Theory of Automata
Kleene’s Theorem

Week-06-Lecture-01

Hafiz Tayyeb Javed

Contents

- Converting Regular Expressions into FAs
- Nondeterministic Finite Automata
- NFAs and Kleene’s Theorem

Converting Regular Expressions into FAs

Proof of Part 3: Converting Regular Expressions into FAs

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

  – We know that every regular expression can be built up from the letters of the alphabet ∑ and Λ by repeated application of certain rules: (i) addition, (ii) concatenation, and (iii) closure.

  – We will show that as we are building up a regular expression, we could at the same time building up an FA that accepts the same language.

- Slides 3 - 30 below show the proof of part 3.

- Before we proceed, let's have a quick review of the **formal definition of regular expressions**.
- The set of **regular expressions** is defined by the following rules:
  - Rule 1: Every letter of the alphabet ∑ can be made into a regular expression by writing it in **boldface: Λ** itself is a regular expression.
  - Rule 2: If r_1 and r_2 are regular expressions, then so are:
    - (r_1)
    - r_1r_2
    - $r_1 + r_2$
    - r_1*
  - Rule 3: Nothing else is a regular expression.

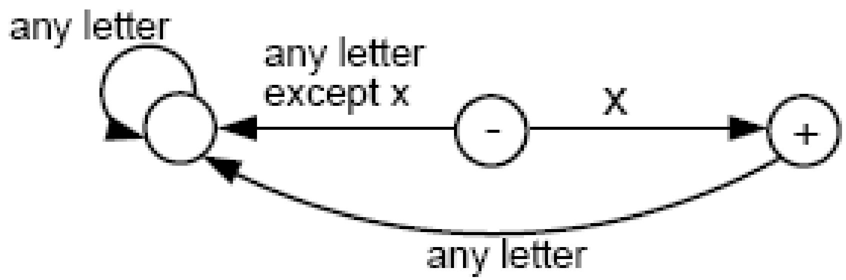
We now present proof of part 3 recursively.

Rule 1

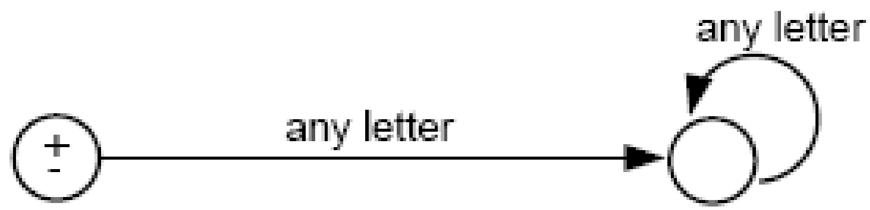
- There is an FA that accepts any particular letter of the alphabet.
- There is an FA that accepts only the word Λ.

Proof of rule 1

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



- The following FA accepts only **λ**:



Rule 2

- If there is an FA called FA_1 that accepts the language defined by the regular expression r_1, and there is an FA called FA_2 that accepts the language defined by the regular expression r_2, then there is an FA that we shall call FA_3 that accepts the language defined by the regular expression $(r_1 + r_2)$.

Proof of Rule 2

- We shall show that FA_3 exists by presenting an algorithm showing how to construct FA_3.
- **Algorithm:**
  - Starting with two machines, FA_1 with states x_1; x_2; x_3;…, and FA_2 with states y_1; y_2; y_3; …, we construct a new machine FA_3 with states z_1; z_2; z_3; … where each z_i is of the form $x_{something}$ or $y_{something}$.
  - The combination state x_{start} or y_{start} is the start state of the new machine FA_3.
  - If either the *x* part or the *y* part is a final state, then the corresponding *z* is a final state.

Algorithm (cont.)

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

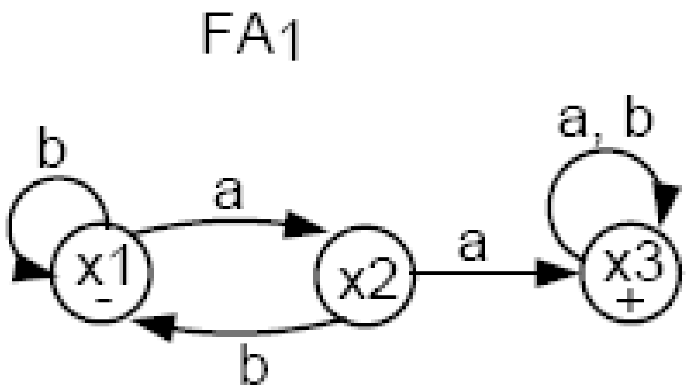
z_{new} after reading letter p = (x_{new} after reading letter p on FA_1) or (y_{new} after reading letter p on FA_2)

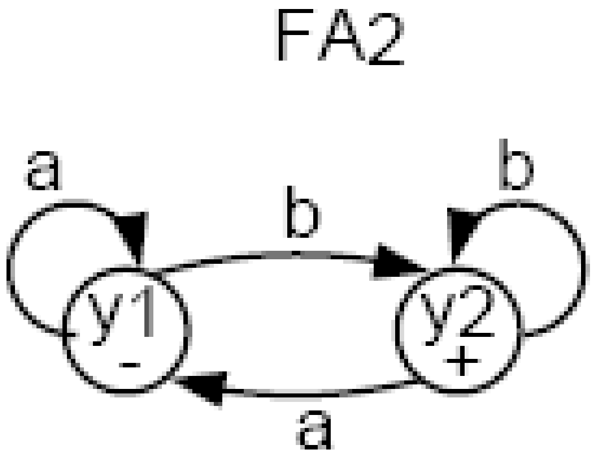
Remarks

- The new machine FA_3 constructed by the above algorithm will simultaneously keep track of where the input would be if it were running on FA_1 alone, and where the input would be if it were running on FA_2 alone.
- If a string traces through the new machine FA_3 and ends up at a final state, it means that it would also end at a final state either on machine FA_1 or on machine FA_2. Also, any string accepted by either FA_1 or FA_2 will be accepted by this FA_3. So, the language FA_3 accepts is the **union** of the languages accepted by FA_1 and FA_2, respectively.
- Note that since there are only finitely many states *x's* and finitely many states *y's*, there can be only finitely many possible states *z's*.
- Let us look at an example illustrating how the algorithm works.

Example

- Consider the following two FAs:





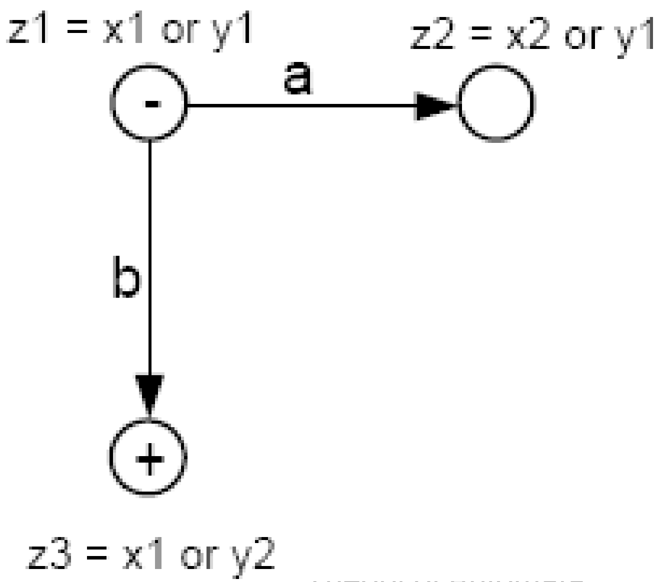
- FA_1 accepts all words with a double a in them.
- FA_2 accepts all words ending with b.
- Let's follow the algorithm to build FA_3 that accepts the union of the two languages.

Combining the FAs

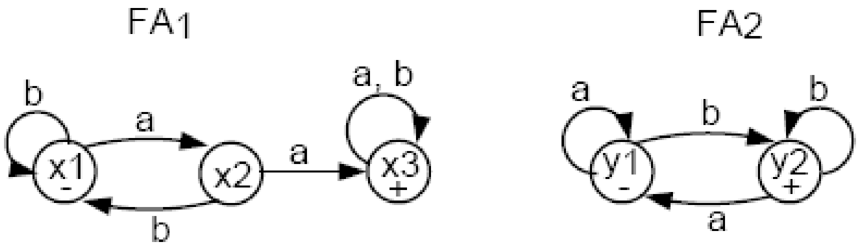
- The start (-) state of FA_3 is $z_1 = x_1$ or y_1.
- In z_1, if we read an ***a***, we go to x_2 (observing FA_1), or we go to y_1 (observing FA_2).

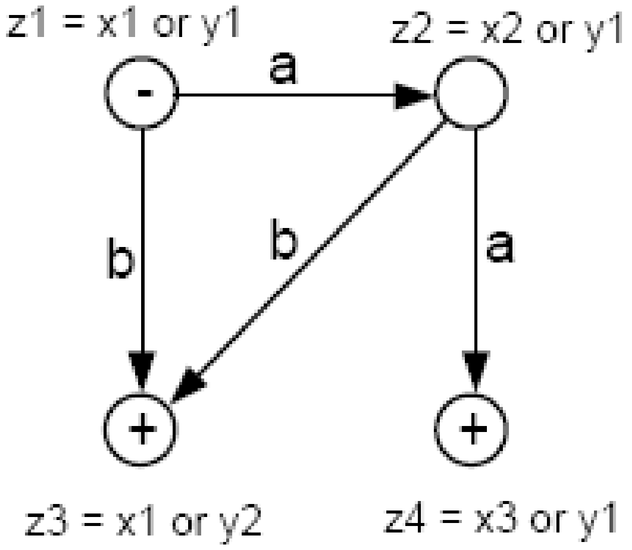
  Let $z_2 = x_2$ or y_1.
- In z_1, if we read a ***b***, we go to x_1 (observing FA_1), or to y_2 (observing FA_2).

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

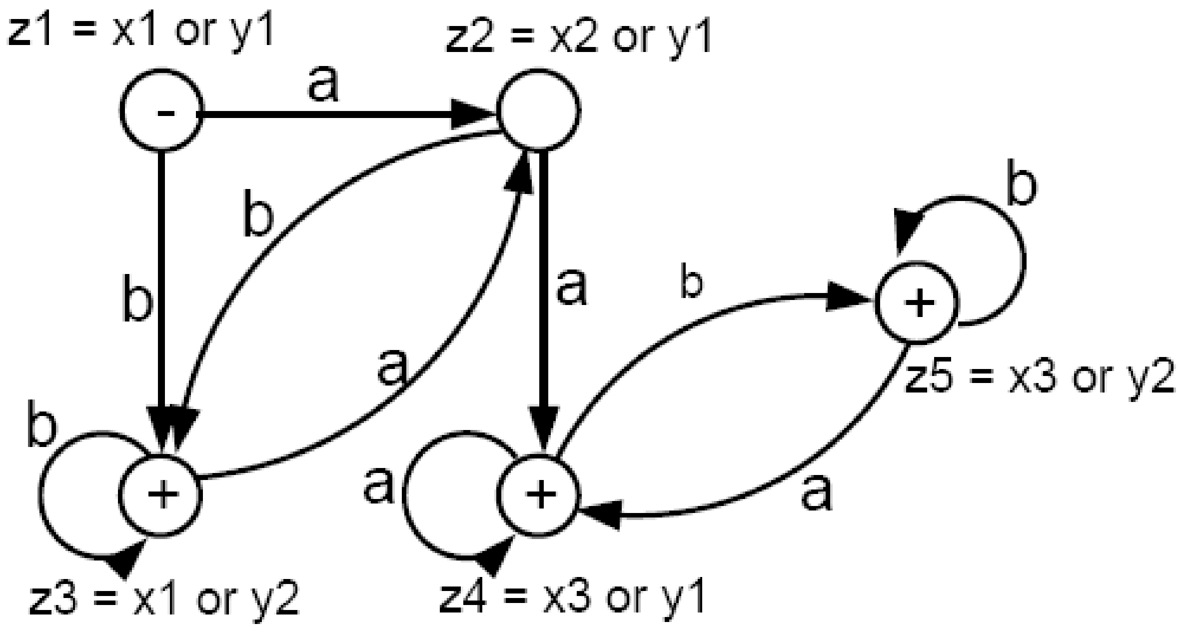


- In z_2, if we read an *a*, we go to x_3 or y_1. Let $z_4 = x_3$ or y_1. z_4 is a final state because x_3 is.
- In z_2, if we read a *b*, we go to x_1 or y_2, which is z_3.





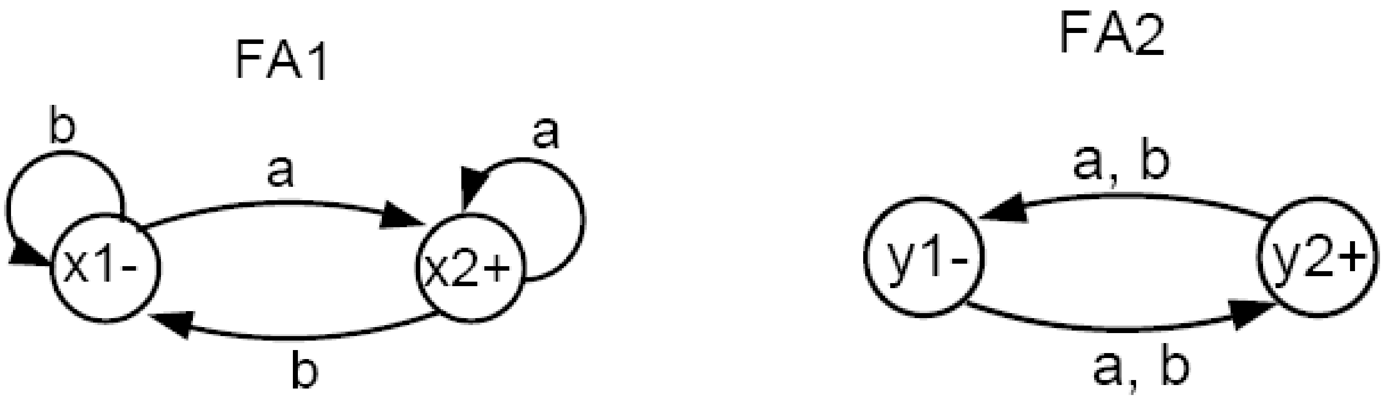
- In z_3, if we read an *a*, we go to x_2 or y_1, which is z_2.
- In z_3, if we read a *b*, we go to x_1 or y_2, which is z_3.
- In z_4, if we read an *a*, we go to x_3 or y_1, which is z_4. Hence, we have an a-loop at z_4.
- In z_4, if we read a *b*, we go to x_3 or y_2. Let $z_5 = x_3$ or y_2. Note that z_5 is a final state because x_3 (and y_2) are.
- In z_5, if we read an *a*, we go to x_3 or y_1, which is z_4.
- In z_5, if we read a *b*, we go to x_3 or y_2, which is z_5. Hence, there is a b-loop at z_5.
- The whole machine looks like the following:



- This machine accepts all words that have a double *a* or that end with *b*.
- The labels $z_1 = x_1$ or y_1, $z_2 = x_2$ or y_1, etc. can be removed if you want.

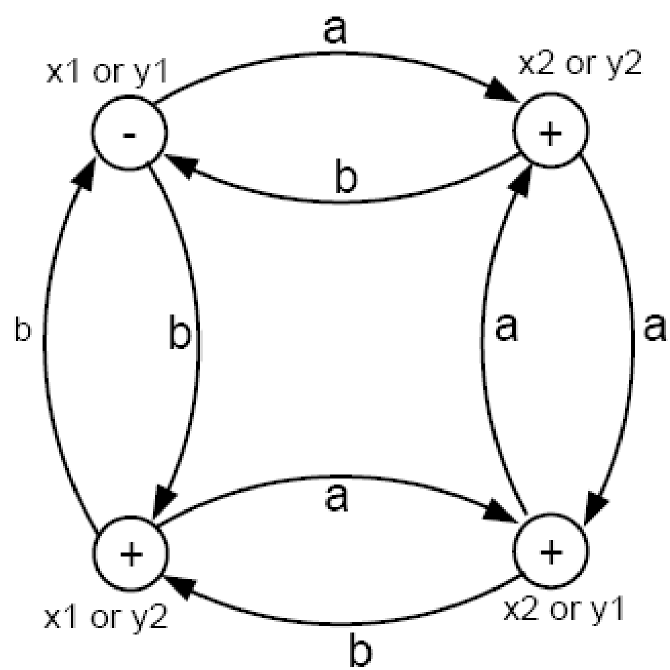
Example

- Consider the following two FAs:



- FA_1 accepts all words that end in *a*.
- FA_2 accepts all words with an odd number of letters (odd length).
- Can you use the algorithm to build a machine FA3 that accepts all words that either have an odd number of letters or end in *a*?

- Using the algorithm, we can produce *FA_3* that accepts all words that either have an odd number of letters or end in a, as follows:



- The only state that is not a + state is the - state. To get back to that start state, a word must have an even number of letters **and** end in *b*.

Rule 3

If there is an FA_1 that accepts the language defined by the regular expression r_1, and there is an FA_2 that accepts the language defined by the regular expression r_2, then there is an FA_3 that accepts the language defined by the (concatenation) regular expression (r_1r_2), i.e. the product language.

- We shall show that such an FA_3 exists by presenting an algorithm showing how to construct it from FA_1 and FA_2.

- The idea is to construct a machine that starts out like FA_1 and follows along it until it enters a final state at which time an option is reached. Either we continue along FA_1, waiting to reach another +, or else we switch over to the start state of FA_2 and begin circulating there.

Algorithm

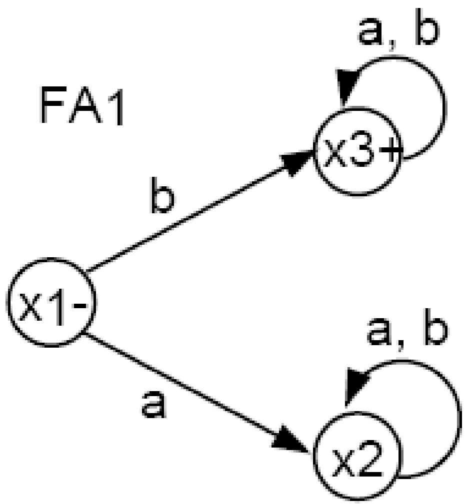
- First, create a state z for every state of FA_1 that we may go through before arriving at a final state.

