RegExp = (DFA,NFA,NFAe)

Sipser pages 66-76

What do we know?

DFA = NFA = ε-NFA

We have seen algorithms to transform

- DFA to NFA (trival)
- NFA to ε–NFA (trivial)
- NFA to DFA (subset construction)
- Ε-NFA to NFA (ε-removal)

The last piece of the puzzle is to show that regular expressions are equivalent to automata.

Outline

We have two results to prove:

Theorem 1. For every regular expression E, there exists an ε-NFA A such that L(E)=L(A).

Theorem 2. For every DFA A, there exists a regular expression E such that L(A)=L(E).

Process

Since we've already seen the "equivalence" of various forms of finite automata, the picture is about to become complete: the same class of languages (REGULAR LANGUAGES, of course) is defined by

1. Any of the 3 types of automata
   (DFA, NFA, ε-NFA)
2. Regular expressions

Proofs of both theorems are constructive; we'll learn algorithms to construct an ε-NFA from a regular expression, and a regular expression from a DFA.

Four Algorithms

1. RegExp -> ε-NFA via delta-extension

   Sipser pages 67-69

2. RegExp -> DFA via GenNFA

3. DFA -> RegExp via state elimnation

   Sipser pages 69-76

4. DFA -> RegExp via GenNFA

Alg 1 - RE to ε-NFA via delta-extension

Sipser pages 67-69

1. Decompose a RegExp into its parts

```
data RegExp a
  = Lambda
  | Empty
  | One a
  | Union (RegExp a) (RegExp a)
  | Cat (RegExp a) (RegExp a)
  | Star (RegExp a)
```

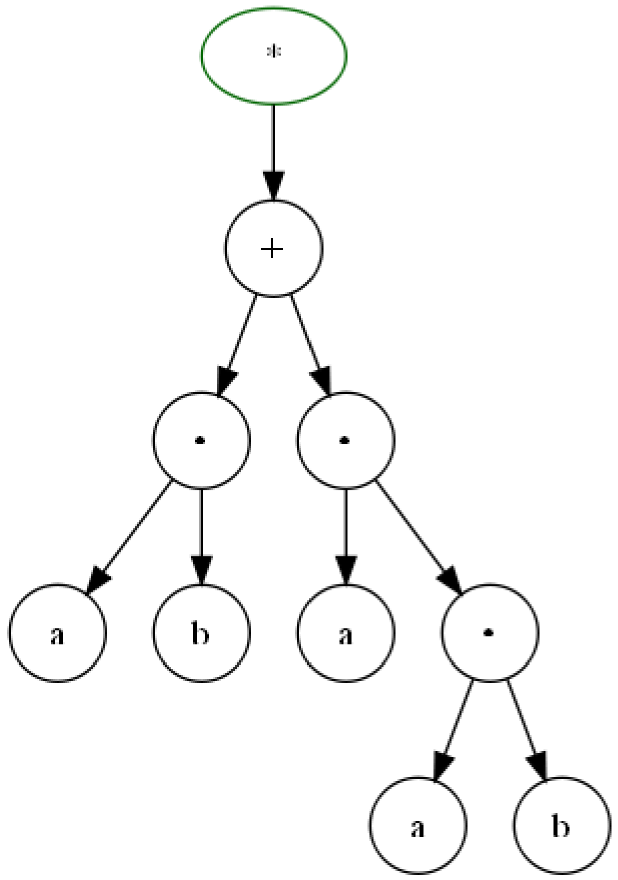
2. Small parts make simple DFAs

3. Combine smaller parts by merging the delta functions of both parts, and by extending the merger (if necessary) with new transitions, and or new states.

RE as Trees

Every RE has a tree like structure. We process the tree bottom-up, strating from the leaves.

"(ab+aab)*"

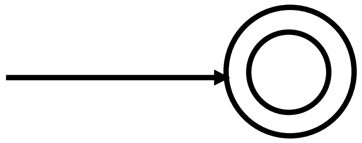


Small (leaf) Cases

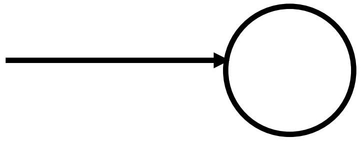
Suppose E is a given regular expression. The construction of the automaton A such that L(RE)=L(A) is defined by induction on the structure of RE.

Base Case: There are three base cases:

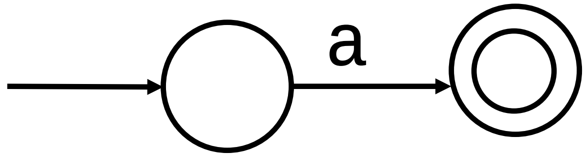
E=ε, E=∅, and E=a, where a∈Σ. Here are the corresponding automata.



E=ε



E=∅



E=a

Induction Step

There are three cases to consider

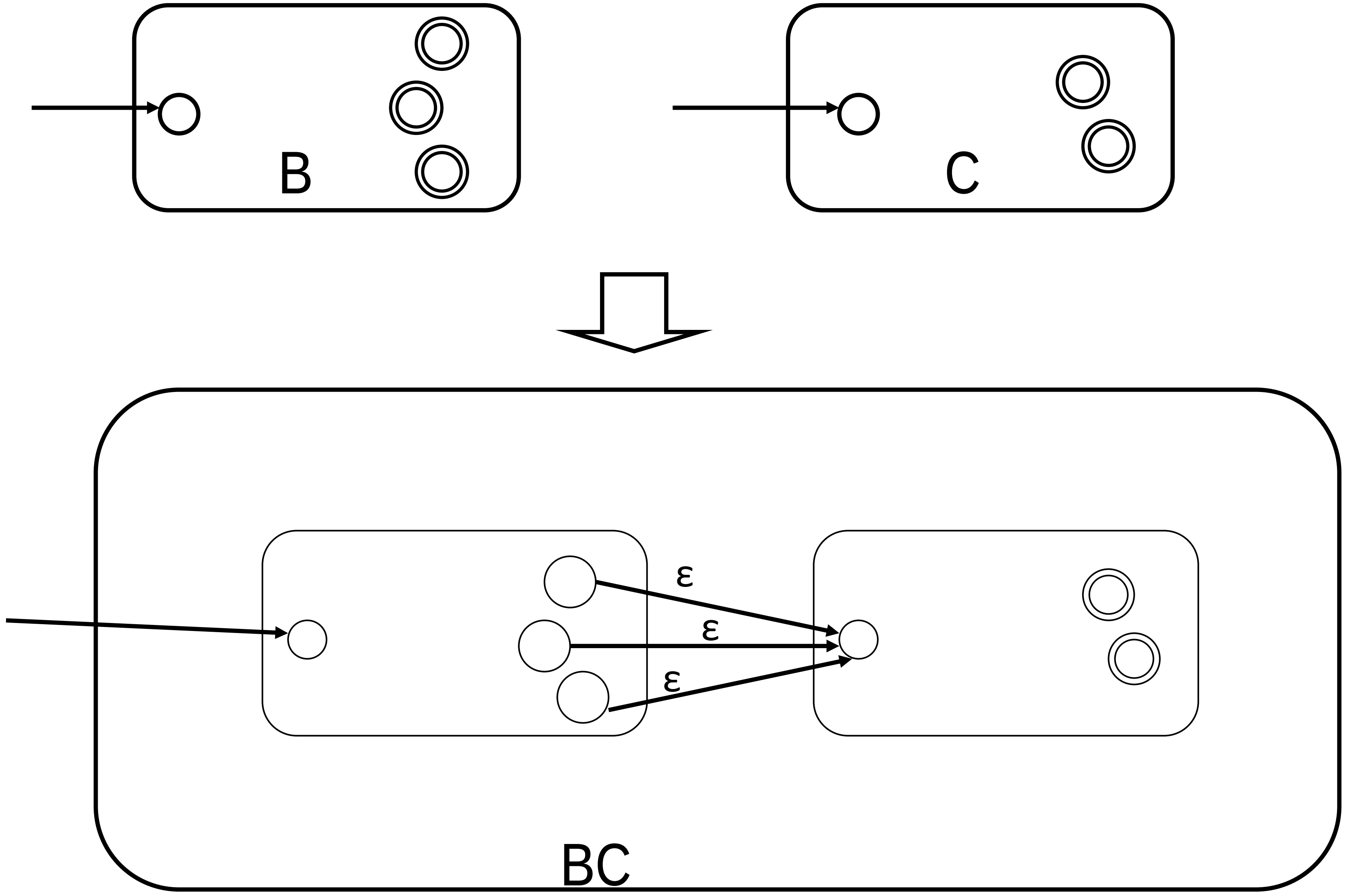
Case 1: E=FG. Suppose (Ind. Hyp.) we have automata B and C such that L(F)=L(B) and L(G)=L(C).

