Theory of Computational

Kleene's Theorem and NFA:

Kleene's Theorem:

Unification
Turning TGs into Regular Expressions
Converting Regular Expressions into FAs
Nondeterministic Finite Automata
NFAs and Kleene's Theorem

Unification:

- We have learned three separate ways to define a language: (i) by **regular expression**, (ii) by **finite automaton**, and (iii) by **transition graph**.
- Now, we will present a theorem proved by Kleene in 1956, which says that **if a language can be defined by any one of these three ways, then it can also be defined by the other two**.
- In other words, Kleene proved that **all three of these methods of defining languages are *equivalent*.**

Theorem 6:(Kleene's Theorem)

- **Any language that can be defined by regular expression, or finite automaton, or transition graph can be defined by all three methods.**

- This theorem is the most important and fundamental result in the theory of finite automata.
- We will take extreme care with its proofs. In particular, we will introduce four algorithms that enable us to construct the corresponding machines and expressions.
- Recall that
  – To prove A = B, we need to prove (i) A B, and (ii) B A.
  – To prove A = B = C, we need to prove (i) A B, (ii) B C, and (iii) C A.
- Thus, 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 of Part 1:

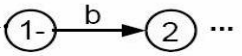
- This is the easiest part.
- From previous lecture, we know that every finite automaton is itself already a transition graph. Therefore, any language that has been defined by a finite automaton has already been defined by a transition graph.

Proof of Part 2: Turning TGs into Regular Expressions

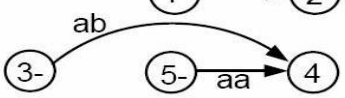
- We prove this part by providing a **constructive algorithm**:
  - We present a algorithm that starts out with a transition graph and ends up with a regular expression that defines the same language.
  - To be acceptable as a method of proof, the algorithm we present will satisfy two criteria: (i) It works for every conceivable TG, and (ii) it guarantees to finish its job in a finite number of steps.

Below present the proof of part 2.

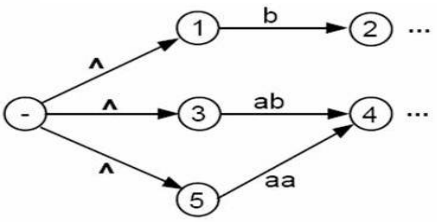
- **Creating A Unique Start State**
- Consider an abstract transition graph T that may have many start states.
- We can simplify T so that it has **only one unique start state that has no incoming edges**.
- We do this by introducing a new start state that we label with the minus sign, and that we connect to all the previous start states by edges labeled with Λ. We then drop the minus signs from the previous start states.
- If a word w used to be accepted by starting at one of the previous start states, then it can now be accepted by starting at the new unique start state.
- The following figure illustrates this idea



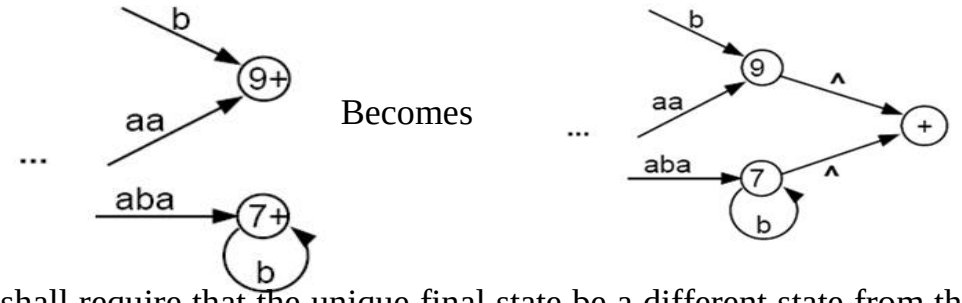
- Consider a fragment of T:



- The above fragment of T can be replaced by



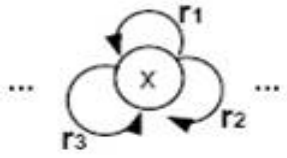
- Let us make another simplification in T so that it has a **unique, unexitable final state**, without changing the language it accepts.
- If T had no final state, then it accepts no strings at all and has no language. So, we need to produce no regular expression other than the null, or empty, expression Φ.
- If T has several final states, we can introduce a new unique final state labeled with a plus sign. We then draw new edges from all the former final states to the new one, dropping the old plus signs, and labeling each new edge with the null string.
- This process is depicted in the next.