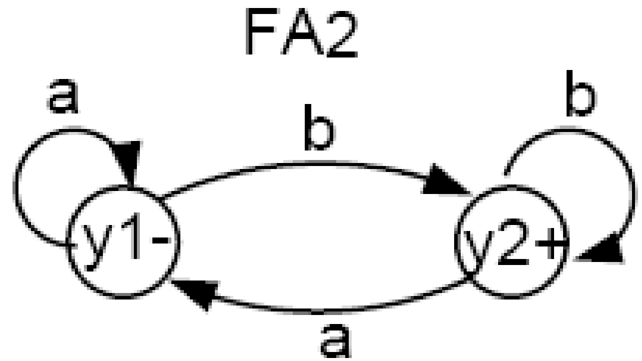
2. For each final state x_{final} of FA_1, add a state $z = x_{final}$ or y_1, where y_1 is the start state of FA_2.
3. From the states added in step 2, add states

$$z = \begin{cases} x_{something} \text{ indicating that we are still continuing on } FA_1 \\ \text{OR} \\ \text{a set of } y_{something} \text{ indicating that we are on } FA_2 \end{cases}$$

4. Label every state z that contains a final state from FA_2 as a final state.

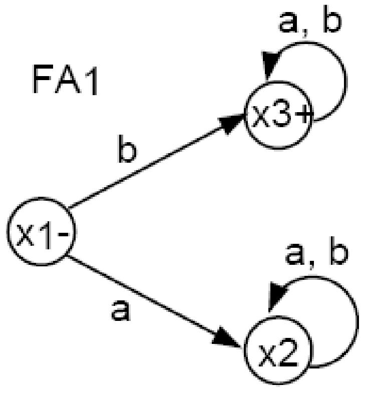
Example

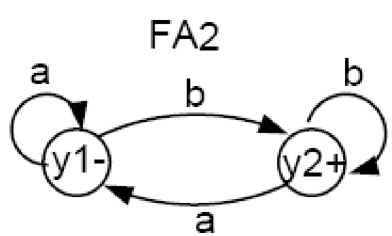


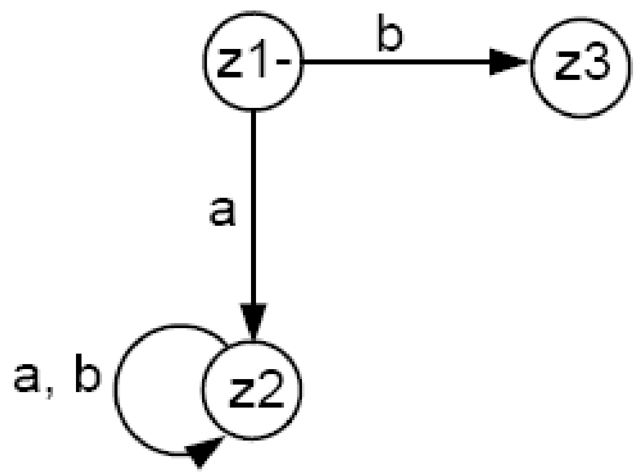


- *FA*$_1$ accepts all words that start with a *b*.
- *FA*$_2$ accepts all words that end with a *b*.
- We will use the above algorithm to construct *FA*$_3$ that accepts the product of the languages of *FA*$_1$ and *FA*$_2$, respectively. That is, *FA*$_3$ will accept all words that both start and end with the letter *b*.

- Initially, we must begin with $x_1 = z_1$.
- In z_1, if we read an *a*, we go to $x_2 = z_2$.
- In z_1, if we read a *b*, we go to x_3, a final state, which gives us the option to jump to y_1. Hence, we label $z_3 = x_3$ or y_1.
- From z_2 just like x_2, both an *a* or a *b* take us back to z_2, i.e., we have a loop here.







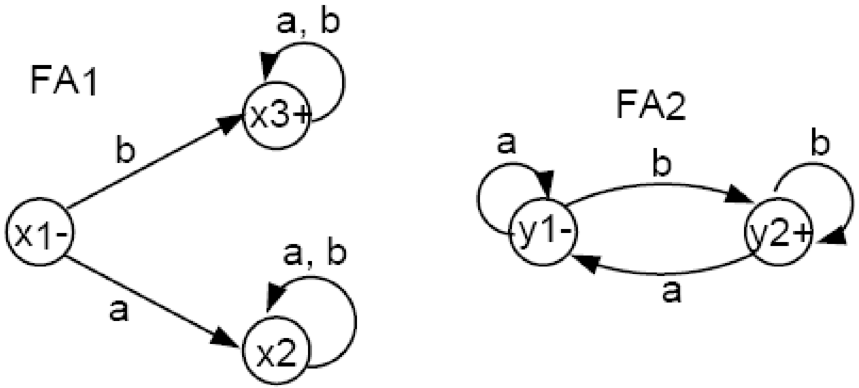
- In z3, if we read an a then the following happens. If z3 is x3, we can stay in x3 or jump to y1 (because x3 is a final state). If z3 is y1, we would loop back to y1. In any of the events, we end up at either x3 or y1, which is still z3. Hence, we have an a-loop at z3.

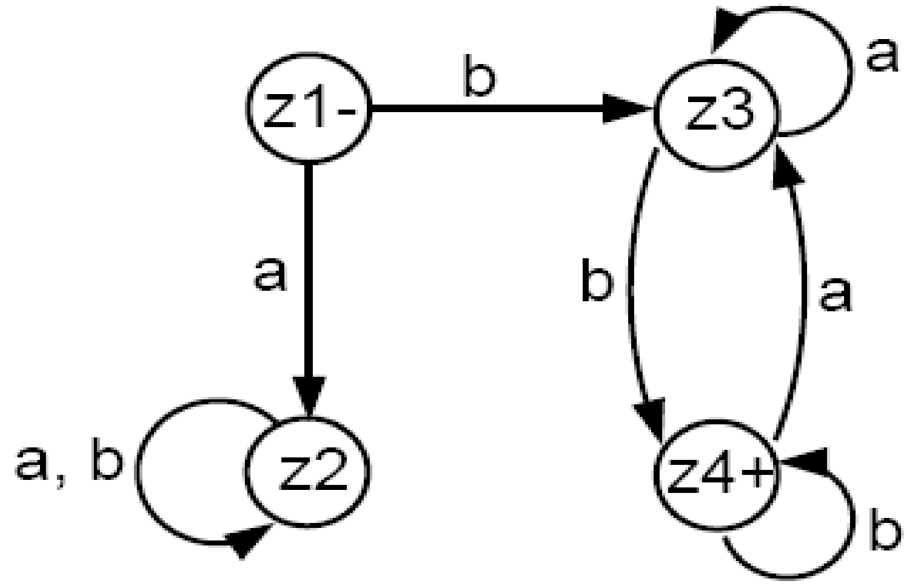
- In z3, if we read a b, then a different event takes place. If z3 is x3 we either stay in x3 or jump to y1. If z3 is y1, then we go to y2, a final state. Hence, we need a new final state z4 = x3 or y1 or y2.

- In z4, if we read an a, what happens? If z4 is x3 then we go back to x3 or jump to y1. If z4 is y1 then we loop back to y1. If z4 is y2, we go to y1. Thus, from z4, an a takes us to x3 or y1, which is z3.

- In z4, if we read a b, what happens? If z4 is x3, we go back to x3 or jump to y1. If z4 is y1, we go to y2, a final state. If z4 is y2, we loop back to y2, a final state. Hence, from z4 a b takes us to x3 or y1 or y2, which is still z4 (i.e., we have a b-loop here).

xW





- This machine accepts all words that both begin and end with the letter b, which is what the product of the two languages (defined by FA_1 and FA_2 respectively) would be.

- If you multiply the two languages in opposite order (i.e. first FA_2 then FA_1), then the product language will be different. What is that language? Can you build a machine for that product language

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

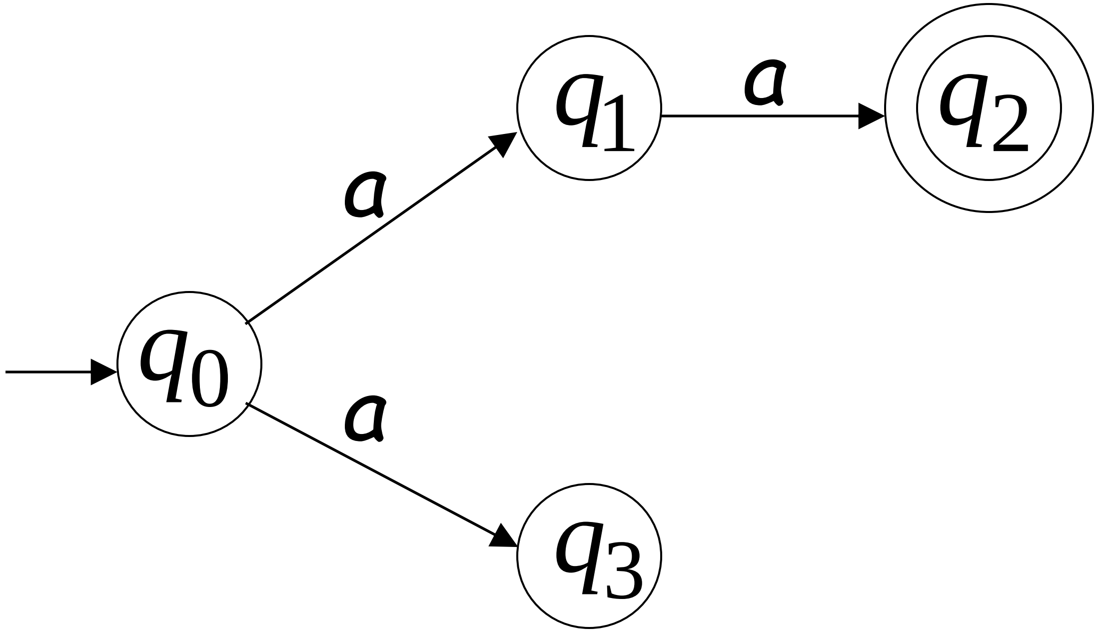
NFA

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)

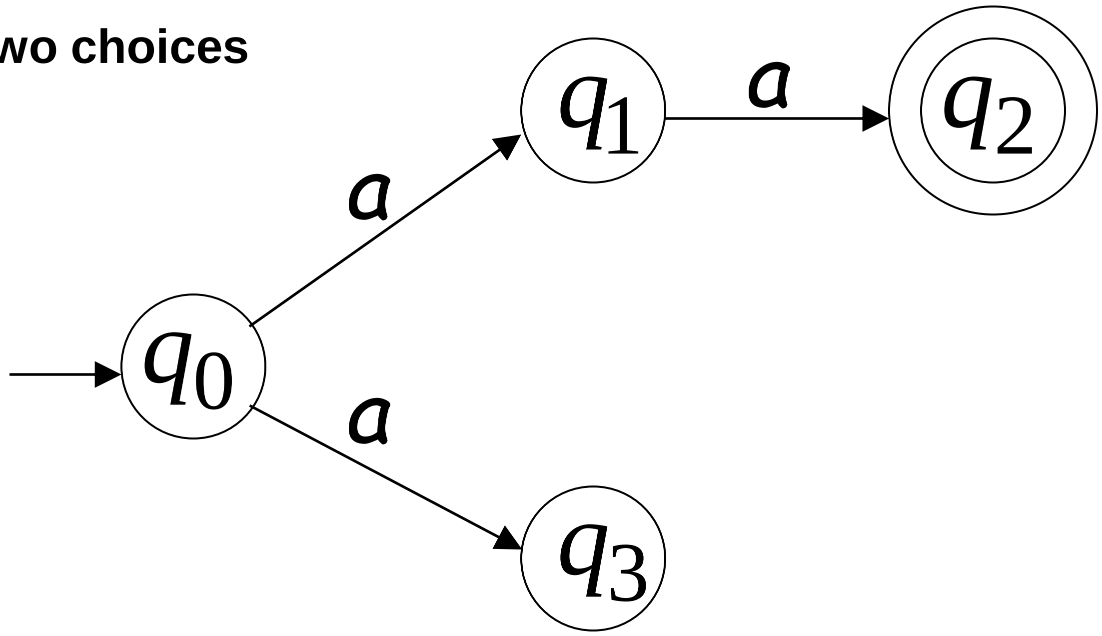
Alphabet = $\{a\}$



Nondeterministic Finite Accepter (NFA)

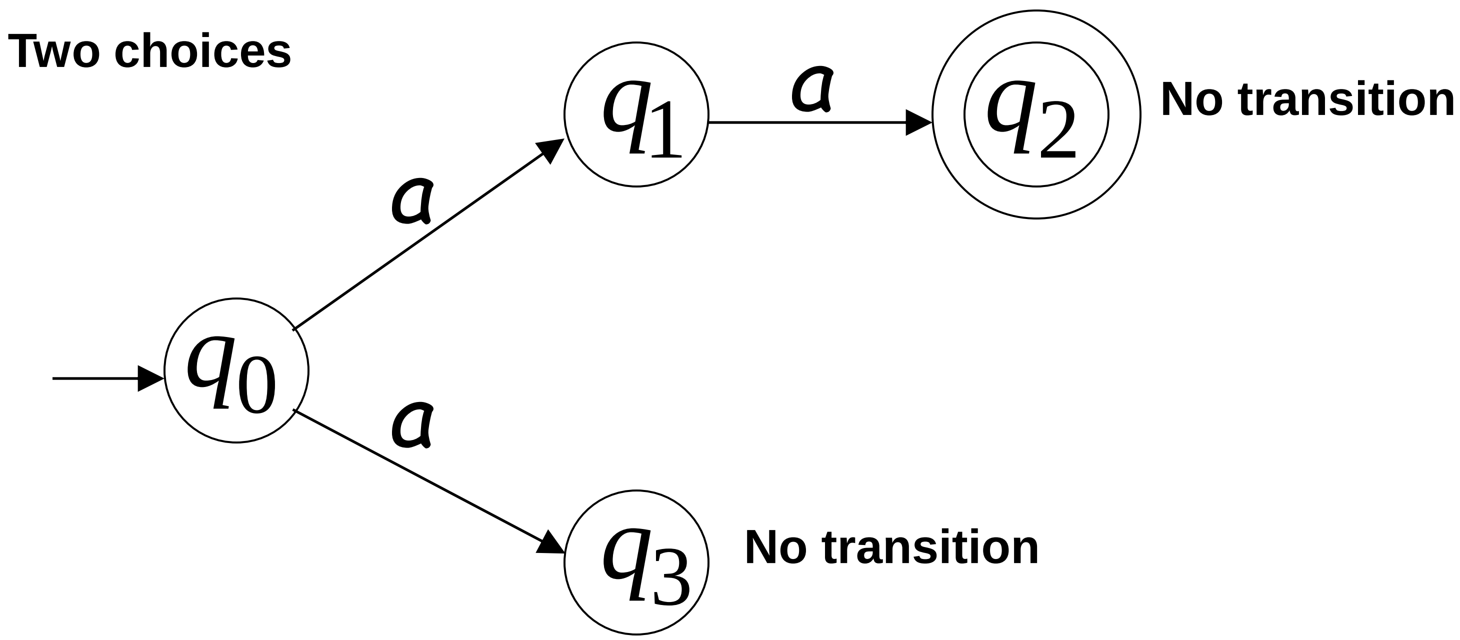
Alphabet = $\{a\}$

Two choices

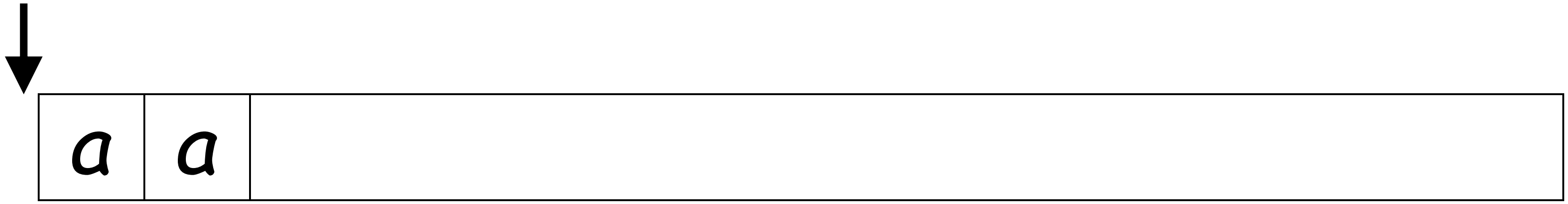


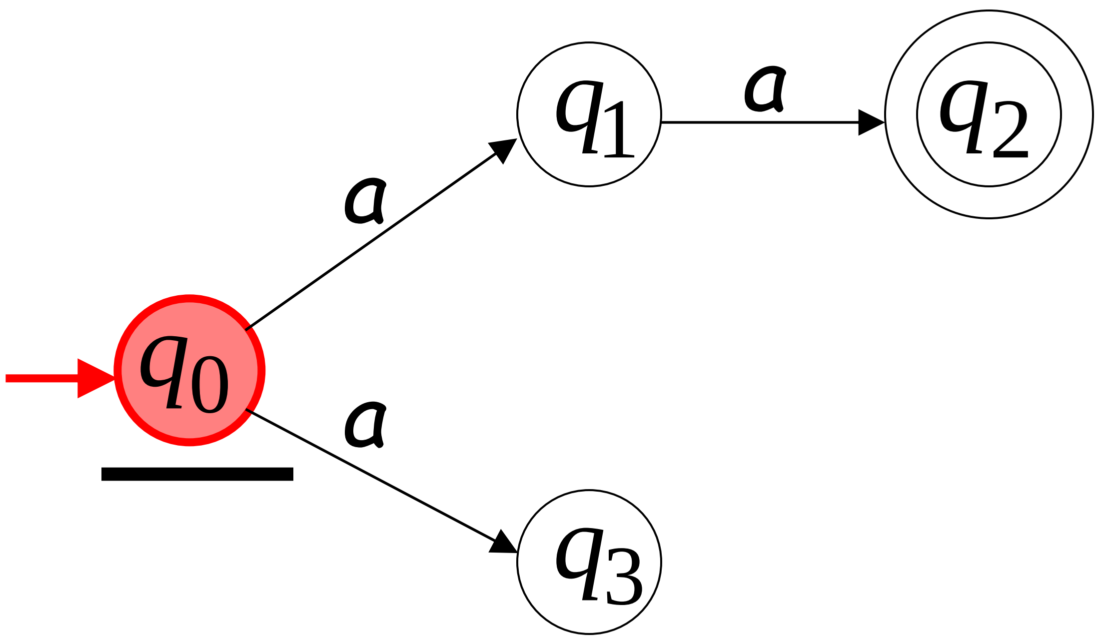
Nondeterministic Finite Accepter (NFA)

