Compiler Design

Dr. Sahar kamal

Lexical Analysis

Lecture 3

Compilers *Principles, Techniques, & Tools*

Second Edition

Finite Automata

•A *recognizer* for a language is a program that takes a string x, and answers “yes” if x is a sentence of that language, and “no” otherwise.
•We call the recognizer of the tokens as a *finite automaton*.
•A finite automaton can be *deterministic(DFA)* or *non-deterministic (NFA)*
•This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
•Both deterministic and non-deterministic finite automaton recognize regular sets.
•Which one?
–deterministic –faster recognizer, but it may take more space
–non-deterministic –slower, but it may take less space
–Deterministic automatons are widely used lexical analyzers.
•First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
–Algorithm1: Regular Expression ➔NFA ➔DFA (two steps: first to NFA, then to DFA)
–Algorithm2: Regular Expression ➔DFA (directly convert a regular expression into a DFA)

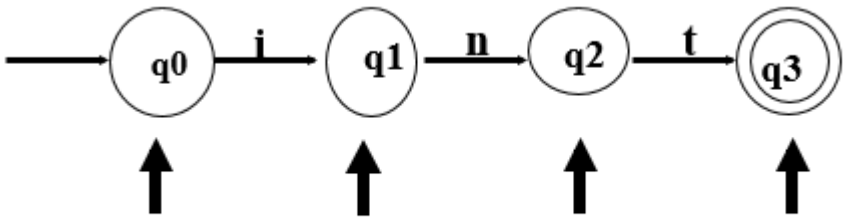
Finite Automata (FA)

- FA also called Finite State Machine (FSM)
  - Decides whether to accept or reject a string.
  - Every regular expression can be represented as a FA and vice versa

- Two types of FAs:

❑ Non-deterministic (NFA): Has more than one alternative action for the same input symbol. A symbol can label several edges out of the same state, and the empty string, is a possible label.

❑ Deterministic (DFA): have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol



Finite Automata

- To convert the regular expression (RE) to Finite Automata (FA), we can use the Subset method.
- Subset method is used to obtain FA from the given RE.

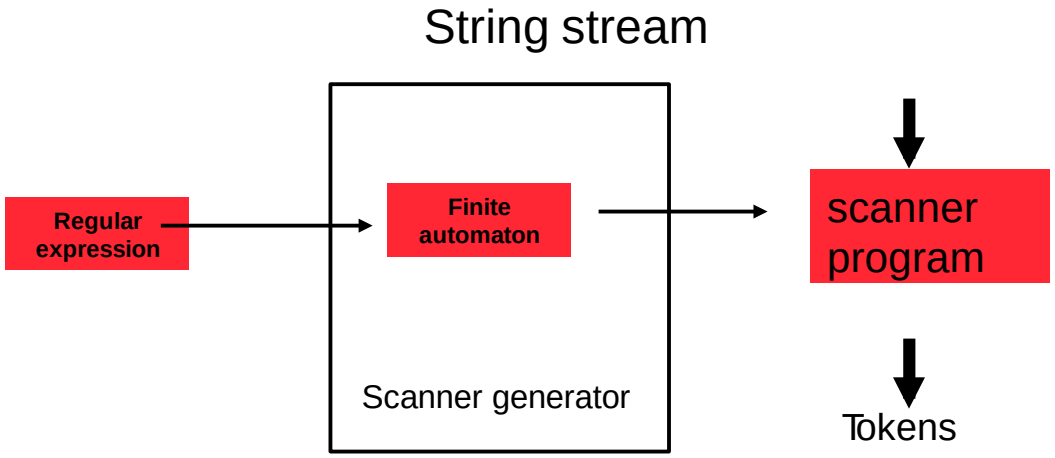
❖**Step 1** − Construct a Transition diagram for a given RE by using Non-deterministic finite automata (NFA) with ε moves.

❖**Step 2** − Convert NFA with ε to NFA without ε.

