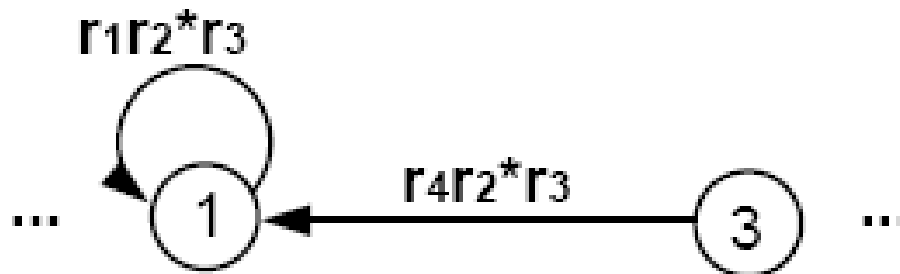


Can be replaced with

# Special Cases

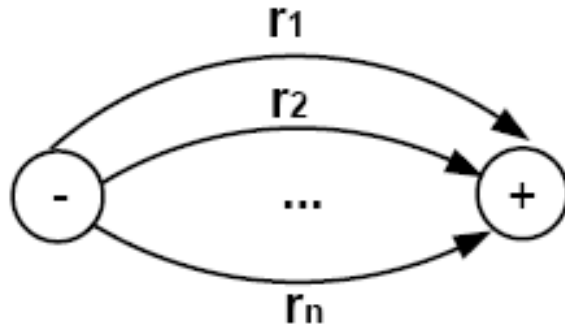


Can be replaced with



# Combining Edges

- We can repeat this bypass and elimination process again and again until we have eliminated all the states from  $T$ , except for the unique start state and the unique final state.

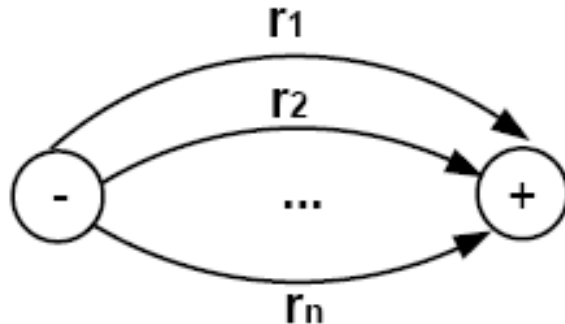


- We can then combine the edges from the above picture one more to produce

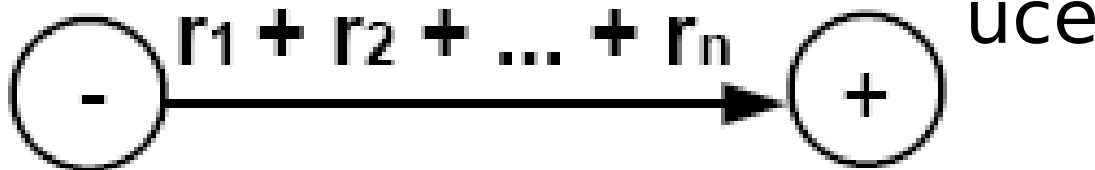


# Combining Edges

- We can repeat this bypass and elimination process again and again until we have eliminated all the states from  $T$ , except for the unique start state and the unique final state.