Let A be the automaton obtained by taking states and transitions of B and C together,

Then add Λ-transitions from all final states of B to the start state of C (delta-extension).

Then declare the start state (of the new automata) to be the start state of B.

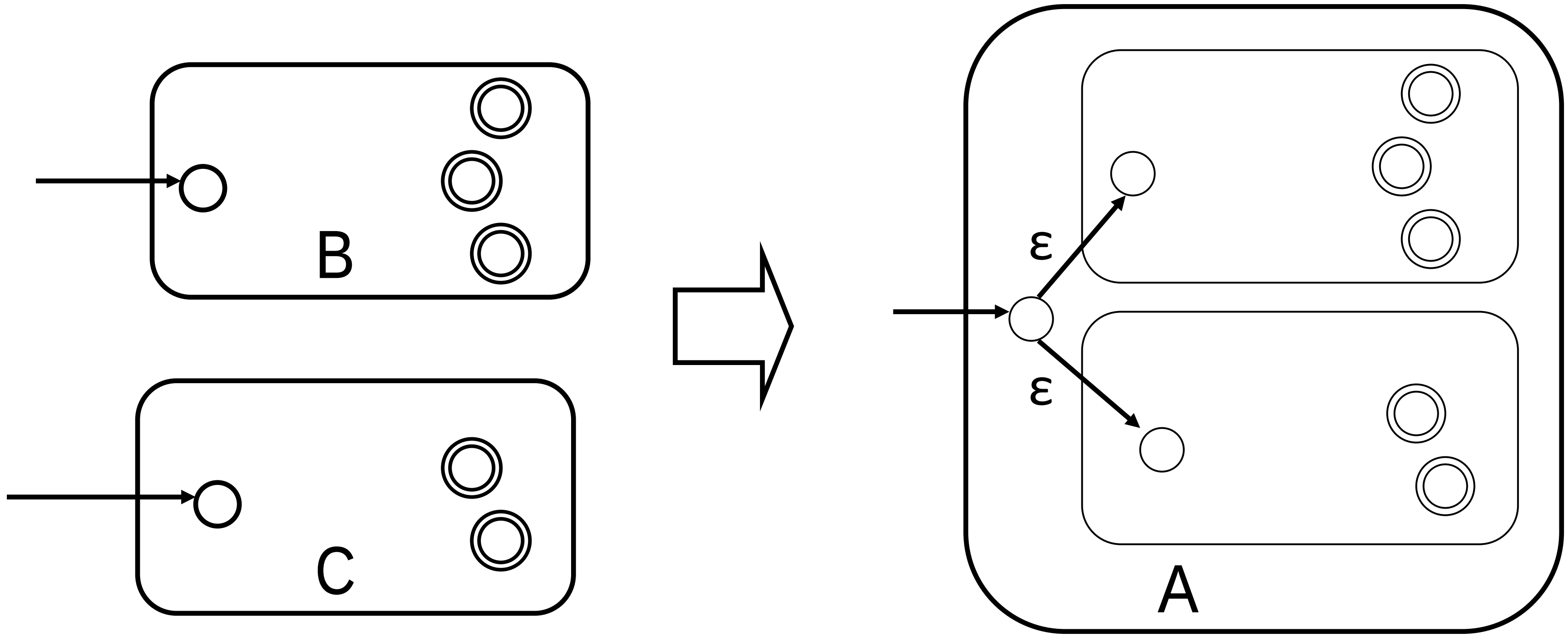
The final states (of the new automata) to be the final states of C.



We need to check that L(A)=L(B)L(C); it will follow then that L(A)=L(E)

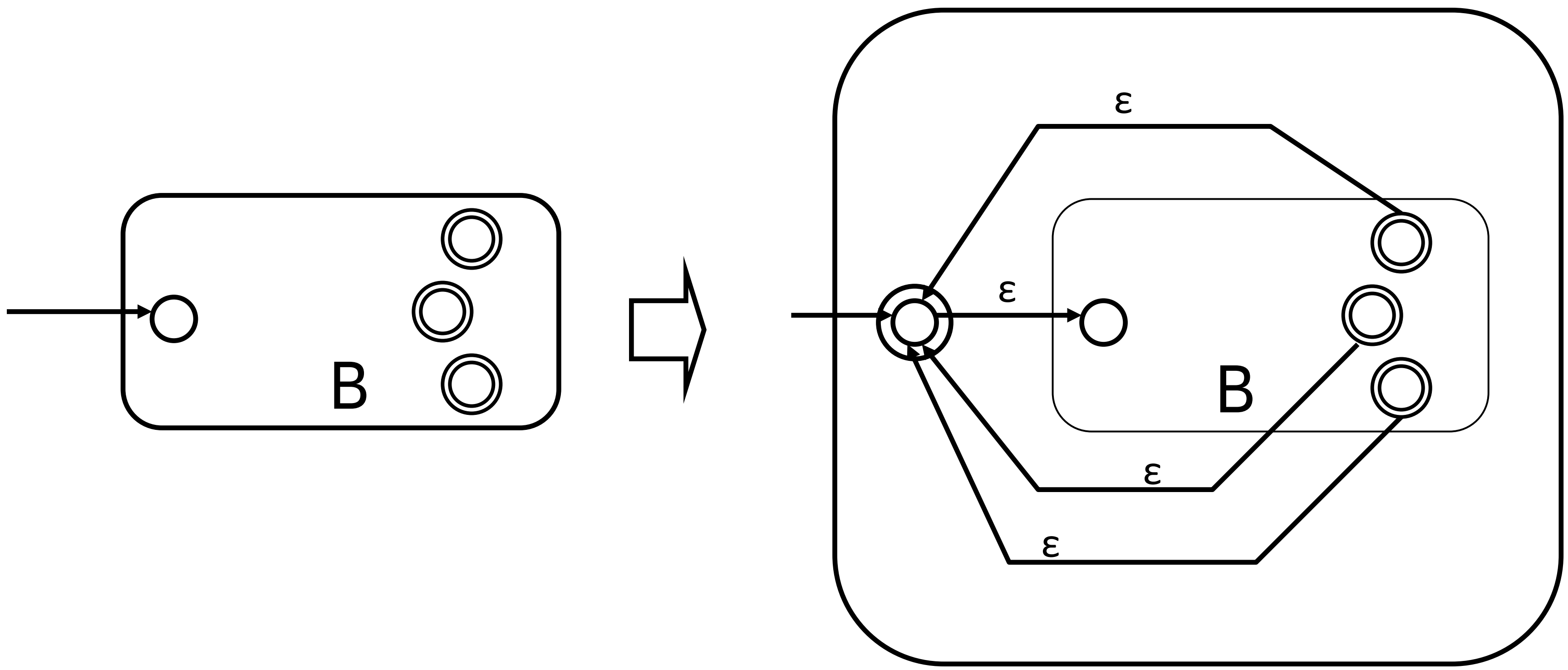
Case 2: E=F+G

Again, take F,G such that L(F)=L(B) and L(G)=L(C). Get A by adding a new start state and ε-transitions from the new state to the start states of B and C. Check that $L(A)=L(B) \cup L(C)$!



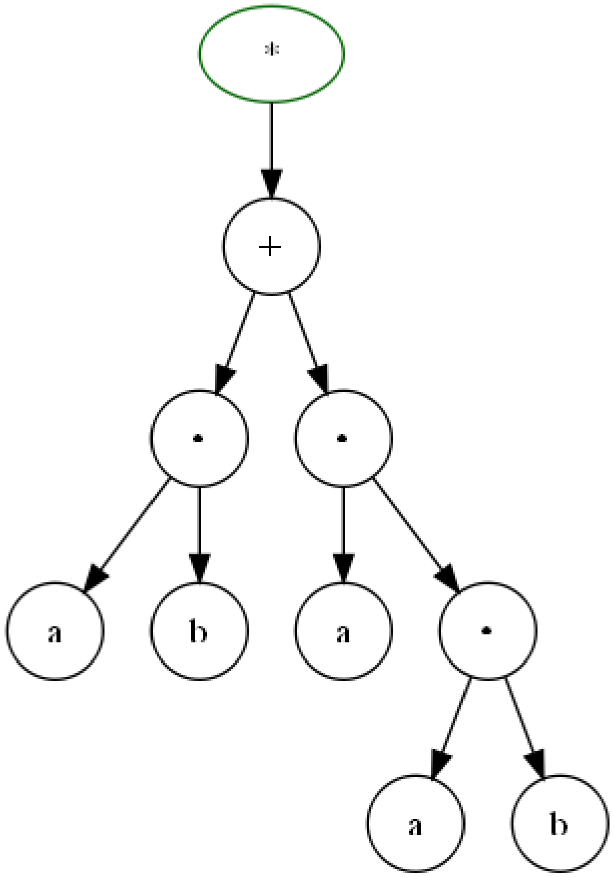
Case 3: E = F*

Take B such that L(B)=L(F)and define A as in the picture. Check that L(A)=L(B)*.