❖**Step 3** − Convert the NFA to the equivalent Deterministic Finite Automata (DFA).

RE and Finite State Automaton (FA)

- Regular expressions are a declarative way to describe the tokens
  - Describes *what* is a token, but not *how* to recognize the token
- FAs are used to describe *how* the token is recognized
  - FAs are easy to simulate in a programs
- There is a correspondence between FSAs and regular expressions
  - A scanner generator bridges the gap between regular expressions and FAs.

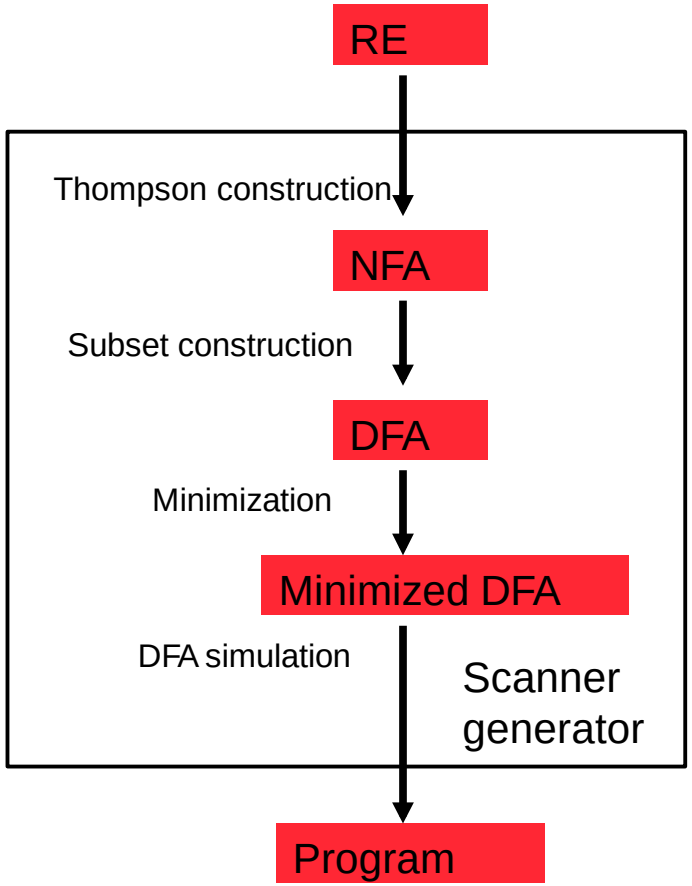


Inside scanner generator

Main components of scanner generation

Convert a regular expression to a non-deterministic finite automaton (NFA)

- Convert the NFA to a deterministic finite automaton (DFA)
- Improve the DFA to minimize the number of states
- Generate a program in C or some other language to "simulate" the DFA



Non-deterministic Finite Automata (NFA)

- NFA (Non-deterministic Finite Automaton) is a 5-tuple (S, Σ, δ, S0, F):
  - Sor Q: a set of states;
  - Σ: the symbols of the input alphabet;
  - δ : a set of transition functions;
    - » move(state, symbol) → a set of states
  - q0: s0 ∈S, the start state;
  - F: F ⊆ S, a set of final or accepting states.

- A Regular Expression is a representation of Tokens. But, to recognize a token, it can need a token Recognizer, which is nothing but a Finite Automata (NFA). So, it can convert Regular Expression into NFA
- **Thompson's construction** is used to transform a regular expression into a NFA

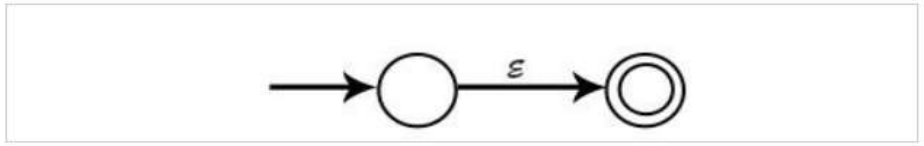
From Regular expression into NFA

Input − A Regular Expression R

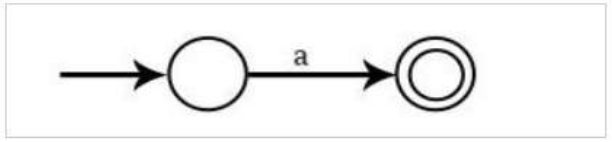
Output − NFA accepting language denoted by R