Alphabet = $\{a\}$

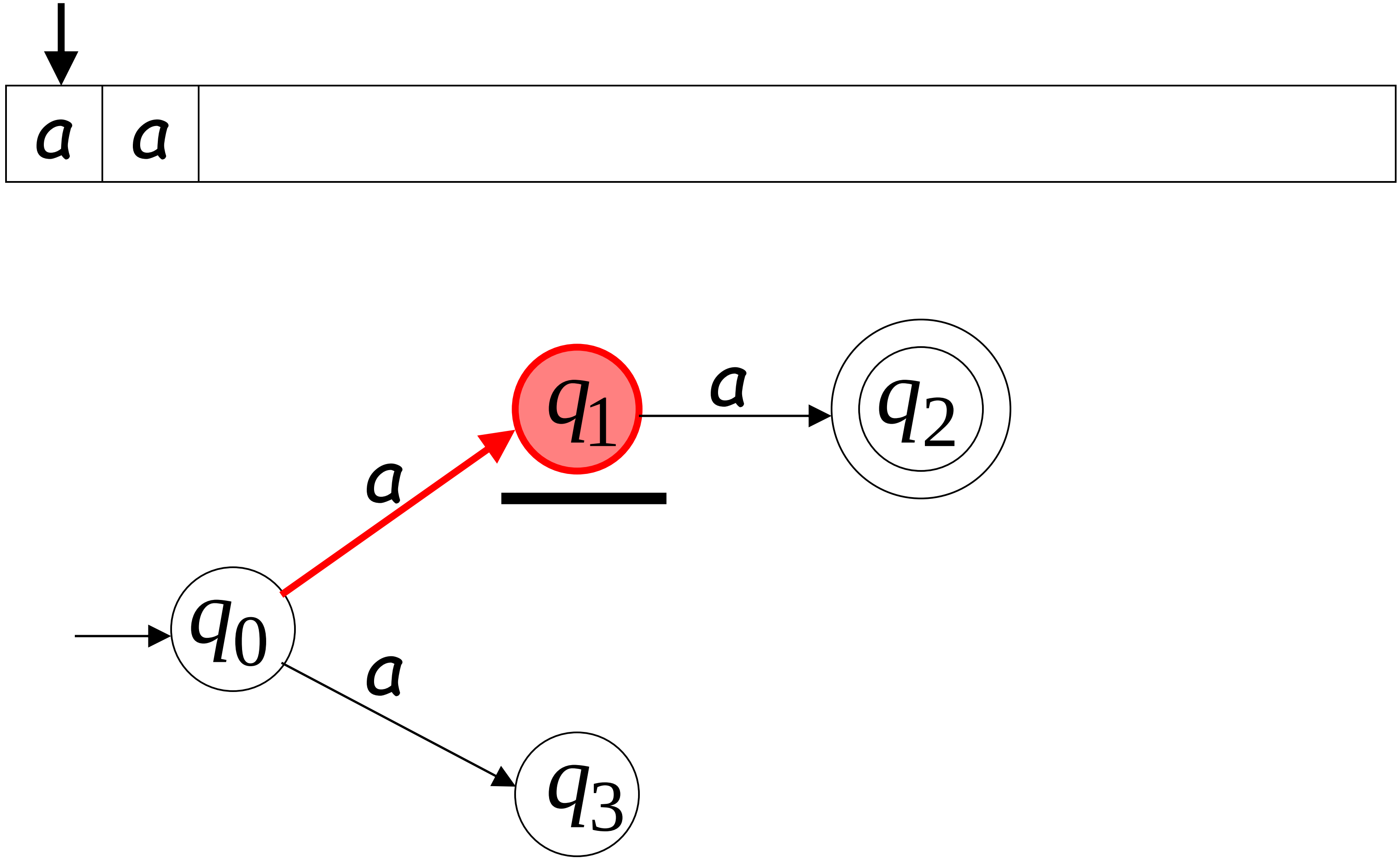


First Choice

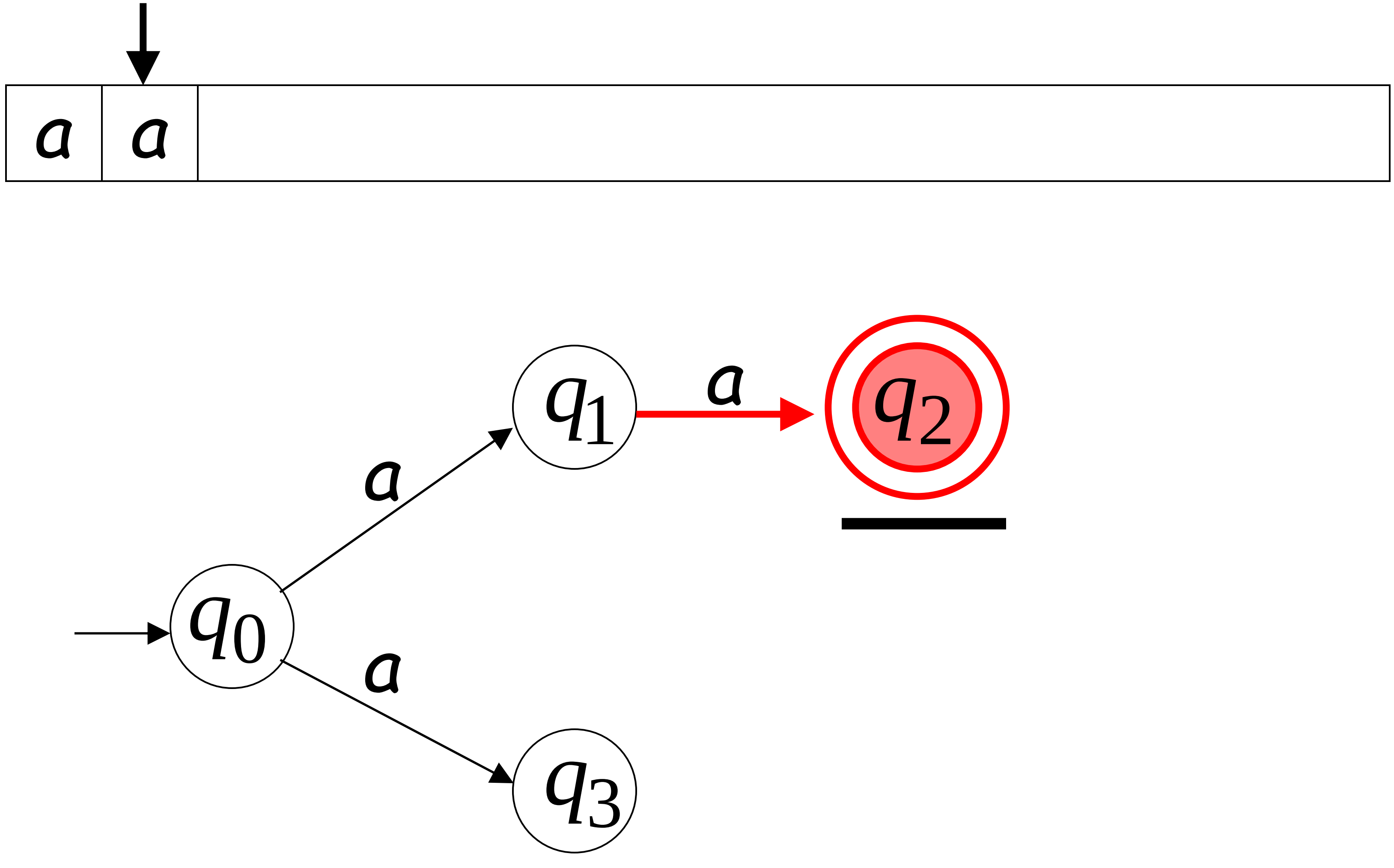




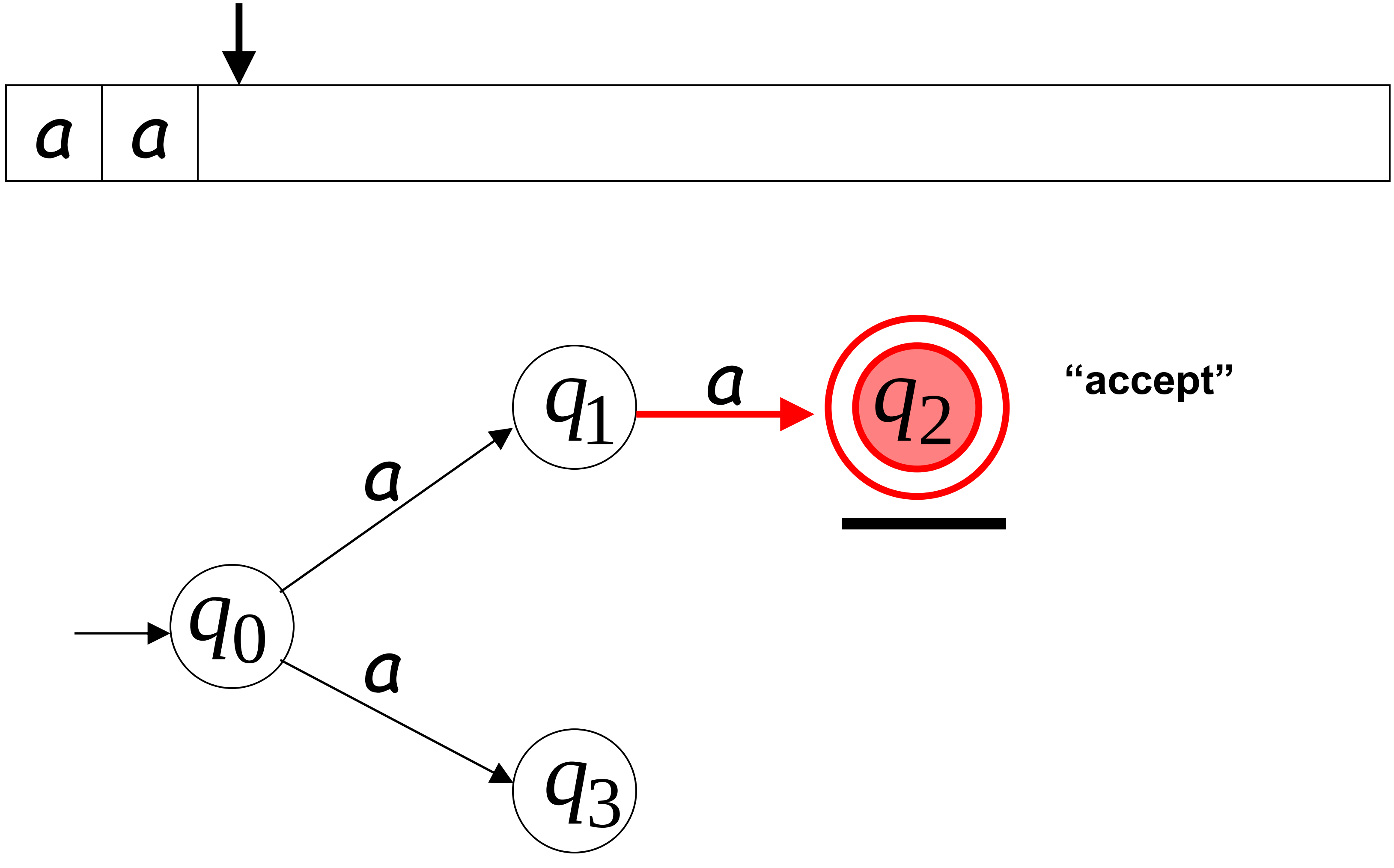
First Choice



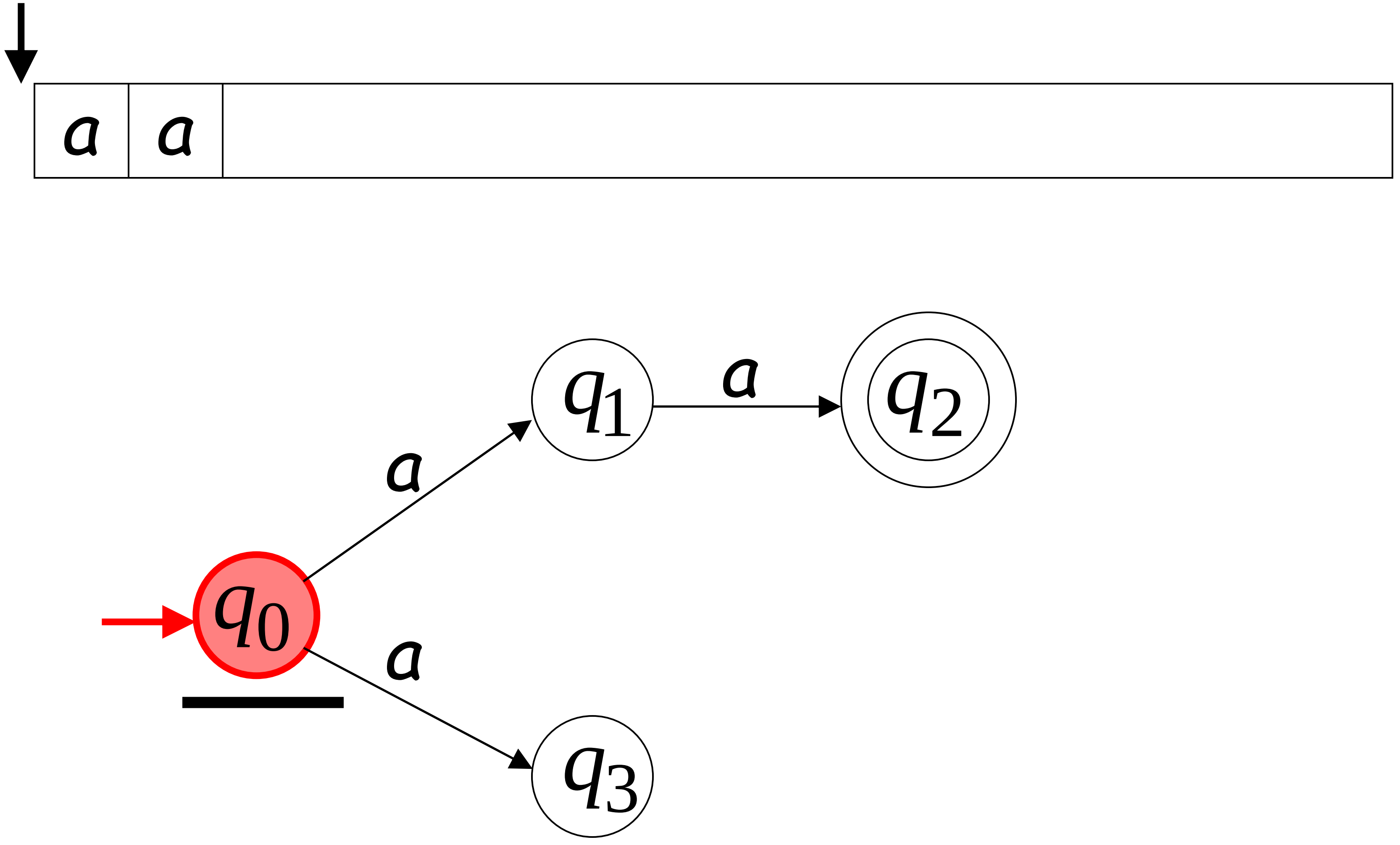
First Choice



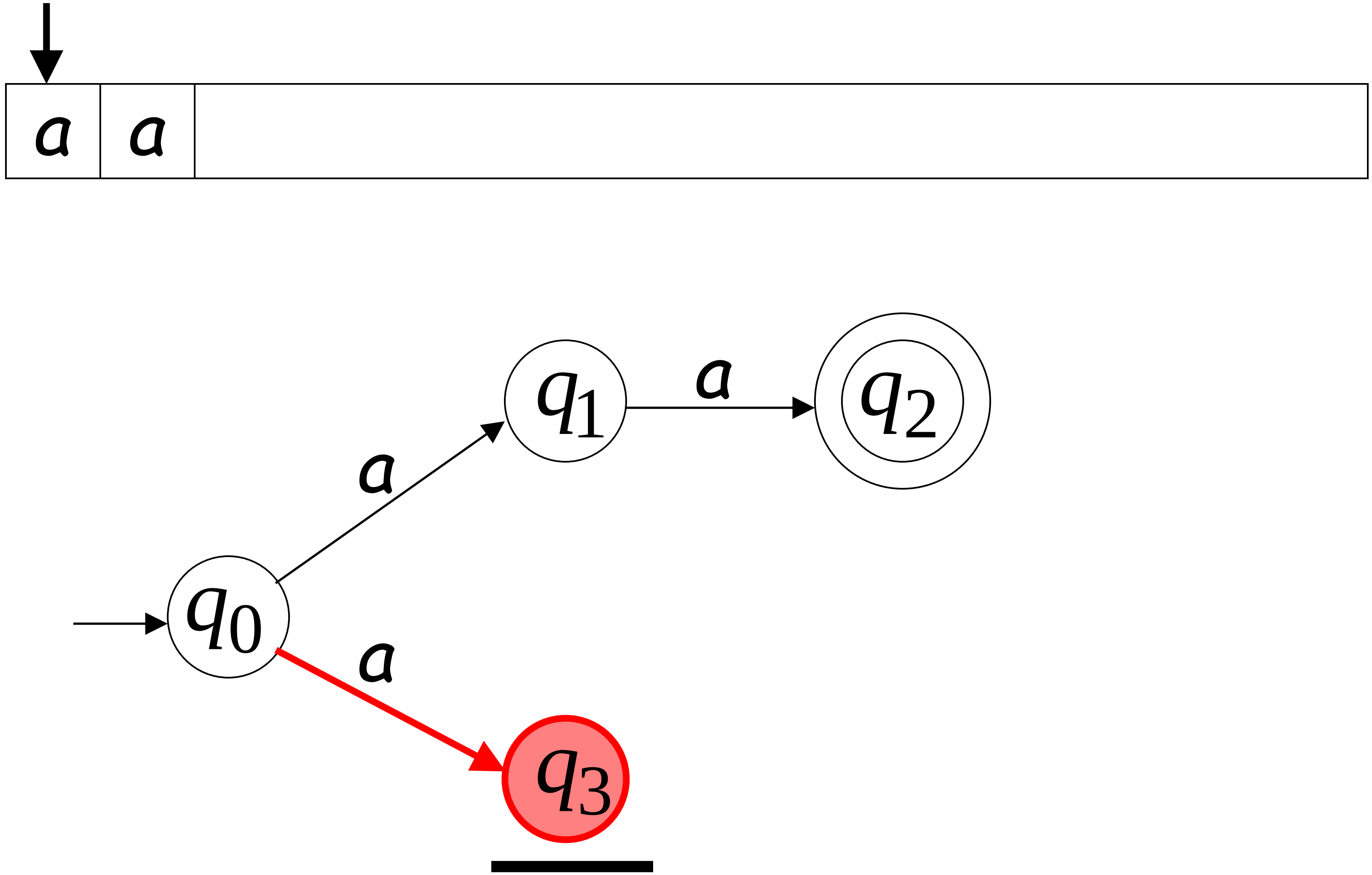
First Choice



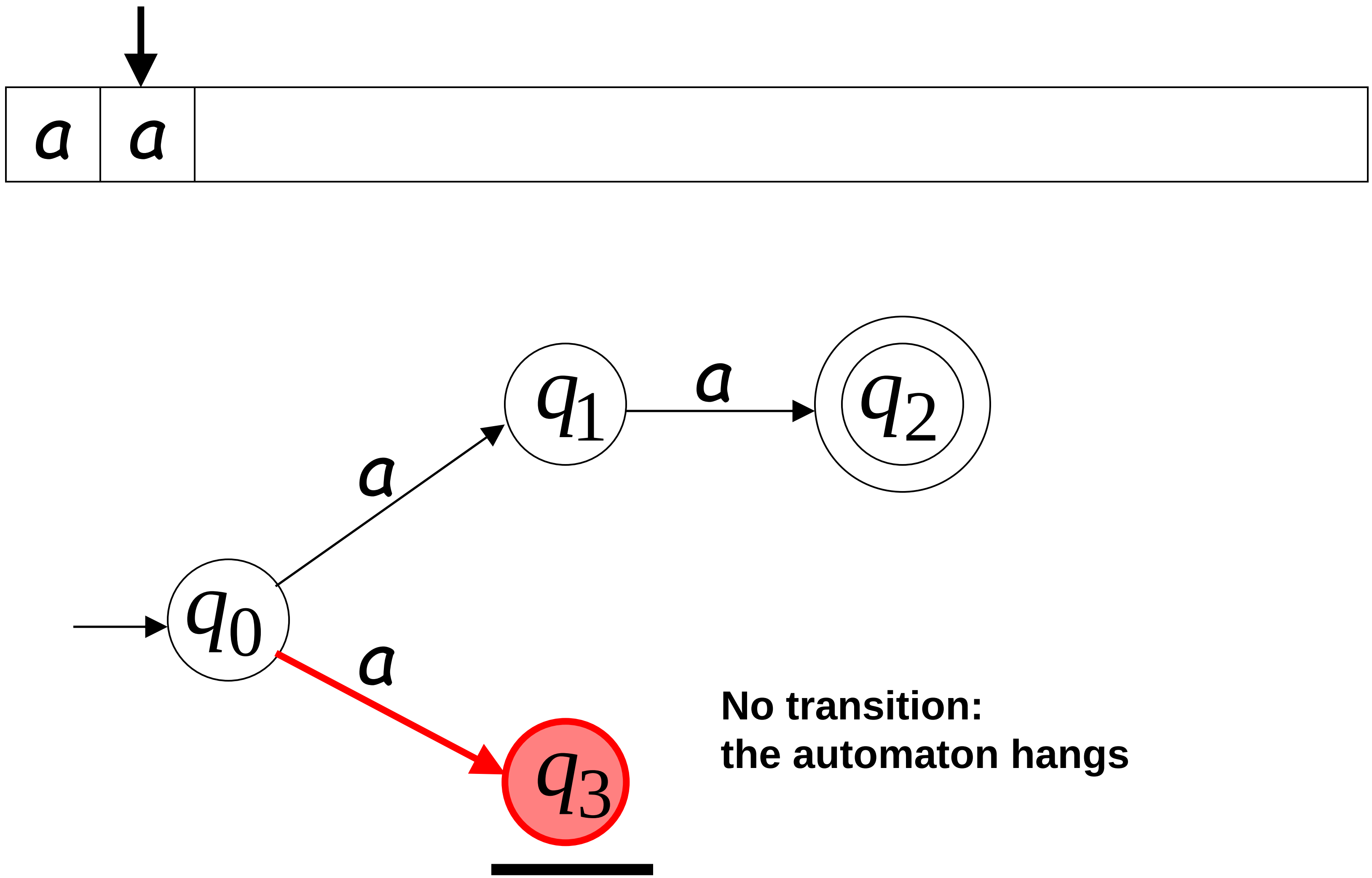