Exercise

Exercise. Construct an ε-NFA accepting the language of the regular expression $(ab+aab)^*$

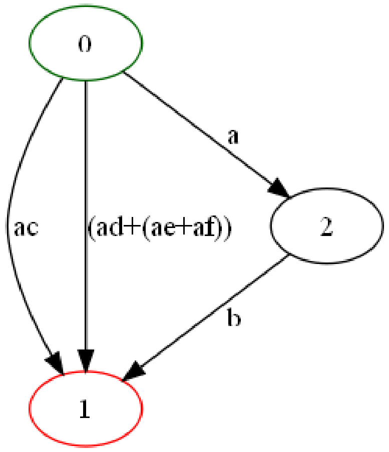


Definition - Generalized NFA

Generalize DFA so that every transition is a regular expression rather than a letter of the alphabet.

An GenNFA is a quintuple $A=(Q,\Sigma,s,F,\delta)$, where the first four components are as in a DFA, and the transition function labels arcs between states with regular expressions.

$$\delta\colon\ Q \times Q \longrightarrow \mathrm{RegExp}(\Sigma)$$



Alg 2 - RegExp to DFA via GenNFA

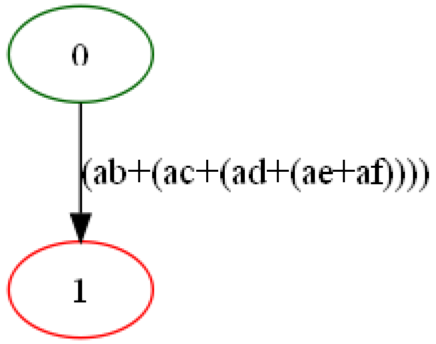
(not covered in Sipser)

Given a RegExp: ab+ac+ad+ae+af

1. Construct a simple GenNFA with two states with one arc between the two states labeled with the regular expression.

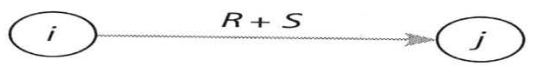
2. The source of the arc is the start state

3. The target of the arc is the final

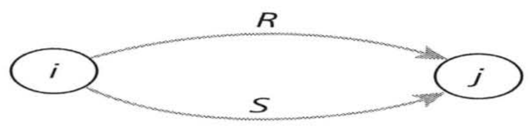


Extend the GenNFA

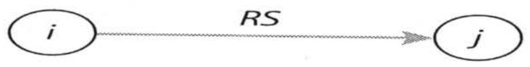
1. If an edge is labeled with ∅, then erase the edge.
2. Transform any diagram like