Method

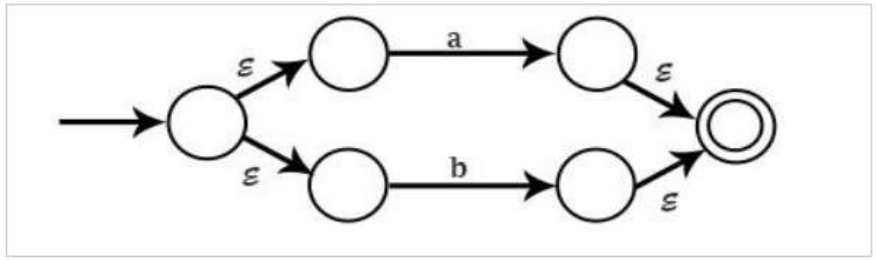
For ε, NFA is



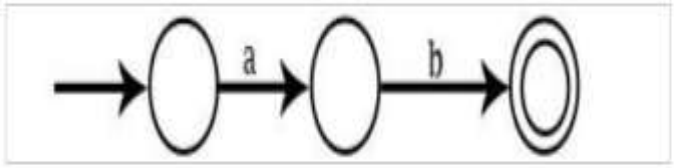
For a NFA is



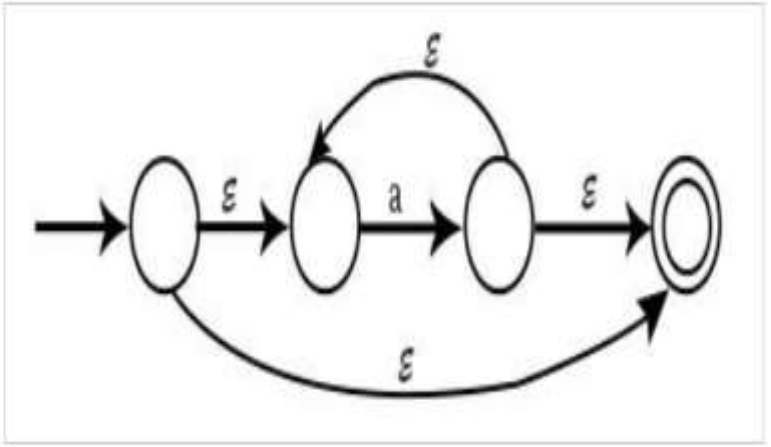