- We shall require that the unique final state be a different state from the unique start state. If an old state used to have ±, 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 now has the following shape: where there are no other - or + states.

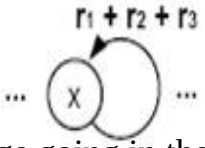
Combining Edges:

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

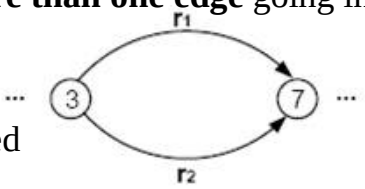


where r_1, r_2, and r_3 are all regular expressions or simple strings.

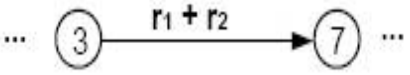
- We can replace the three loops by one loop labeled with a regular expression:



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



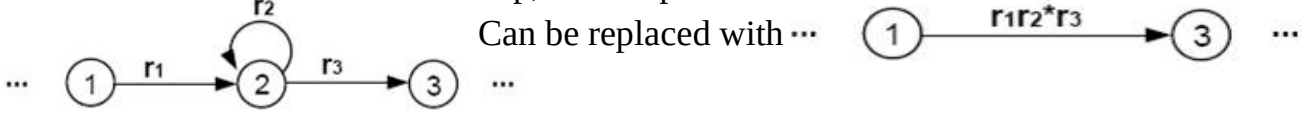
- We can replace this with a single edge labeled with a regular expression:



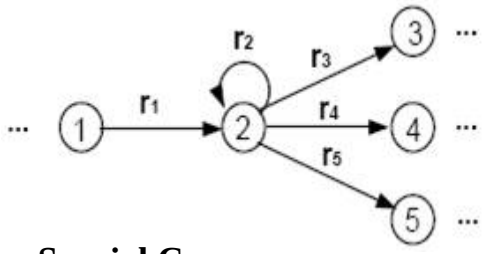
Bypass and State Elimination:

- 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 examples:

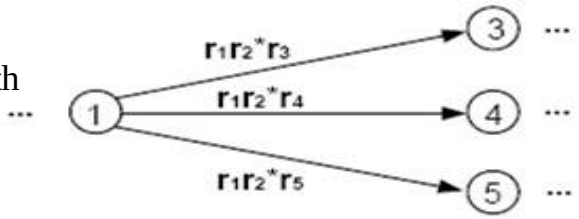
- If the middle state has a loop, we can proceed as follows:



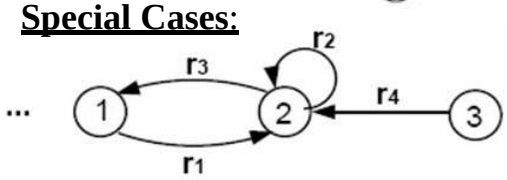
- If the middle state is connected to more than one state, then the bypass and elimination process can be done as follows:



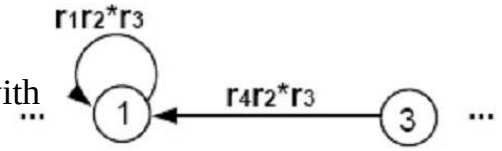
Can be replaced with



Special Cases:

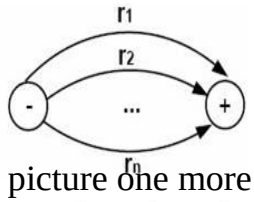


Can be replaced with



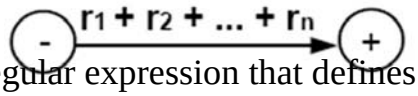
Combining Edges:

- 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. What we come down to is a picture that looks like this:



with each edge labeled by a regular expression.

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

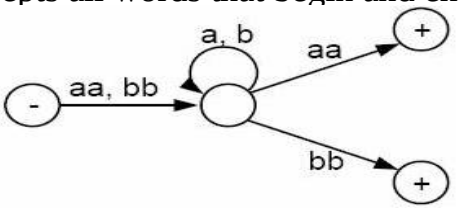


in which the resultant expression is the regular expression that defines the same language as *T* did originally.

- Recall that all words accepted by *T* are paths through the picture of *T*. If we change the picture but preserve all paths and their labels, we must keep the language unchanged.