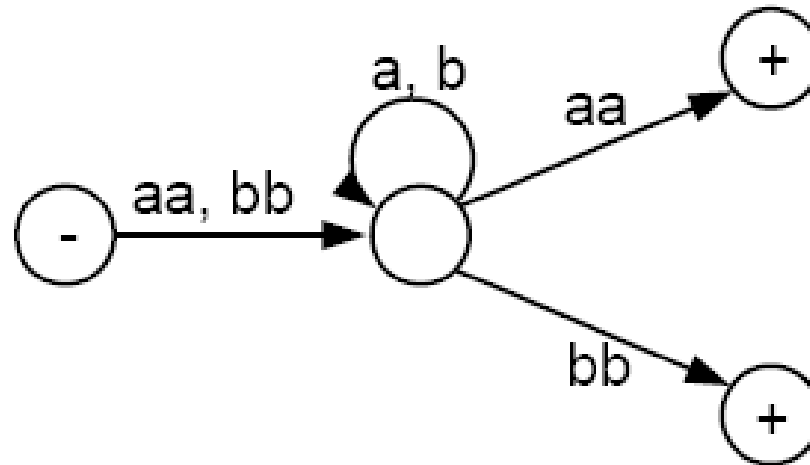


- We can then combine the edges from the above



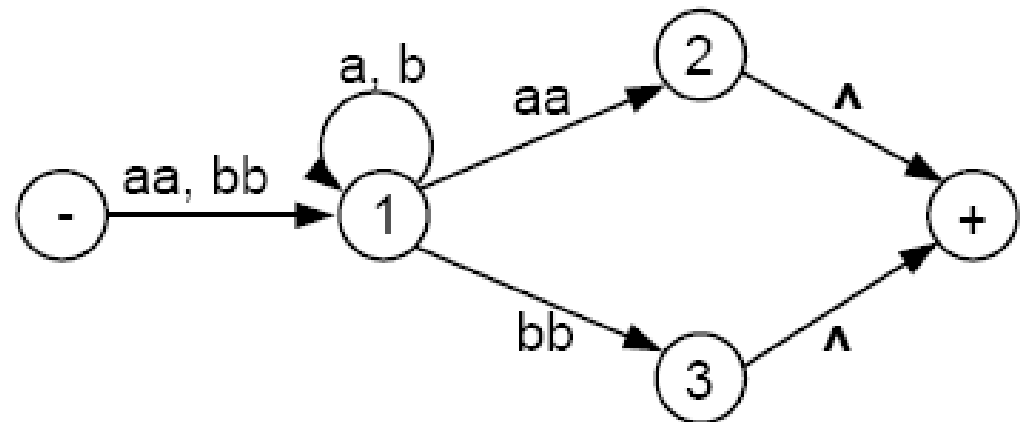
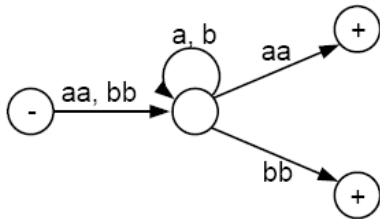
# Example 1

- Consider the following TG that accepts all words that begin and end with double letters (having at least length 4):



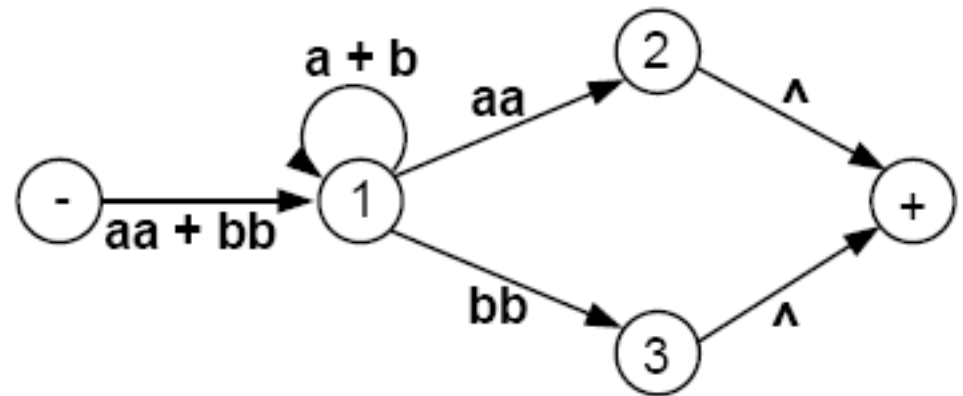
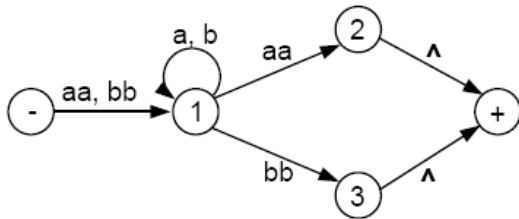
# Example 1

- This TG has only one start state with no incoming edges, but has two final states. So, we must introduce a new unique final state



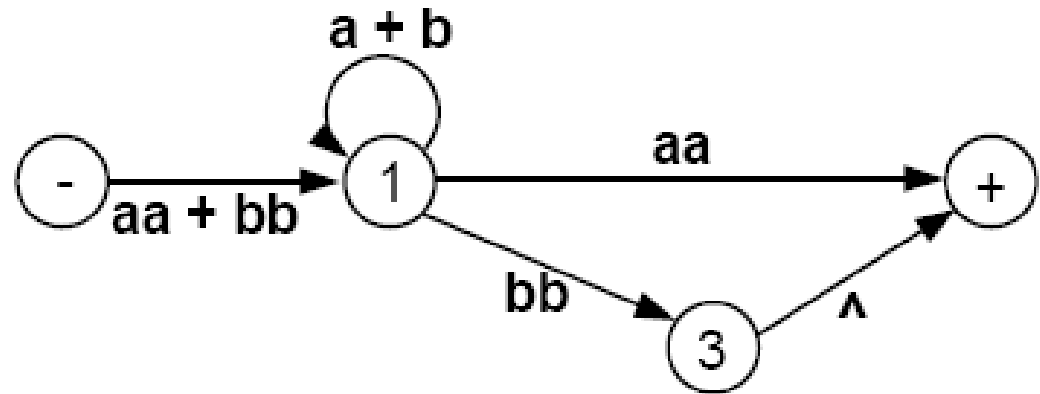
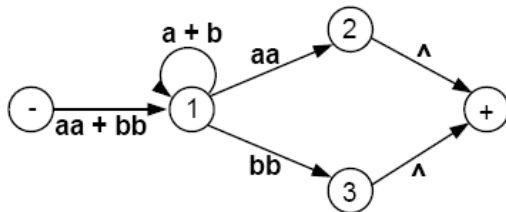
# Example 1