For a + b, or a | b NFA is



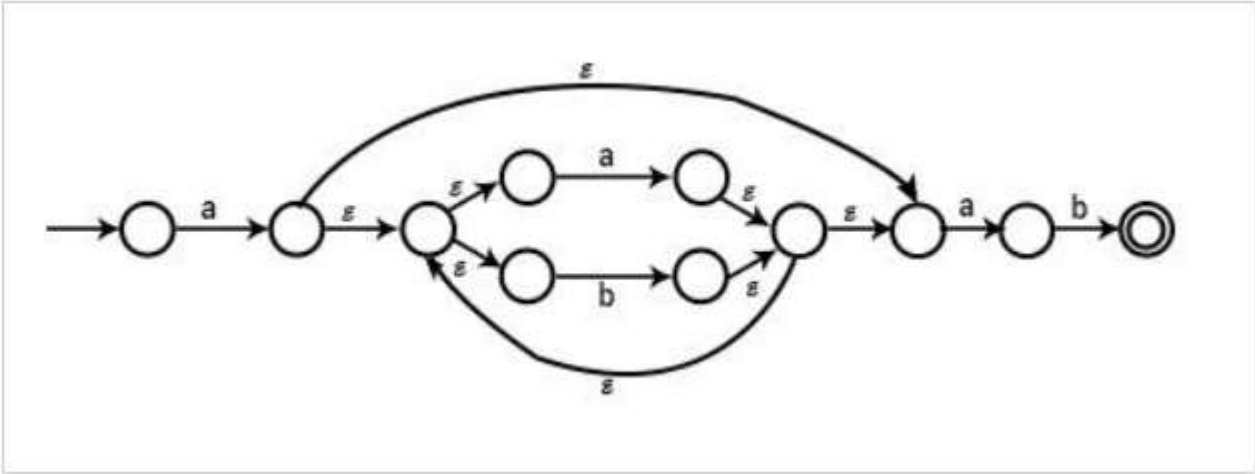
For ab, NFA is



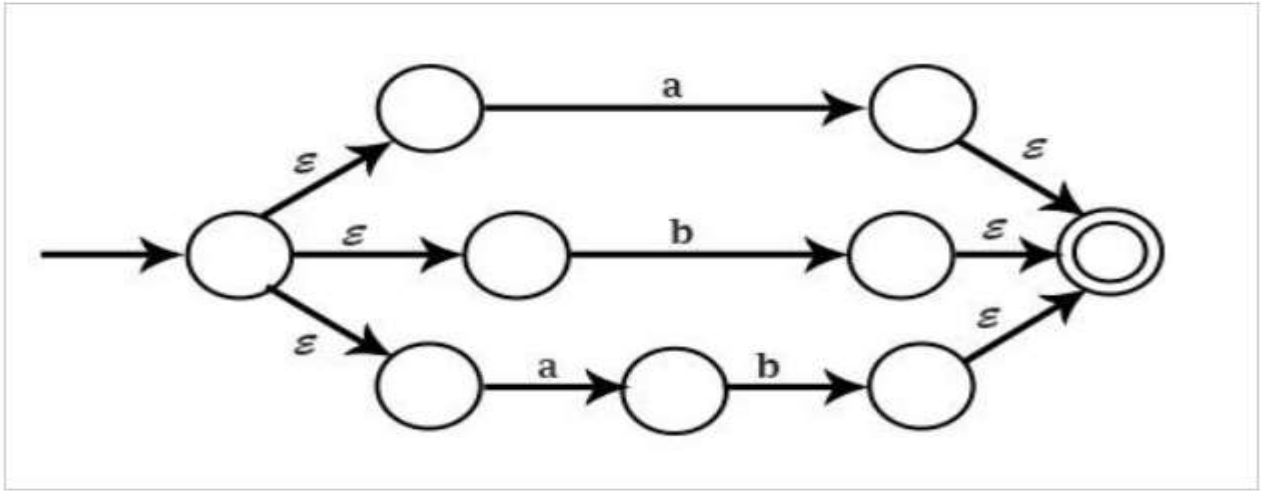
For a*, NFA is



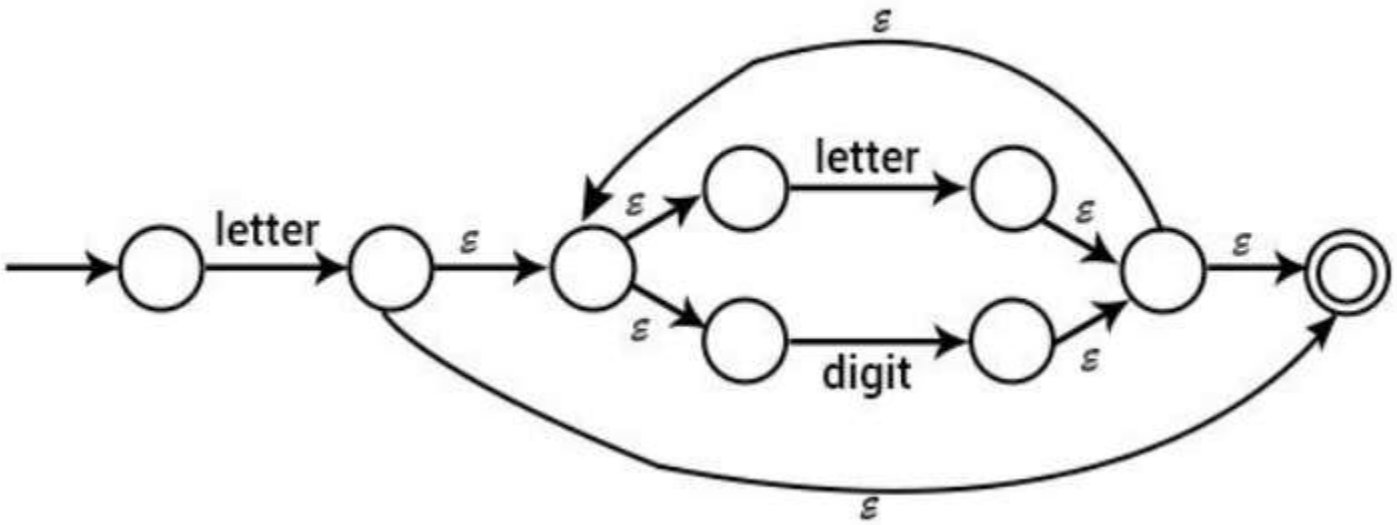