Second Choice



Second Choice

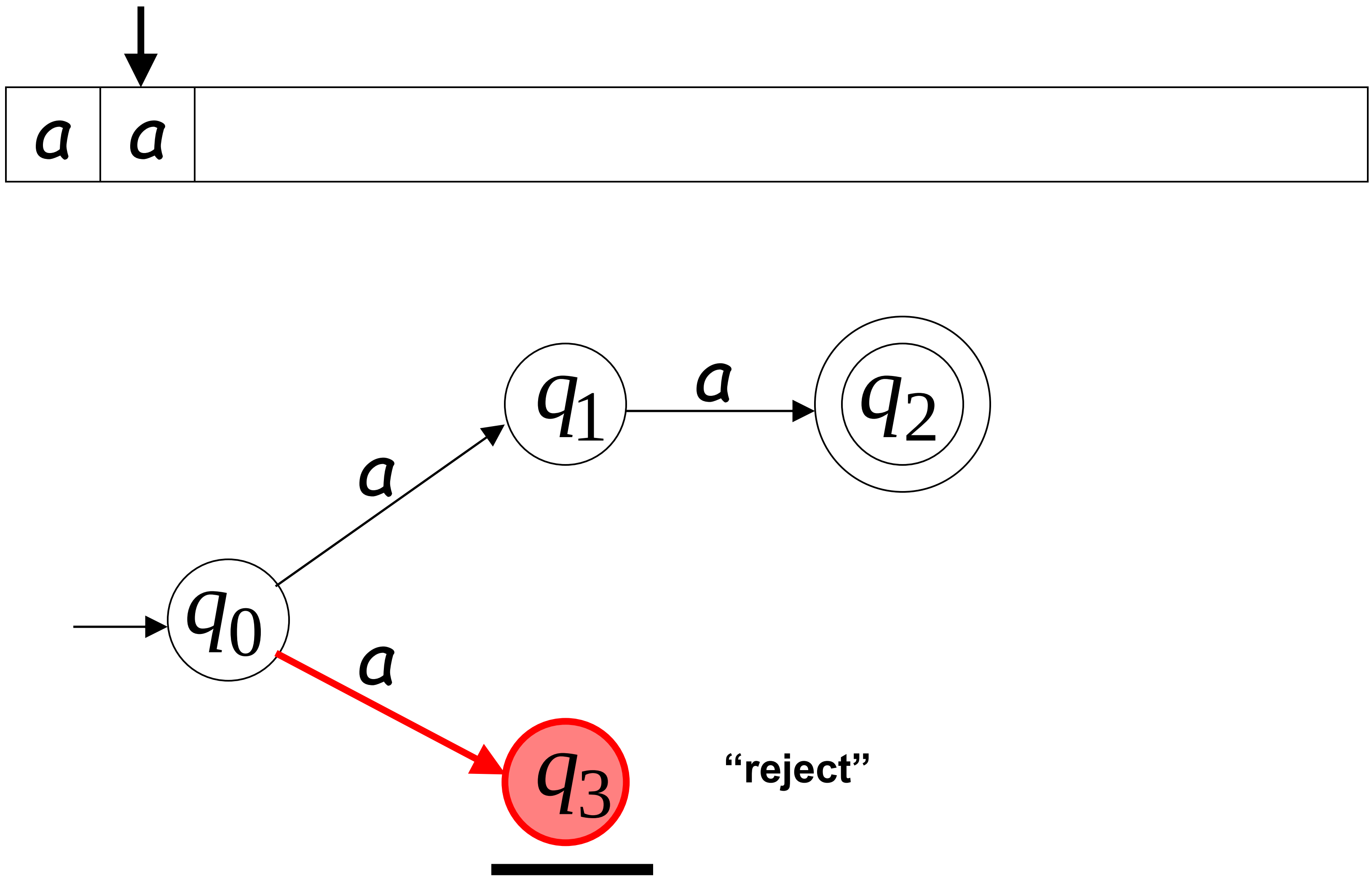


Second Choice



No transition:
the automaton hangs

Second Choice



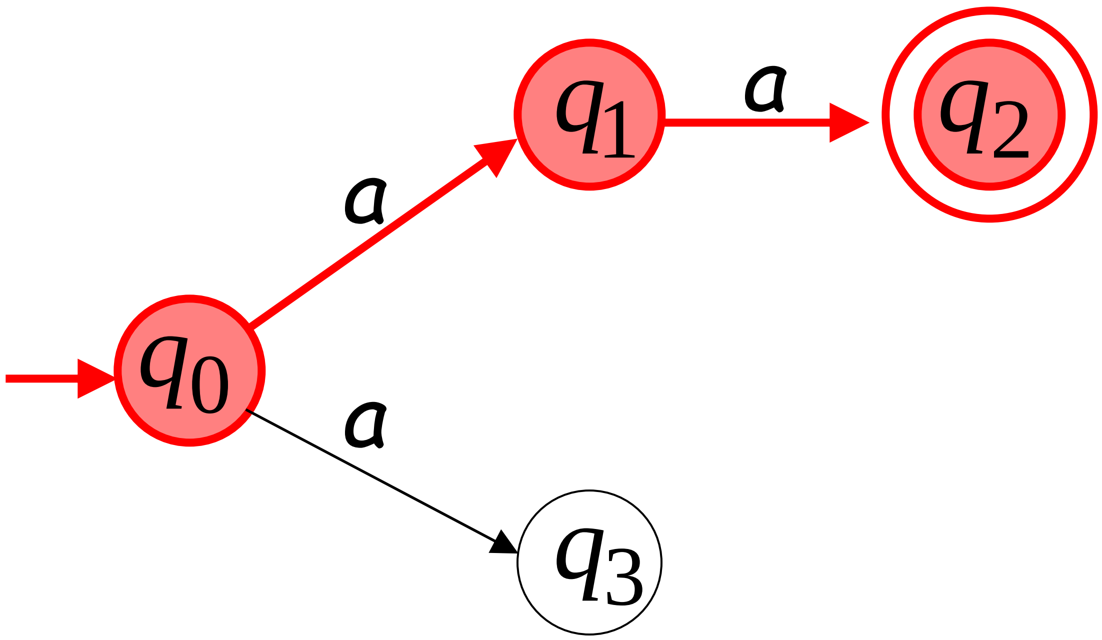
Observation

An NFA accepts a string:

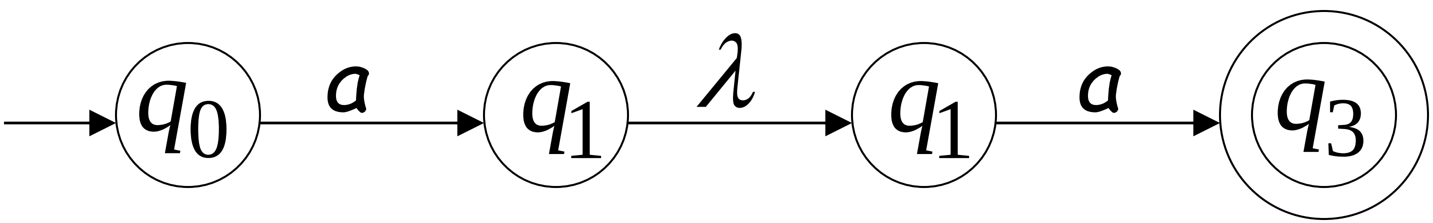
if
there is a computation of the NFA
that accepts the string

