Theory of Automata
Kleene's Theorem

Week 4

Contents

- 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.

Kleene’s Theorem

- 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) $A \subset B$, and (ii) $B \subset A$.
  - To prove A = B = C, we need to prove (i) $A \subset B$, (ii) $B \subset C$, and (iii) $C \subset A$.

Kleene’s Theorem

- 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.

Turning TGs into Regular Expressions

Proof of Part 2: Turning TGs into Regular Expressions

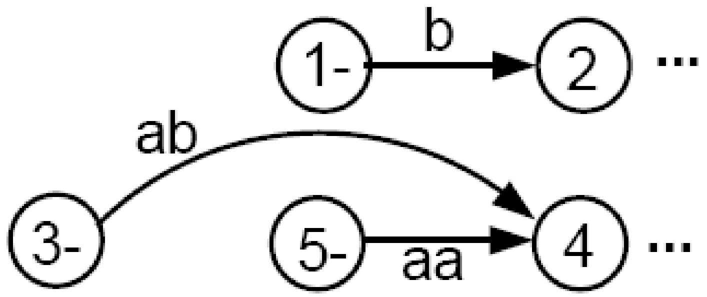
- We prove this part by providing a **constructive algorithm**:
  - We present a algorithm that starts out with a transition graph and ends up with a regular expression that defines the same language.
  - To be acceptable as a method of proof, the algorithm we present will satisfy two criteria:
    - (i) It works for every conceivable TG, and
    - (ii) It guarantees to finish its job in a finite number of steps.
- Slides 8 - 27 below present the proof of part 2.

Step-1 Creating A Unique Start State

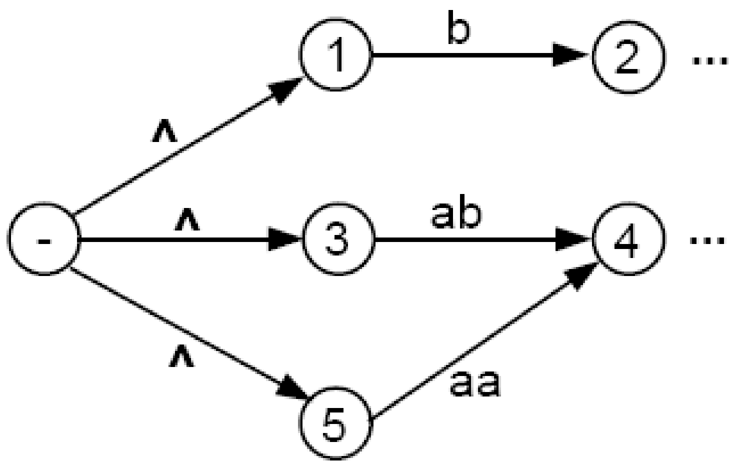
Consider an abstract transition graph T that may have many start states.

- 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

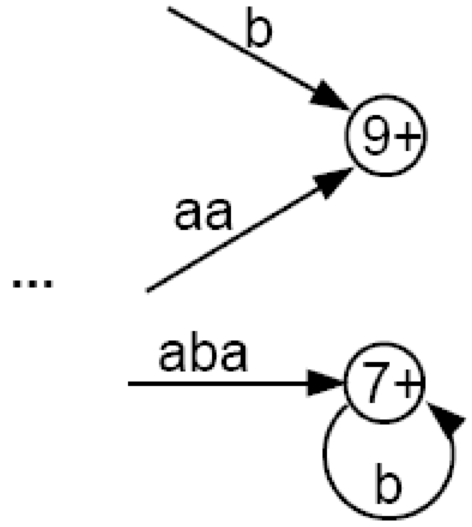


Step 2 - Creating a Unique Final State

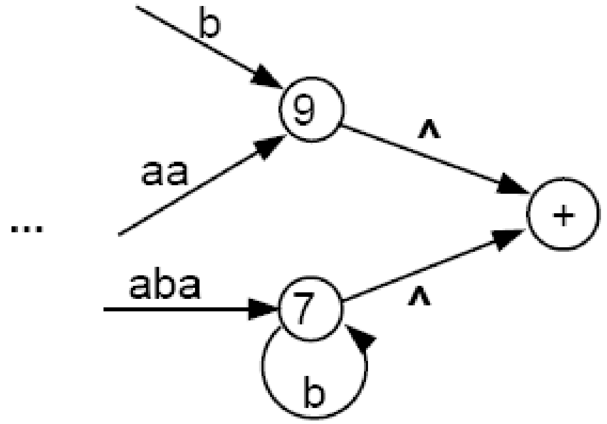
- Let us make another simplification in T so that it has a **unique, un-exitable final state**, without changing the language it accepts.

- 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(see page 36 of the text).

- 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 slide.



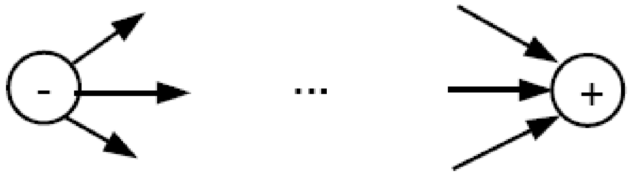
becomes



- 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.

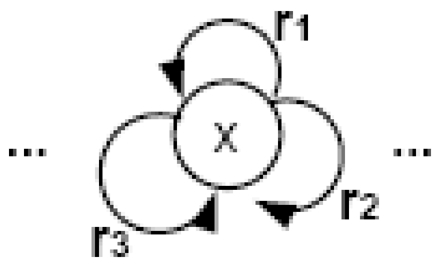
- The machine now has the following shape:



where there are no other - or + states.

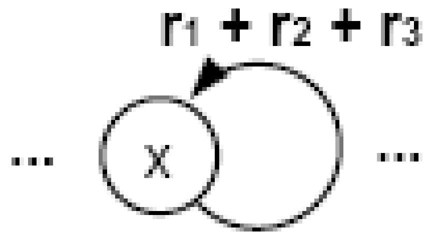
Step 3 -- Combining Edges

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



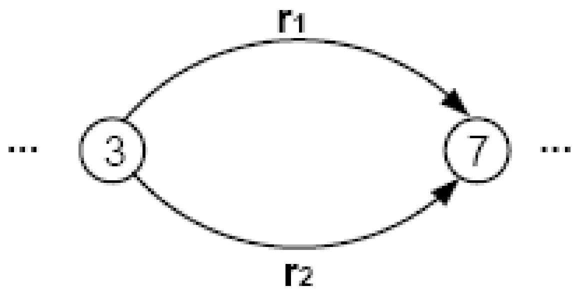
where r_1, r_2, and r_3 are all regular expressions or simple strings.

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

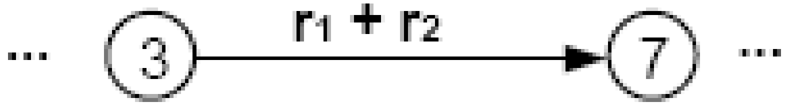


Combining Edges

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

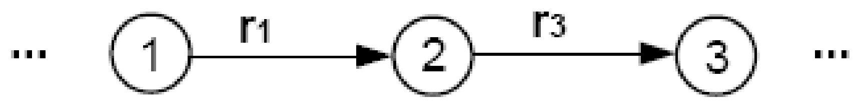


- We can replace this with a single edge labeled with a regular expression:

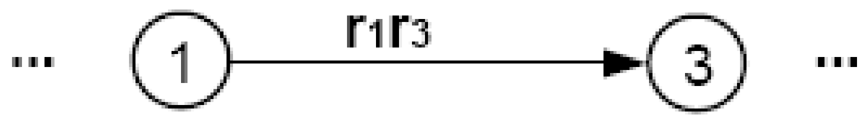


Bypass and State Elimination

- 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:

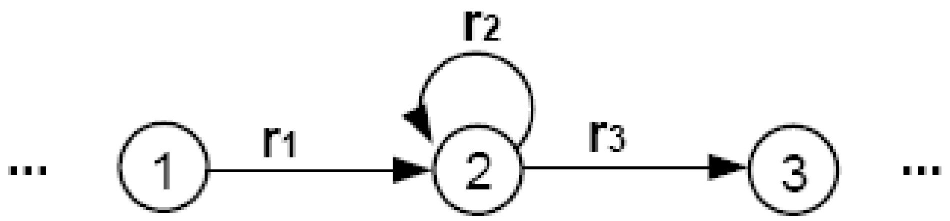


Can be replaced with

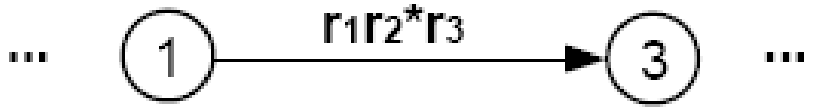


Bypass and State Elimination

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

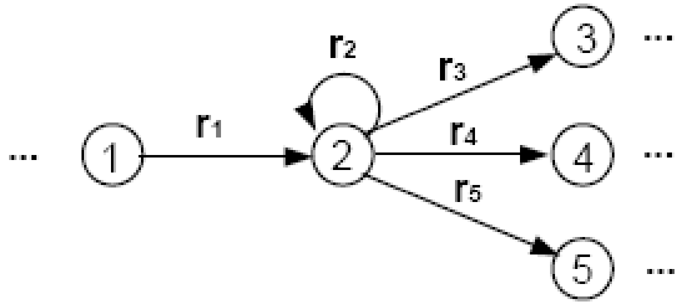


Can be replaced with

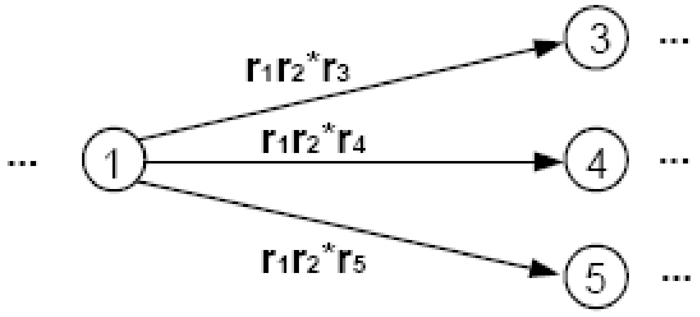


Bypass and State Elimination

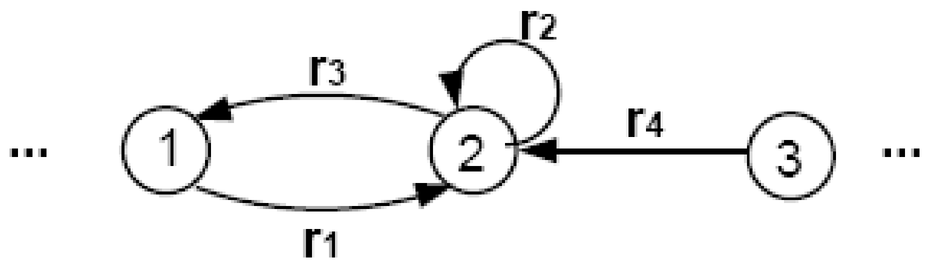
- If the middle state is connected to more than one state, then the bypass and elimination process can be done as follows:



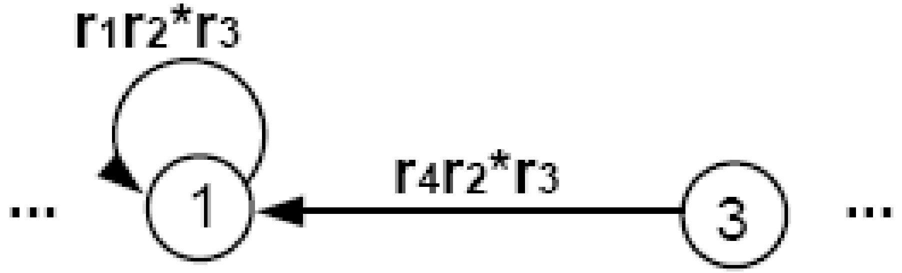
Can be replaced with



Special Cases

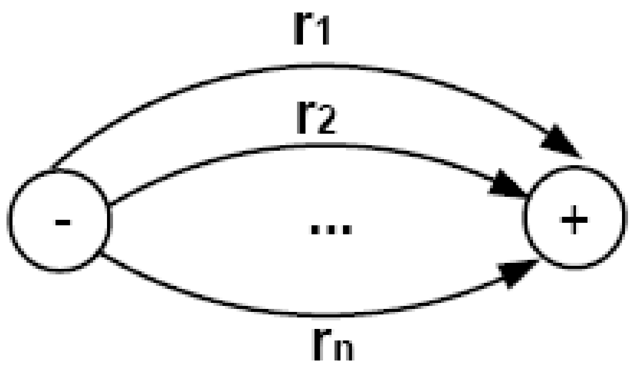


Can be replaced with



Combining Edges

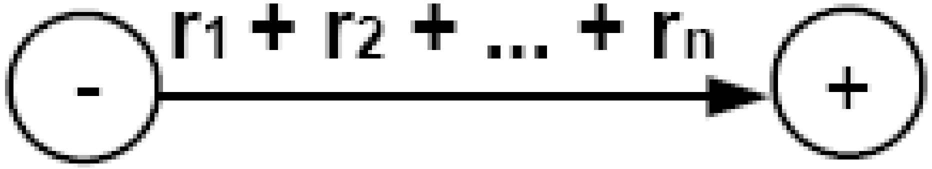
- We can repeat this bypass and elimination process again and again until we have eliminated all the states from T, except for the unique start state and the unique final state. What we come down to is a picture that looks like this:



with each edge labeled by a regular expres...

Combing Edges

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

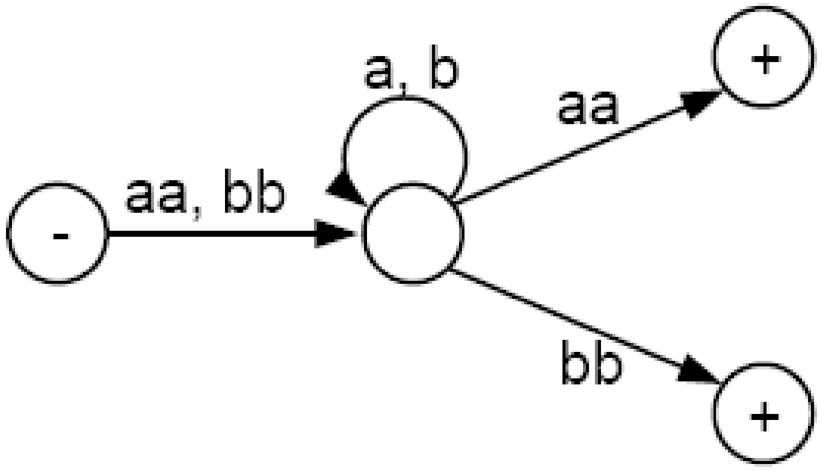


in which the resultant expression is the regular expression that defines the same language as *T* did originally.

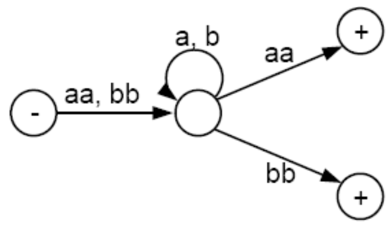
- Recall that all words accepted by *T* are paths through the picture of *T*. If we change the picture but preserve all paths and their labels, we must keep the language unchanged.

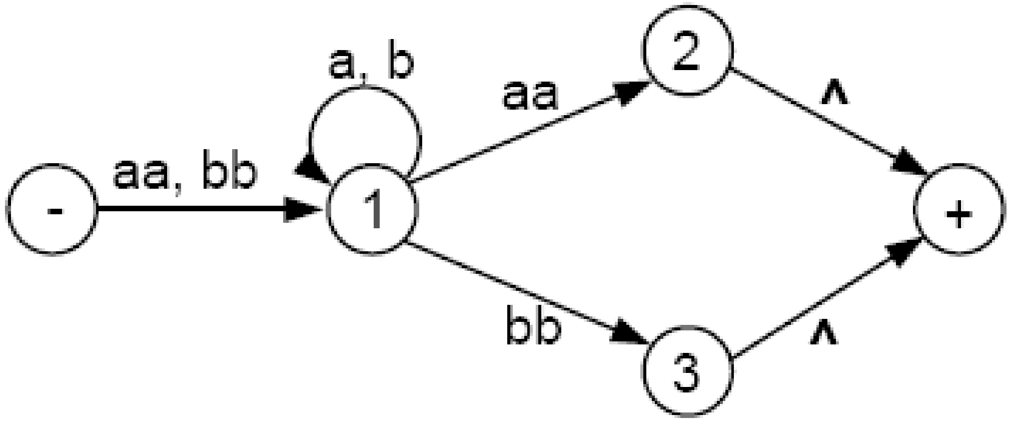
Example

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

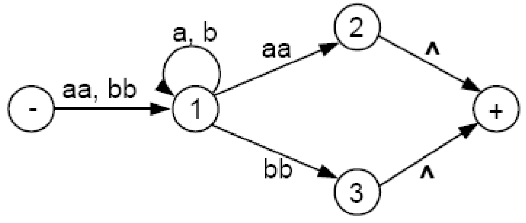


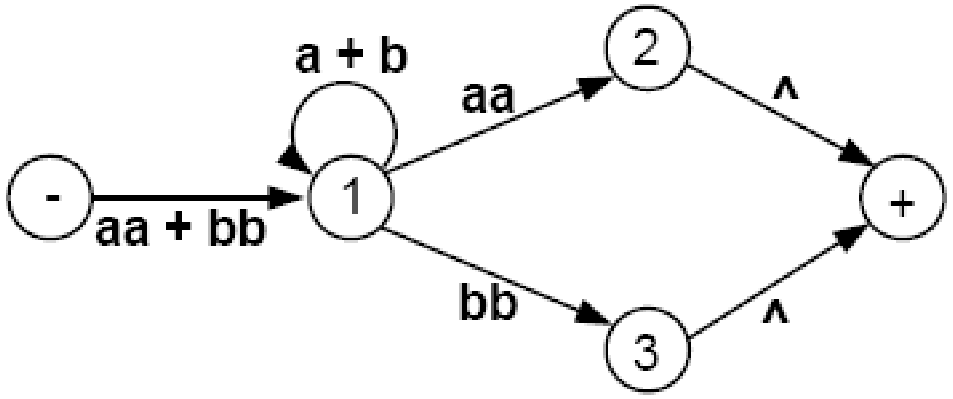
- This TG has only one start state with no incoming edges, but has two final states. So, we must introduce a new unique final state:



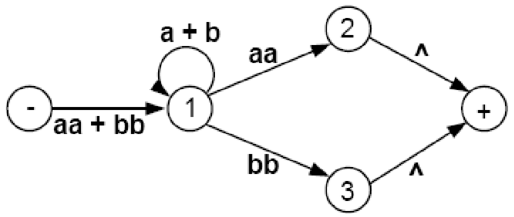


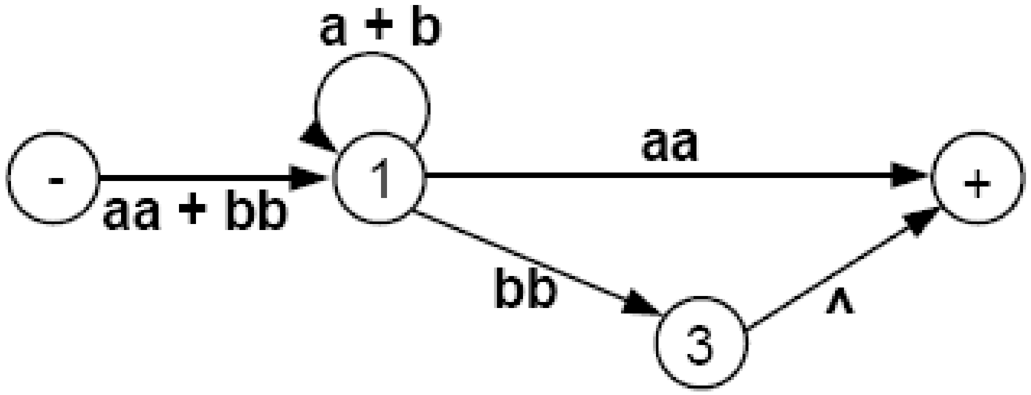
- Now we build regular expressions piece by piece:



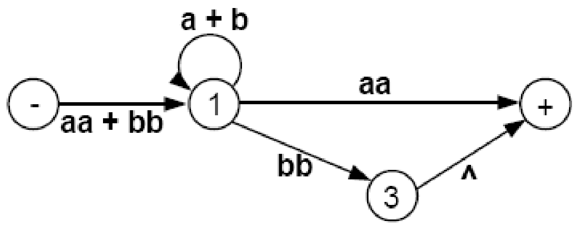


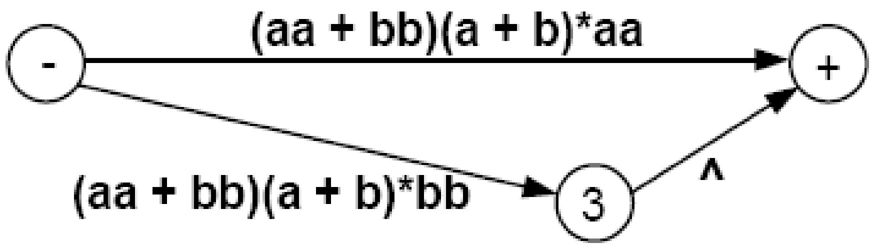
- Eliminate state 2:



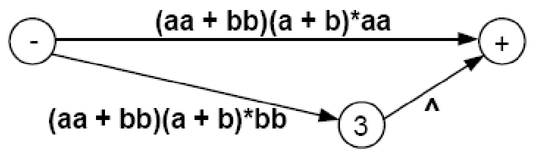


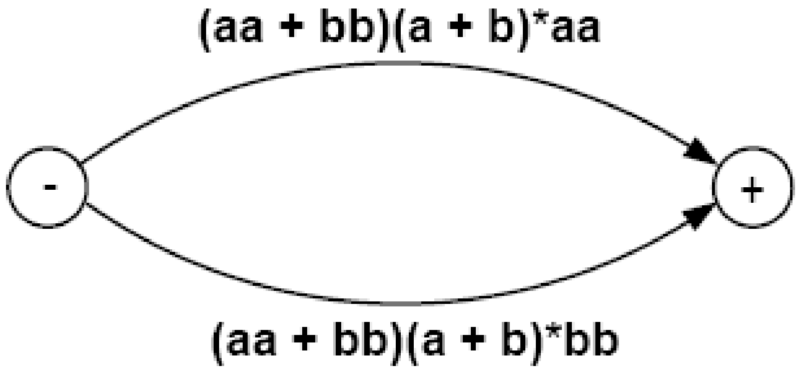
Eliminate state 1:





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

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

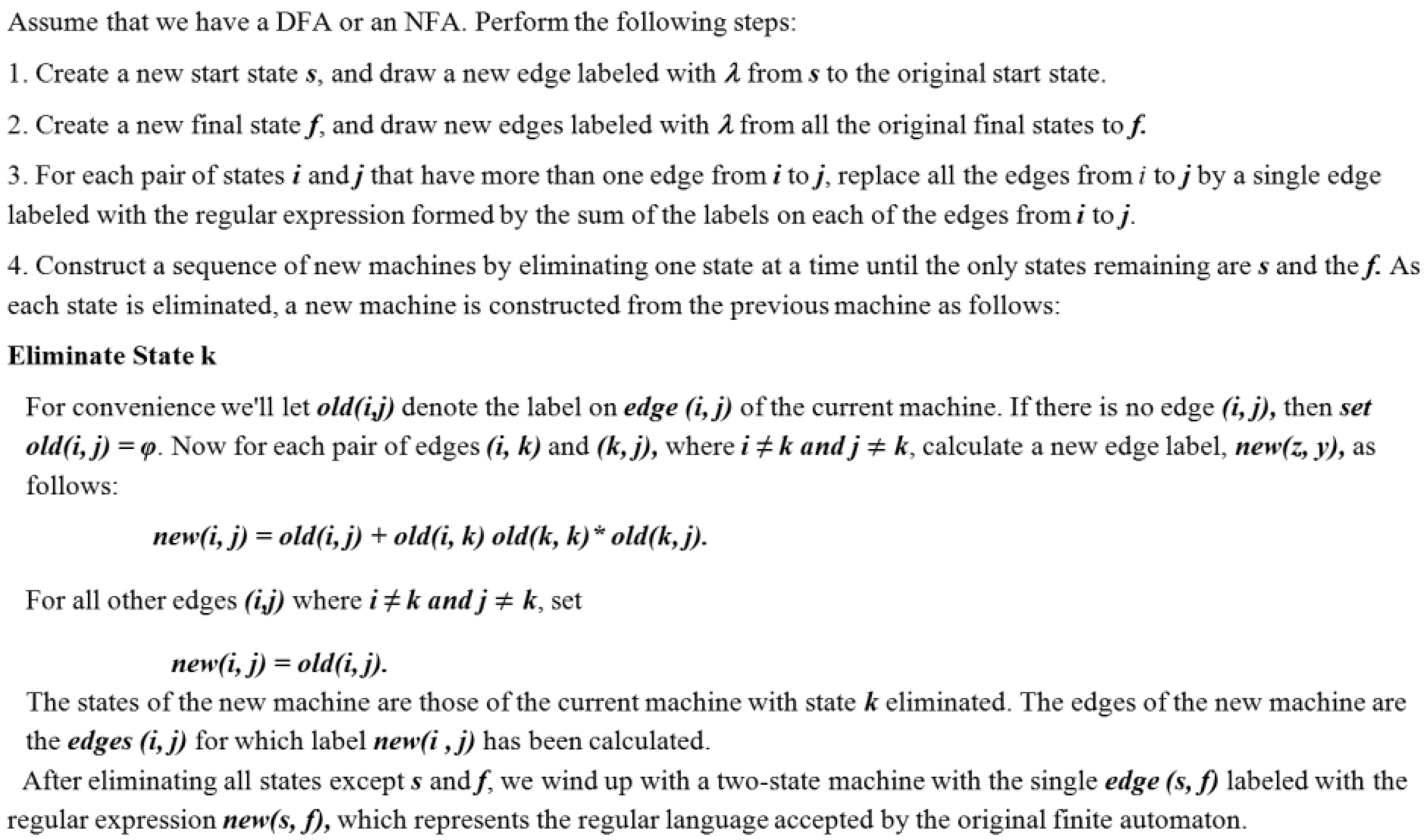
- If we eliminated the states in a different order, we could end up with a different-looking regular expression. But by the logic of the elimination process, these expressions would all have to represent the same language.

- 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.

Algorithm

- Step 1: Create a unique, unenterable minus state and a unique, unleaveable plus state.
- Step 2: One by one, in any order, bypass and eliminate all the non-minus or non-plus states in the TG. A state is bypassed by connecting each incoming edge with each outgoing edge. The label of each resultant edge is the concatenation of the label on the incoming edge with the label on the loop edge (if there is one) and the label on the outgoing edge.
- Step 3: When two states are joined by more than one edge going in the same direction, unify them by adding their labels.
- Step 4: Finally, when all that is left is one edge from - to +, the label on that edge is a regular expression that generates the same language as was recognized by the original TG.

Algorithm for *Finite Automaton to Regular Expression* Conversion 1a



Assume that we have a DFA or an NFA. Perform the following steps:

1. Create a new start state ***s***, and draw a new edge labeled with λ from ***s*** to the original start state.

2. Create a new final state ***f***, and draw new edges labeled with λ from all the original final states to ***f.***

3. For each pair of states ***i*** and ***j*** that have more than one edge from ***i*** to ***j***, replace all the edges from *i* to ***j*** by a single edge labeled with the regular expression formed by the sum of the labels on each of the edges from ***i*** to ***j***.

4. Construct a sequence of new machines by eliminating one state at a time until the only states remaining are ***s*** and the ***f.*** As each state is eliminated, a new machine is constructed from the previous machine as follows:

Eliminate State k

For convenience we'll let ***old(i,j)*** denote the label on ***edge (i, j)*** of the current machine. If there is no edge ***(i, j),*** then ***set old(i, j)*** $= \varphi$. Now for each pair of edges ***(i, k)*** and ***(k, j),*** where $i \neq k$ ***and*** $j \neq k$, calculate a new edge label, ***new(z, y),*** as follows:

$$new(i, j) = old(i, j) + old(i, k)\ old(k, k)^*\ old(k, j).$$

For all other edges ***(i,j)*** where $i \neq k$ ***and*** $j \neq k$, set

$$new(i, j) = old(i, j).$$

The states of the new machine are those of the current machine with state ***k*** eliminated. The edges of the new machine are the ***edges (i, j)*** for which label ***new(i , j)*** has been calculated.

After eliminating all states except ***s*** and ***f***, we wind up with a two-state machine with the single ***edge (s, f)*** labeled with the regular expression ***new(s, f),*** which represents the regular language accepted by the original finite automaton.

Converting Regular Expressions into FAs

FA	NFA	TG
no relaxation	2 relaxation [1 and 2 only]	4 relaxations 1) Δ - transitions 2) non-determinism 3) multiple start states 4) multiple letters on edge

[RE to FA conversion]

a) RE → NFA

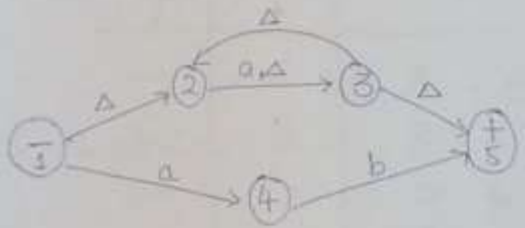
b) NFA → FA

c) FA → minimized FA

FA = DFA

Ex : 1 $a^* + ab$

a) RE → NFA



b) NFA to FA :

λ - closure $\{1\} = \{1, 2, 3, 5\}$ — $\boxed{A}$ +

$\text{gotoset}_a(A) = \{3, 4\}_{(B)}$

$\text{gotoset}_b(A) = \{\ \}$

λ-closure $\{3,4\} = \{3,4,5,2\}$ — $\left(\begin{smallmatrix}B\\+\end{smallmatrix}\right)$

$\text{gotoset}_a(B) = \{3\}_{(C)}$

$\text{gotoset}_b(B) = \{5\}_{(D)}$

λ-closure $\{3\} = \{3,2,5\}$ — $(C+)$

$\text{gotoset}_a(C) = \{3\}_{(C)}$

$\text{gotoset}_b(C) = \{\ \}$

λ-closure $\{5\} = \{5\}$ — $(D+)$

$\text{gotoset}_a(D) = \{\ \}$

$\text{gotoset}_b(D) = \{\ \}$

---

		a	b
+ -	A	B	—
+	B	C	D
+	C	C	—
+	D	—	—

$$\begin{tikzcd}
A\mp \arrow[rr, "a"] & & B+ \arrow[dll, "b"'] \arrow[d, "a"] \\
D+ & & C+ \arrow[loop below, "a"]
\end{tikzcd}$$

Ex2: a^*

a) RE → NFA

$$\begin{tikzcd} -1 \arrow[r, bend left, "a{,}\Lambda"] & +2 \arrow[l, bend left, "\Lambda"] \end{tikzcd}$$

b) NFA → FA

λ-closure {1} = {1, 2} — Ⓐ

$goto_a(A)$ = $\{2\}_{(B)}$

λ-closure {2} = {2, 1} — Ⓑ

$goto_a(B)$ = $\{2\}_{(B)}$

---

	a
∓ A	B
+ B	B

$$\begin{tikzcd} A\mp \arrow[r, "a"] & +B \arrow[loop right, "a"] \end{tikzcd}$$

c) NFA → minimized FA

	a
∓ A	~~B~~ A
~~+ B~~	~~B~~

} duplication
A = B

⇒

	a
± A	A

$$\begin{tikzcd} \pm \arrow[loop right, "a"] \end{tikzcd}$$

Ex 3: $(ab+c)^* d$

a) RE → NFA

$$\begin{tikzcd}
 & 2 \arrow[dr, "b"] & & \\
-1 \arrow[ur, "a"] \arrow[rr, bend right, "{c,\Delta}"'] & & 3 \arrow[ll, bend right, "\Delta"'] \arrow[r, "d"] & +4
\end{tikzcd}$$

b) NFA → FA

λ- closure $\{1\} = \{1, 3\}$ — A

$goto_a(A) = \{2\}_B$

$goto_b(A) = \{\ \}$

$goto_c(A) = \{3\}_A$

$goto_d(A) = \{4\}_C$

λ- closure $\{2\} = \{2\}$ — B

$goto_a(B) = \{\ \}$

$goto_b(B) = \{3\}_A$

$goto_c(B) = \{\ \}$

$goto_d(B) = \{\ \}$

λ- closure $\{3\} = \{\ ,\ \}$ — A

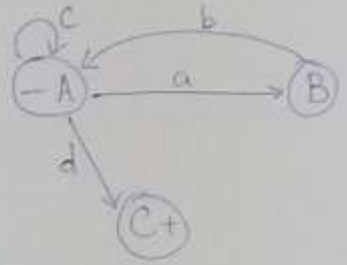
λ- closure $\{4\} = \{4\}$ — C+

$goto_a(C) = \{\ \}$ $\quad$ $goto_c(C) = \{\ \}$

$goto_b(C) = \{\ \}$ $\quad$ $goto_d(C) = \{\ \}$

		a	b	c	d
–	A	B	–	A	C
	B	–	A	–	–
+	C	–	–	–	–

[Already minimized]


$$\begin{tikzcd}
-A \arrow[loop above, "c"] \arrow[r, "a"] \arrow[d, "d"] & B \arrow[l, bend right, "b"'] \\
C+ &
\end{tikzcd}$$

NFA - Non-Deterministic Finite Automata

NFA

Definition: An NFA is a TG with a unique start state and with the property that each of its edge labels is a single alphabet letter.

- The regular deterministic finite automata are referred to as DFAs, to distinguish them from NFAs.
- As a TG, an NFA can have arbitrarily many a-edges and arbitrarily many b-edges coming out of each state.
- An input string is accepted by an NFA if there exists any possible path from - to +.

NFA

- An NFA is quintuple $M=\{Q,\Sigma,q_0, F, \delta\}$
  - Q is set of finite states
  - Σ is a finite set of symbols called ***alphabet***
  - q_0 belongs to Q is distinguished ***Start State***
  - F is subset of Q called the ***Final*** or Accepting states
  - δ is a total function from Q x Σ to P(Q) known as ***transition function***, such that, an input symbol may cause more than one next states, i.e., to one state out of a set of possible next states.
  - P(Q) is the power set of Q, that is, 2^Q

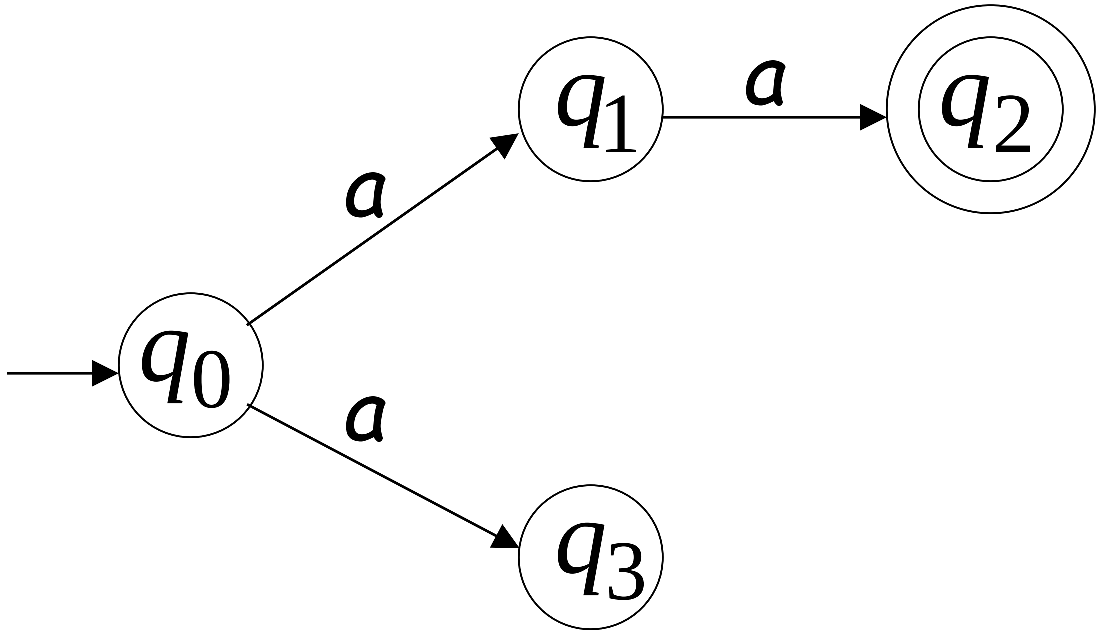
NFA

Definition: A nondeterministic finite automaton (or NFA) is a TG with a unique start state and with the property that each of its edge labels is a single alphabet letter.

- The regular deterministic finite automata are referred to as DFAs, to distinguish them from NFAs.
- As a TG, an NFA can have arbitrarily many a-edges and arbitrarily many b-edges coming out of each state.
- An input string is accepted by an NFA if there exists any possible path from - to +.

Nondeterministic Finite Accepter (NFA)

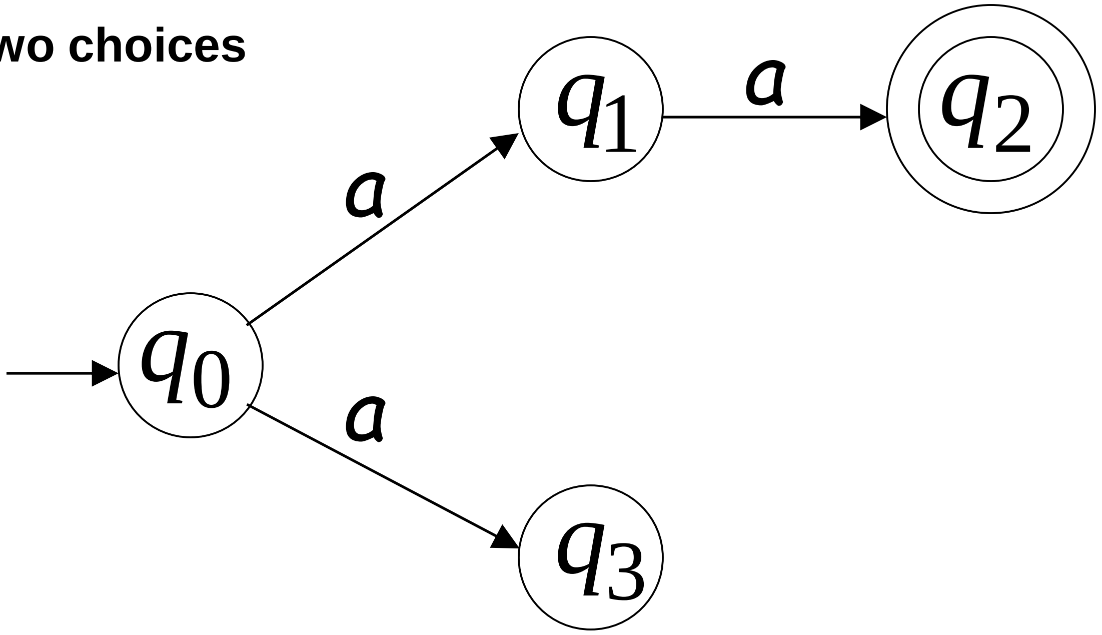
Alphabet = $\{a\}$



Nondeterministic Finite Accepter (NFA)

Alphabet = $\{a\}$

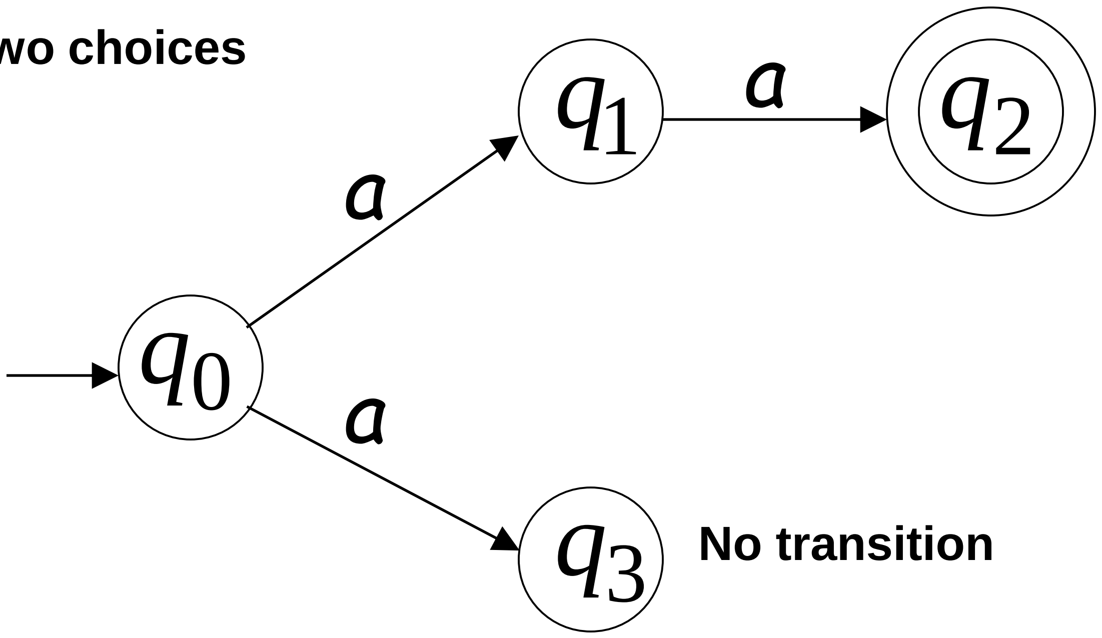
Two choices



Nondeterministic Finite Accepter (NFA)