Example1 – Draw NFA for the Regular Expression a(a|b)*ab
Solution



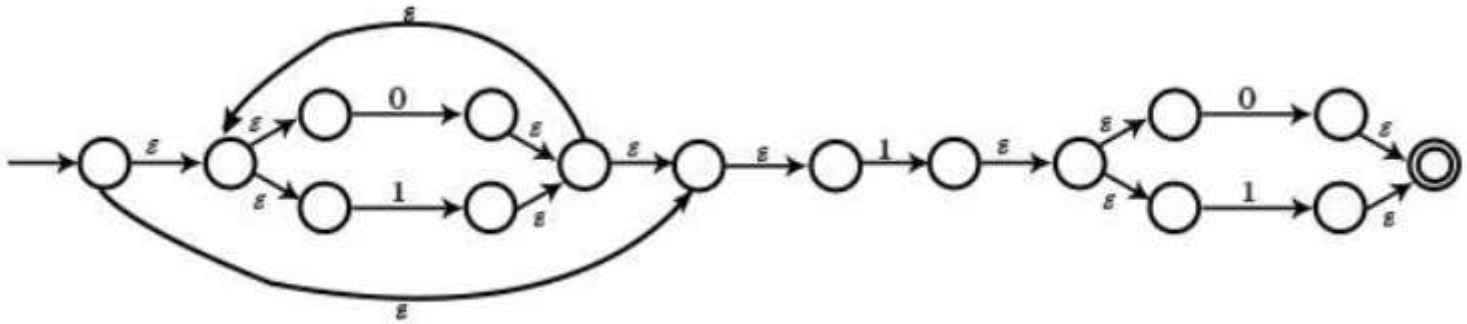
Example2 – Draw NFA for a | b | ab
Solution