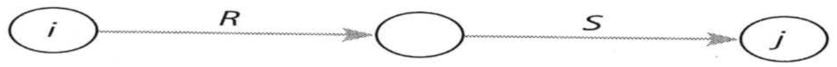
into the diagram



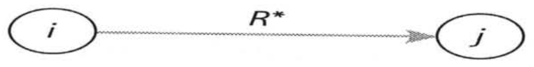
3. Transform any diagram like



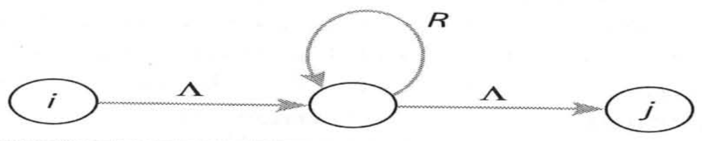
into the diagram



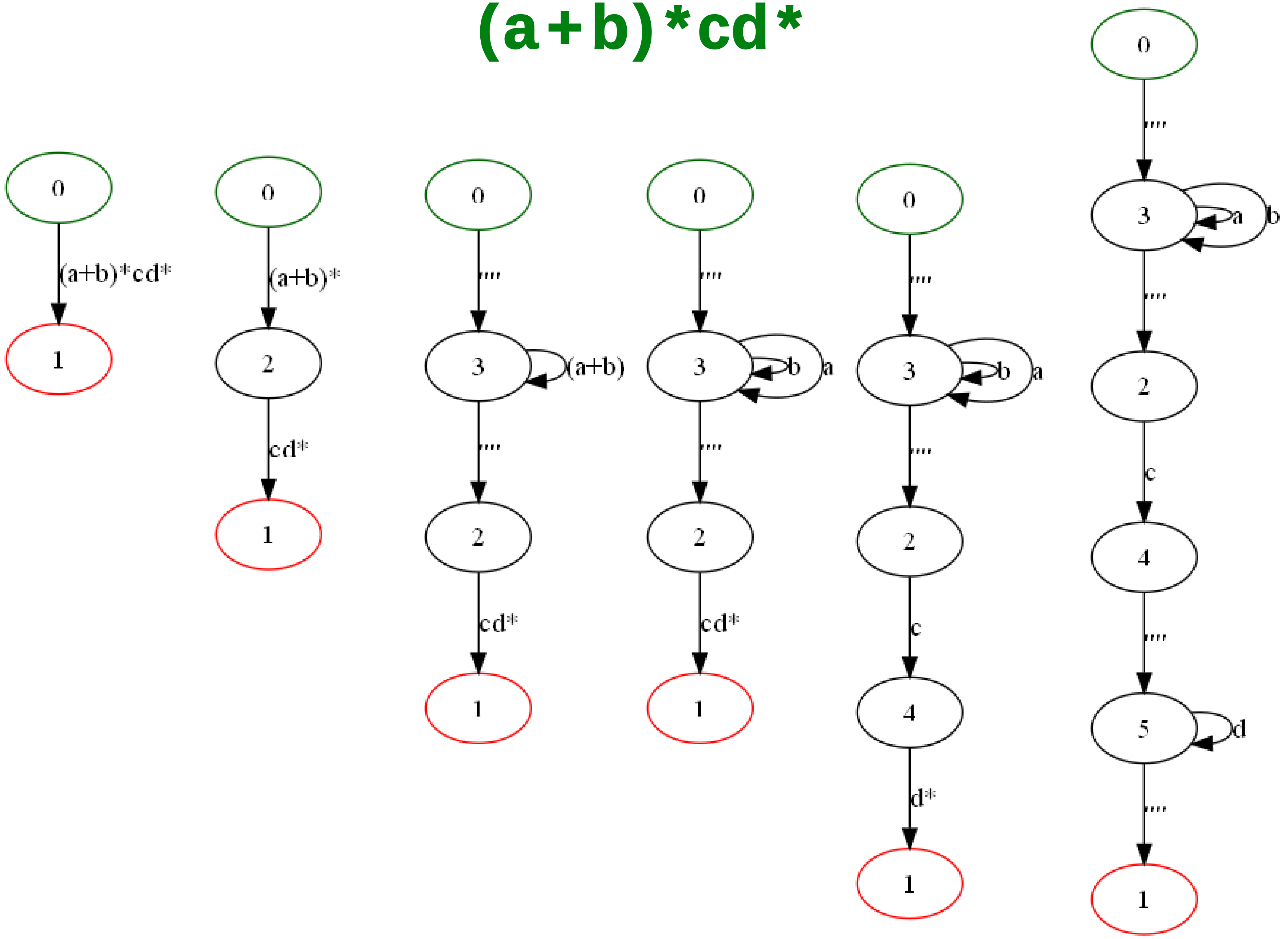
4. Transform any diagram like



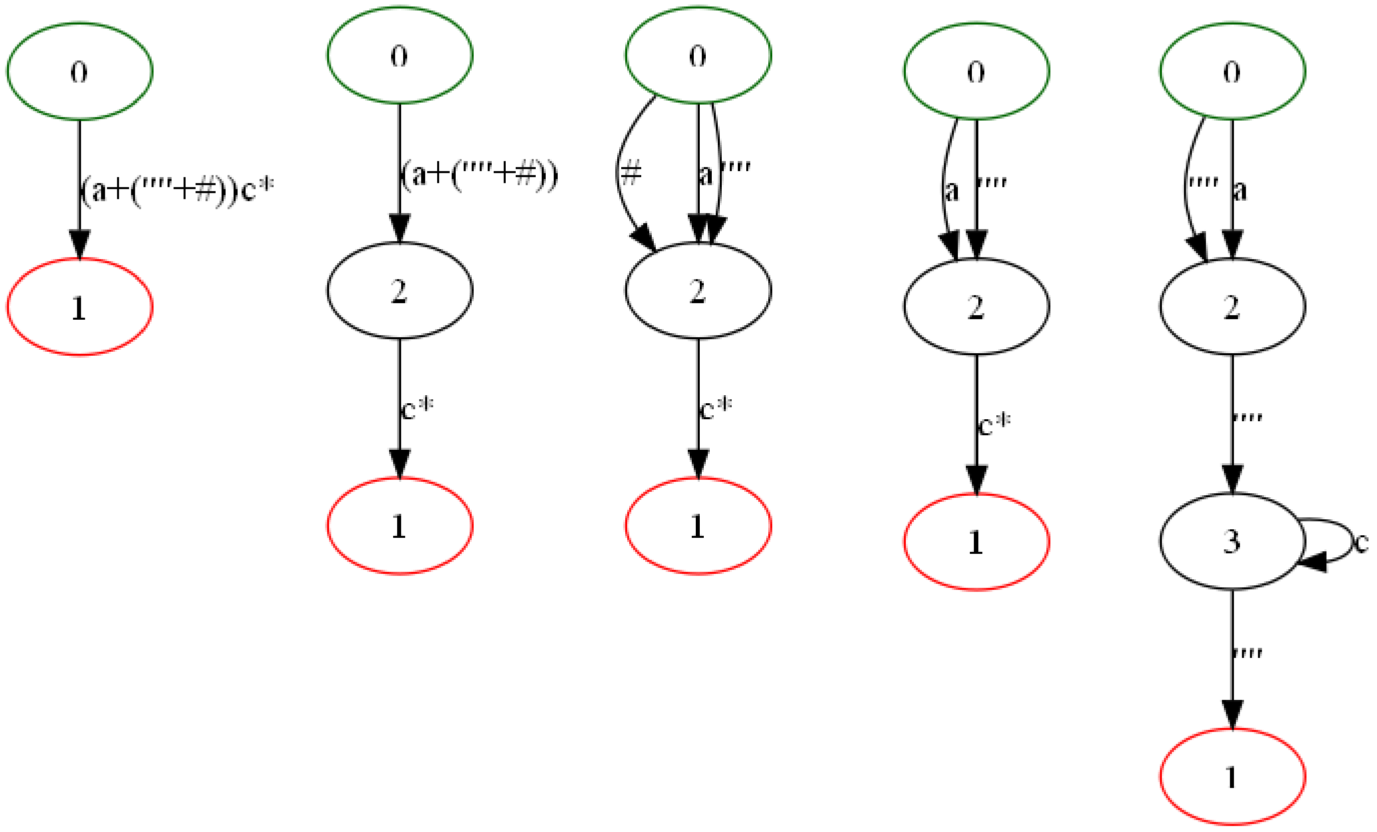
into the diagram



(a+b)*cd*



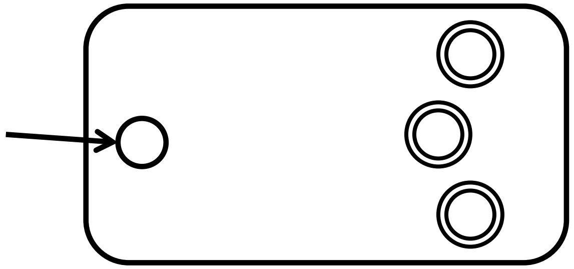
(a+Λ+∅)c*

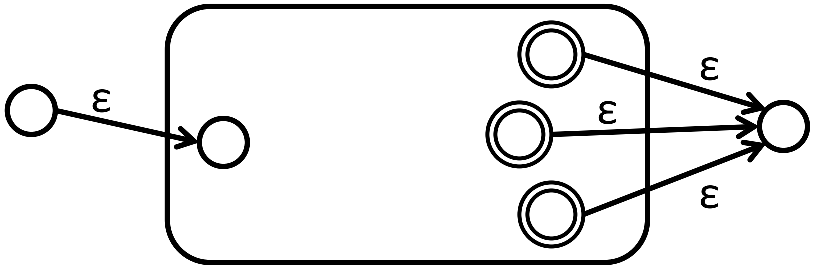


Alg 3 - From DFA to RE by state elimination

(Sipser pages 69-76)

1. The last construction was in fact algorithmic in both directions. We can improve the algorithm as follows. With an input NFA A, it will produce a regular expression E such that L(E)=L(A).
2. First we *standardize* the automaton A so that:
   - there is only one final state
   - no arcs go out of the final state
   - no arcs come into the initial state
3. Do this by using ε-transitions





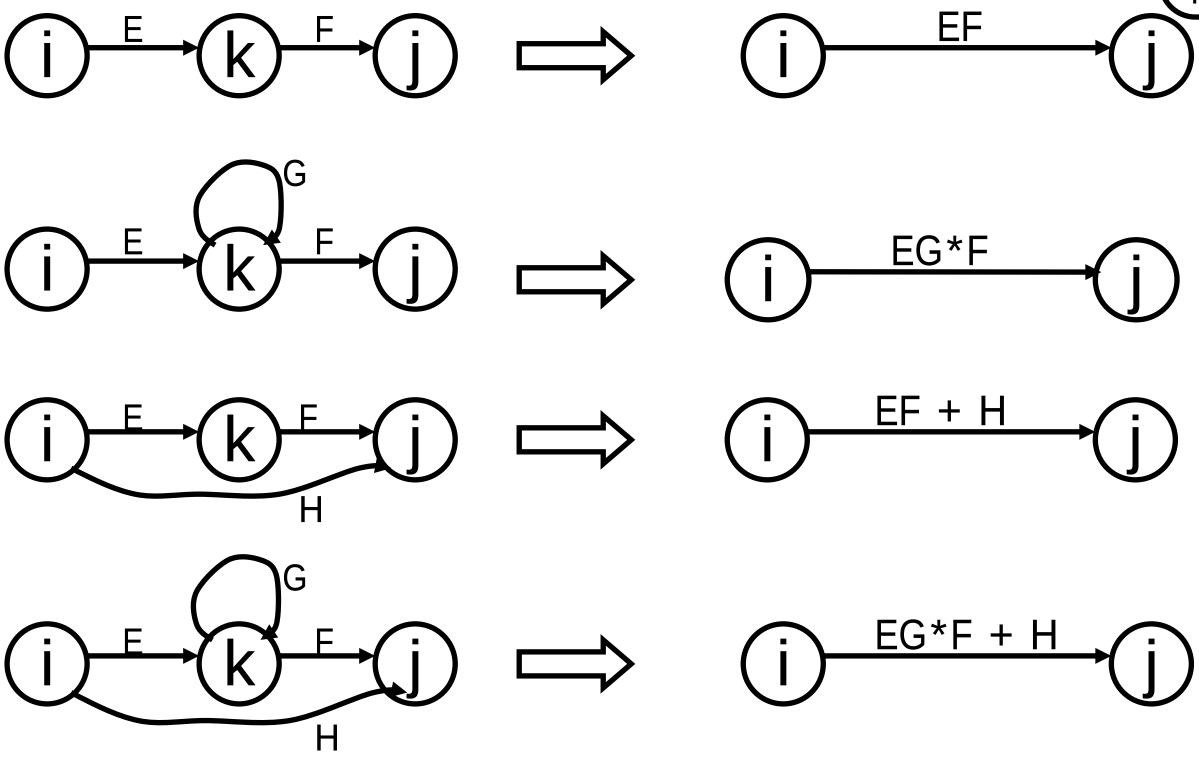
Assume the initial and final states are n-1 and n. We eliminate the states 1,2,...,n-2 one by one, producing intermediate automata whose arcs are *labeled by regular expressions* (i.e GenNFA).

We end up with an automaton that has two states n-1 and n, and only one arc (from n-1 to n) whose label is the required regular expression E.