- Now we build regular expressions piece by piece:



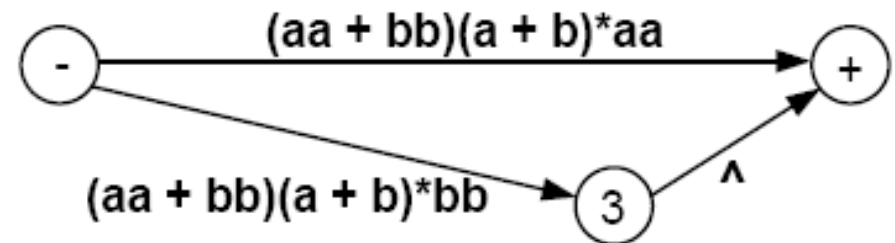
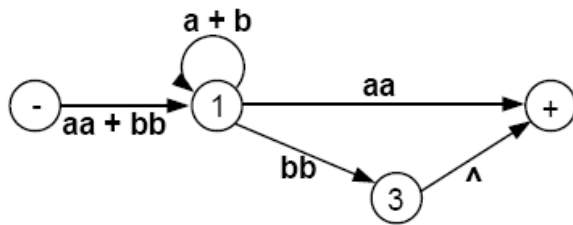
# Example 1

- Eliminate state 2:



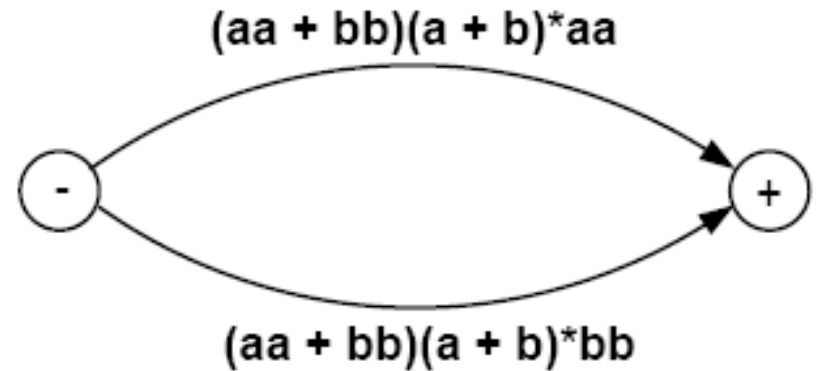
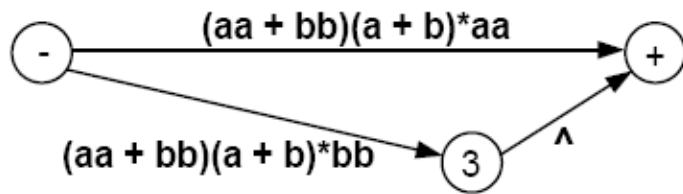
# Example 1

Eliminate state 1:



# Example 1

Eliminate state 3:



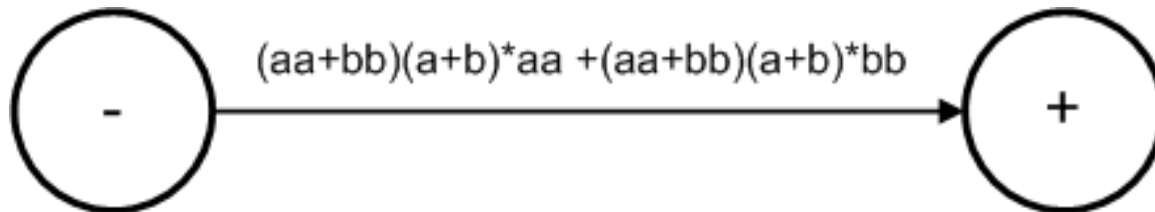
# Example 1

Hence, this TG defines the same language as the regular expression

$(aa + bb)(a + b)^*(aa) + (aa + bb)(a + b)^*(bb)$

or equivalently

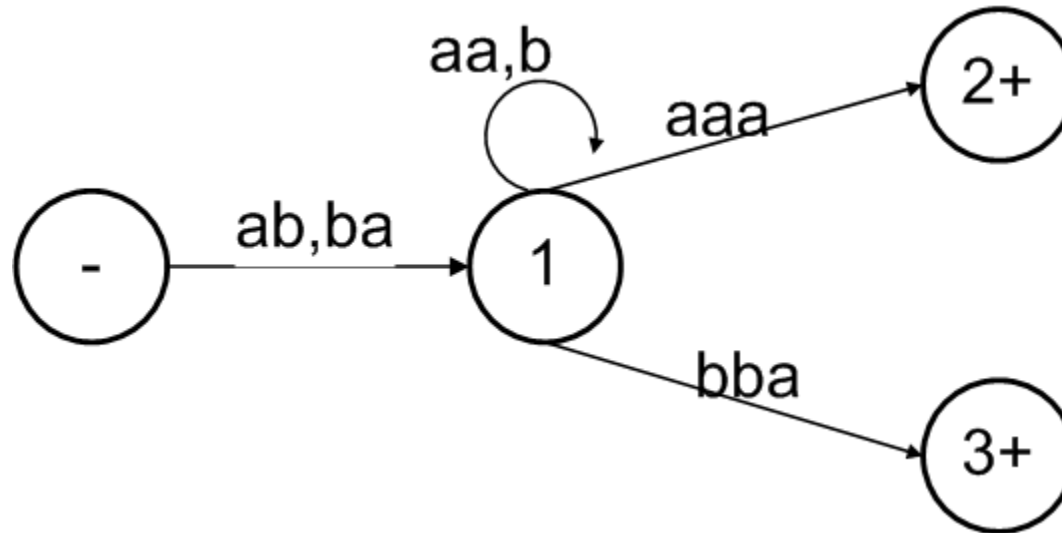
$(aa + bb)(a + b)^*(aa + bb)$





# Example 2

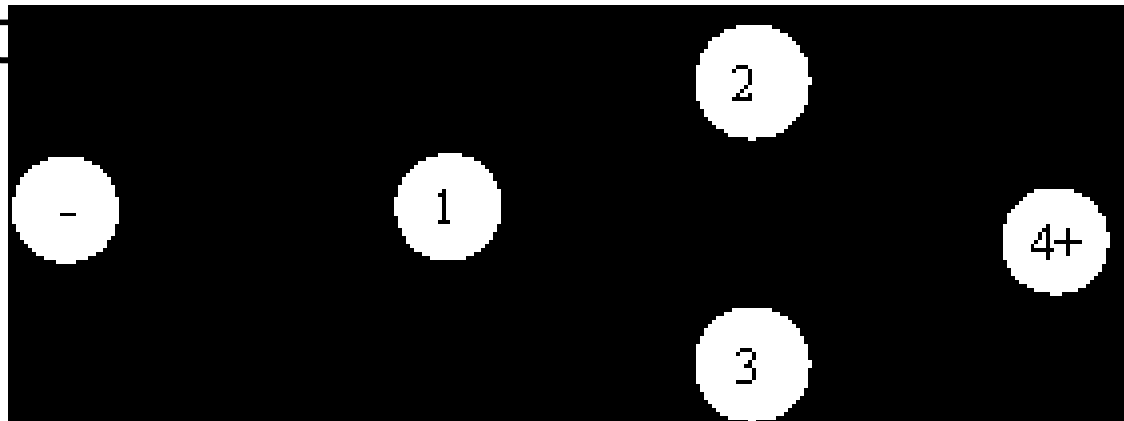
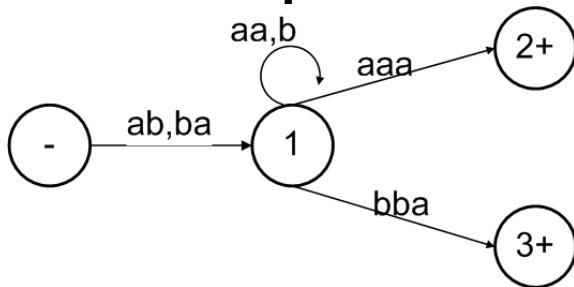
- Consider the following TG



- To have single final state, the above TG can be reduced to the following

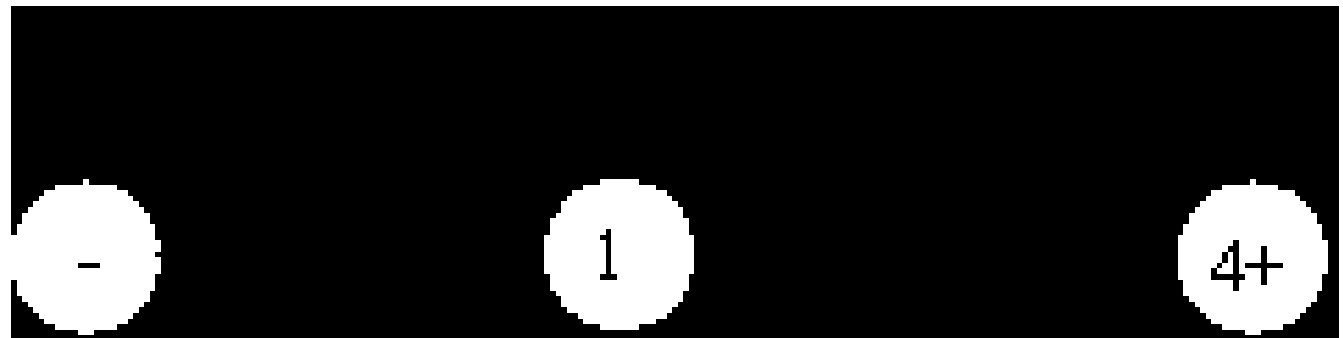
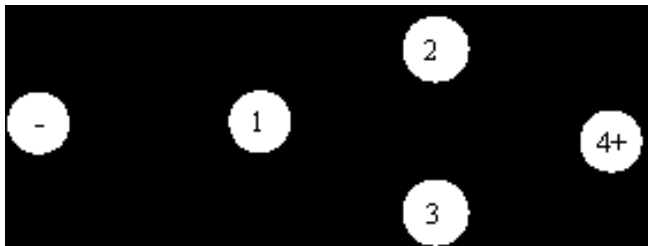
# Example 2