Alphabet = $\{a\}$

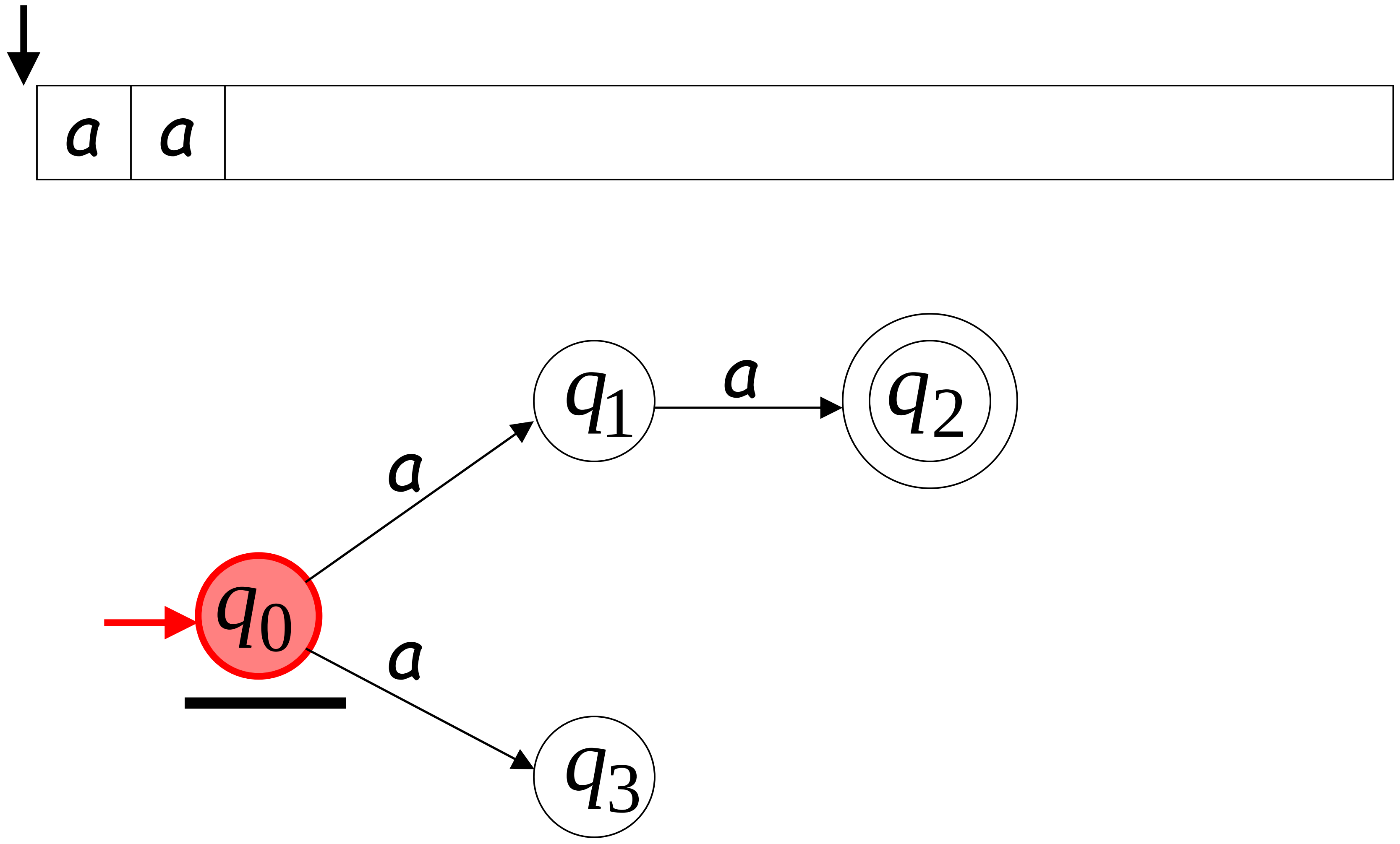
Two choices



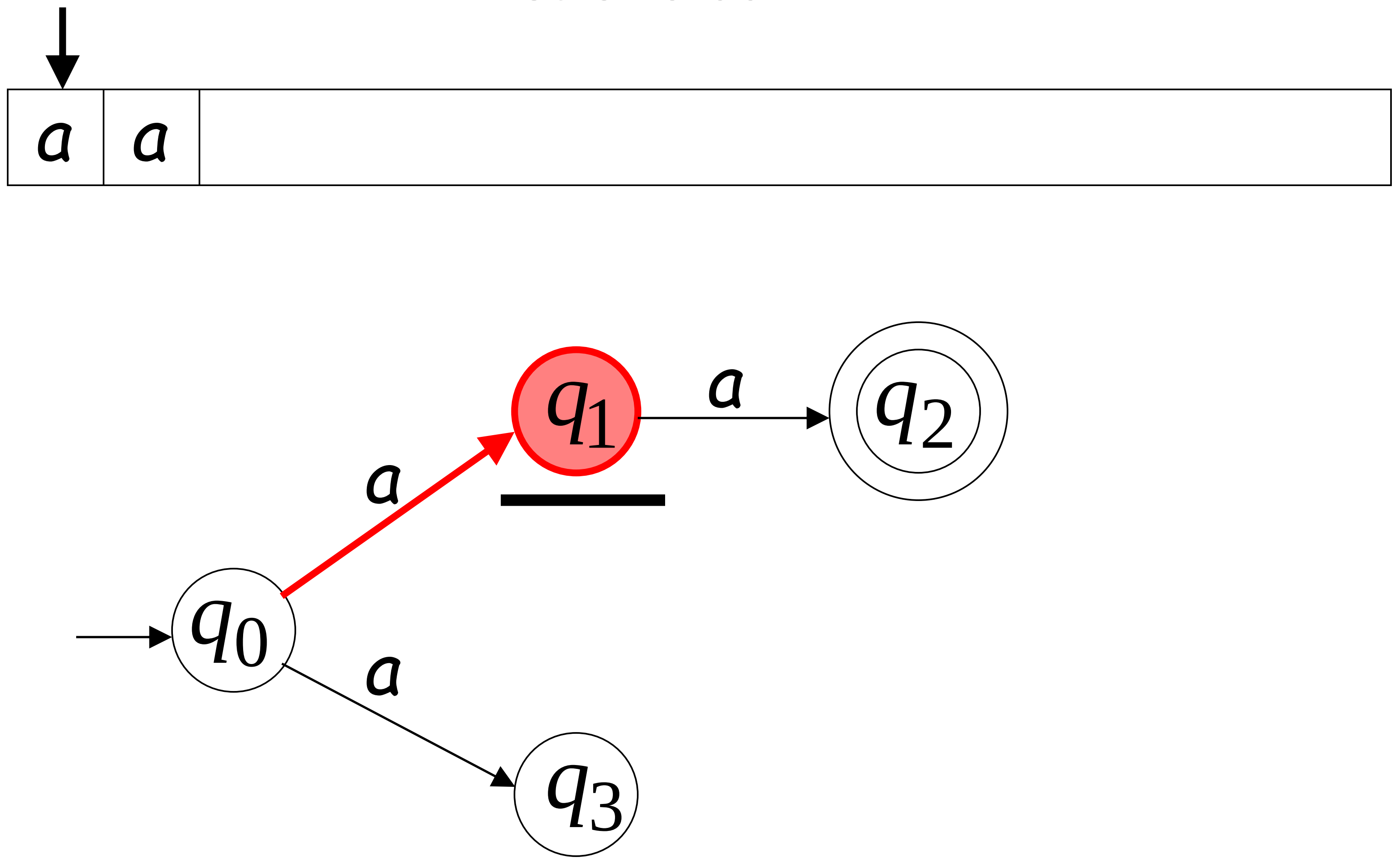
No transition

No transition

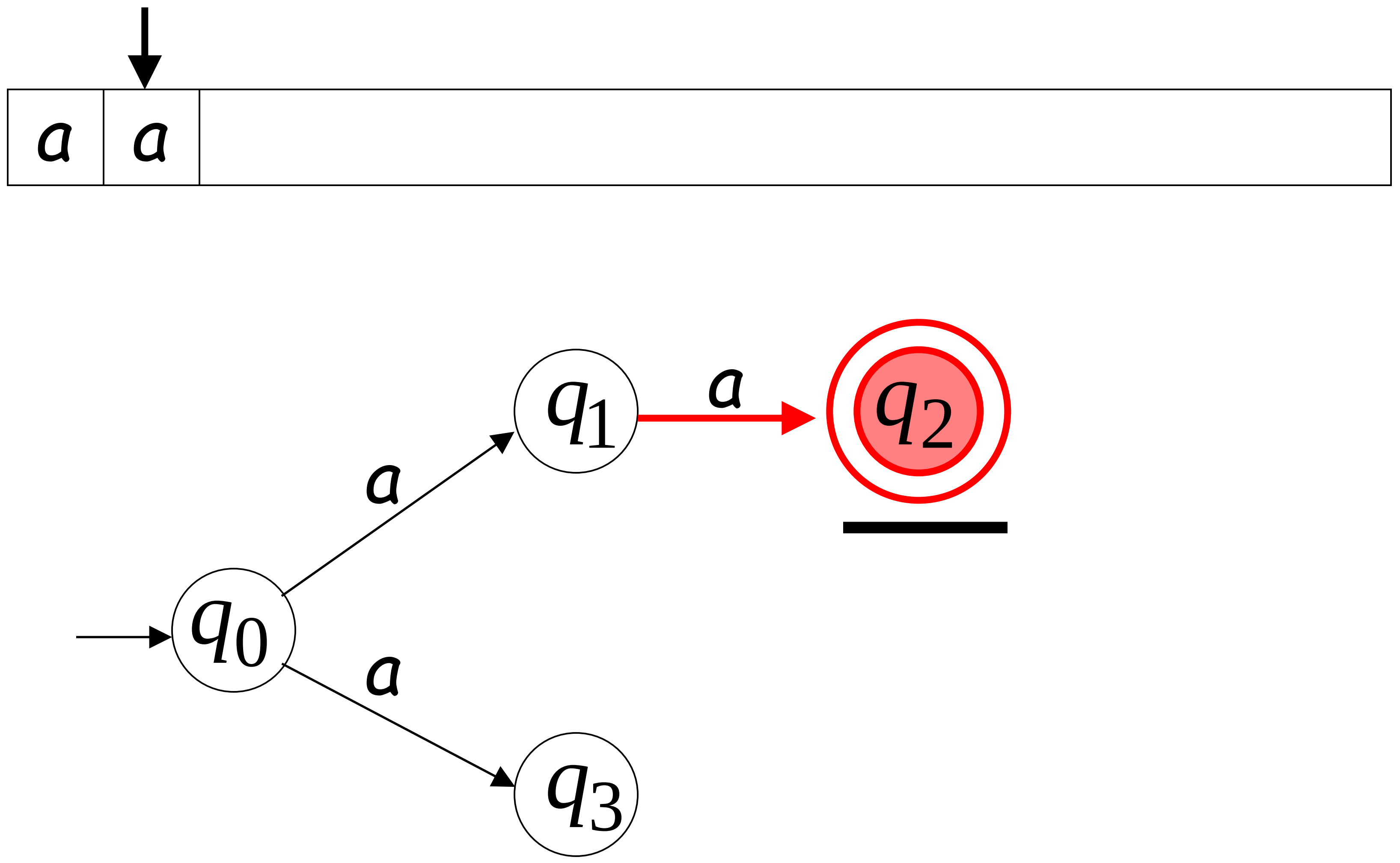
First Choice



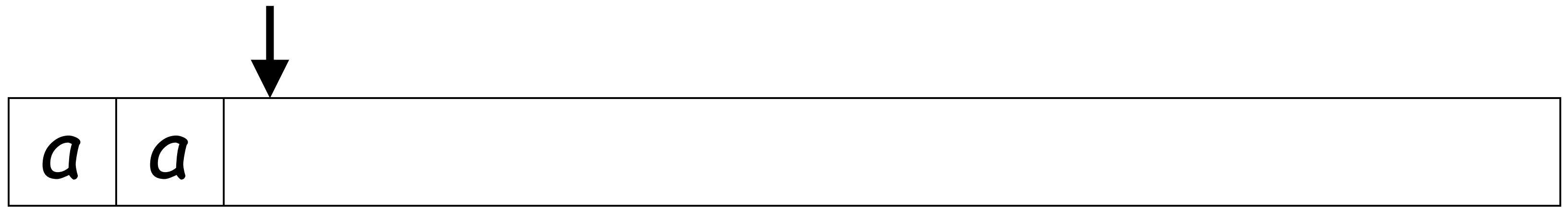
First Choice

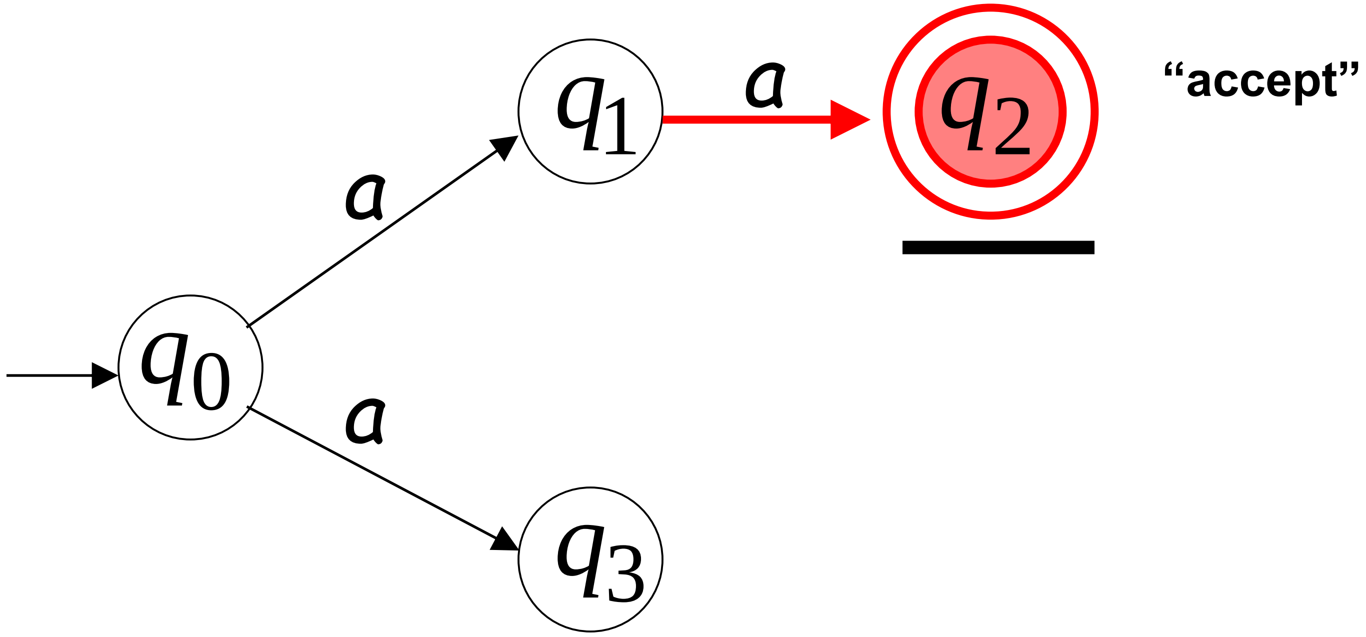


First Choice

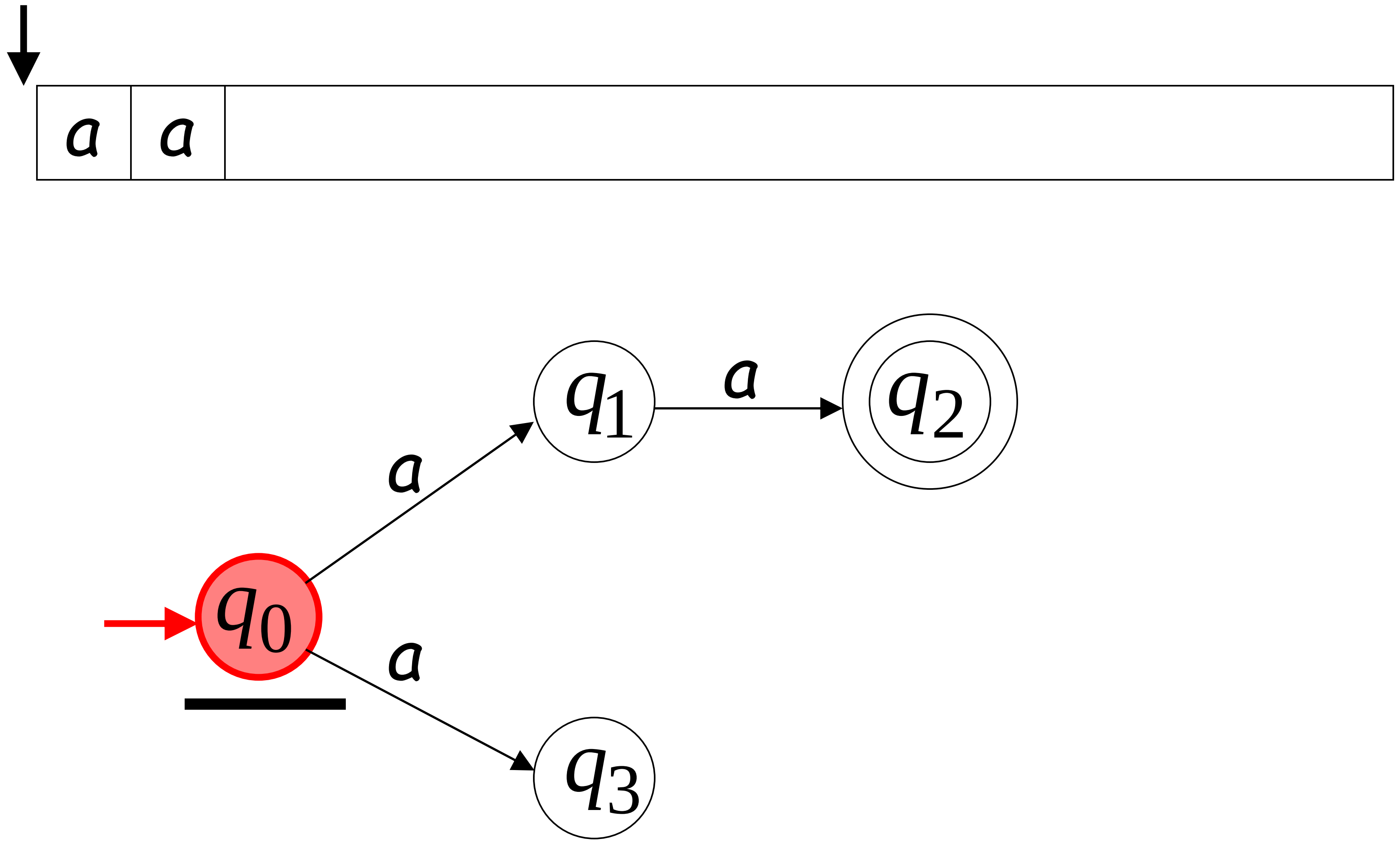


First Choice

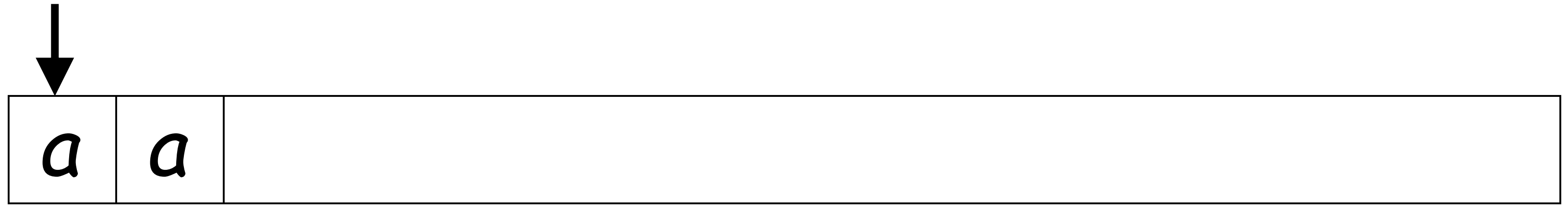


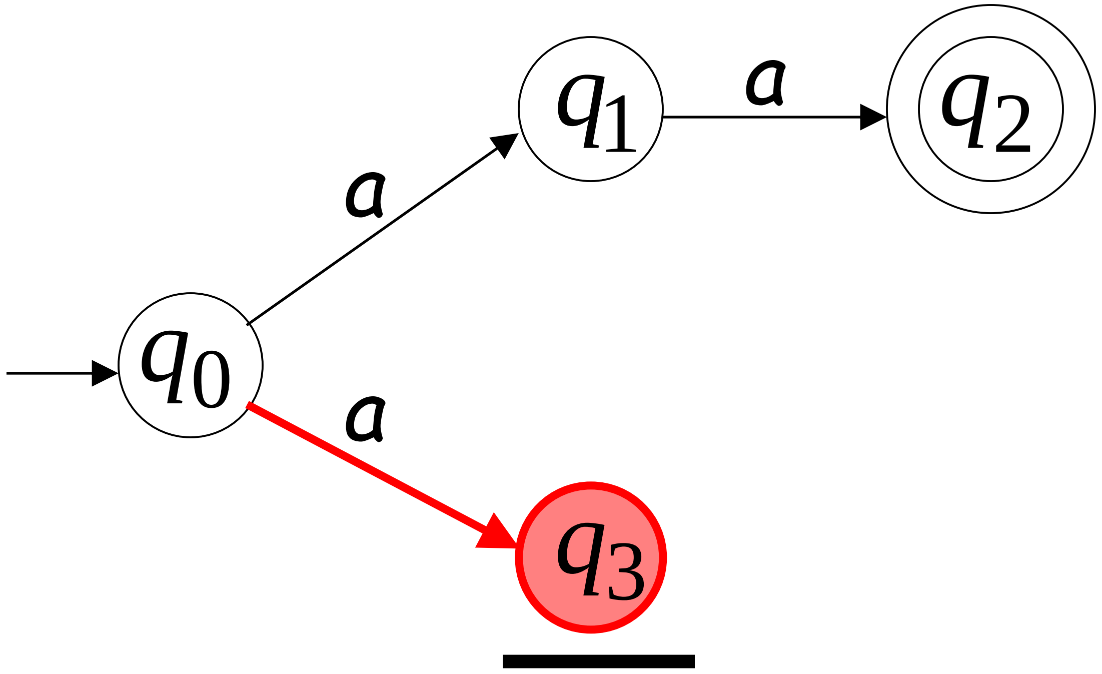


Second Choice

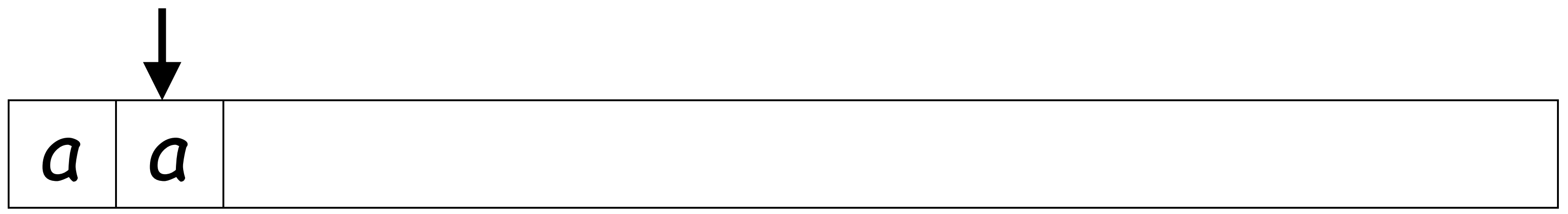


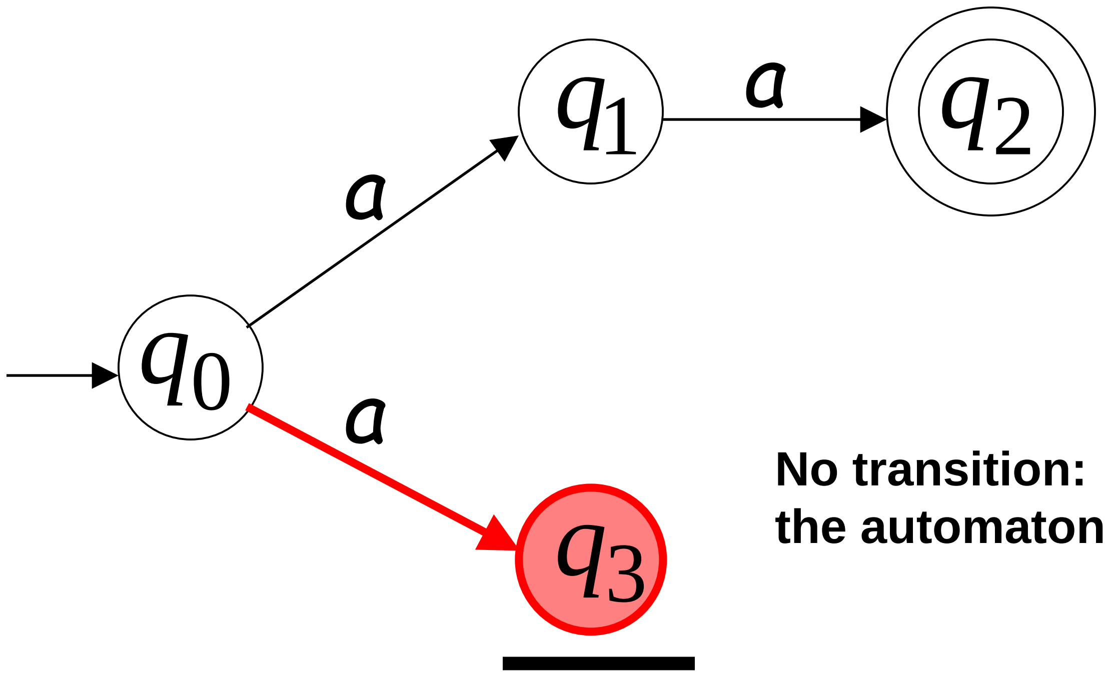
Second Choice





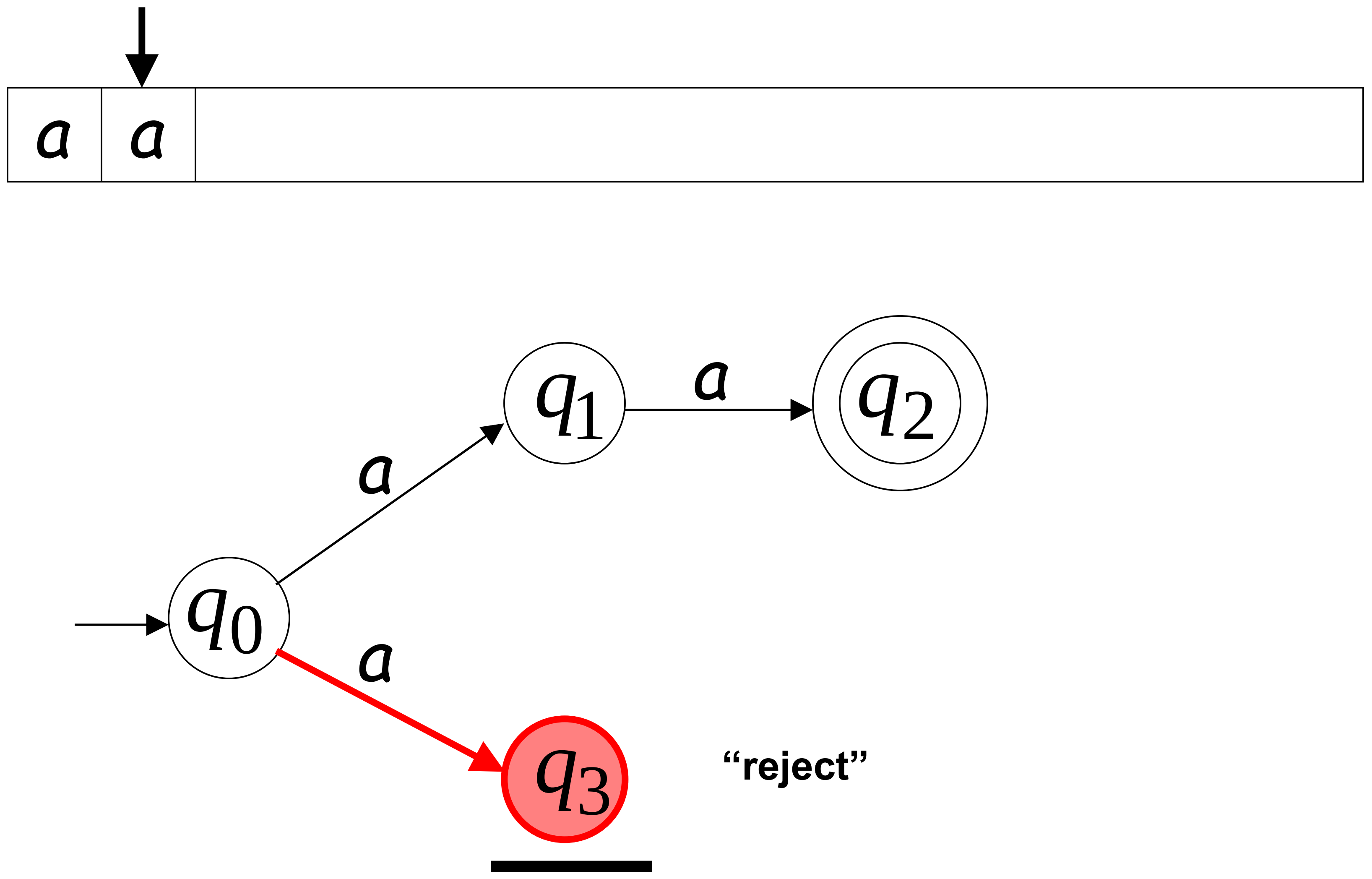
Second Choice





No transition:
the automaton hangs

Second Choice