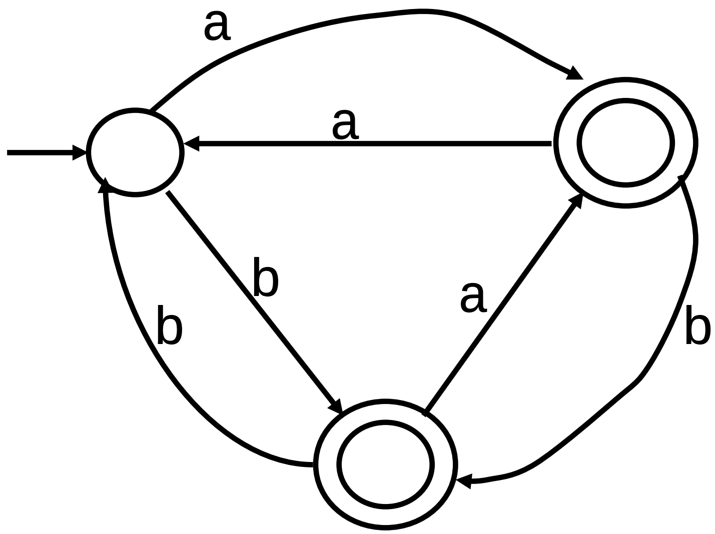
Algorithm

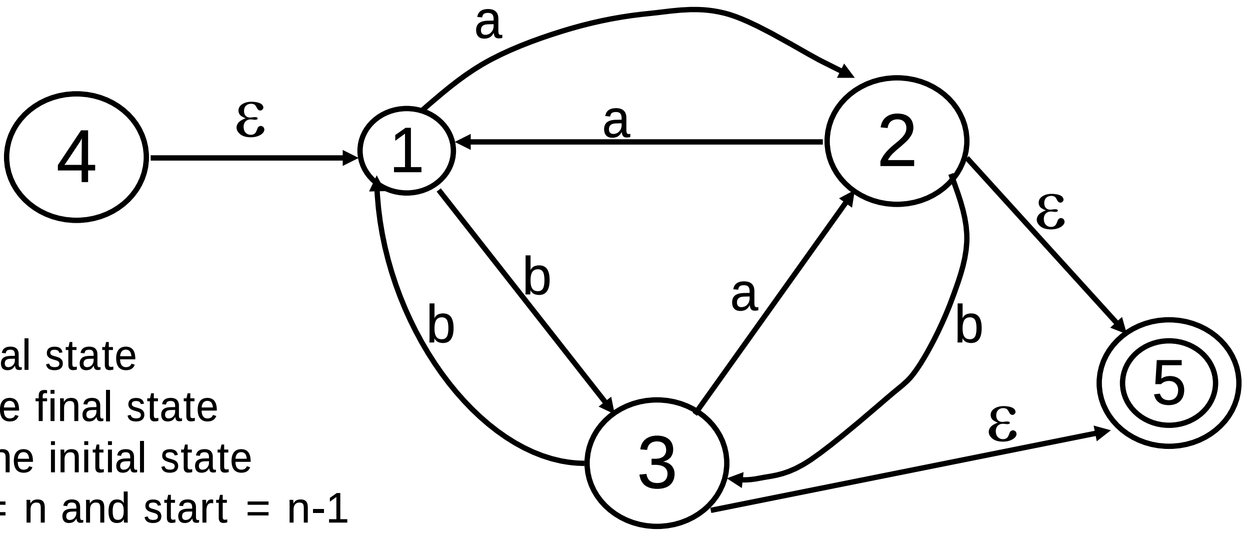
```
For k = 1 to n-2      (* eliminate each node, k in turn *)
  For i = k+1 to n-1      (* i flows into k *)
    For j = k+1 to n    (* k flows into j *)
      case  paths i to k to j  of
```

Each case removes k, and adds a new label from i to j.



Example

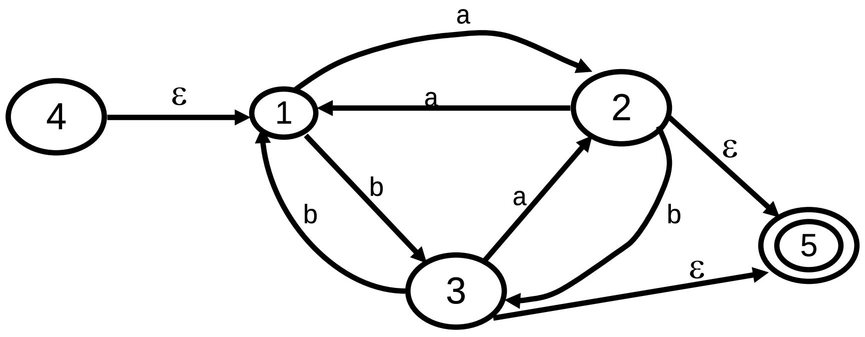


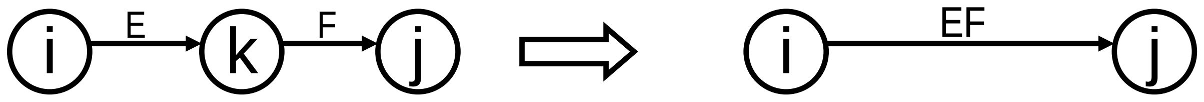


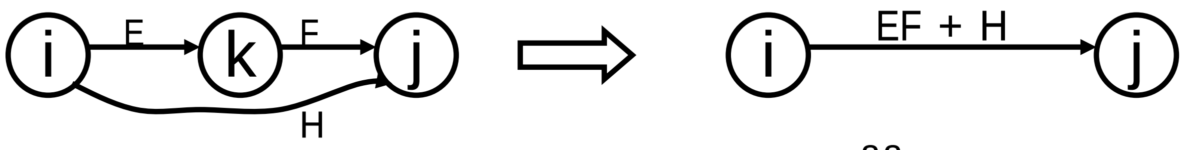
1. there is only one final state
2. no arcs go out of the final state
3. no arcs come into the initial state
4. Renumber so final = n and start = n-1

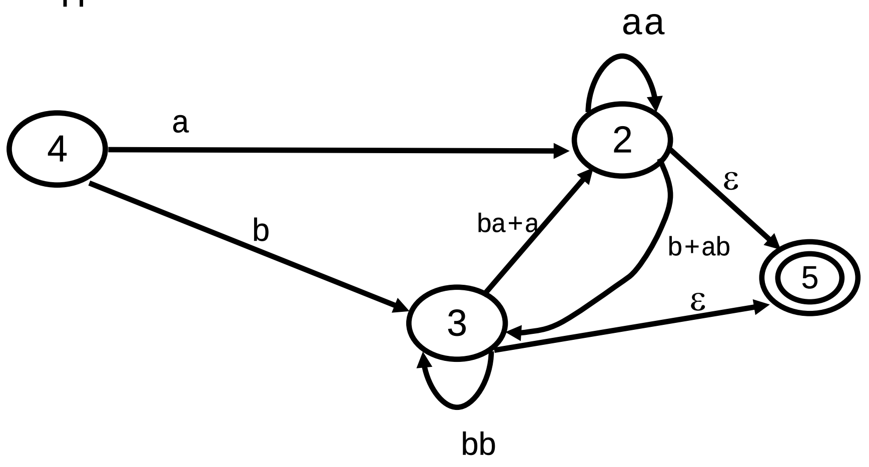
Eliminate k=1

2-1-2	2-(aa)-2
2-1-3	2-(b+ab)-3
2-1-4	∅
2-1-5	∅
3-1-2	3-(ba+a)-2
3-1-3	3-(bb)-3
3-1-4	∅
3-1-5	∅
4-1-2	4-(a)-2
4-1-3	4-(b)-3
4-1-4	∅
4-1-5	∅