- This TG has only one start state with no incoming edges, but has two final states. So, we must introduce a new unique final state



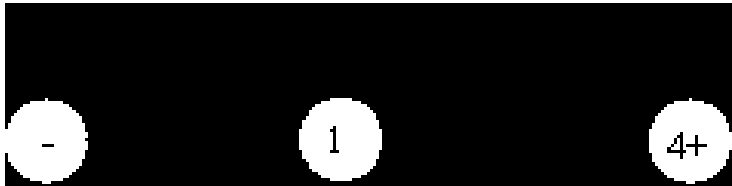
# Example 2

- To eliminate states 2 and 3, the above GTG can be reduced to the following

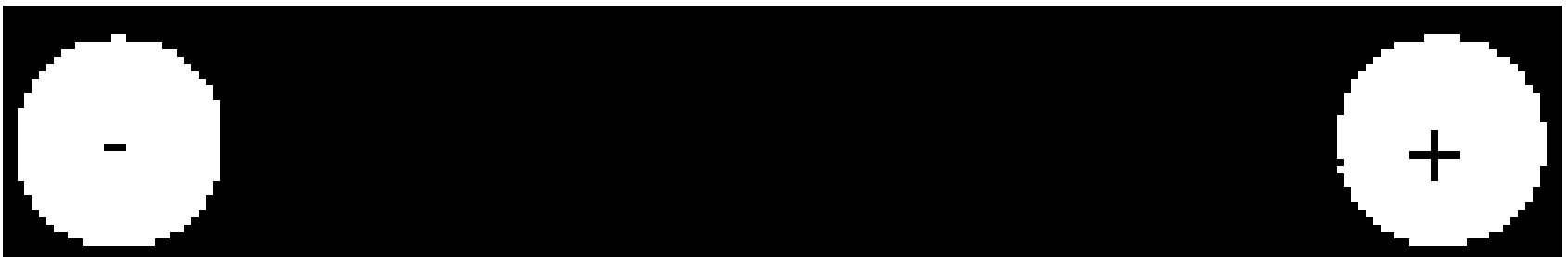


# Example 2

- To eliminate state 1 the above TG can be reduced to the following

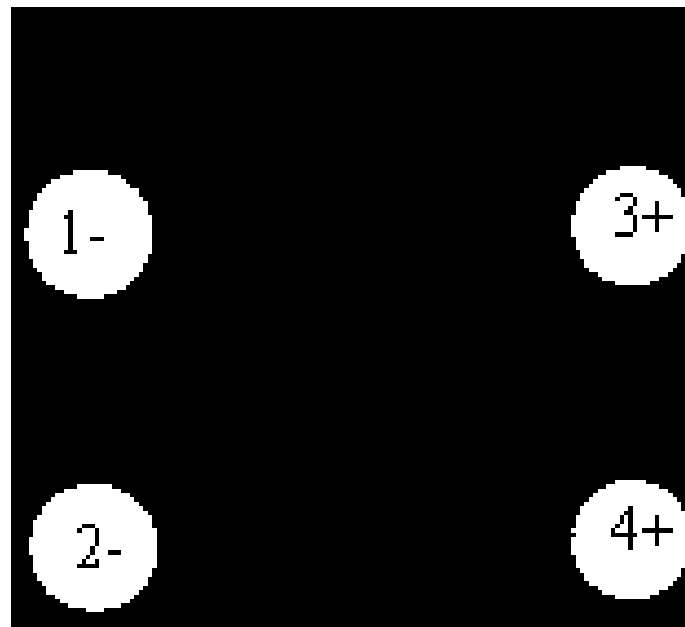


- Hence the required **RE** is  
 $(ab+ba)(aa+b)^*(aaa+bba)$