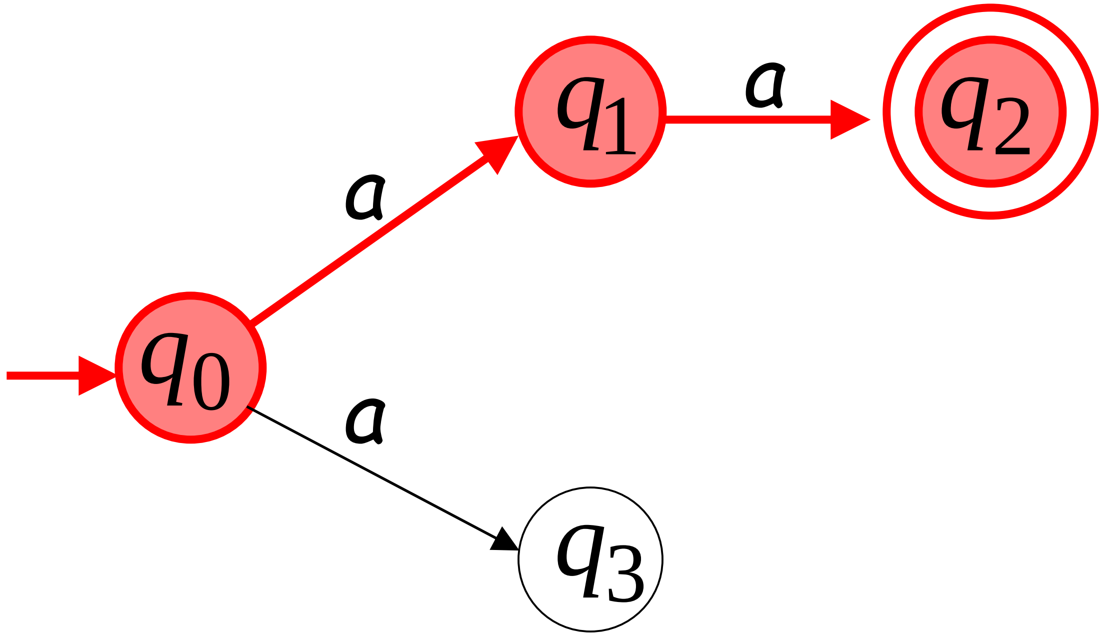
Observation

An NFA accepts a string:

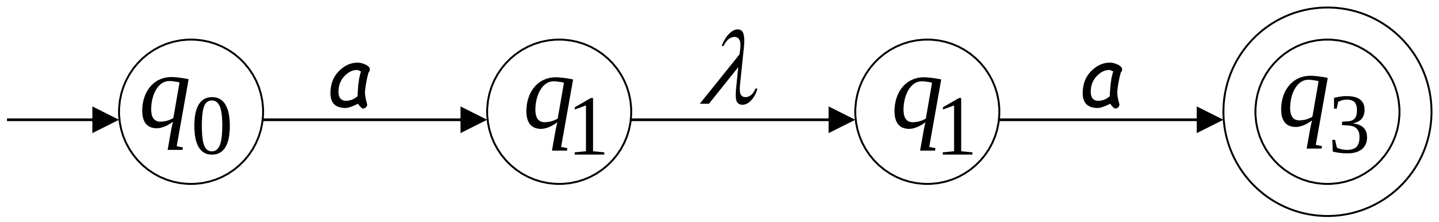
if
there is a computation of the NFA
that accepts the string

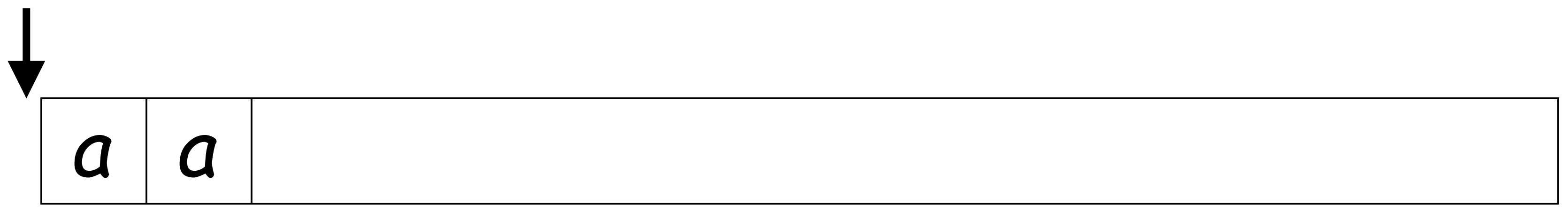
Example

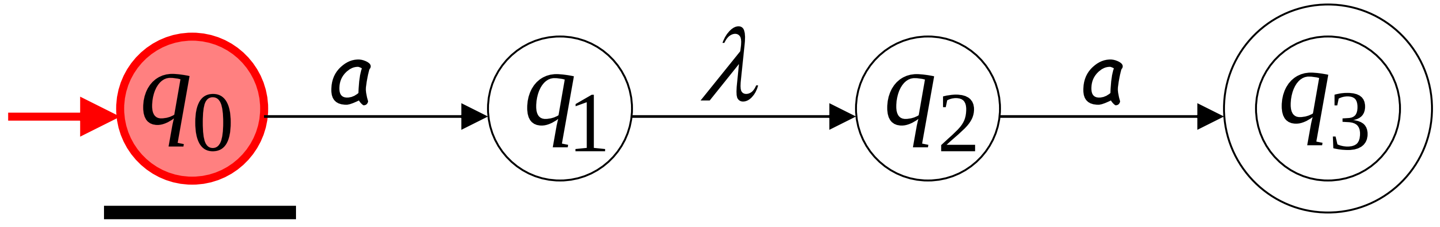
aa **is accepted by the NFA:**

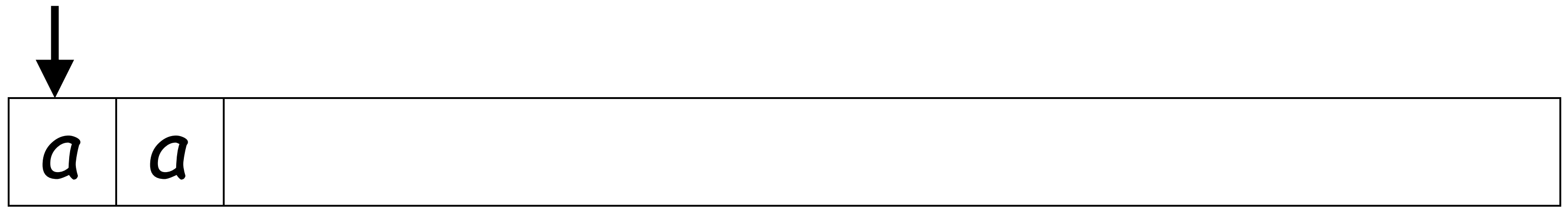


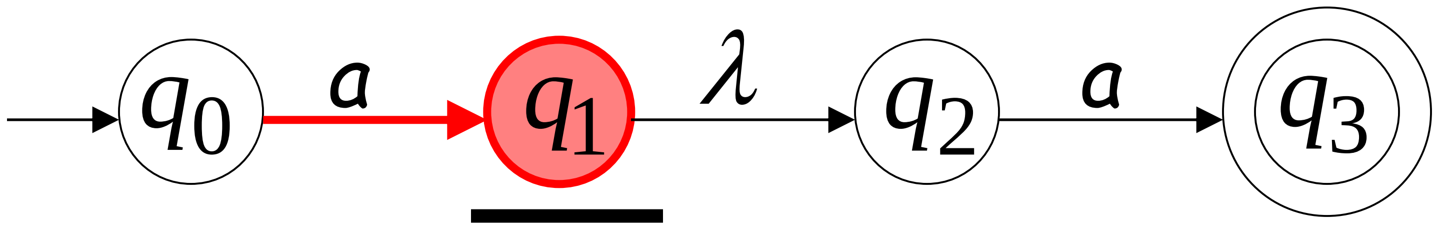
Lambda Transitions



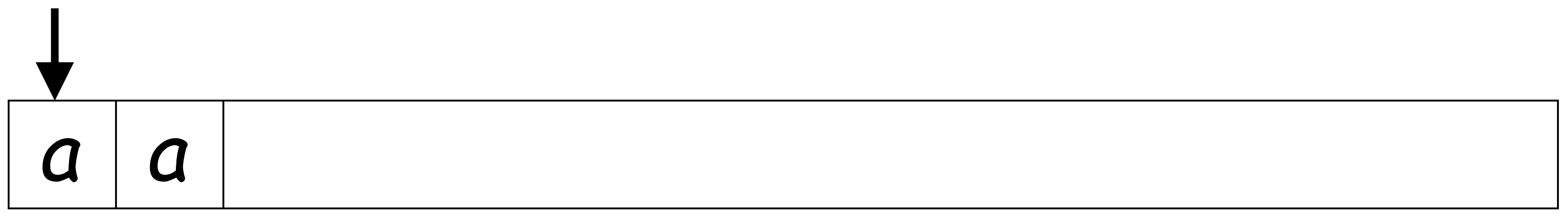

a
a

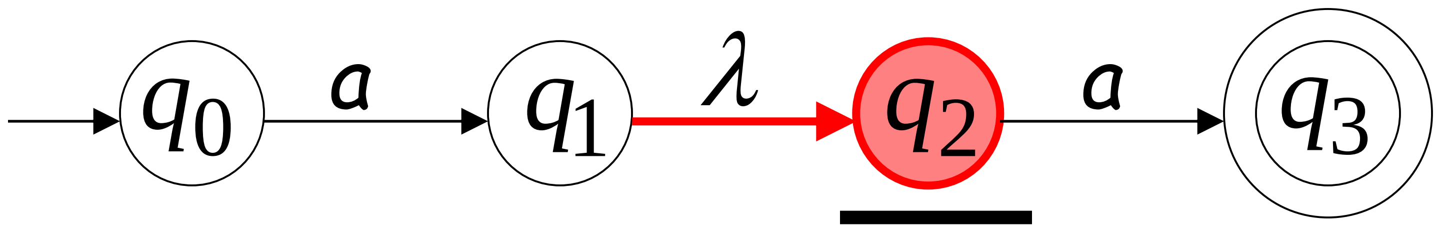

q_0
a
q_1
λ
q_2
a
q_3

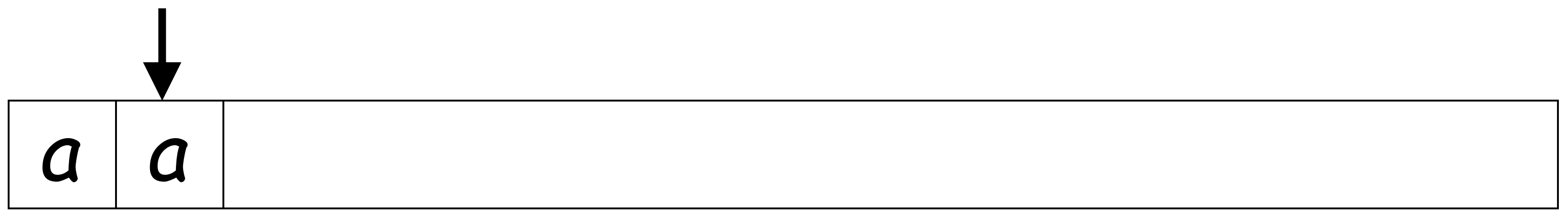

a
a

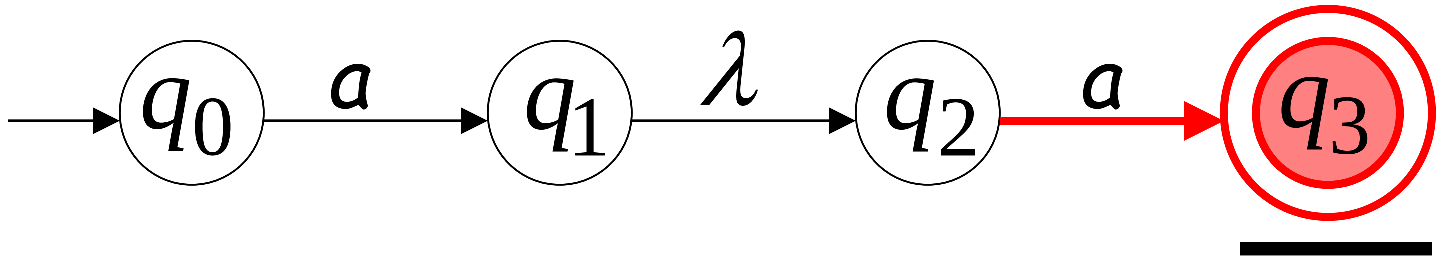

q_0
a
q_1
λ
q_2
a
q_3

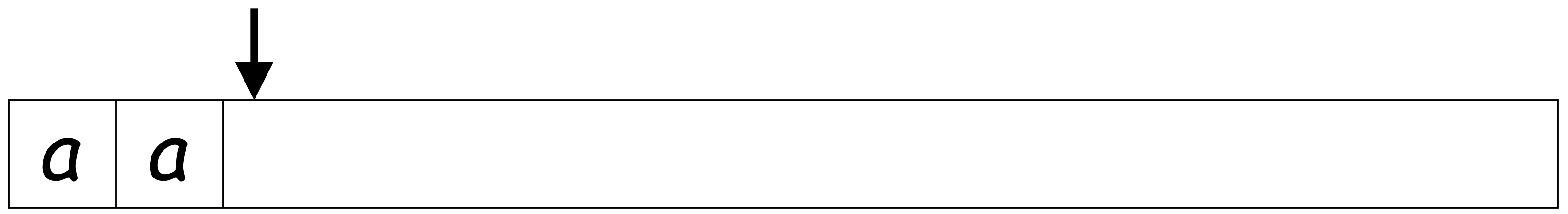
(read head doesn't move)