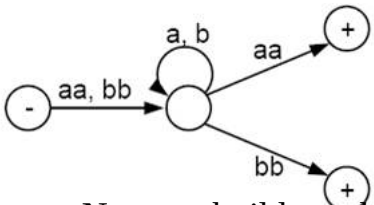
Example:

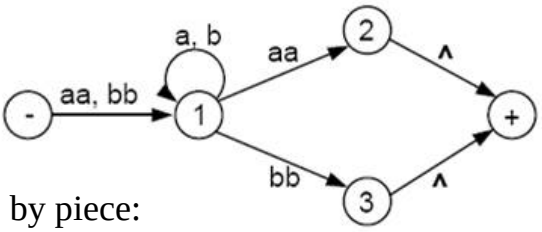
- Consider the following TG that accepts all words that begin and end with double



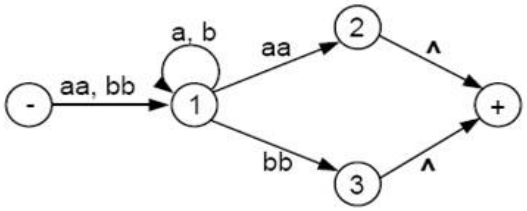
letters (having at least length 4):

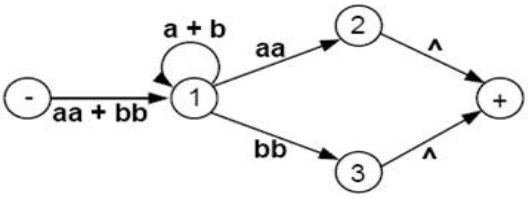
- This TG has only one start state with no incoming edges, but has two final states. So, we must introduce a new unique final state:



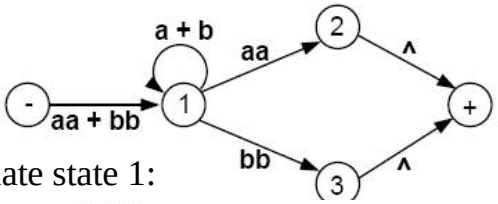


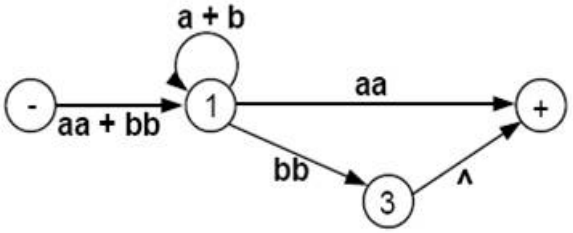
- Now we build regular expressions piece by piece:



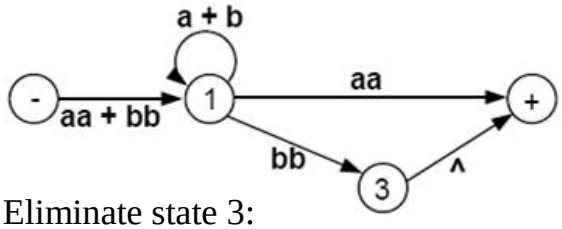


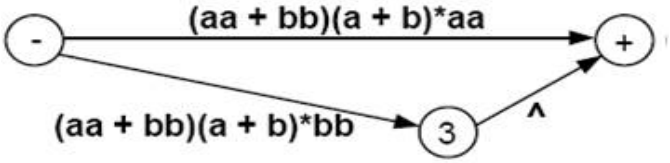
- Eliminate state 2:



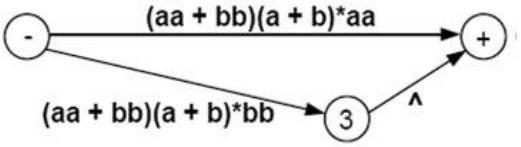


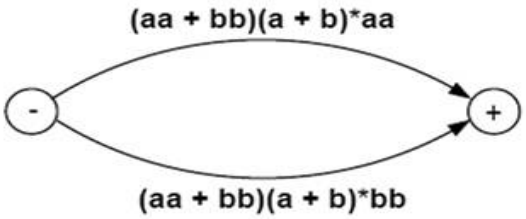
Eliminate state 1:





Eliminate state 3:





- 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)
- If we eliminated the states in a different order, we could end up with a different-looking regular expression. But by the logic of the elimination process, these expressions would all have to represent the same language.
- We are now ready to present the constructive algorithm that proves that all TGs can be turned into regular expressions that define the exact same language.