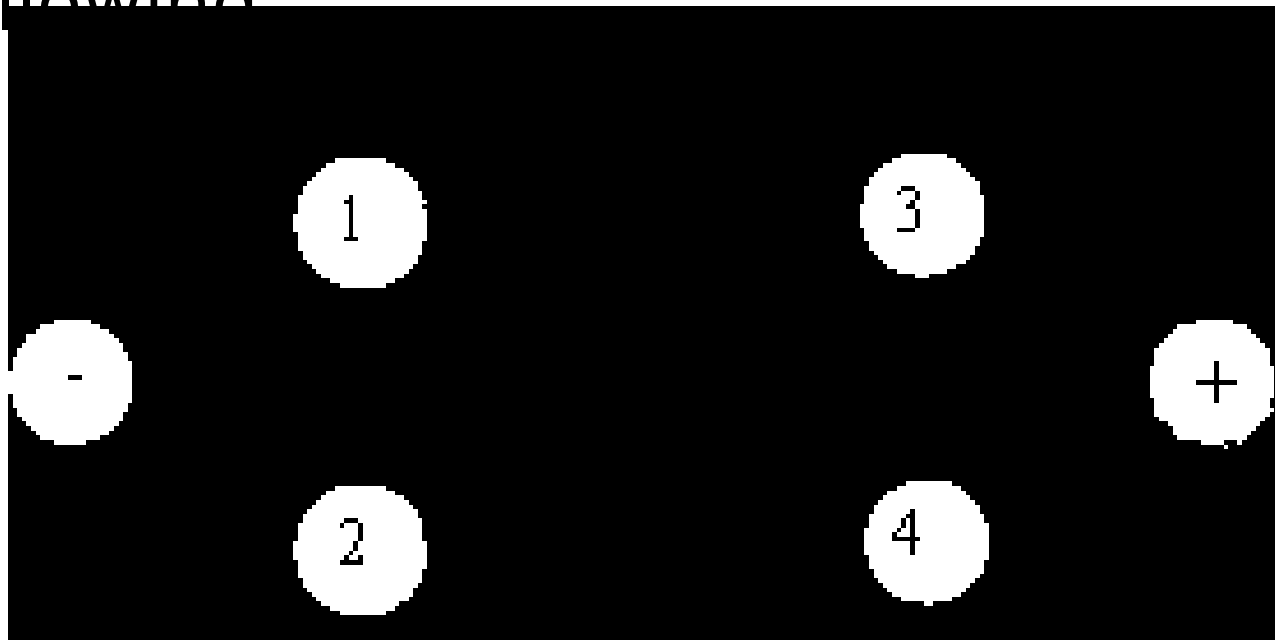
# Example 3

- Consider the following TG



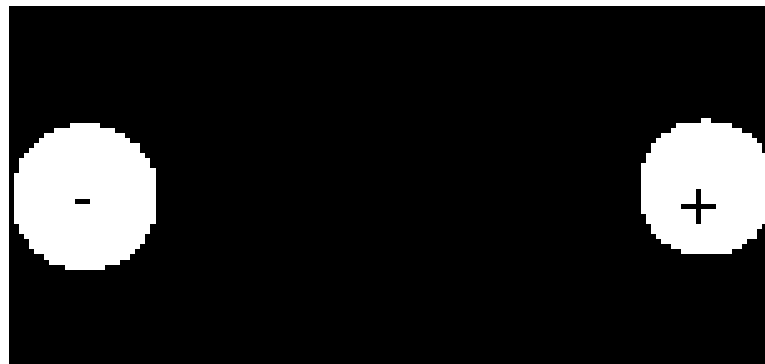
# Example 3

- To have single initial and single final state the above TG can be reduced to the following



# Example 3

- To eliminate states 1,2,3 and 4, the above TG can be reduced to the following TG



# Example 3

- To connect the initial state with the final state by single transition edge, the above TG can be reduced to the following