Example3 – Draw NFA for letter (letter|digit)*
Solution

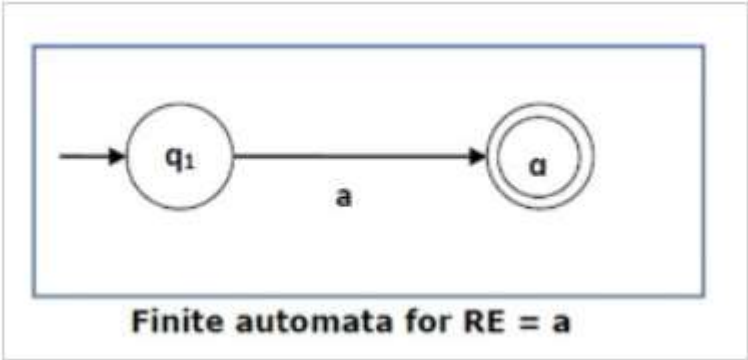


Example4 – Draw NFA corresponding to (0|1)*1(0|1)
Solution

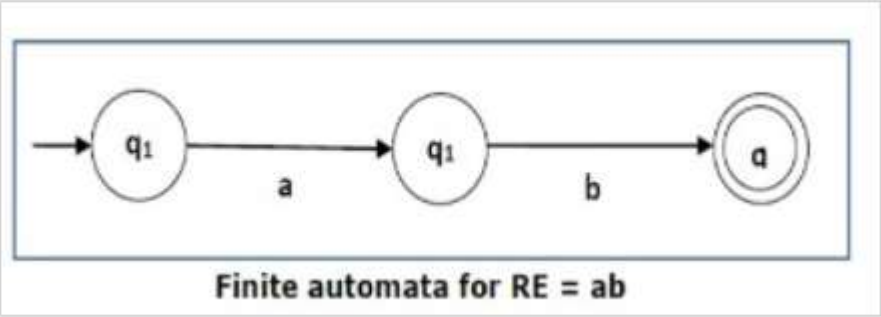


Step2: Convert NFA with ε to NFA without ε.

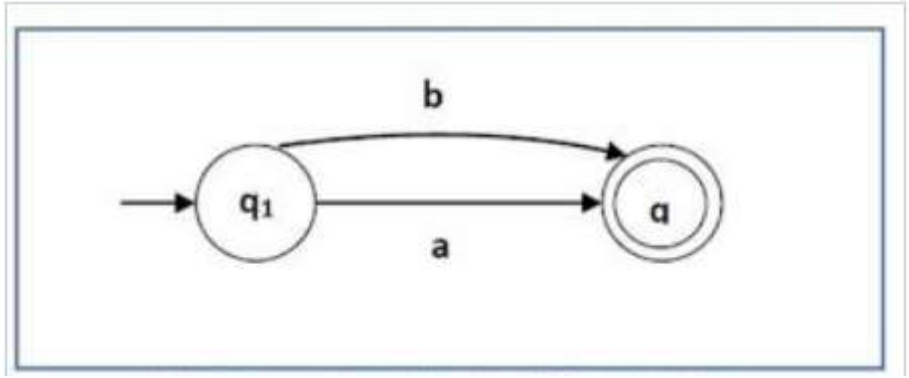
Case 1 − For a regular expression ‘a’, we can construct FA as shown below −



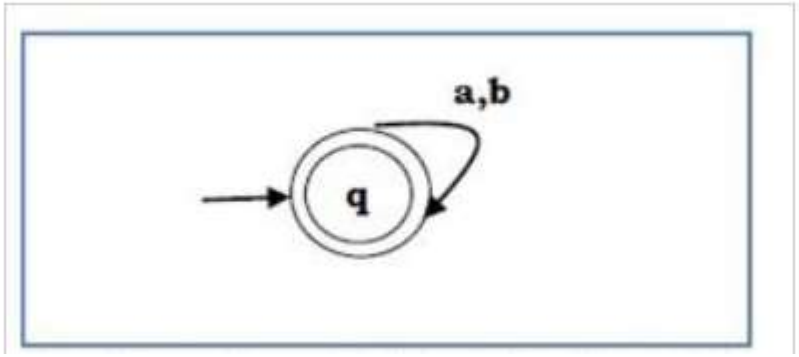
Case 2 − For a regular expression ‘ab’ we can construct FA, as given below −