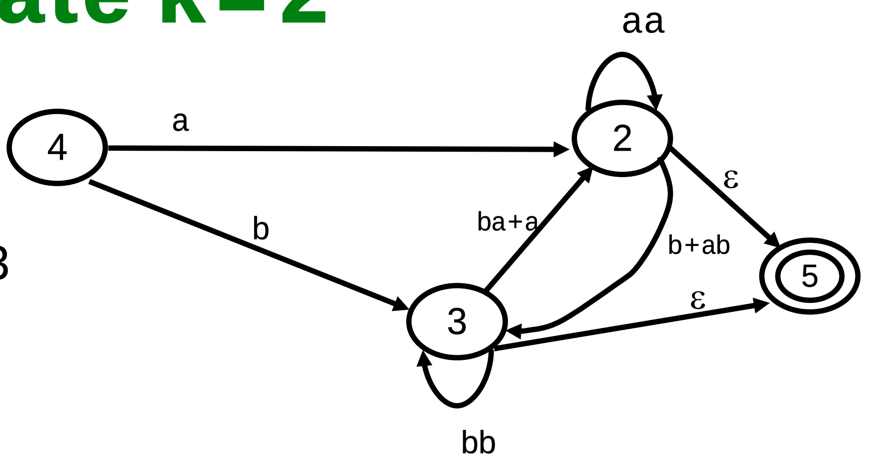








Eliminate k=2



3-2-3 3-((ba+a)(aa)*(b+ab) +bb)-3

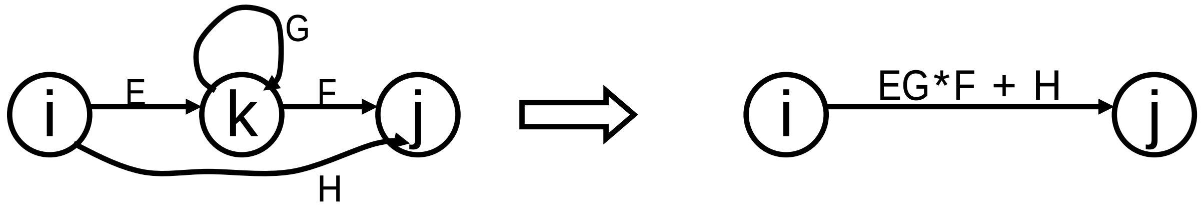
3-2-4 ∅

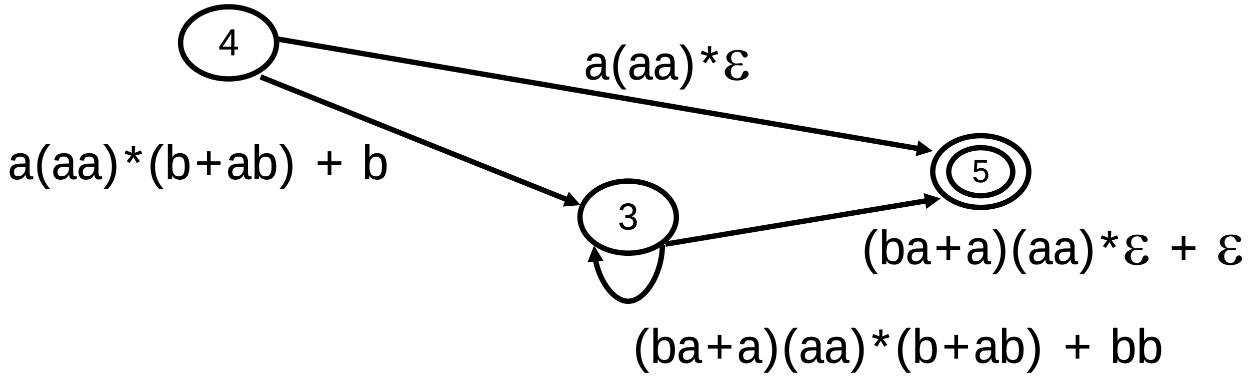
3-2-5 3-((ba+a)(aa)*ε + ε)-5

4-2-3 4-(a(aa)*(b+ab) + b)-3

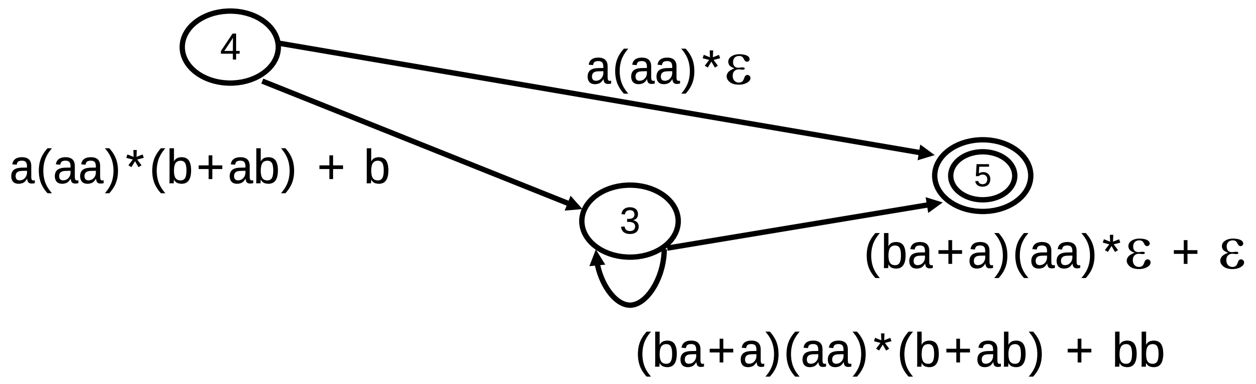
4-2-4 ∅

4-2-5 4-(a(aa)*ε)-5





K=3



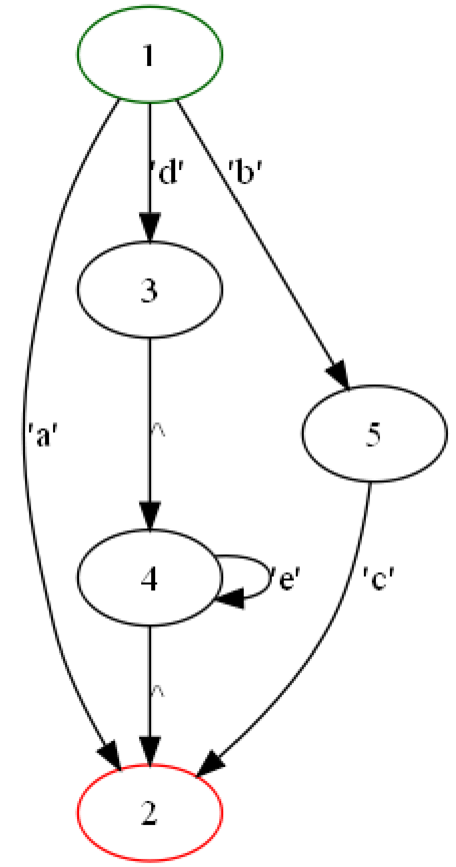
4-3-4 ∅

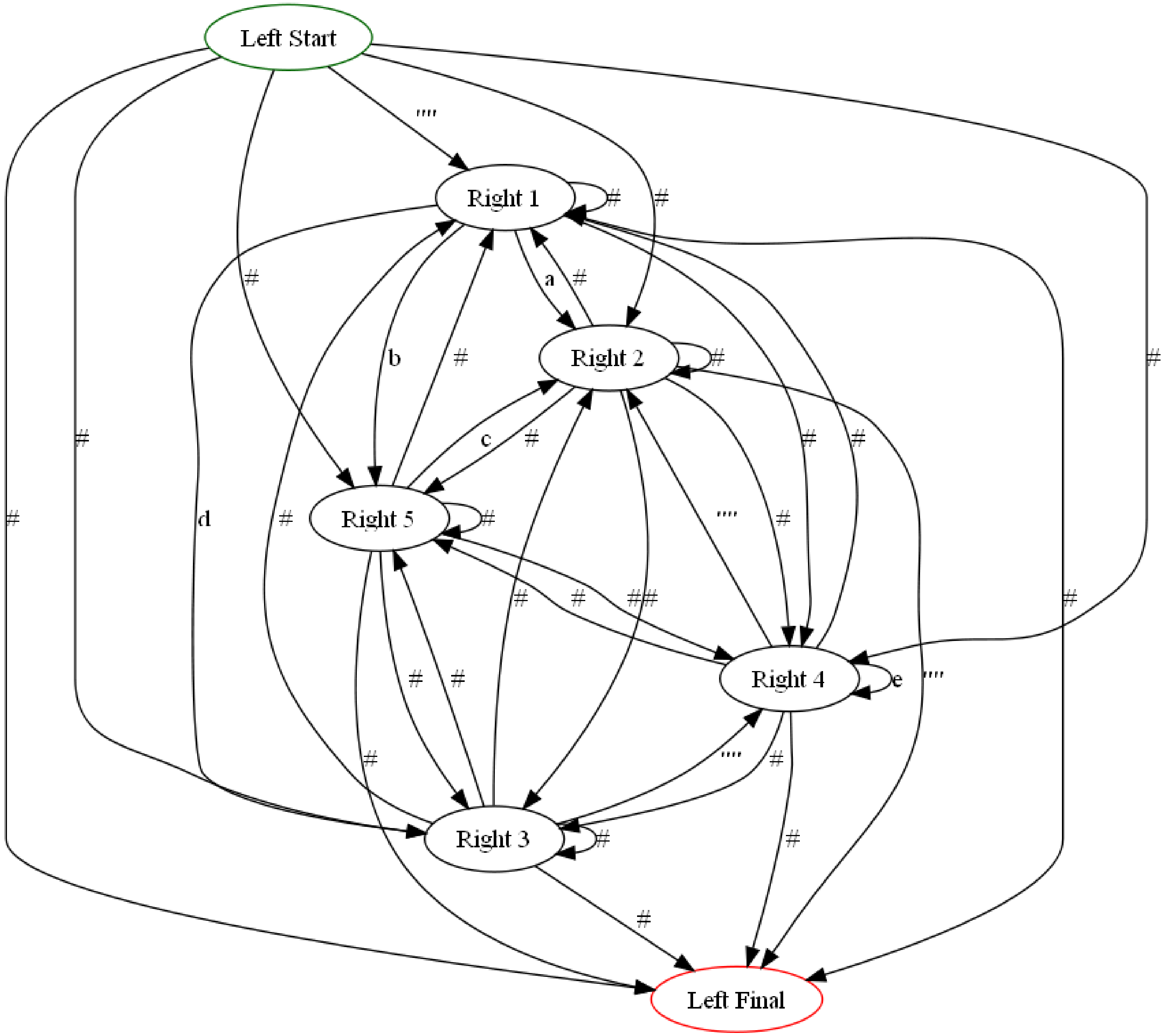
4-3-5 4-((a(aa)*(b+ab) + b)