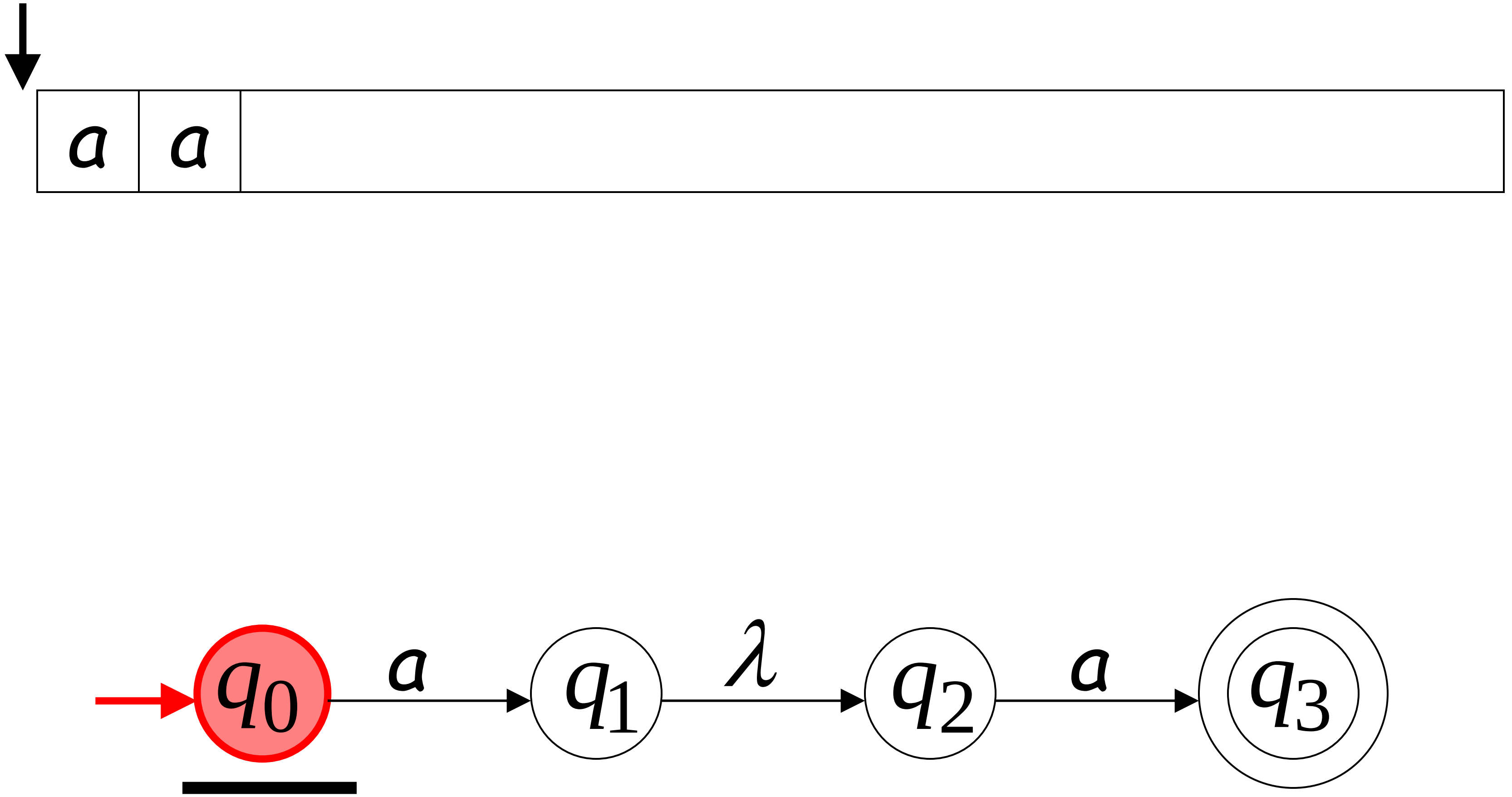
Example

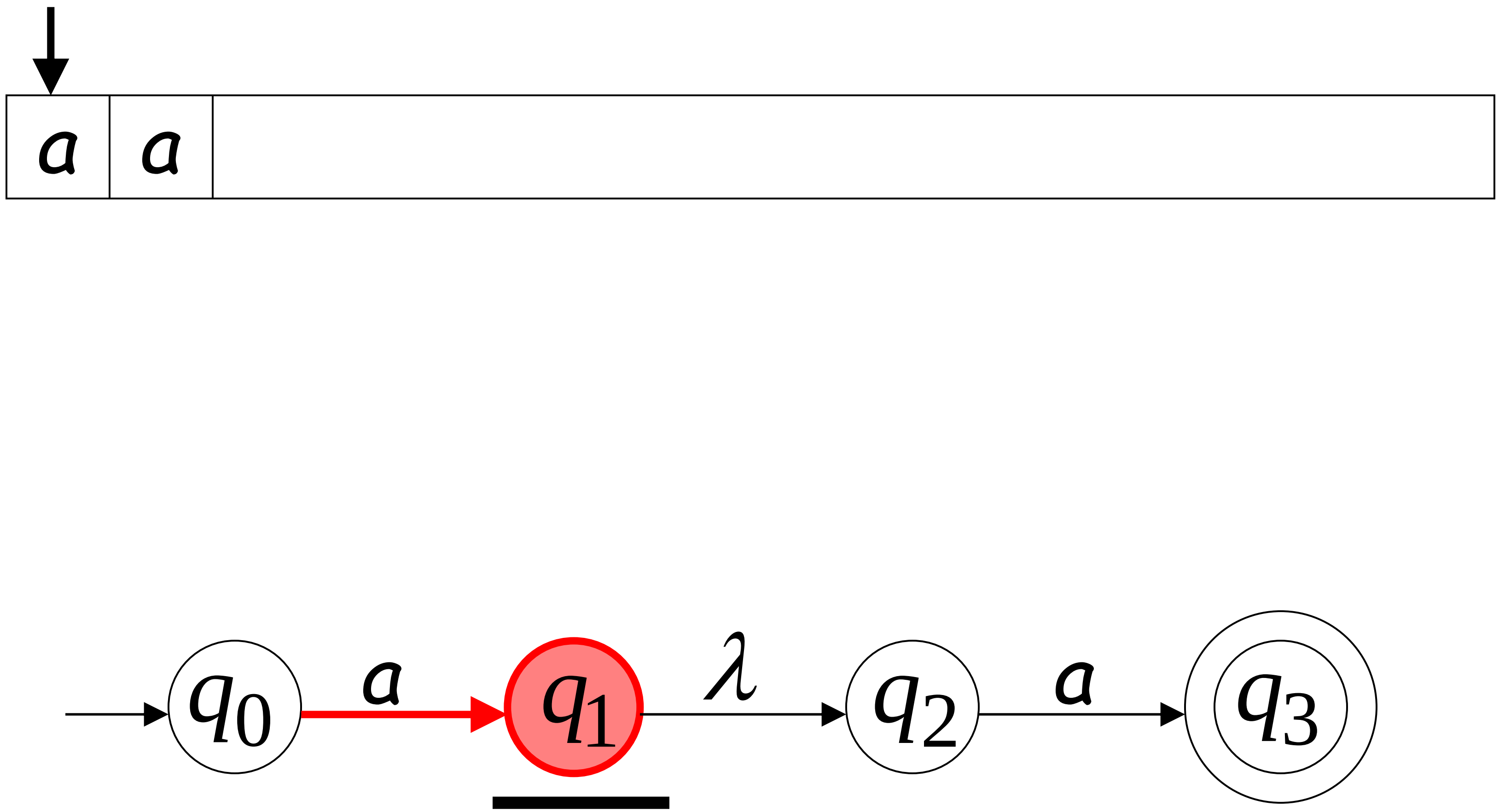
aa **is accepted by the NFA:**



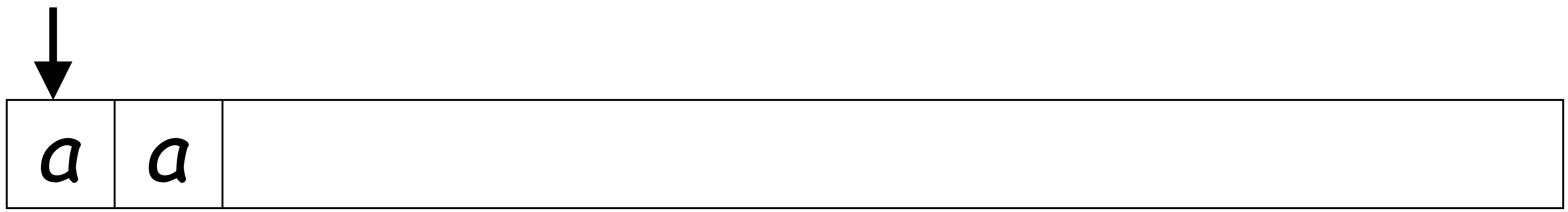
Lambda Transitions

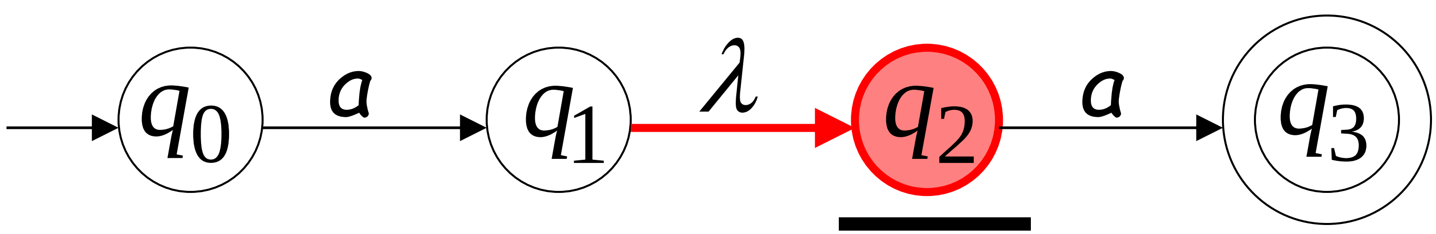


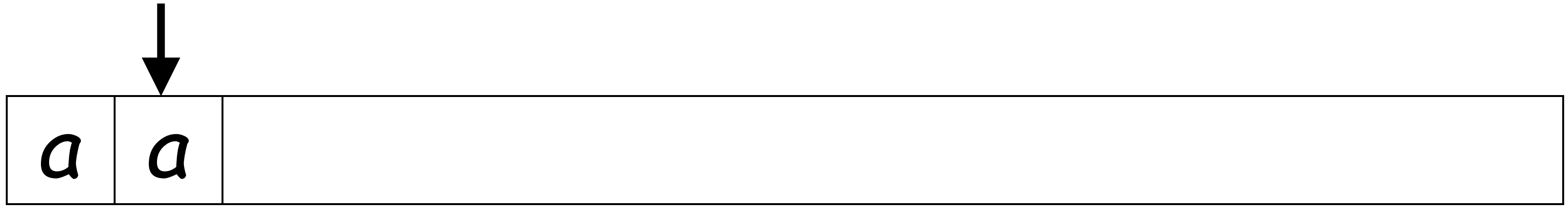

a
a
a
λ
a
q_0
q_1
q_2
q_3

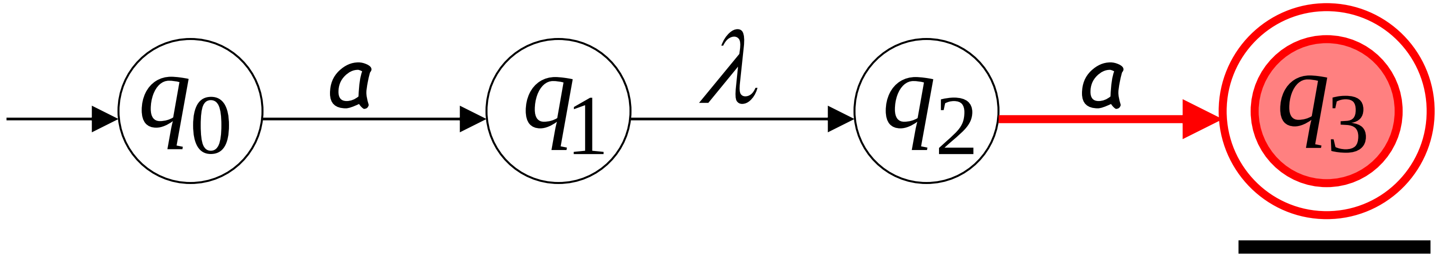

a a
q_0
a
q_1
λ
q_2
a
q_3

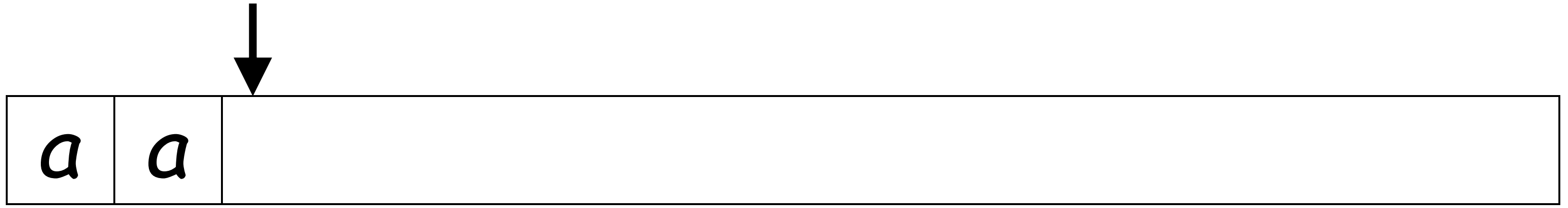
(read head doesn't move)