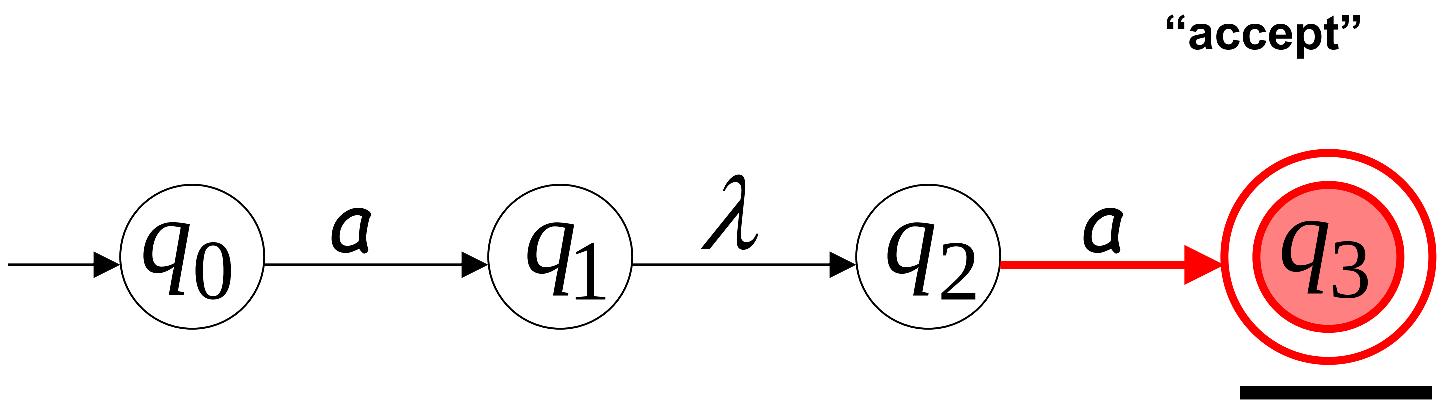





a
a

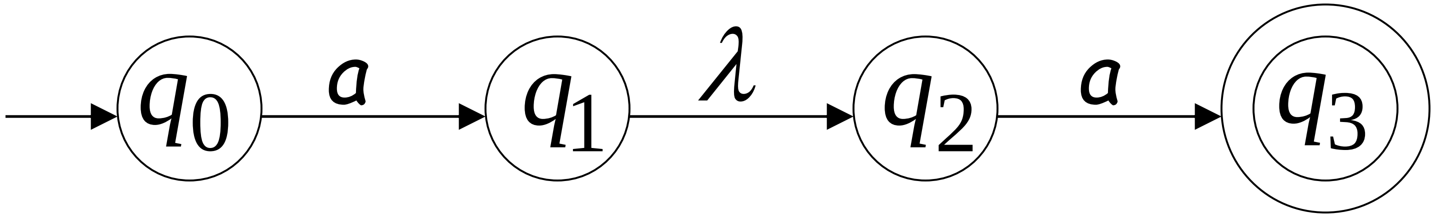

q_0
a
q_1
λ
q_2
a
q_3



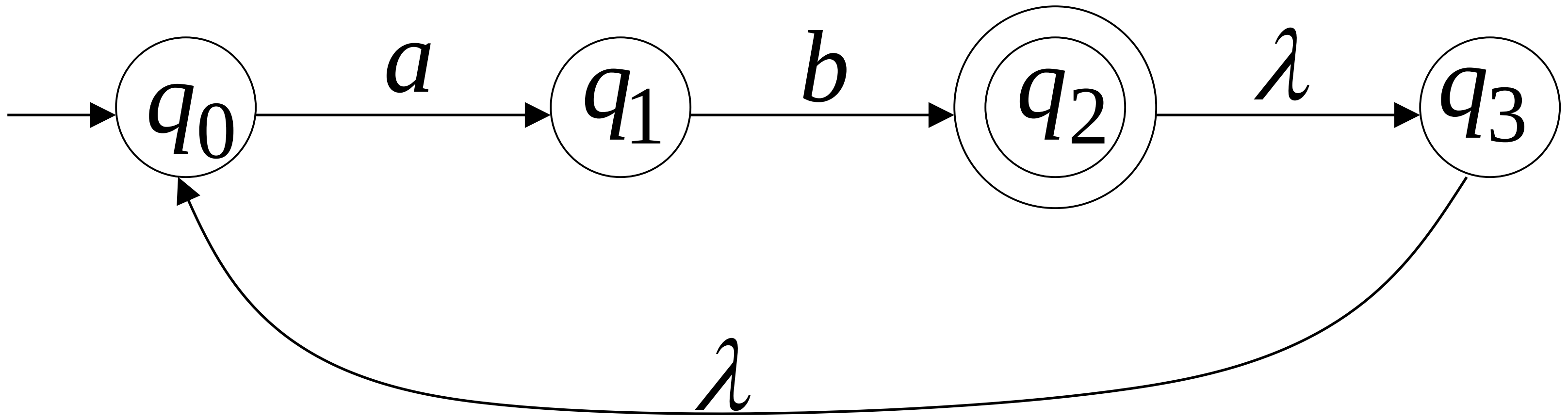


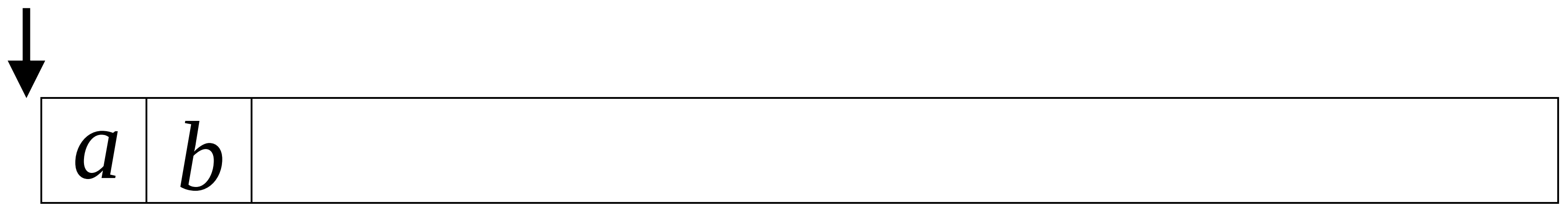
String aa is accepted

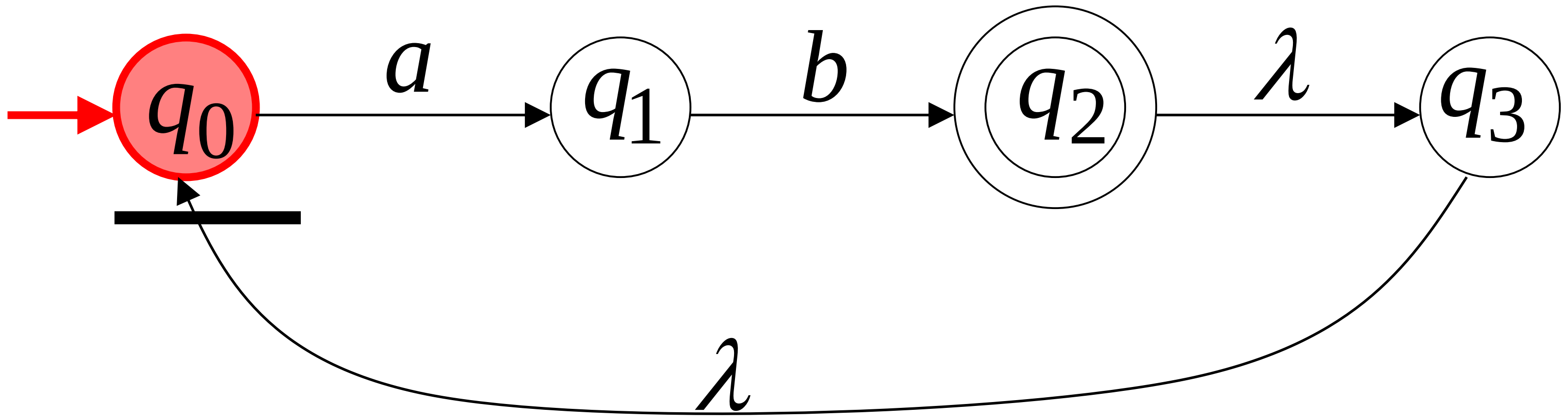
Language accepted: $L = \{aa\}$

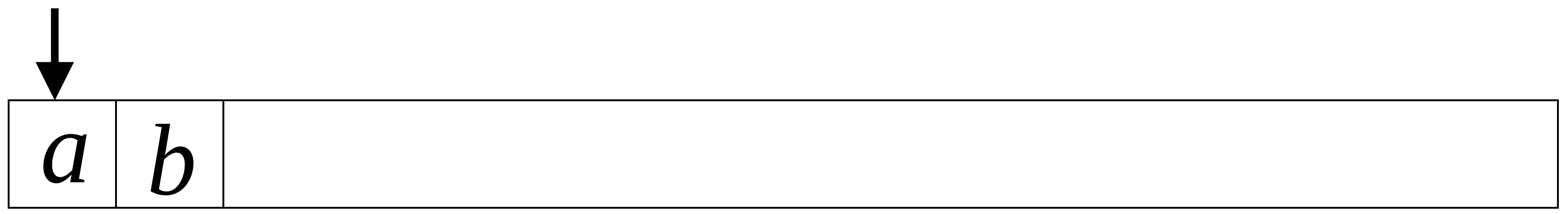

$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "a"] & q_1 \arrow[r, "\lambda"] & q_2 \arrow[r, "a"] & \boxed{q_3} \end{tikzcd}$$