Case 3 − For a regular expression (a|b) we can construct FA as follows −



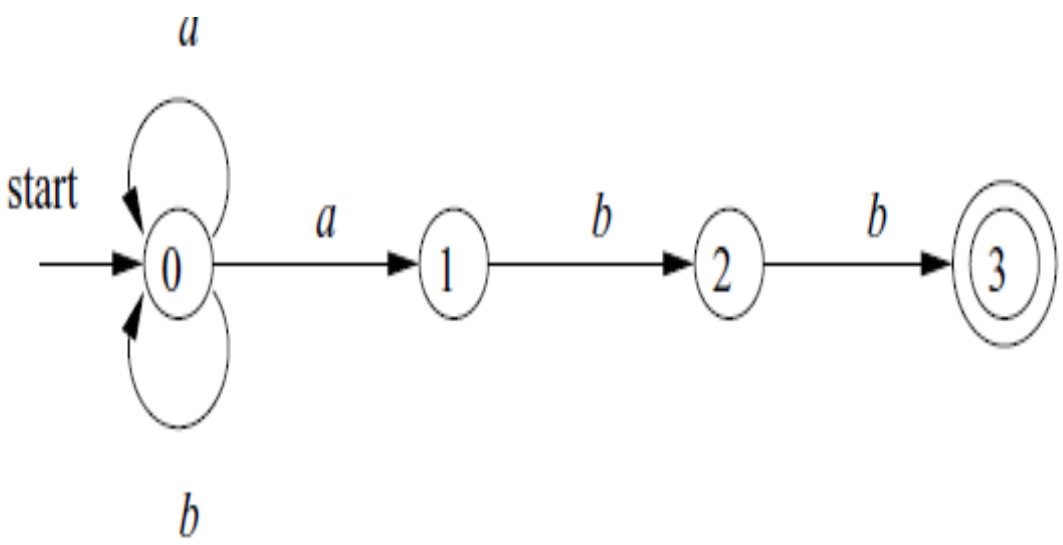
Case 4 − For a RE (a|b)*, We can construct FA as mentioned below −



EX: The transition graph for an NFA recognizing the language of regular expression (a | b)abb

STATE	a	b
0	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

Transition table for the NFA

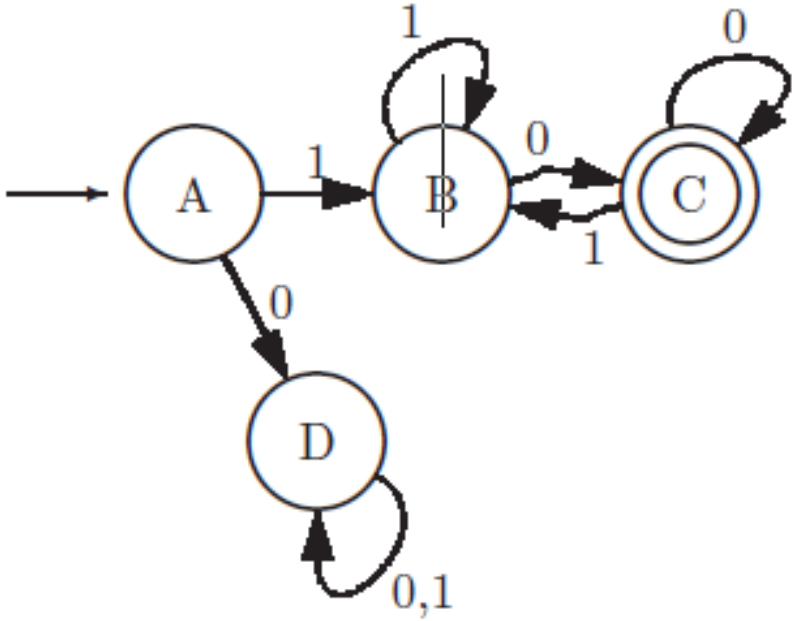


Deterministic Finite Automata (DFA)

- **Deterministic Finite Automata(DFA)**
- A DFA is a special kind of finite automaton that has exactly one transition out of each state for each input symbol. Also, transitions on empty input are disallowed. The DFA is easily simulated and makes a good implementation of a lexical analyzer, similar to a transition diagram.
- A deterministic finite automaton (DFA) is a special case of an NFA
  1. There are no moves on input ε
  2. For each state s and input symbol a, there is exactly one edge out of s labeled
- If we are using a transition table to represent a DFA, then each entry is a single state. We may therefore represent this state without the curly braces that we use to form sets.