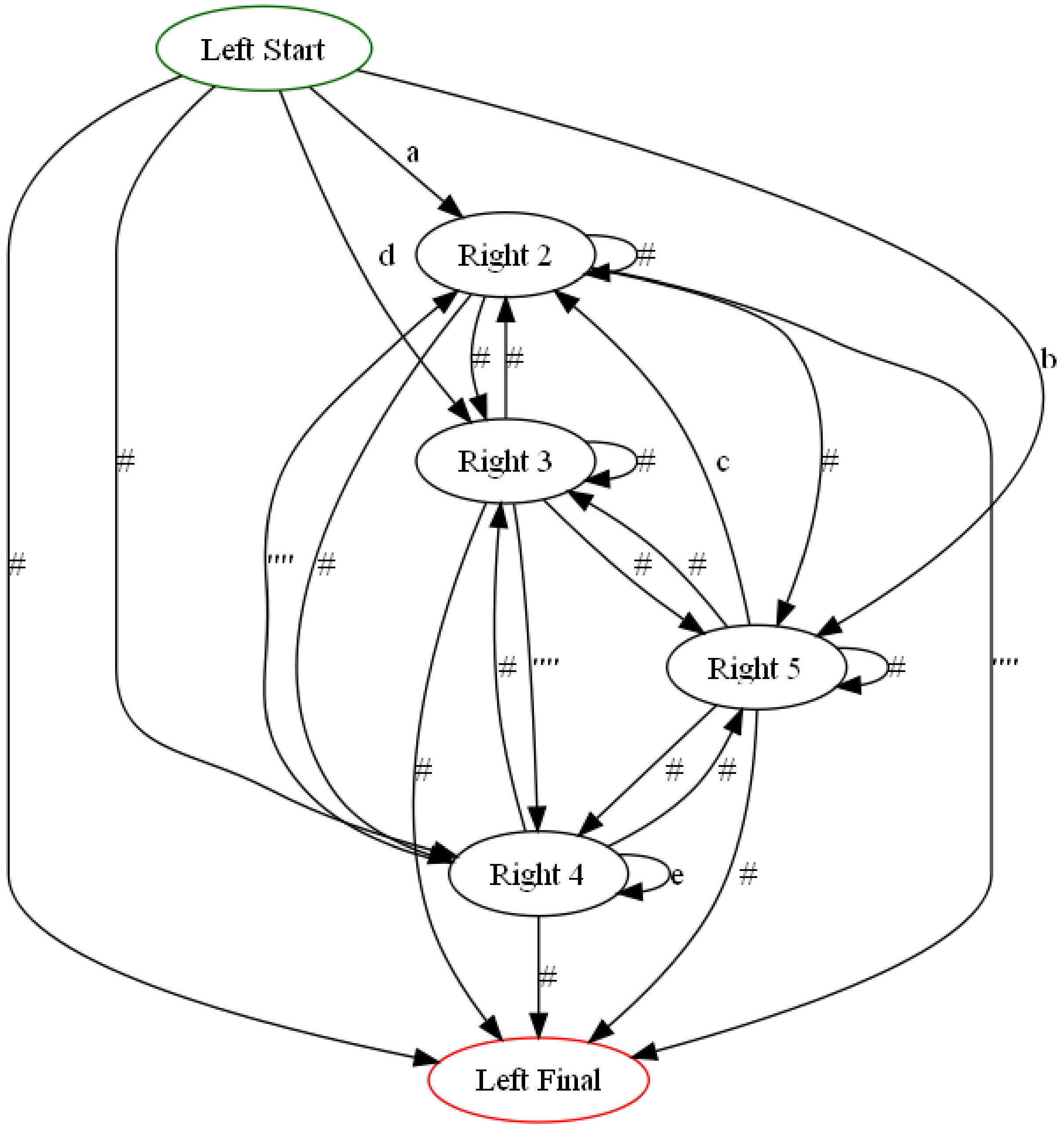
((ba+a)(aa)*(b+ab) + bb)*

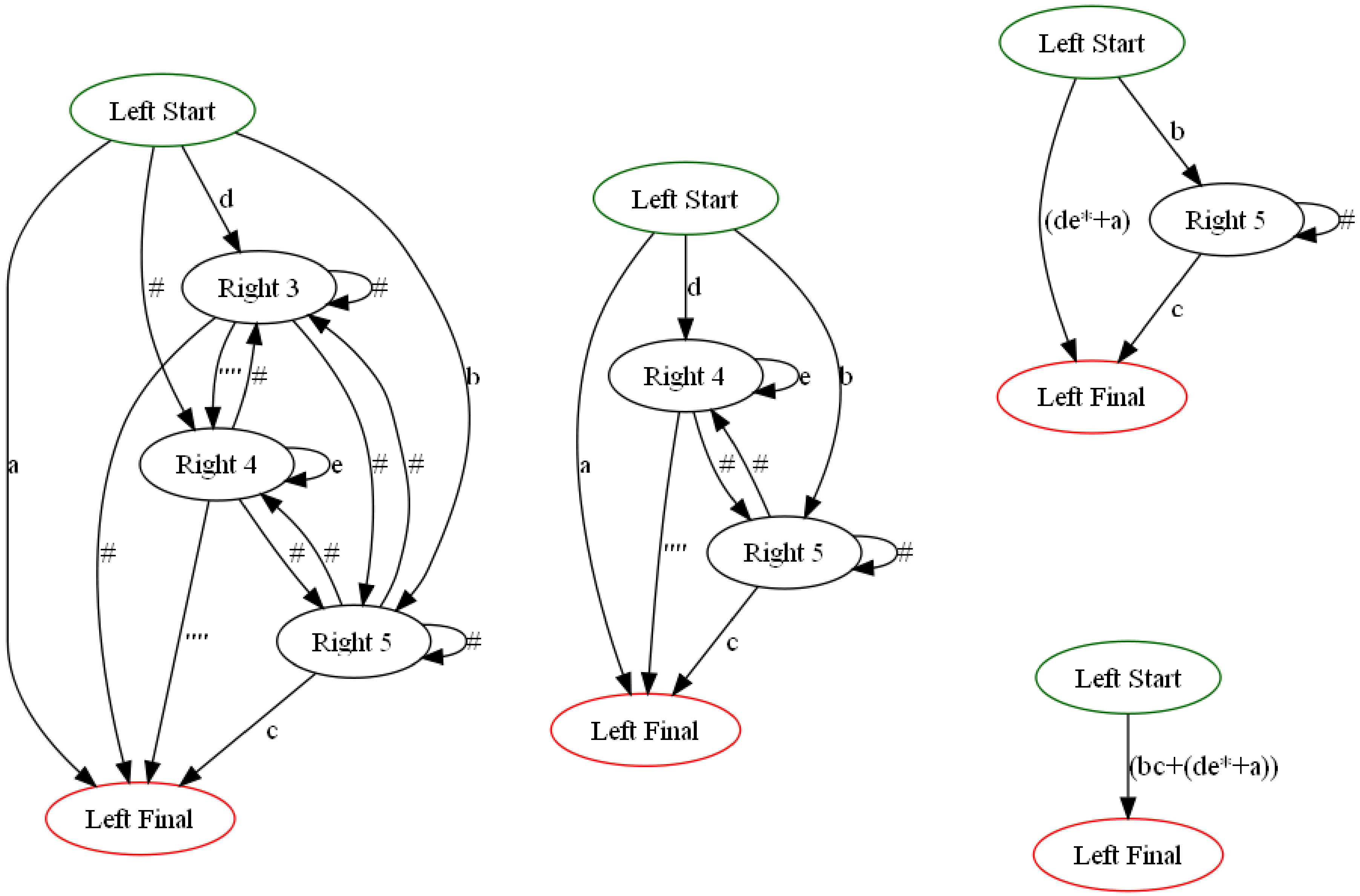
((ba+a)(aa)*ε + ε) + a(aa)*ε)-5

Another example




Left Start
Right 1
Right 2
Right 5
Right 4
Right 3
Left Final
""
#
a
b
c
d
e


Left Start
Right 2
Right 3
Right 5
Right 4
Left Final
a
b
c
d
e
#
""


Left Start
Right 3
Right 4
Right 5
Left Final
d
#
a
b
e
""
c
Left Start
Right 4
Right 5
Left Final
d
a
b
e
#
""
c
Left Start
Right 5
Left Final
b
(de*+a)
#
c
Left Start
Left Final
(bc+(de*+a))

Alg 4 - From DFA to RE via path labels

(not covered in Sipser)

Suppose A is a given DFA. Our goal is to find a regular expression E such that L(E)=L(A).

Assume the state set of A is {1,2,...,n}$. Let L_{ij} be the language consisting of *labels of all paths from i to j*.

Note that $L(A) = \cup_{q \in F} L_{sq}$

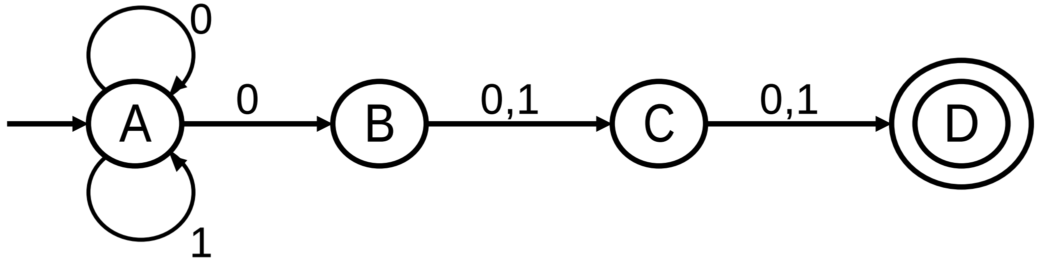
s is the start state,

F is the set of final states,

L_{sq} is the labels of all paths from s to q

We'll be done if we can find regular expressions E_{ij} such that $L_{ij} = L(E_{ij})$.

Exercise: Compute

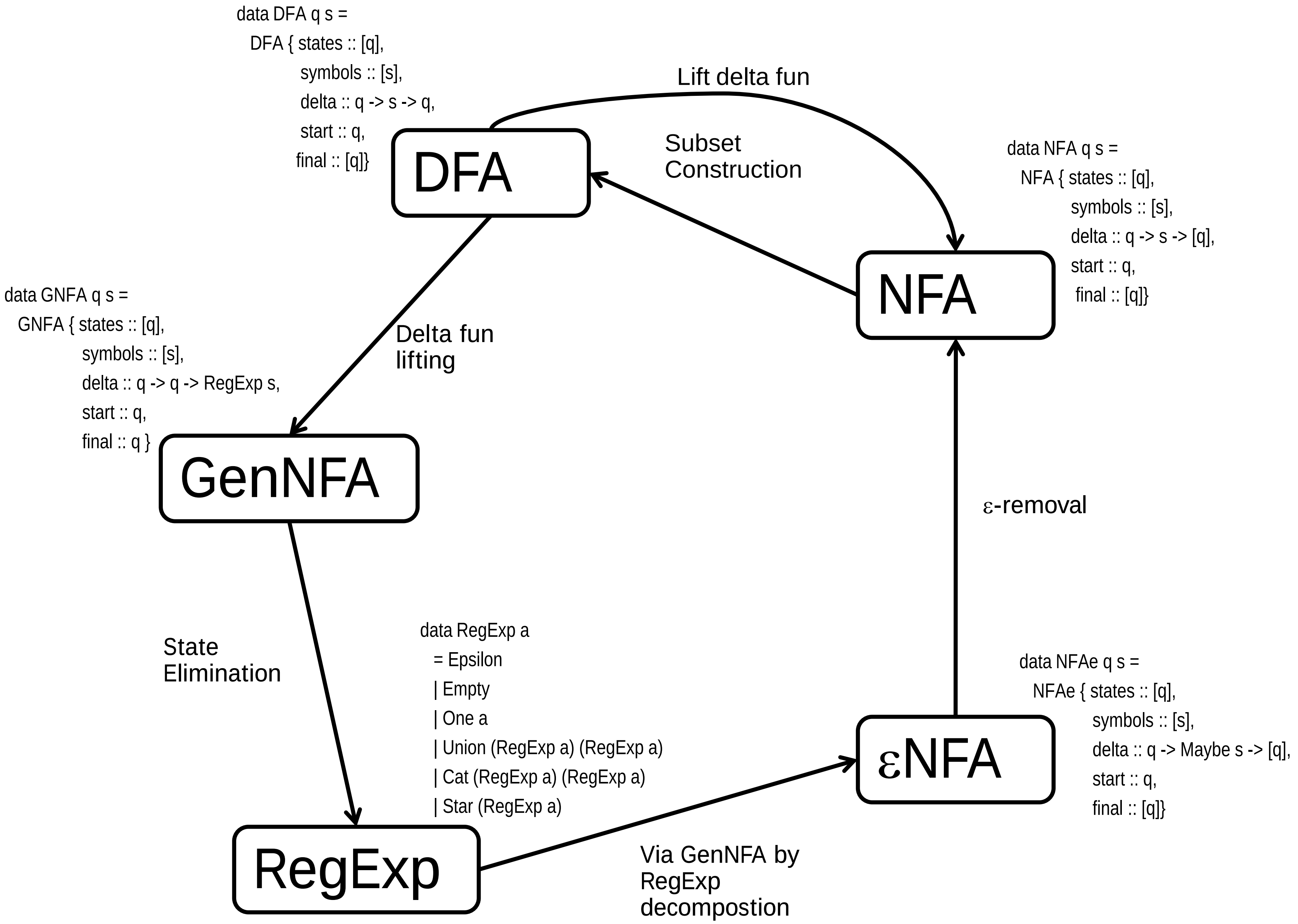


L_{BC} = {“0”, ”1”}

L_{BD} = ______________________________