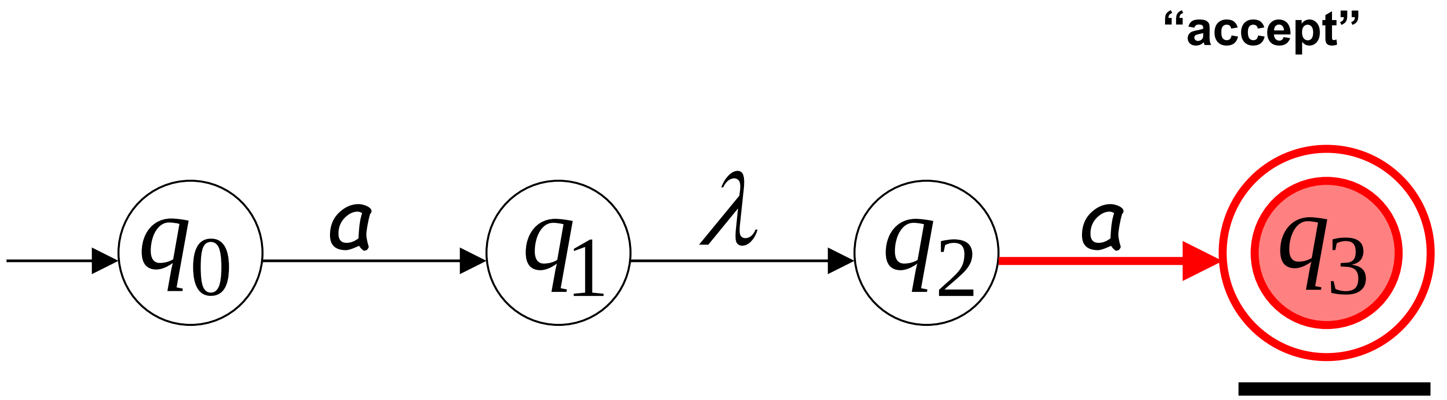





a
a

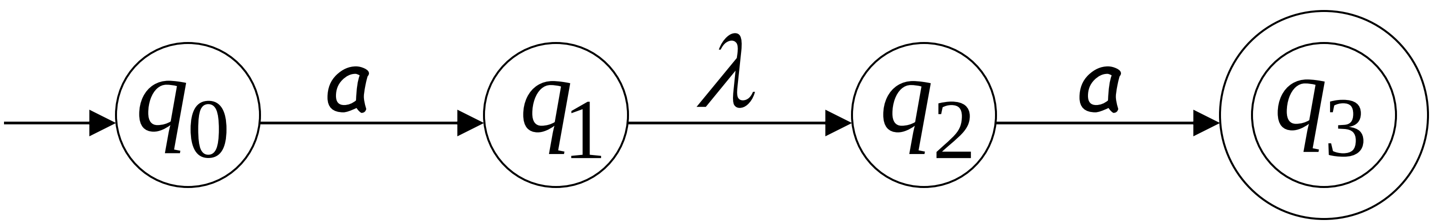

q_0
a
q_1
λ
q_2
a
q_3



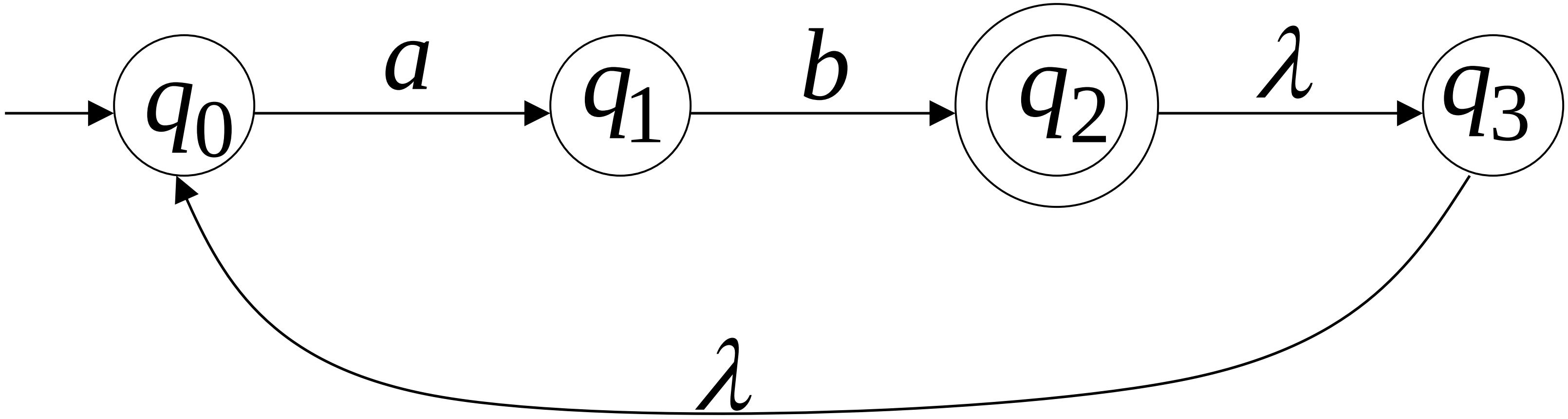


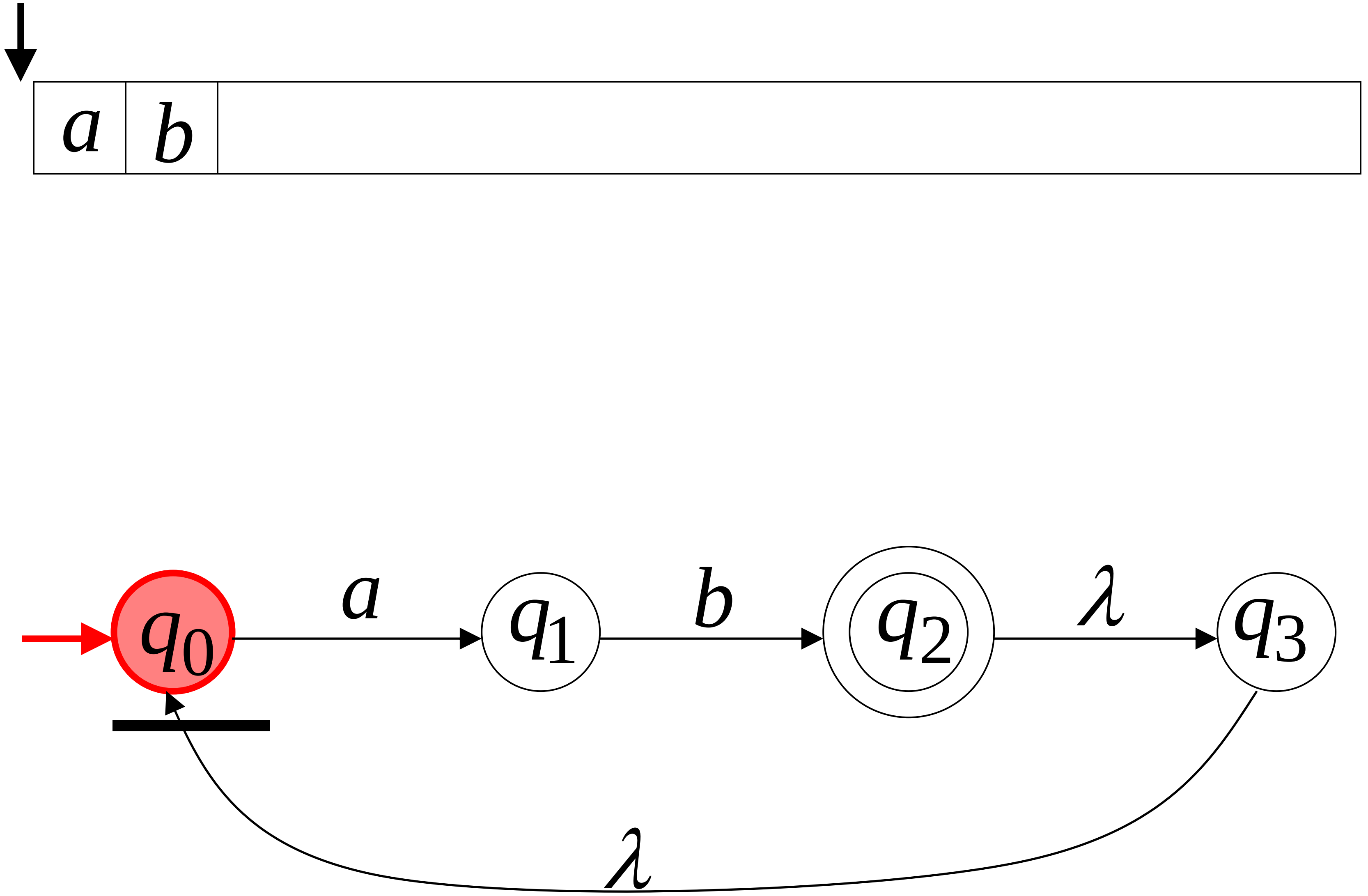
String aa **is accepted**

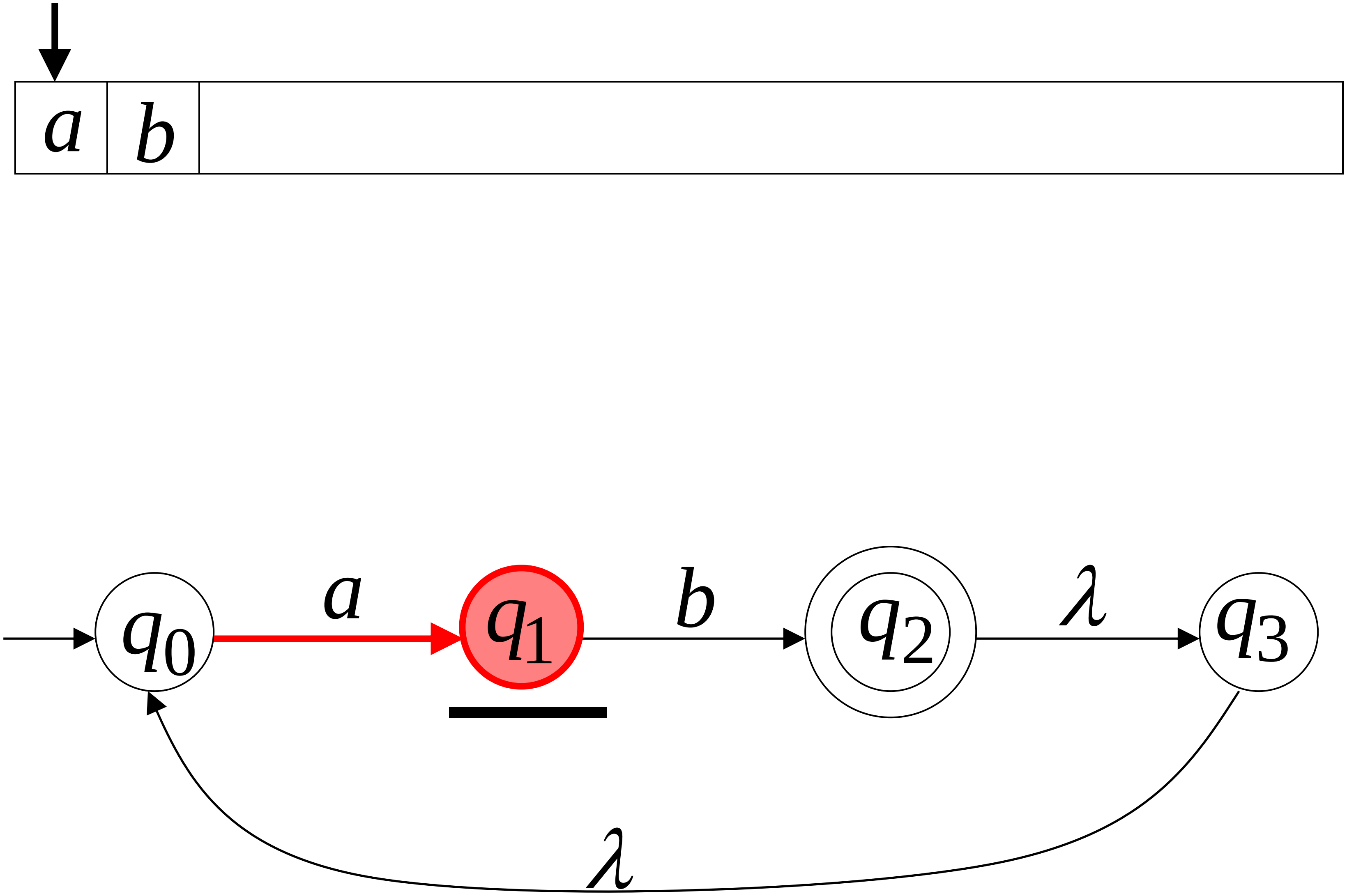
Language accepted: $L = \{aa\}$

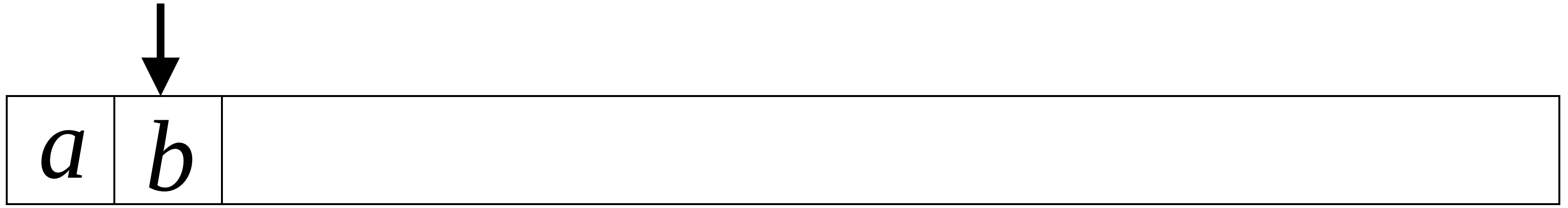


Another NFA Example

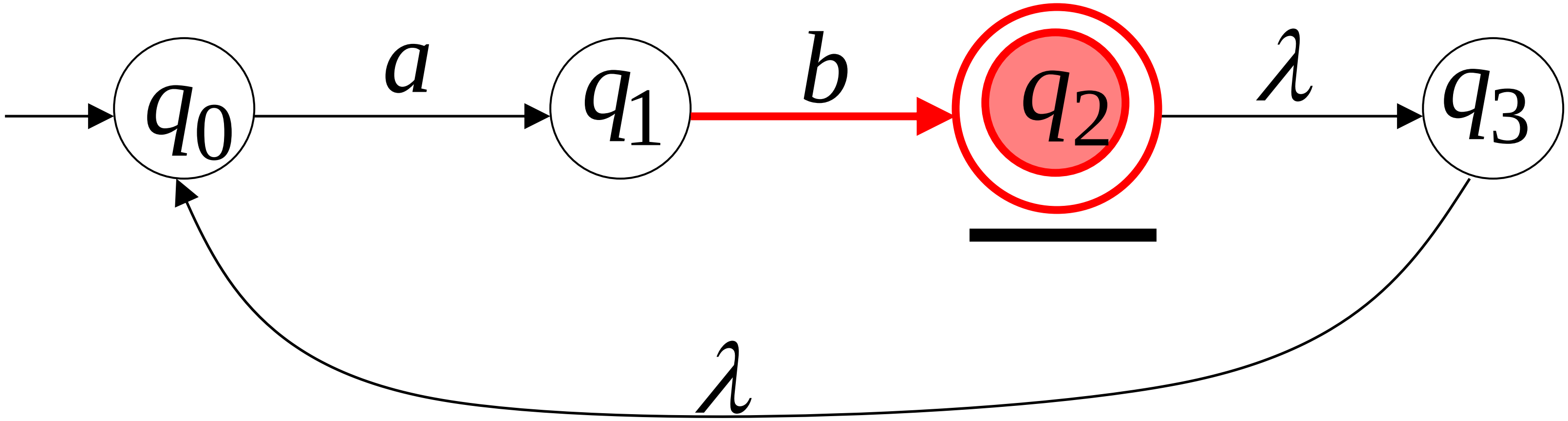



a
b
a
q_0
q_1
b
q_2
λ
q_3
λ

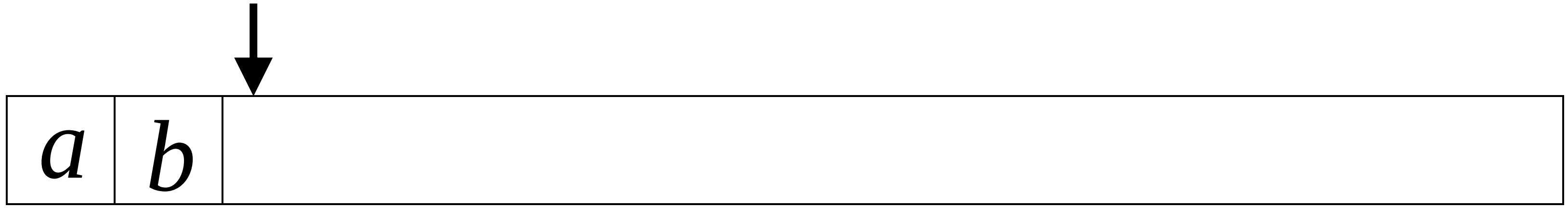

a
b
q_0
a
q_1
b
q_2
λ
q_3
λ



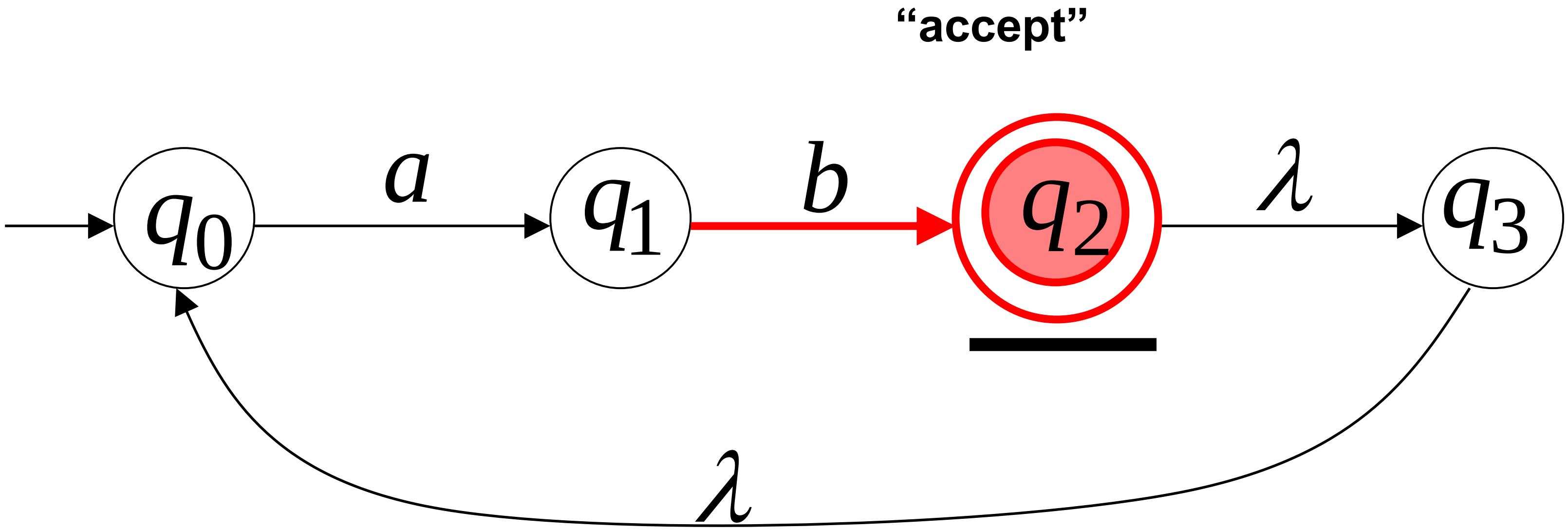
a
b



q_0
a
q_1
b
q_2
λ
q_3
λ

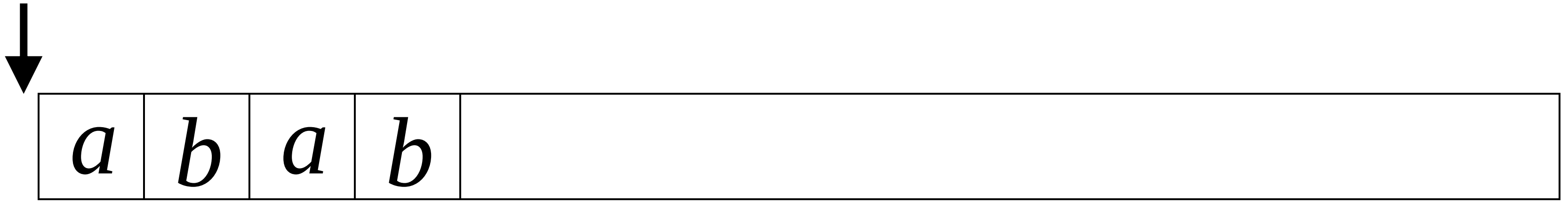


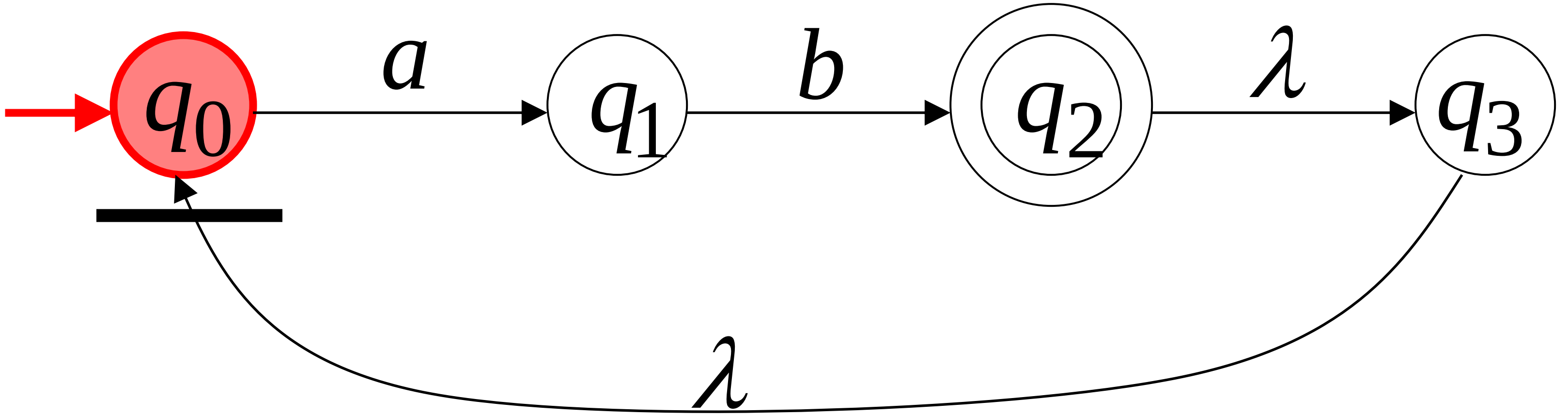
a
b

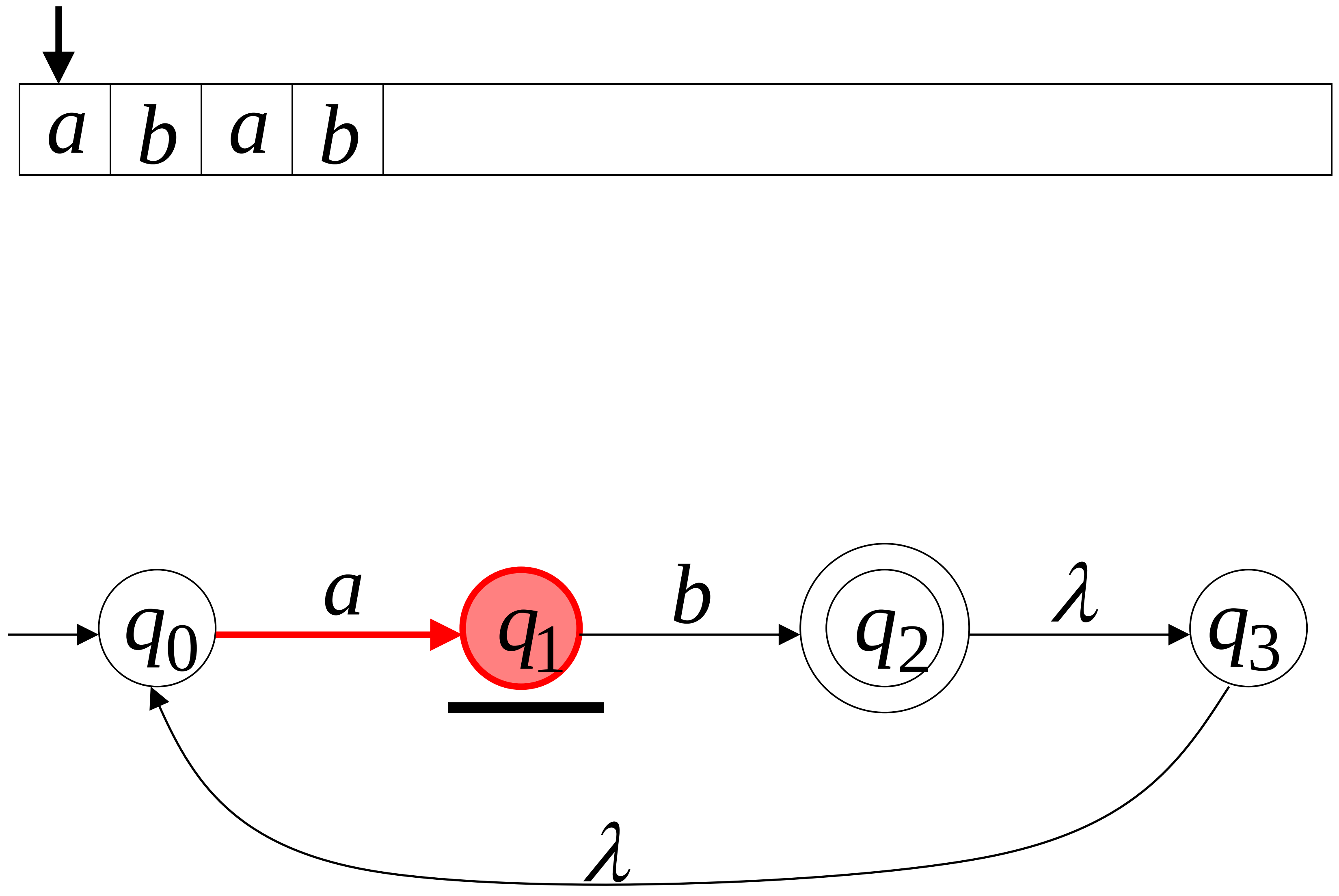


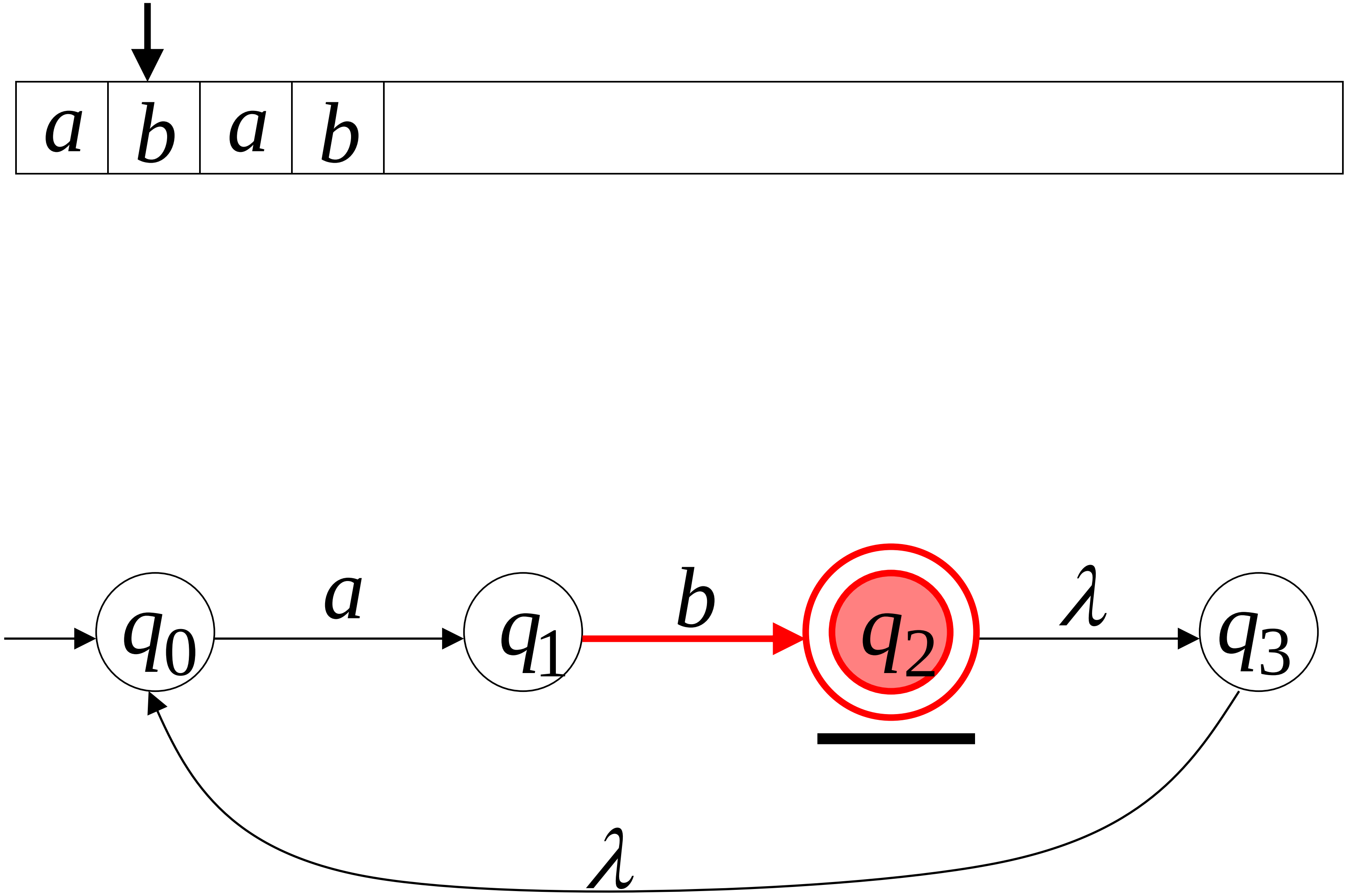
“accept”
q_0
a
q_1
b
q_2
λ
q_3
λ

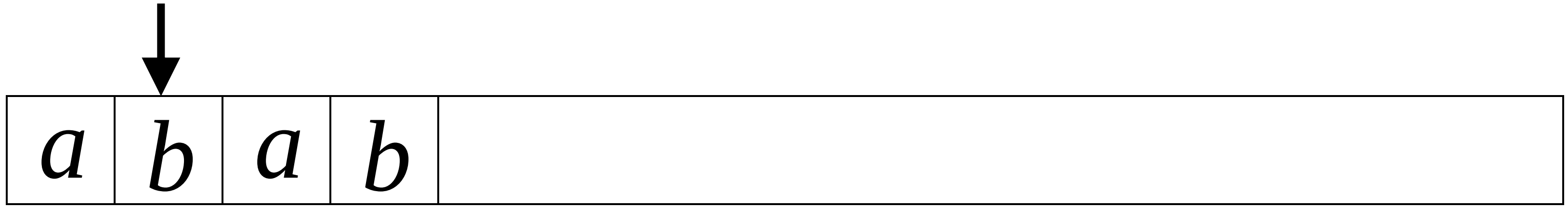
Another String

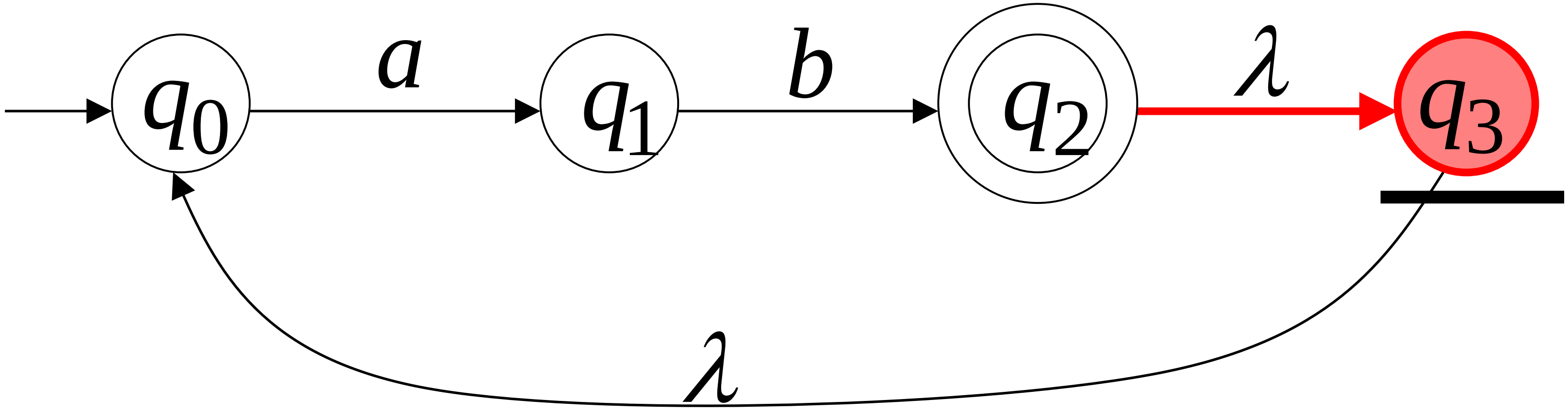


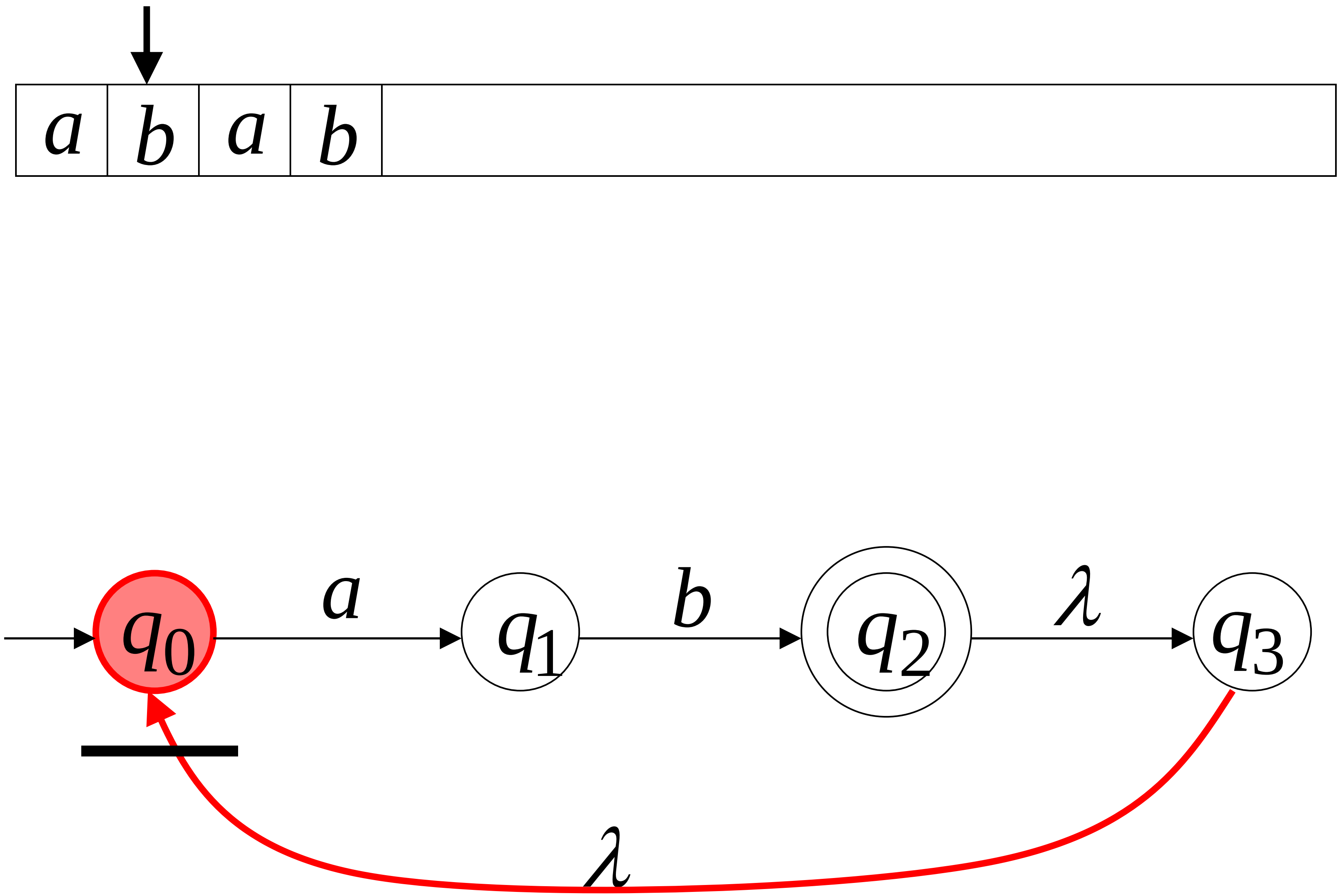