Another NFA Example

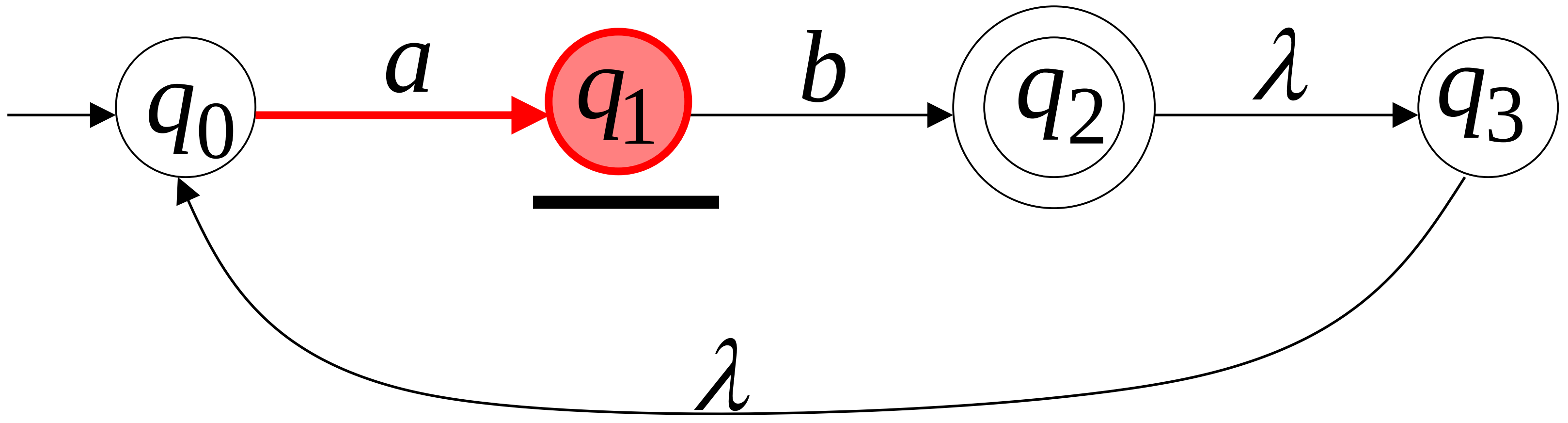



a
b

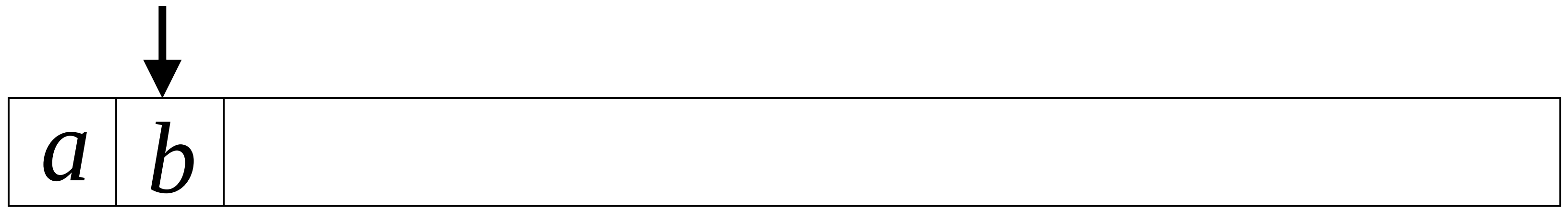

q_0
a
q_1
b
q_2
λ
q_3
λ



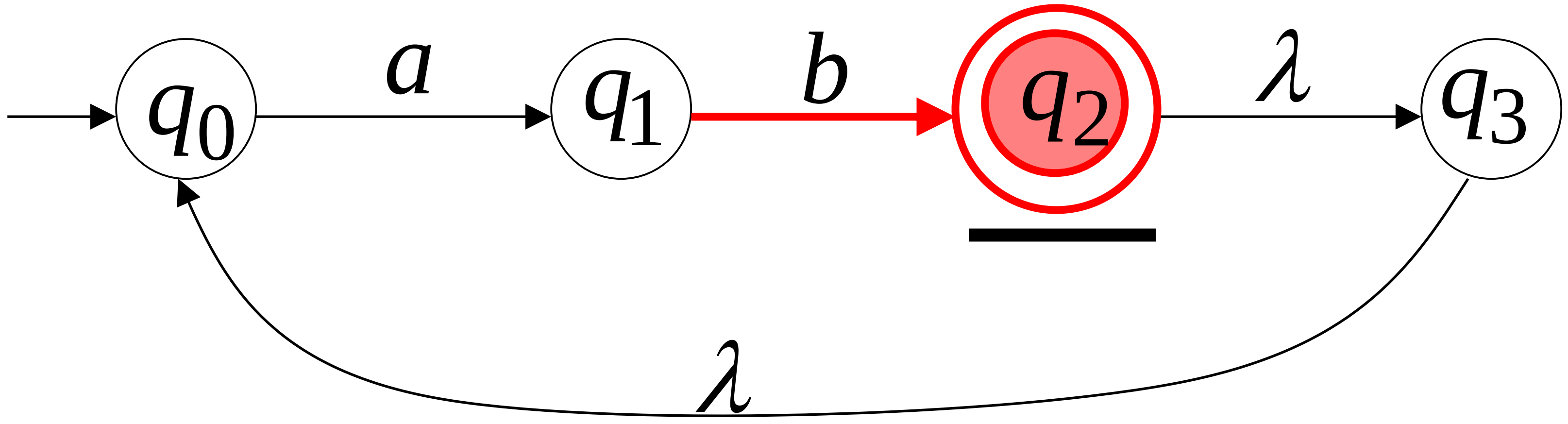
a
b



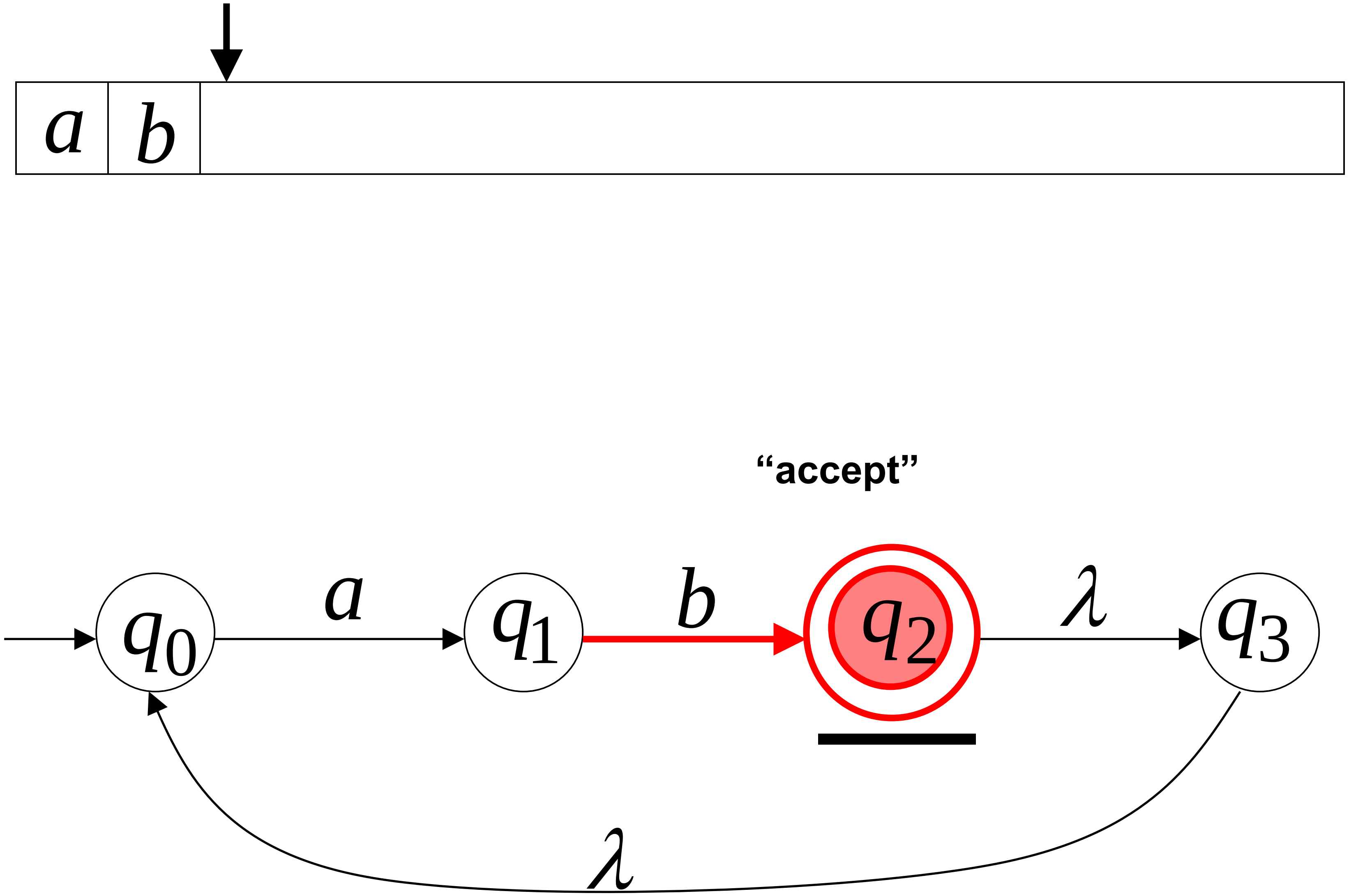
q_0
a
q_1
b
q_2
λ
q_3
λ



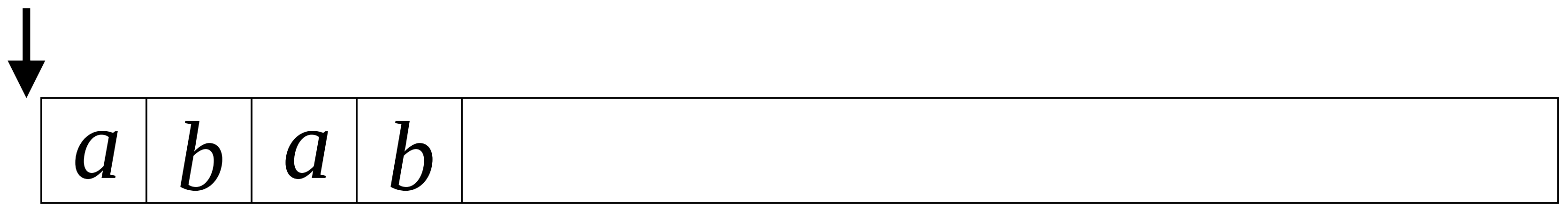
a
b

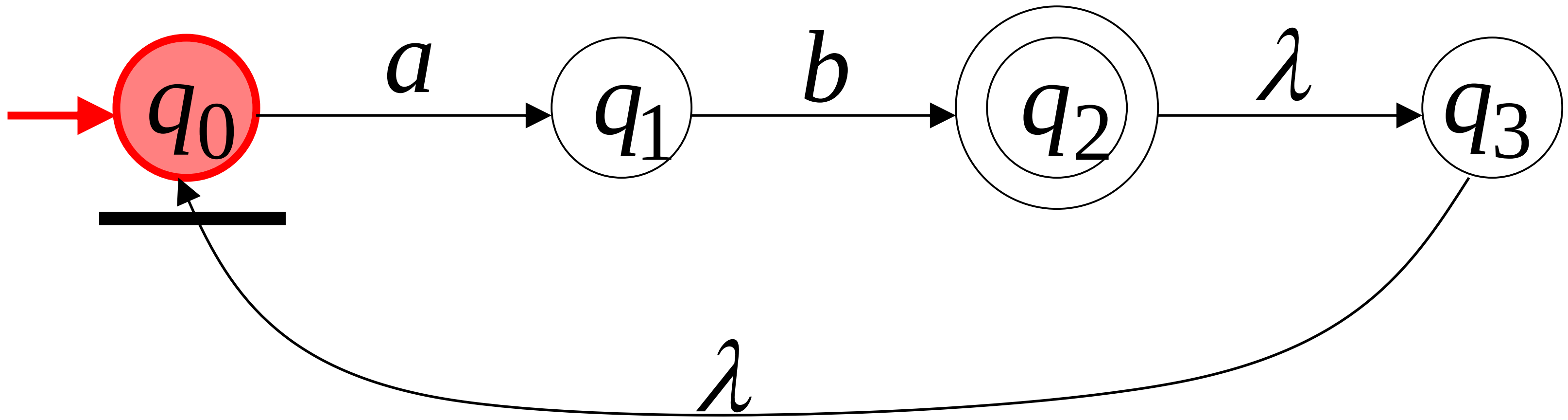


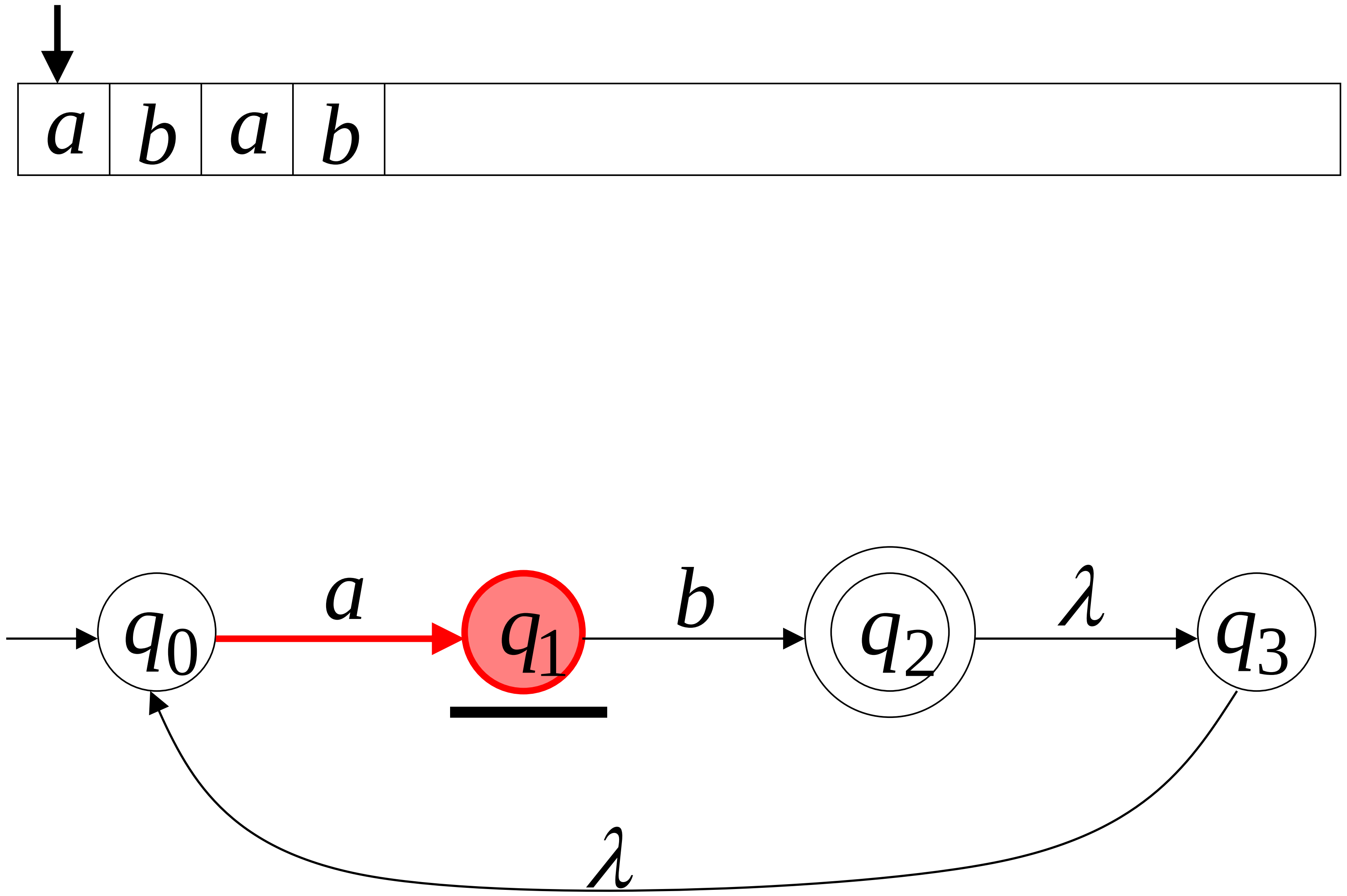
q_0
a
q_1
b
q_2
λ
q_3
λ

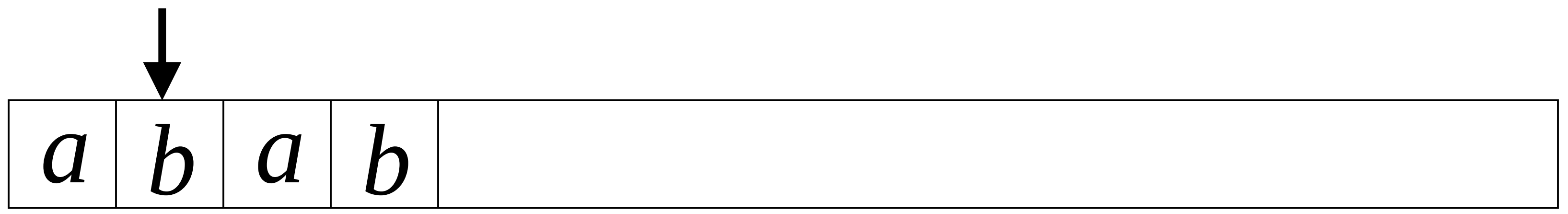

a
b
"accept"
q_0
q_1
q_2
q_3
a
b
λ
λ

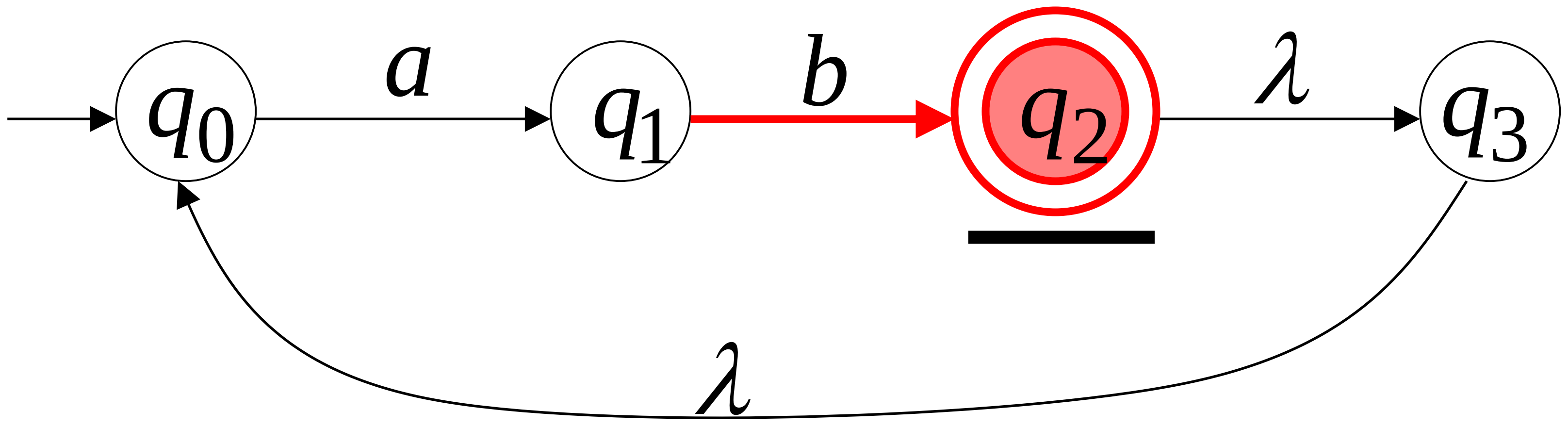
Another String

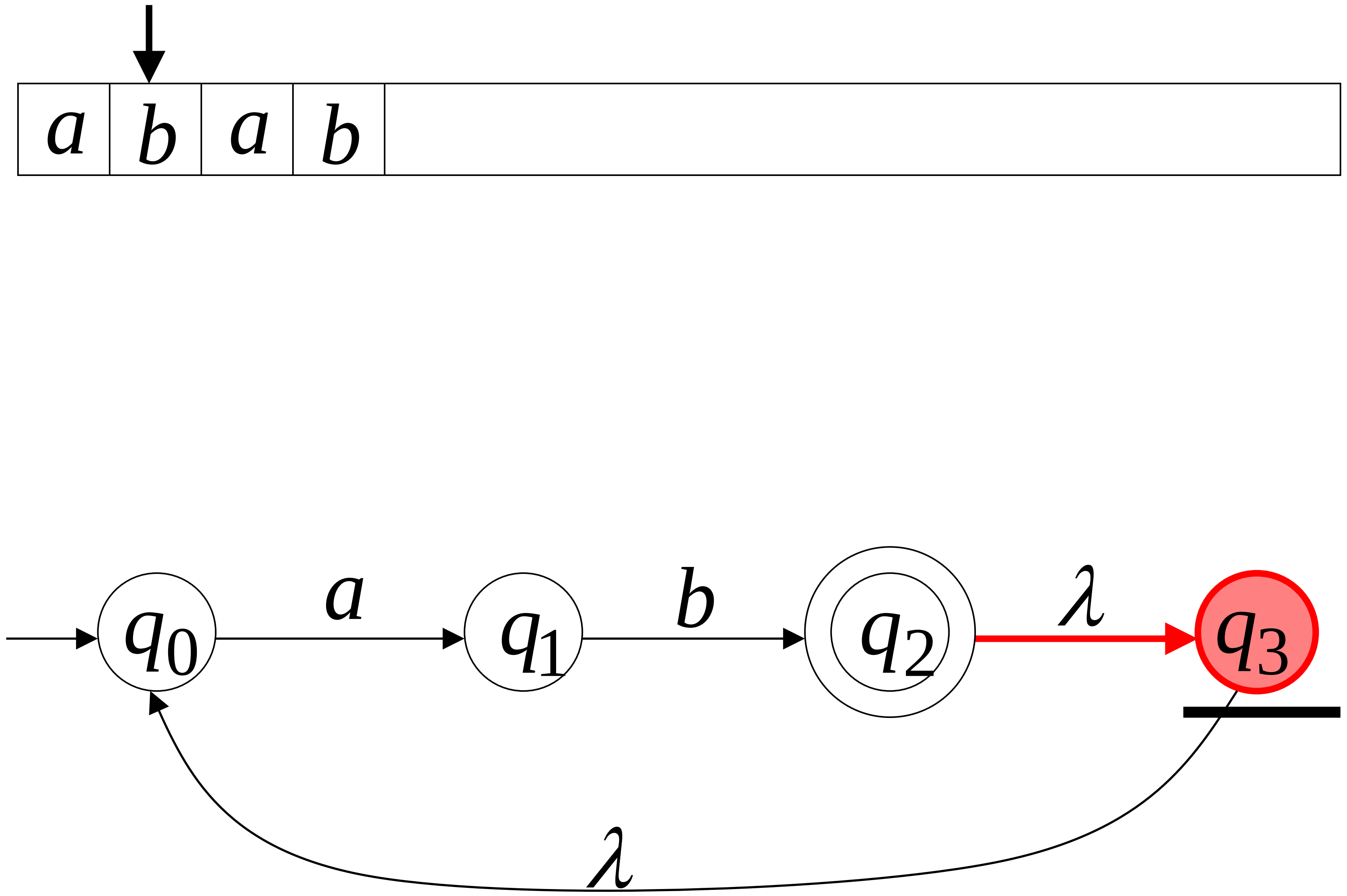


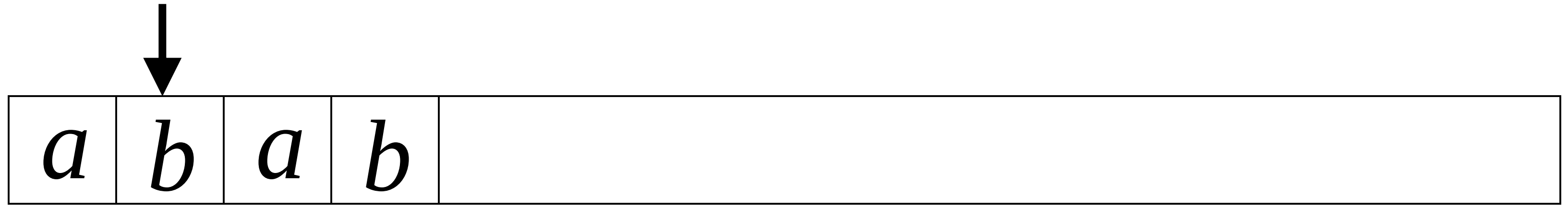


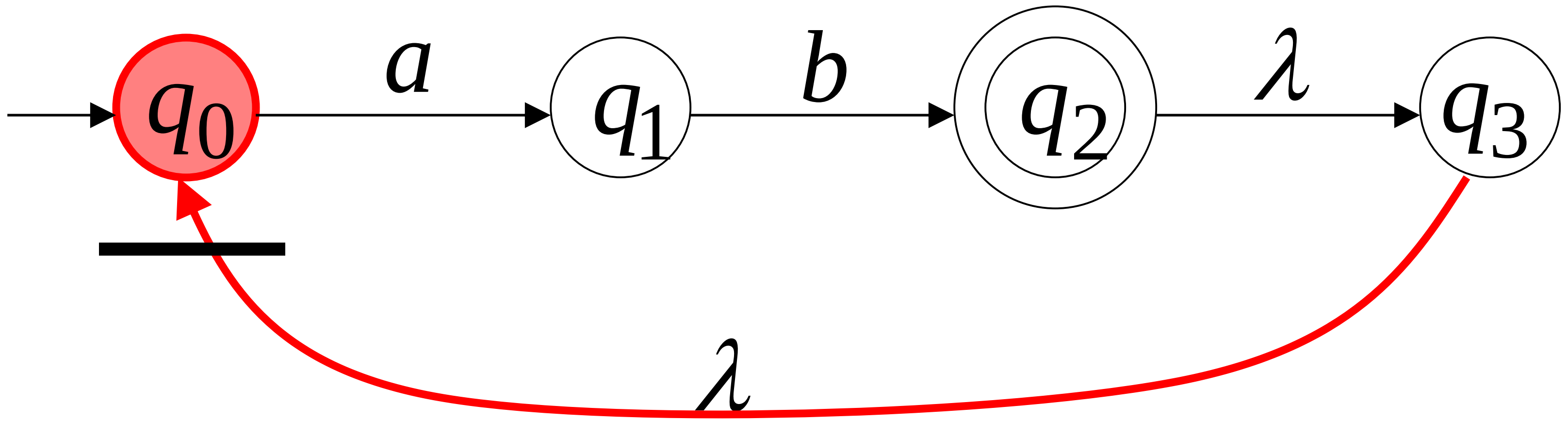

a
b
a
b
q_0
a
q_1
b
q_2
λ
q_3
λ

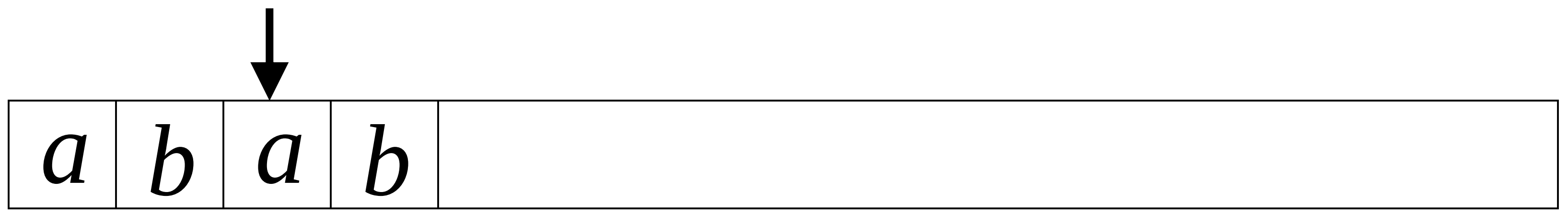

a
b
a
b

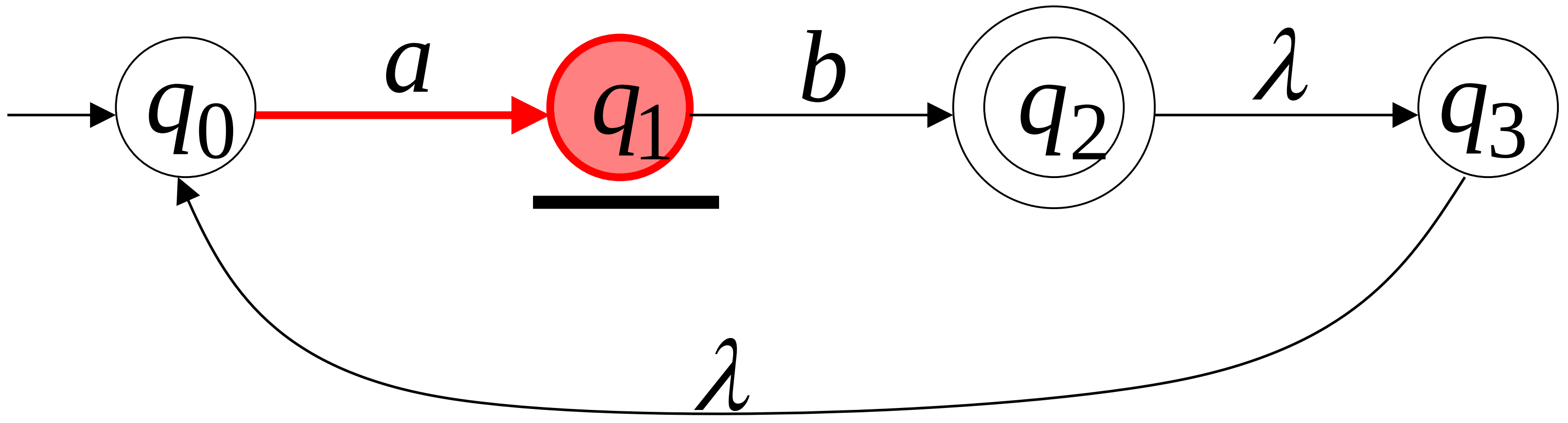

q_0
a
q_1
b
q_2
λ
q_3
λ

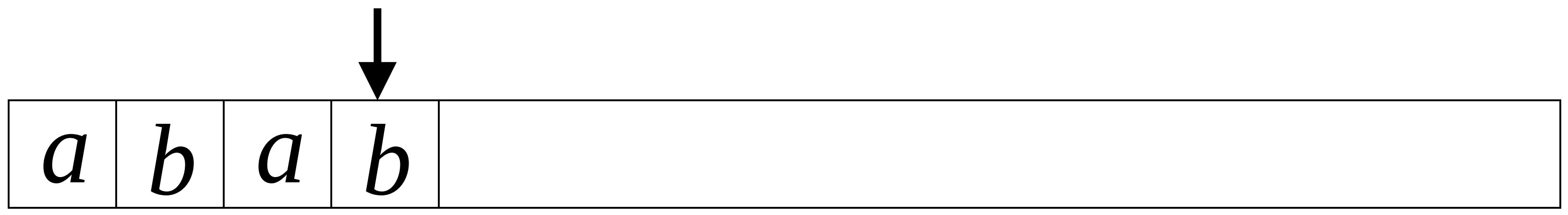

a
b
a
b
a
b
λ
λ
q_0
q_1
q_2
q_3

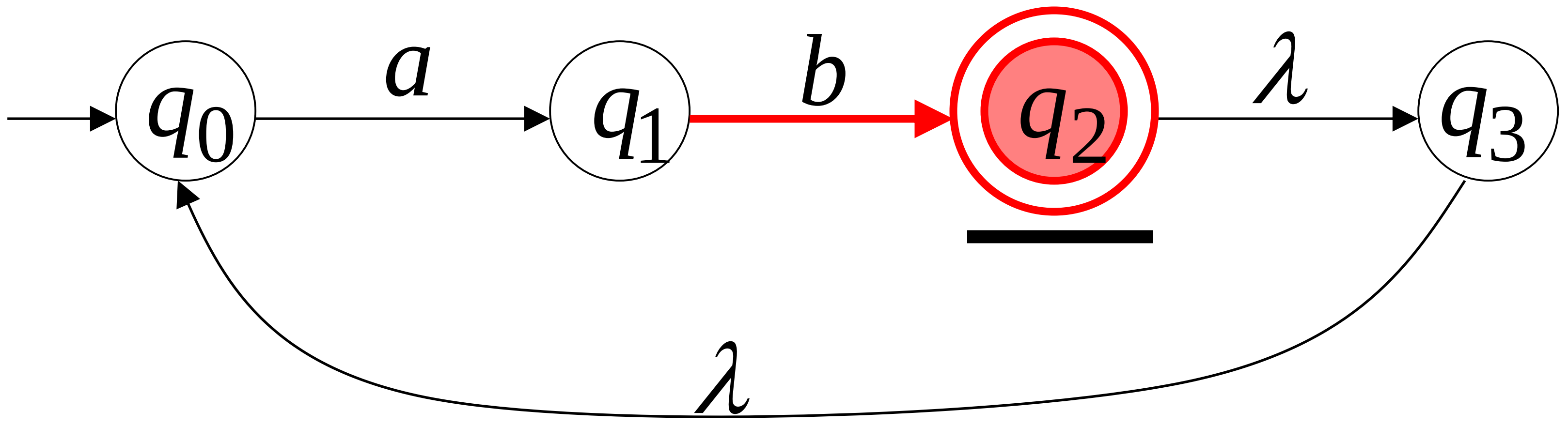

a b a b

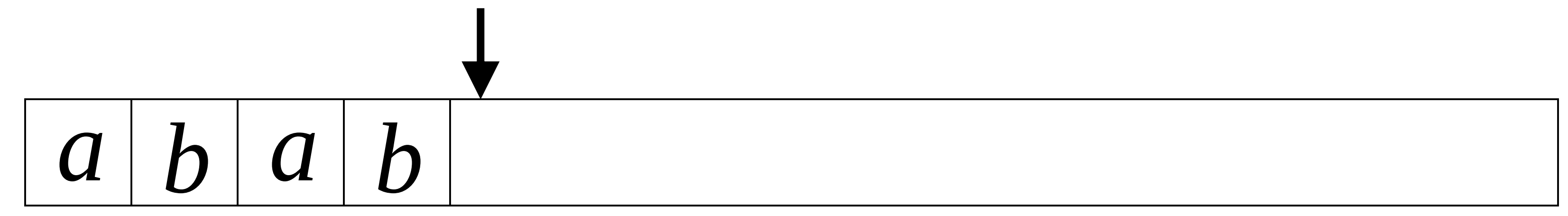

q_0
a
q_1
b
q_2
λ
q_3
λ


a
b
a
b

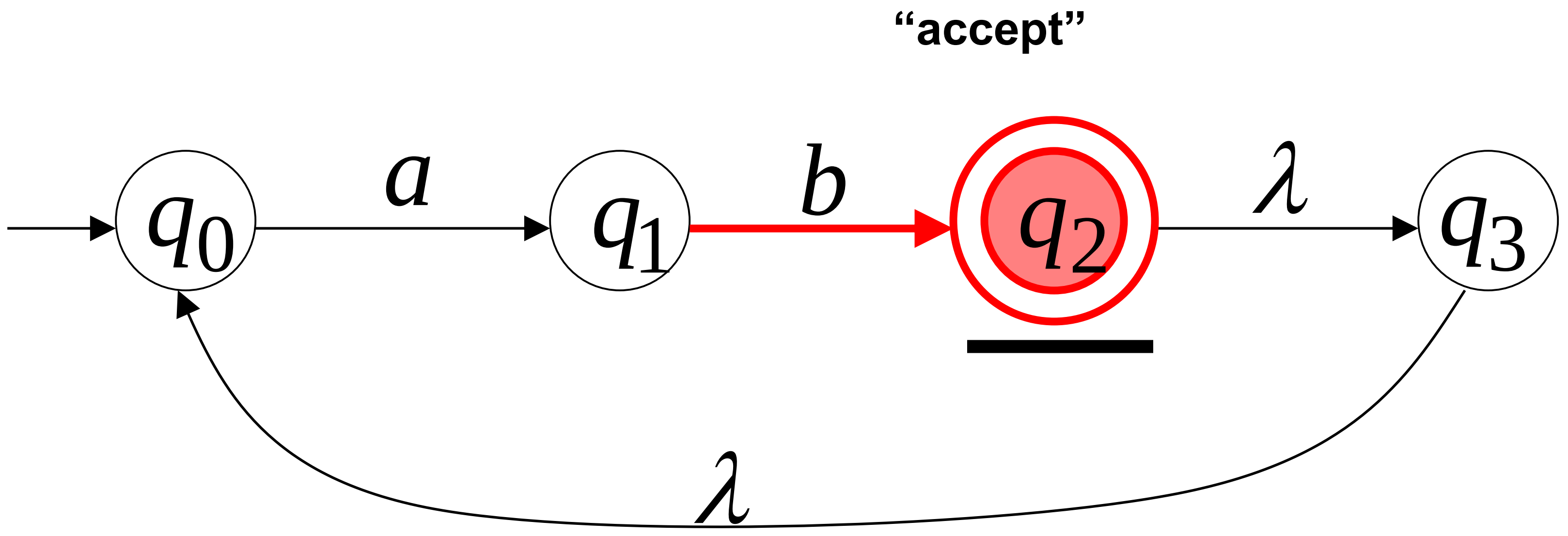