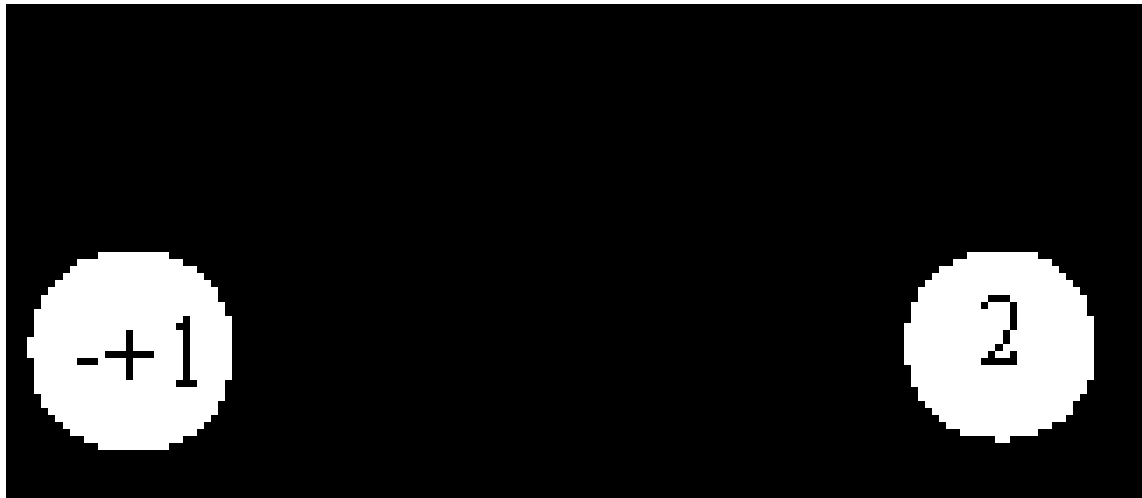


- Hence the required **RE** is  $(b+aa)b^*+(a+bb)a^*$

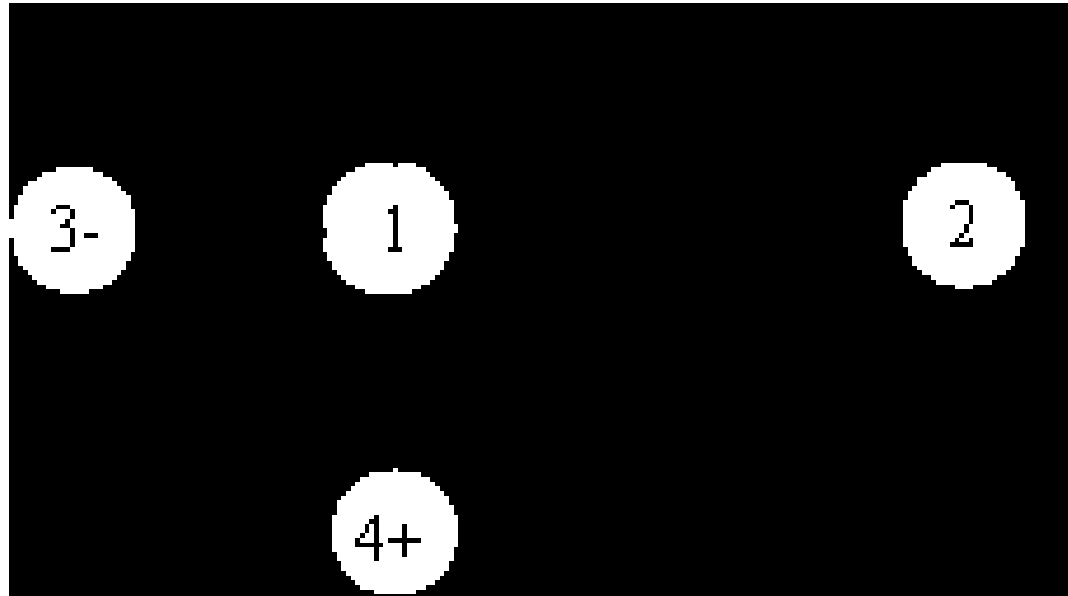


# Example 4

- Consider the following TG, accepting EVEN-EVEN language

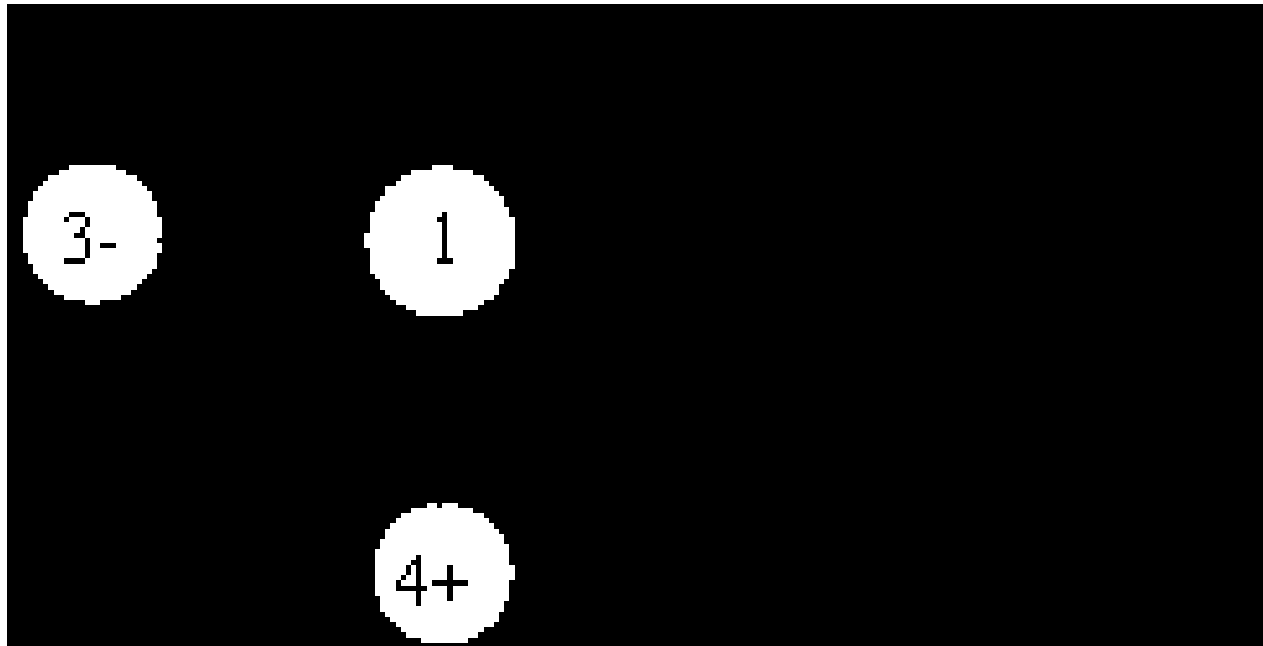


# Example 5



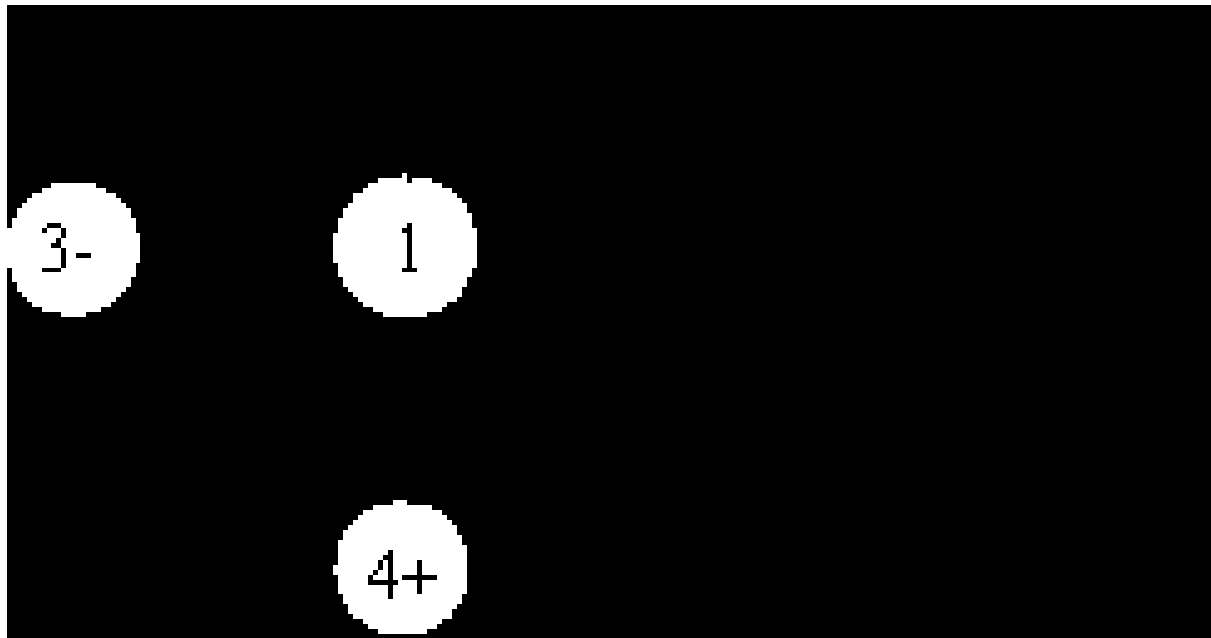