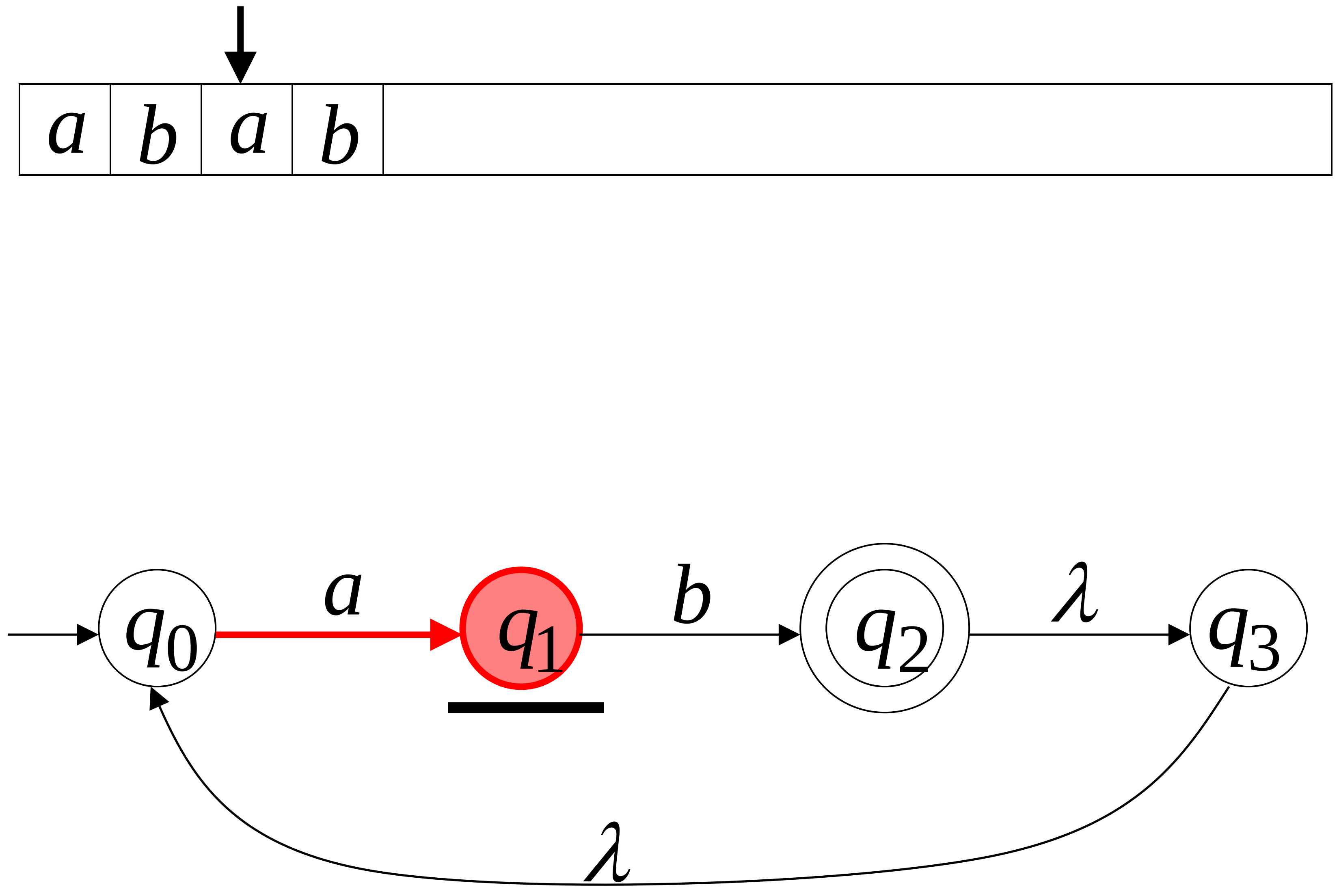

a
b
a
b
q_0
a
q_1
b
q_2
λ
q_3
λ

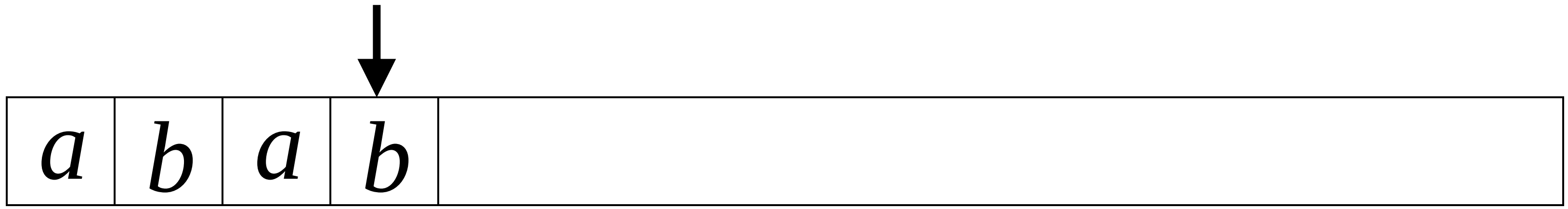

a
b
a
b
q_0
a
q_1
b
q_2
λ
q_3
λ

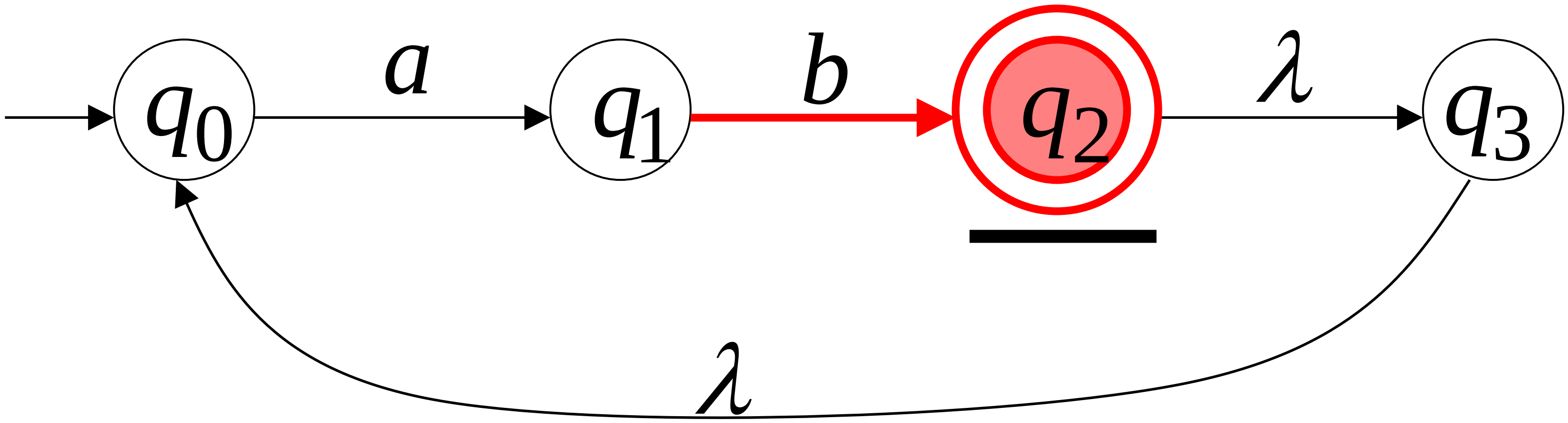

a
b
a
b

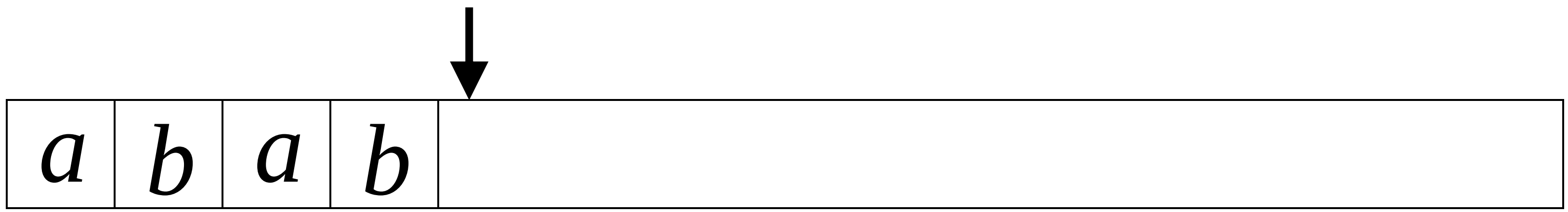

q_0
a
q_1
b
q_2
λ
q_3
λ

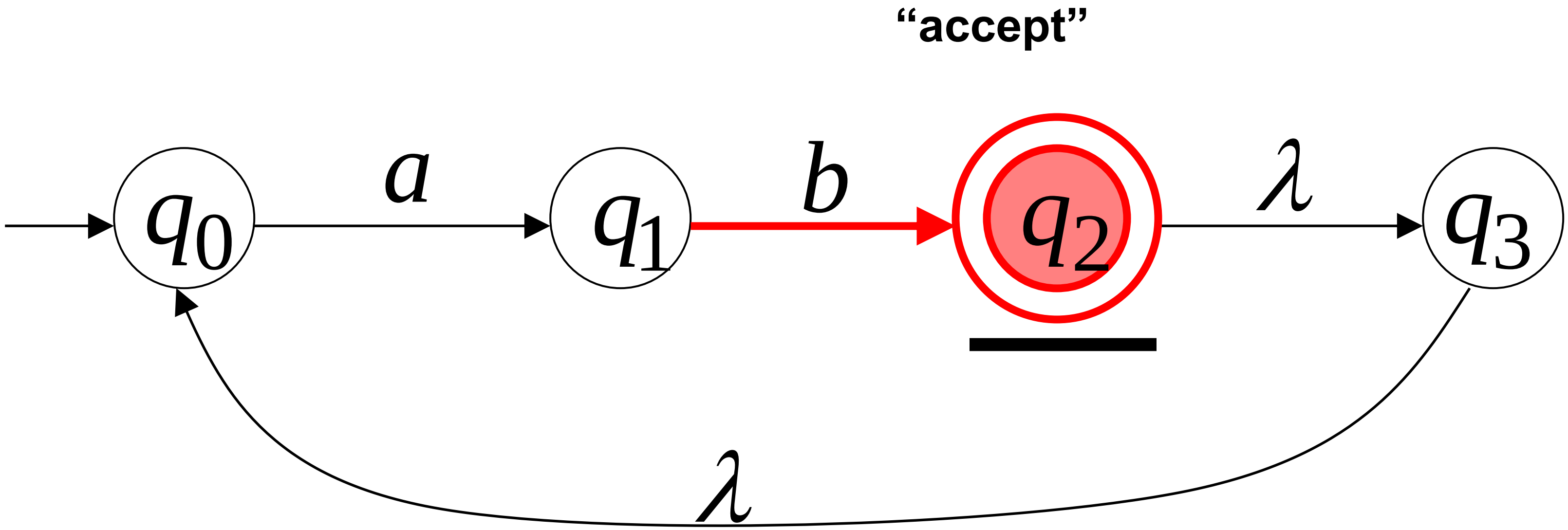

a
b
a
b
a
b
λ
λ
q_0
q_1
q_2
q_3


a
b
a
b
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

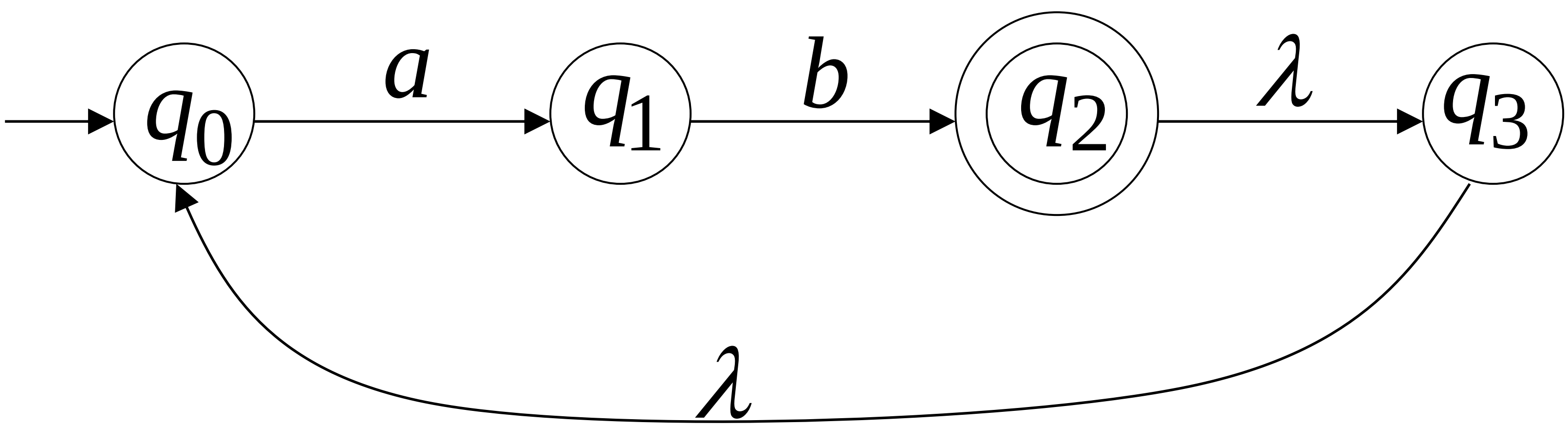

a
b
a
b


"accept"
a
b
λ
λ
q_0
q_1
q_2
q_3

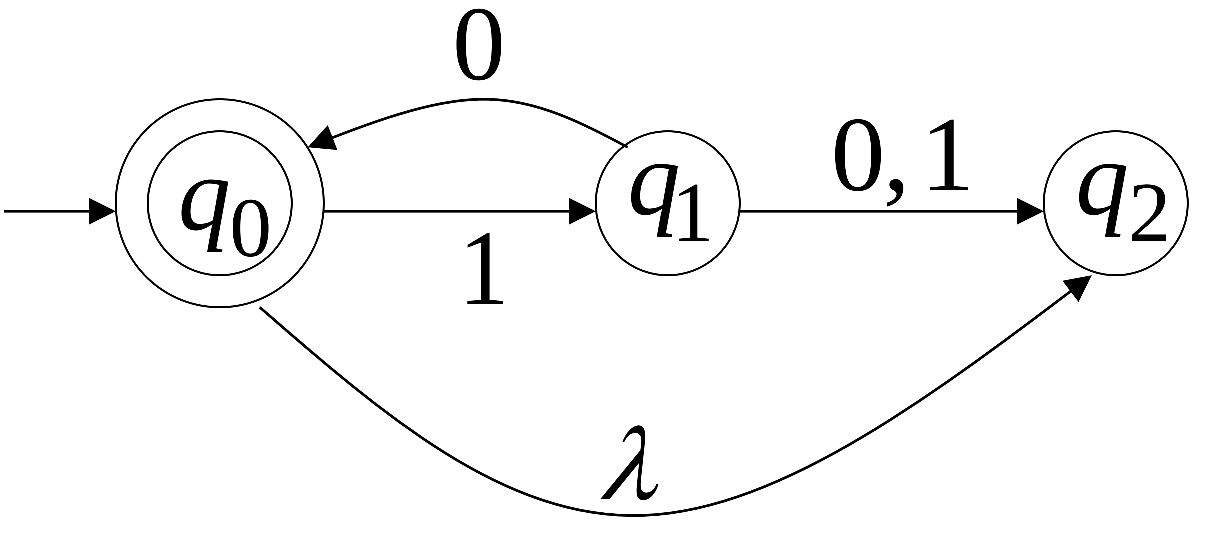
Language accepted

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

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

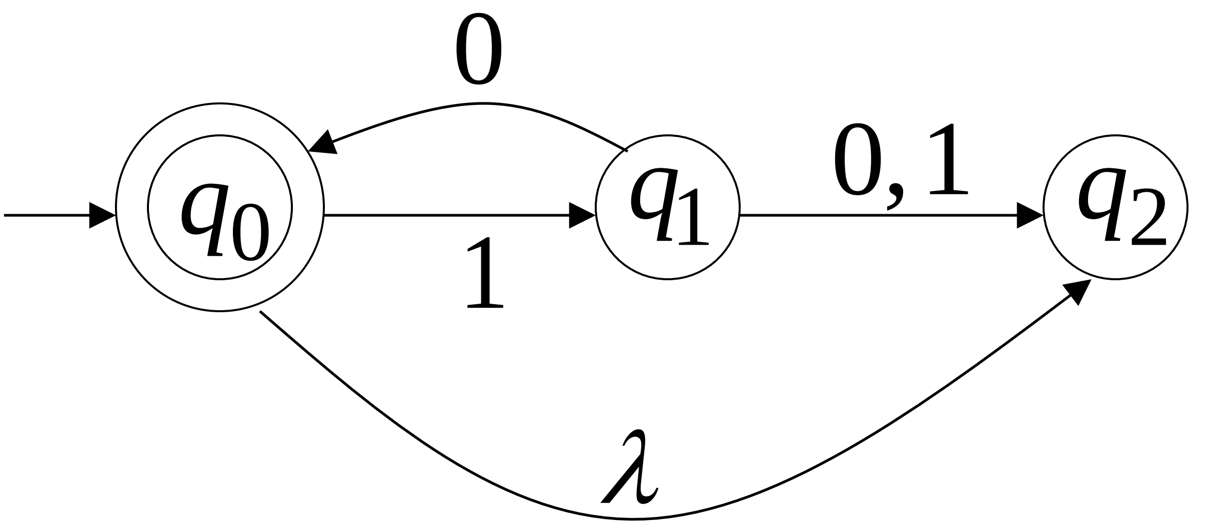


Another NFA Example

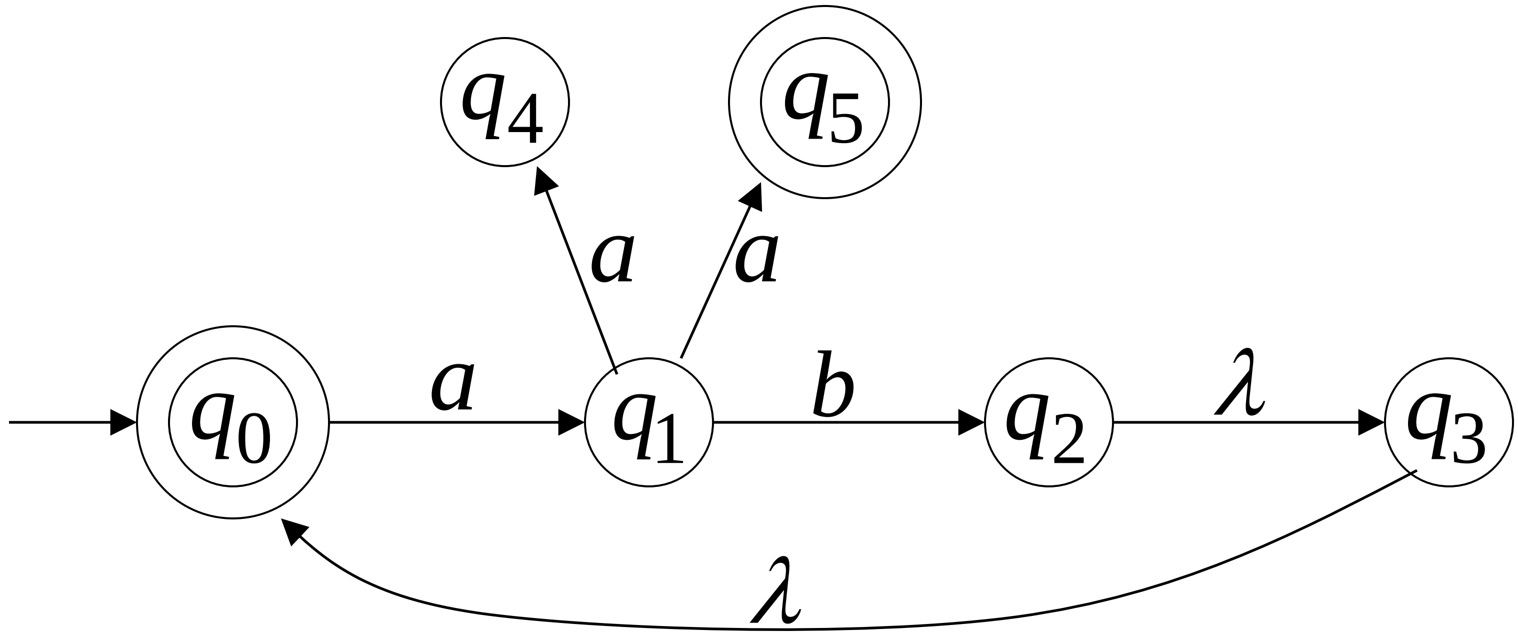


Language accepted

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

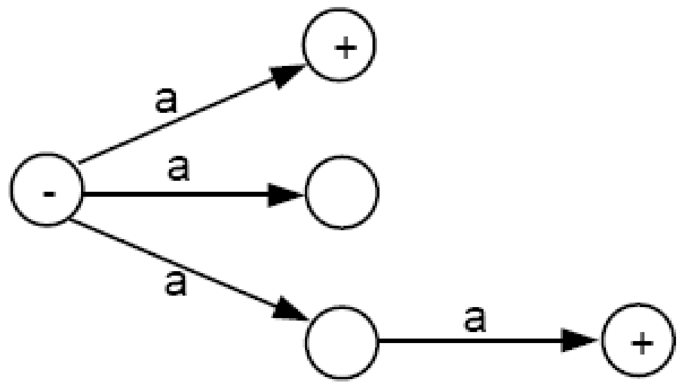


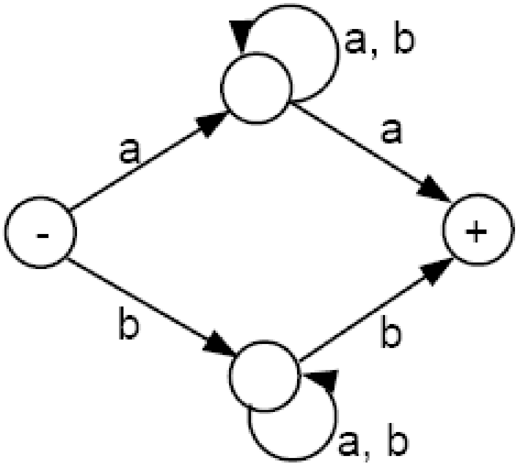
•