q_0
a
q_1
b
q_2
λ
q_3
λ


a
b
a
b


q_0
a
q_1
b
q_2
λ
q_3
λ



a
b
a
b

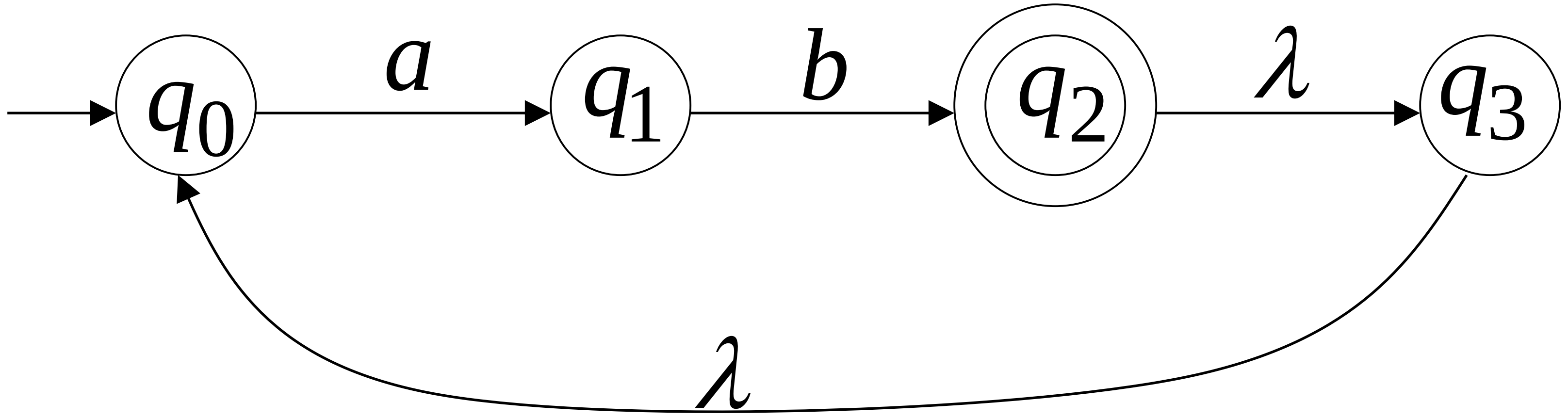


“accept”
q_0
a
q_1
b
q_2
λ
q_3
λ

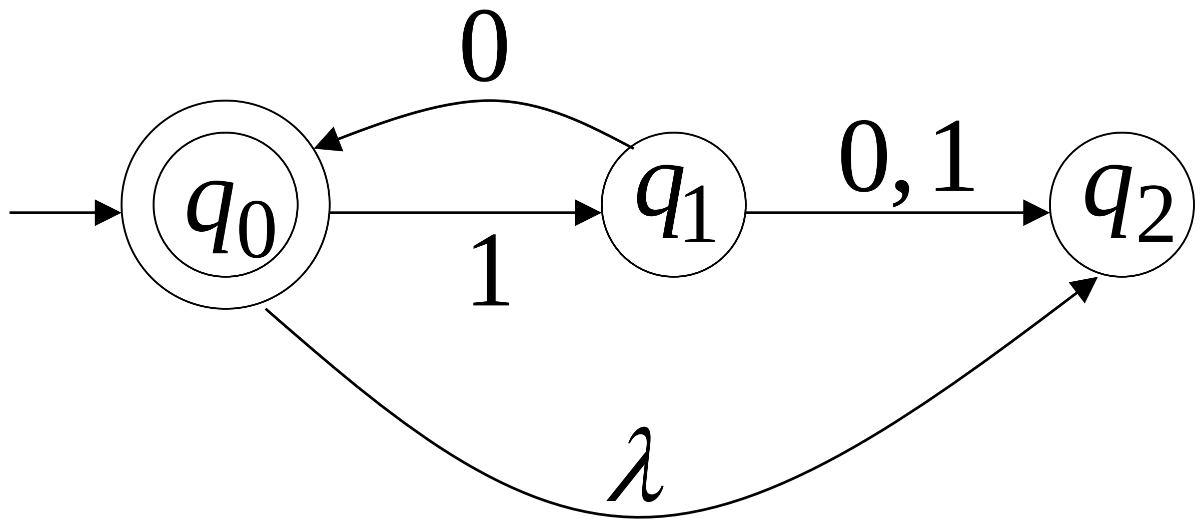
Language accepted

$$L = \{ab,\ abab,\ ababab,\ \ldots\}$$

$$= \{ab\}^{+}$$

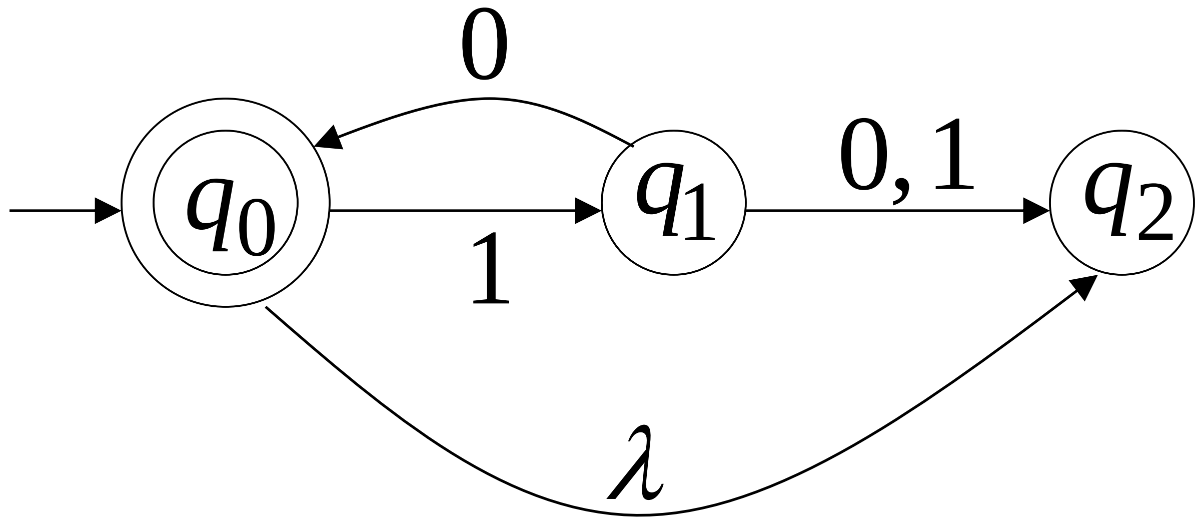

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "a"] & q_1 \arrow[r, "b"] & \boxed{q_2} \arrow[r, "\lambda"] & q_3 \arrow[lll, bend left, "\lambda"]
\end{tikzcd}$$

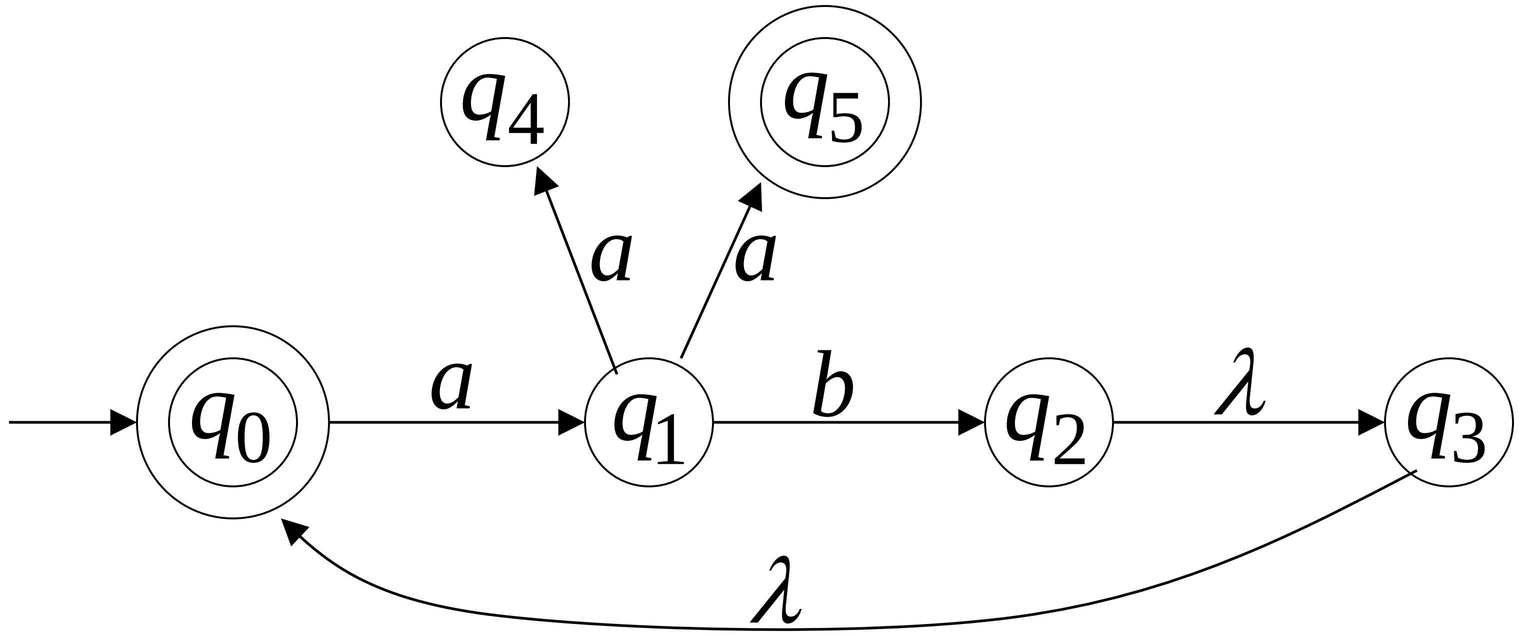
Another NFA Example



Language accepted

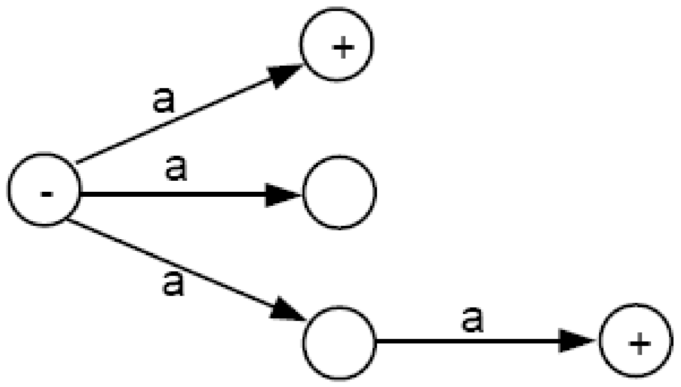
$$L = \{\lambda,\ 10,\ 1010,\ 101010,\ \ldots\}$$
$$= \{10\}^*$$

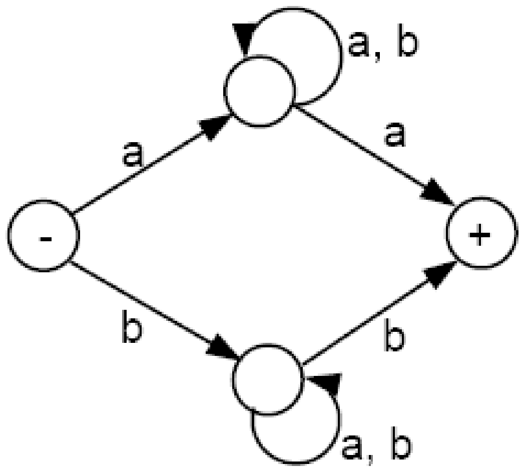

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "1"'] \arrow[rr, bend right, "\lambda"'] & q_1 \arrow[l, bend right, "0"'] \arrow[r, "{0,1}"] & q_2
\end{tikzcd}$$

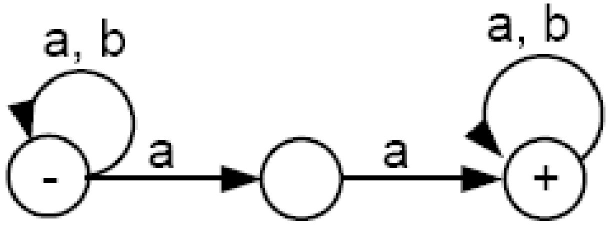


$$L(M) = \{aa\} \cup \{ab\}^* \cup \{ab\}^+\{aa\}$$

Examples of NFAs







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.