L_{AB} = ______________________________

L_{AD} = ______________________________



```
data DFA q s =
  DFA { states :: [q],
        symbols :: [s],
        delta :: q -> s -> q,
        start :: q,
       final :: [q]}
```

```
data NFA q s =
  NFA { states :: [q],
        symbols :: [s],
        delta :: q -> s -> [q],
        start :: q,
         final :: [q]}
```

```
data GNFA q s =
  GNFA { states :: [q],
         symbols :: [s],
         delta :: q -> q -> RegExp s,
         start :: q,
         final :: q }
```

```
data RegExp a
  = Epsilon
  | Empty
  | One a
  | Union (RegExp a) (RegExp a)
  | Cat (RegExp a) (RegExp a)
  | Star (RegExp a)
```

```
data NFAe q s =
  NFAe { states :: [q],
         symbols :: [s],
         delta :: q -> Maybe s -> [q],
         start :: q,
         final :: [q]}
```

Decidability of the Regular Languages

Since we can produce an equivalent DFA for any regular expression or ε-NFA, we will know how to recognize whether any two given representations of regular languages are equivalent.

Transform both into into DFAs: X and Y

Minimize X and Y to Xmin and Ymin

Test if Xmin and Ymin are isomorphic.

(problem 1.52, page 91 Sipser)