- To eliminate state 2, the above TG may be reduced to the following

# Example 5



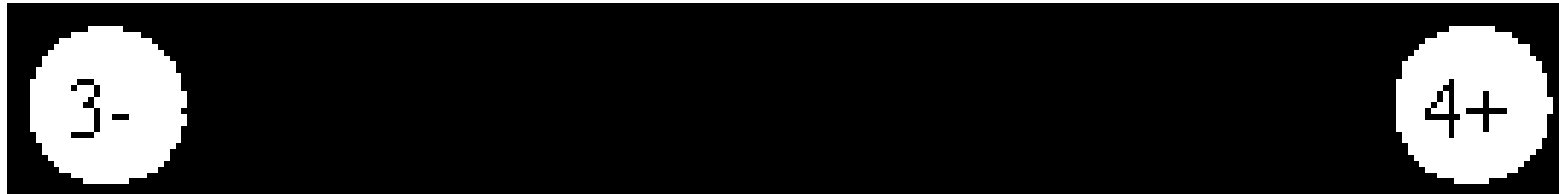
- To have single loop at state 1, the above TG may be reduced to the following

# Example 5



- To eliminate state 1, the above TG may be reduced to the following

# Example 5



- Hence the required **RE** is  
$$(aa + bb + (ab+ba)(aa+bb)^*(ab+ba)^*)^*$$