Distinguished Features

FA/DFA

1. One start state ONLY & Zero or more Final States

NFA

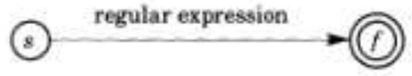
1. One start state ONLY & Zero or More Final States
2. Null transitions
3. Non-determinism

TG

1. One or More Start States
2. Zero or More Final States
3. Multiple letters on the edges
4. Null transitions
5. Non-determinism

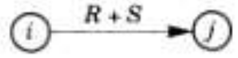
Regular Expression to Finite Automaton (5.4)

Given a regular expression, we start the algorithm with a machine that has a start state, a single final state, and an edge labeled with the given regular expression as follows:

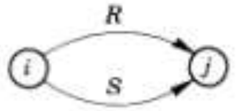


Now transform this machine into a DFA or an NFA by applying the following rules until all edges are labeled with either a letter or Λ:

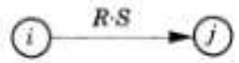
1. If an edge is labeled with ∅, then erase the edge.
2. Transform any diagram like



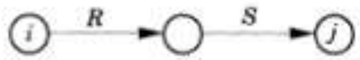
   into the diagram



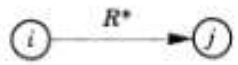
3. Transform any diagram like



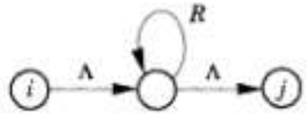
   into the diagram



4. Transform any diagram like

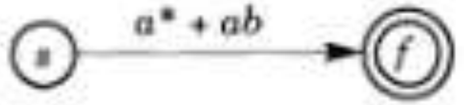


   into the diagram

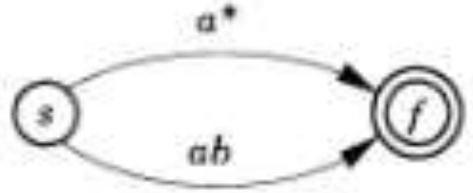


End of Algorithm

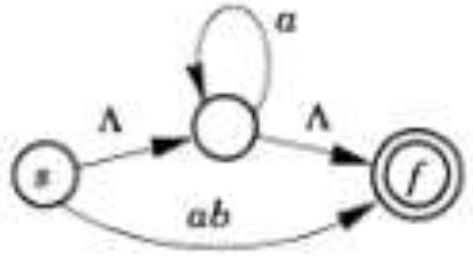
EXAMPLE 4. To construct an NFA for $a^* + ab$, we'll start with the diagram



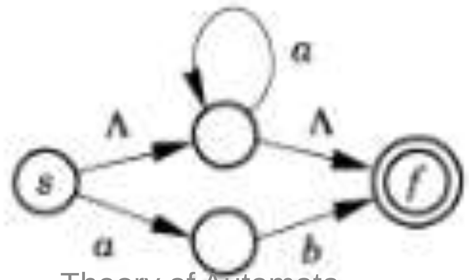
Next we apply rule 2 to obtain the following NFA:



Next we'll apply rule 4 to a^* to obtain the following NFA:

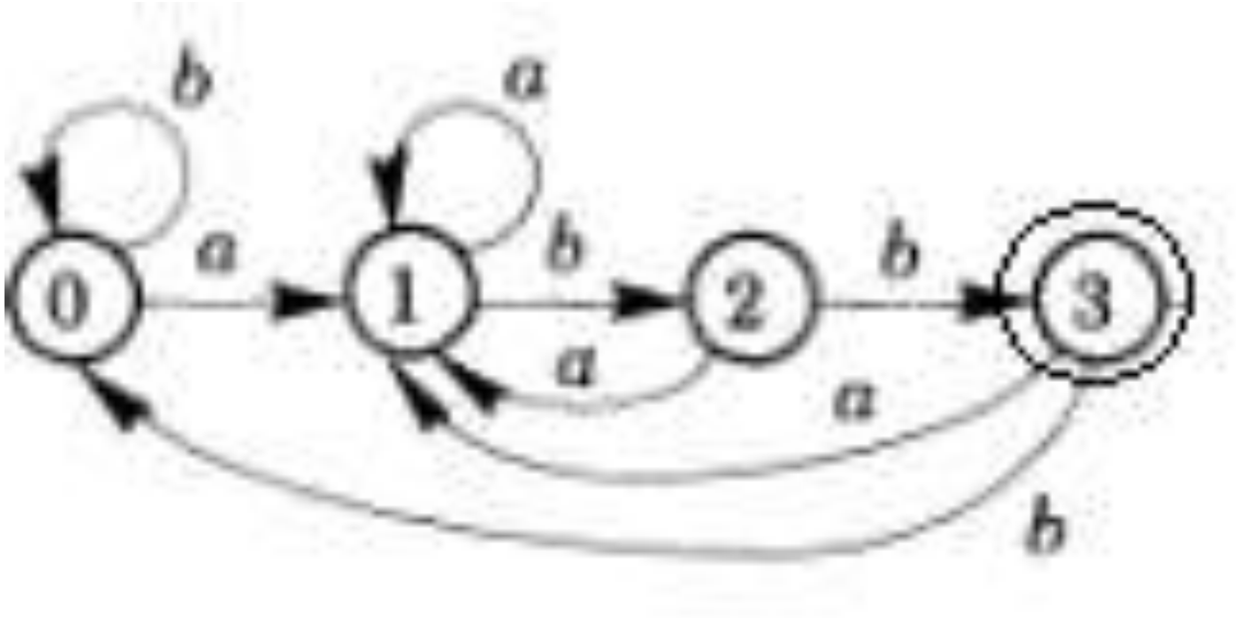


Finally, we apply rule 3 to ab to obtain the desired NFA for $a^* + ab$:

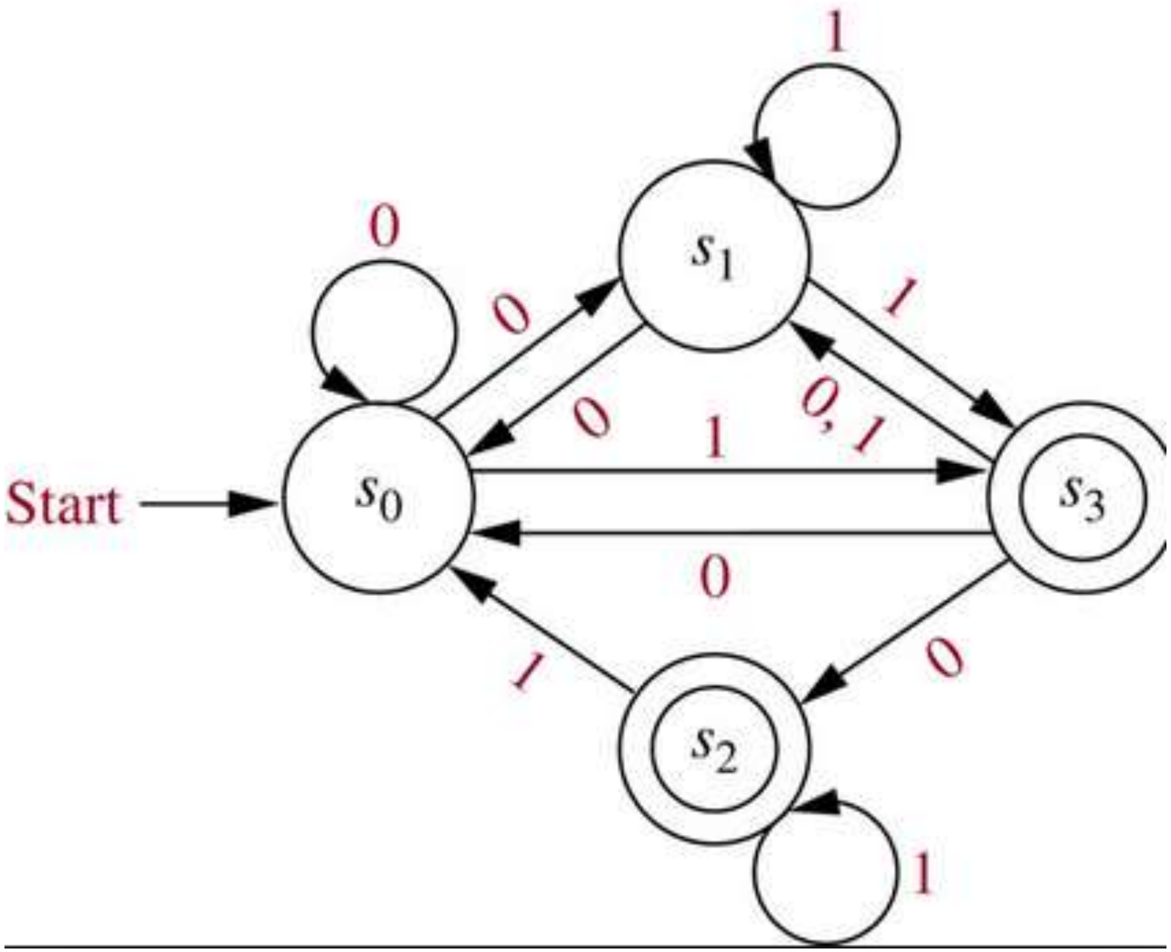


♦

Show that the following DFA is equivalent to R.E (a+b)*abb

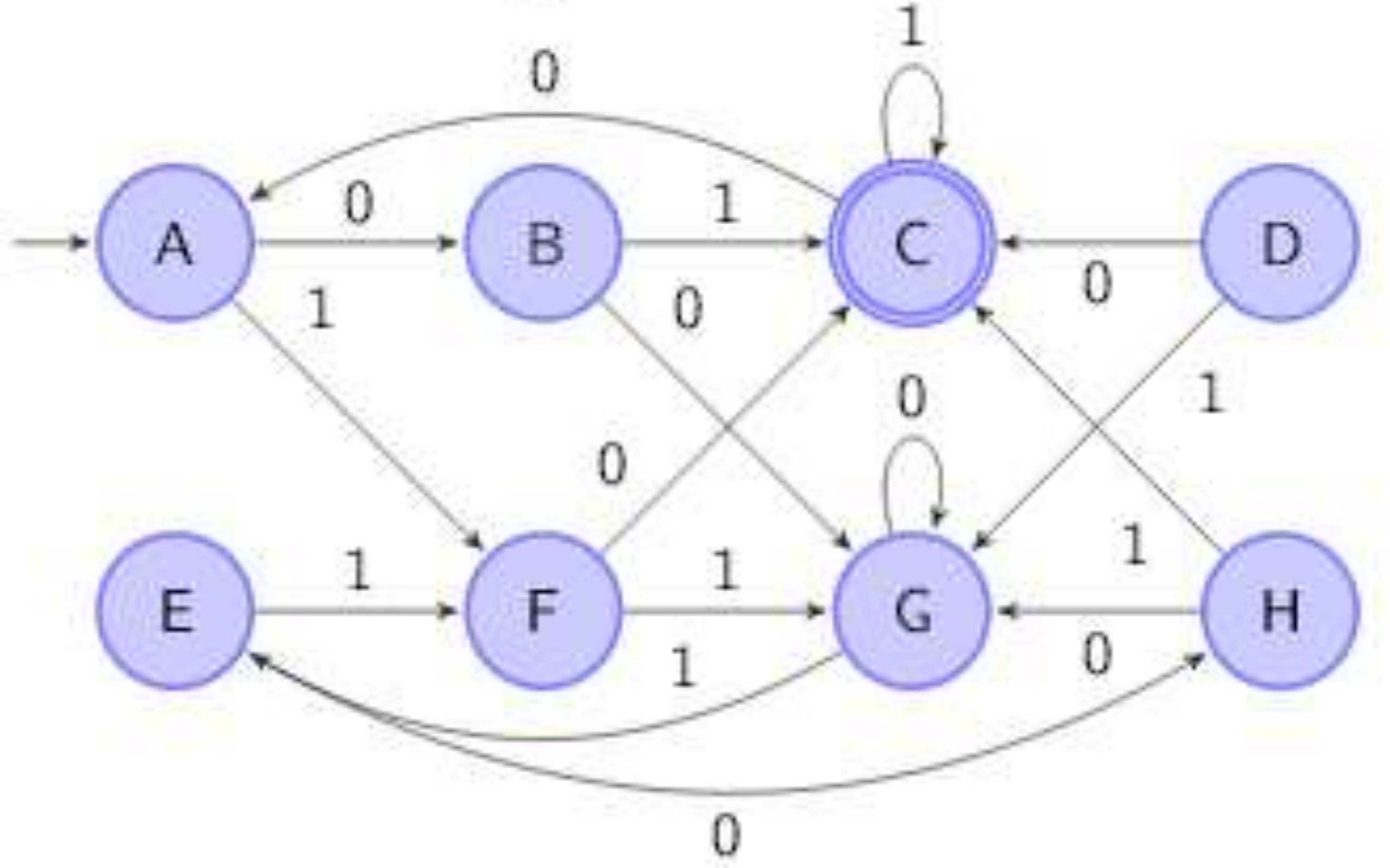


NFA to DFA Conversion

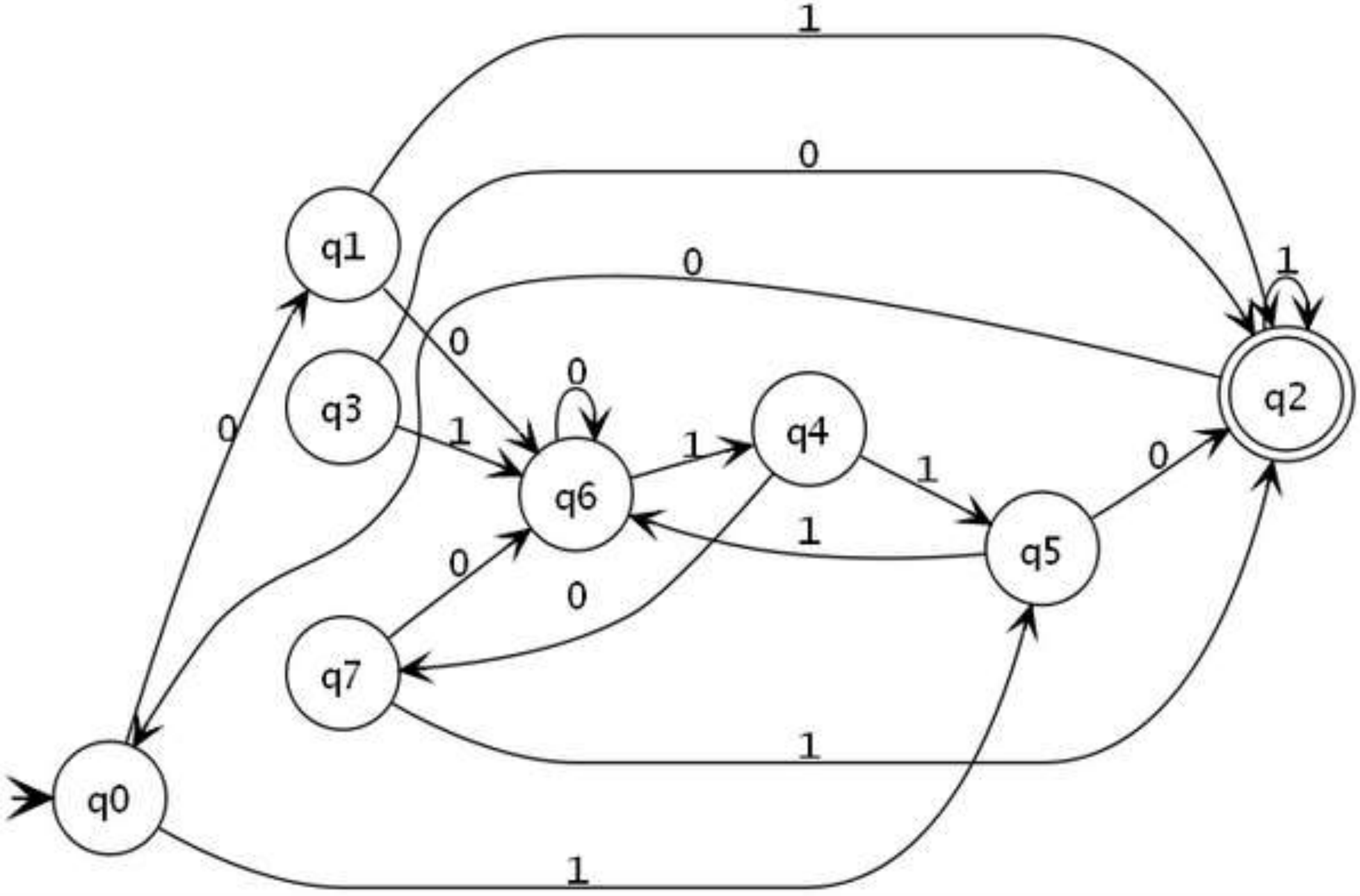


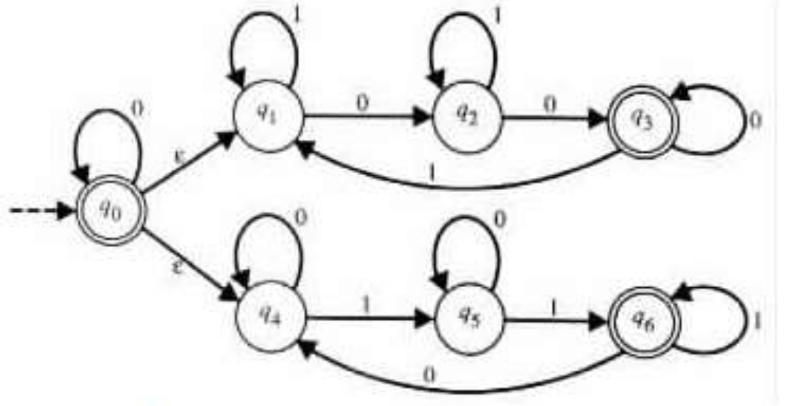
DFA Minimization

Minimize the following DFA:



DFA Minimization





automaton.

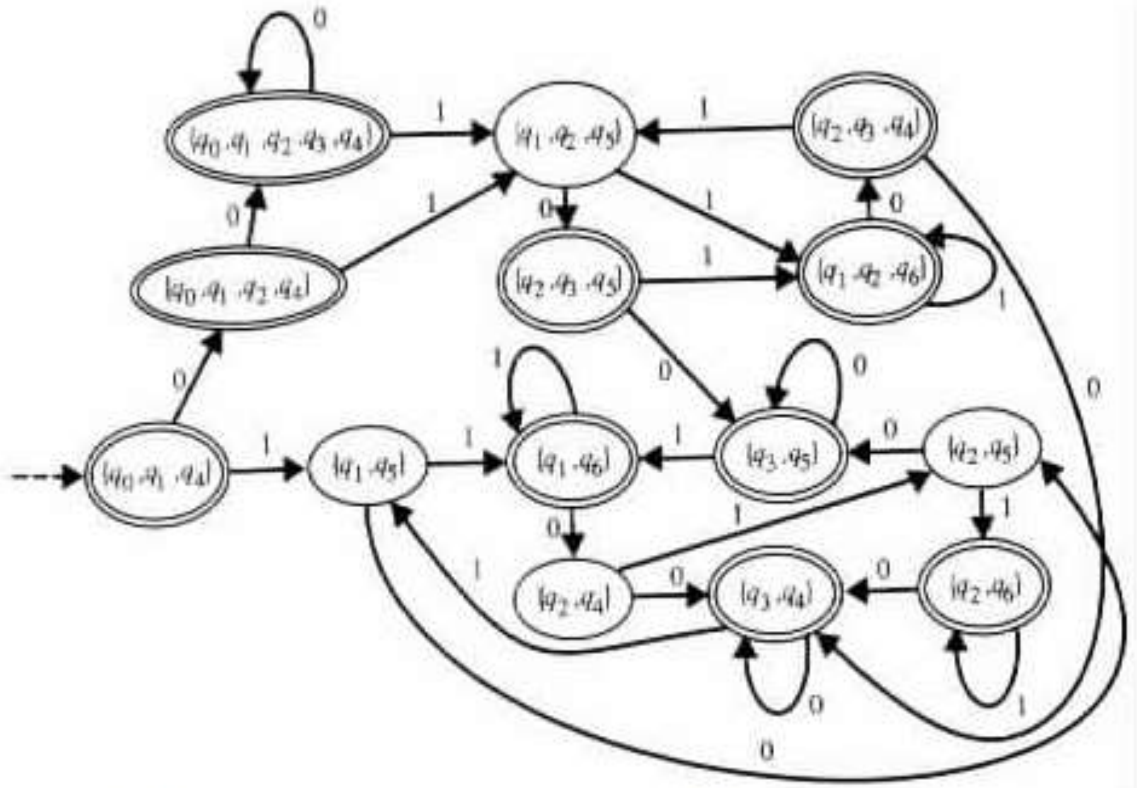


Figure 2.3.5 A transition diagram of an ϵ-free, deterministic finite-state automaton that is equivalent