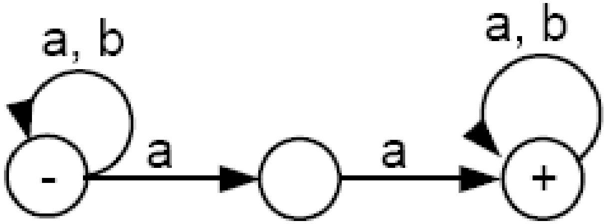


$$L(M) = \{aa\} \cup \{ab\}^* \cup \{ab\}^+\{aa\}$$

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

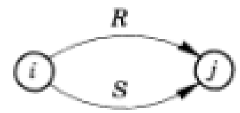
Regular Expression to Finite Automaton (5.4)

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

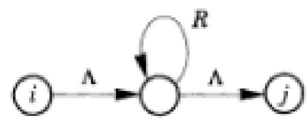


3. Transform any diagram like

into the diagram

4. Transform any diagram like

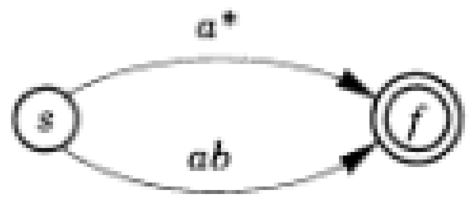
into the diagram



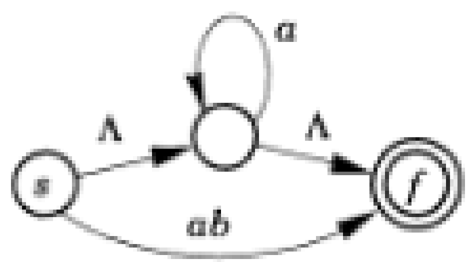
End of Algorithm

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

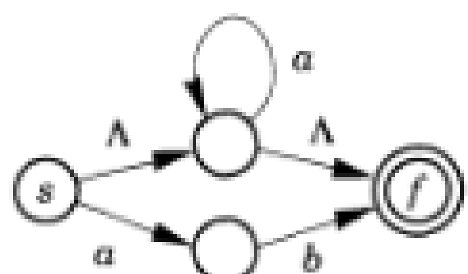
Next we apply rule 2 to obtain the following NFA:



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

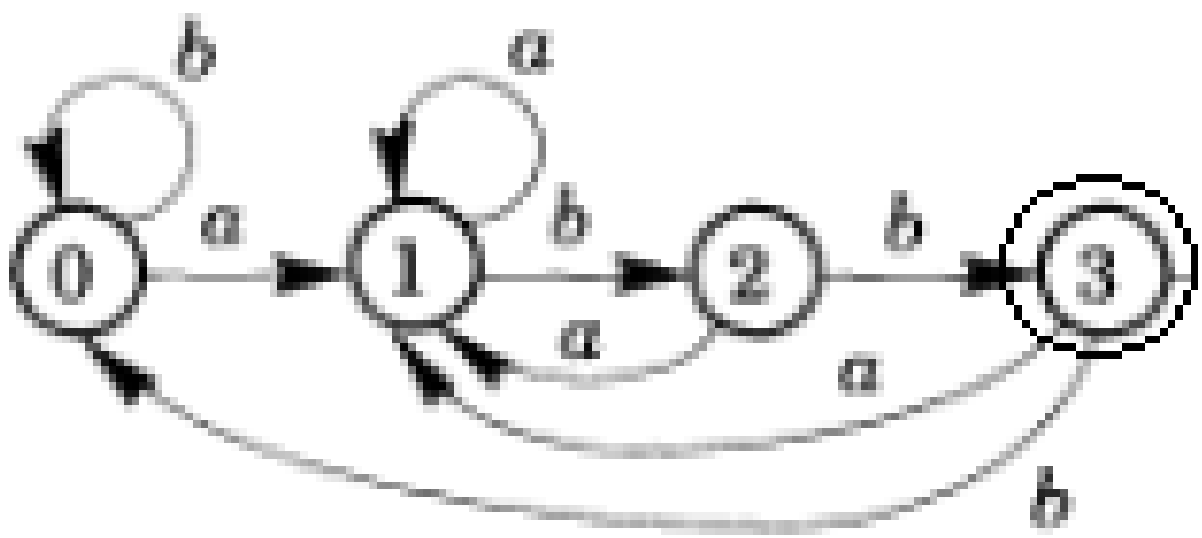


Finally, we apply rule 3 to ab to obtain the desired NFA for $a^* + ab$:



Show that the following DFA is equivalent to R.E

(a+b)*abb



NFA to DFA Conversion