Finite State Machines

1. Q = {A,B,C,D}
2. ∑ = alphabet = {0,1}
3. δ = state diagram / table
4. Q_0= First state={A}
5. F= Final or accepted state= {C}

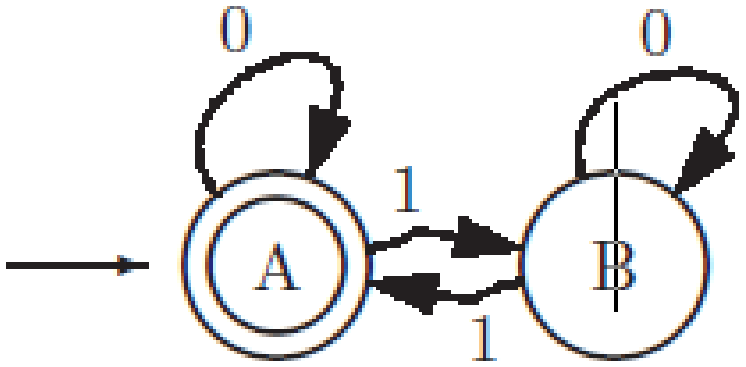


This machine accepts any string of zeroes and ones which begins with a one and ends with a zero,

	0	1
A	D	B
B	C	B
*C	C	B
D	D	D

Finite State Machines

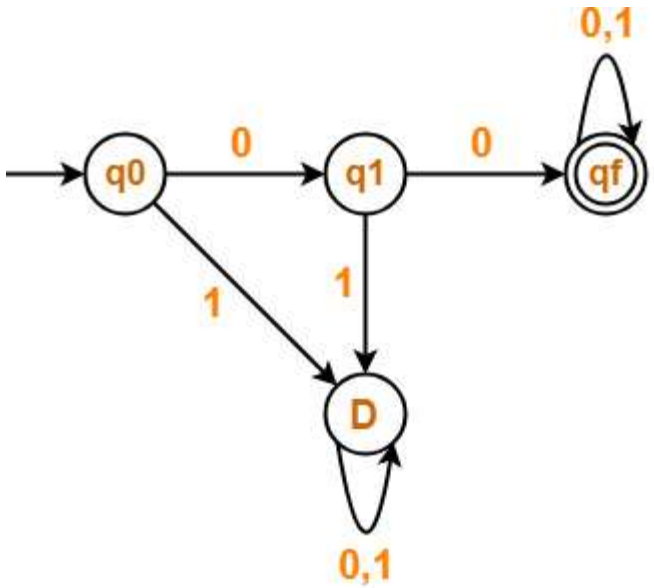
1. Q = {A,B}
2. ∑ = alphabet = {0,1}
3. **δ** = State diagram / table
4. Q_o= First state={A}
5. F= Final or accepted state= {A}



This machine accepts any string of zeroes and ones which contains an even number of ones . Such a machine is called a parity checker.

	0	1
*A	A	B
B	B	A

- EX3: Draw a DFA that accepts a language L over input alphabets ∑ = {0, 1} such that L is the set of all strings starting with ’00’.
- **Solution-**
- Regular expression for the given language = 00(0 | 1)*
- **Step-01:**
- All strings of the language starts with substring “00”. So, length of substring = 2.
- Thus, Minimum number of states required in the DFA = 2 + 2 = 4.
- It suggests that minimized DFA will have 4 states.
- **Step-02:**
- We will construct DFA for the following strings-
- 00,000, 00000
- **Step-03:**
- The required DFA is-



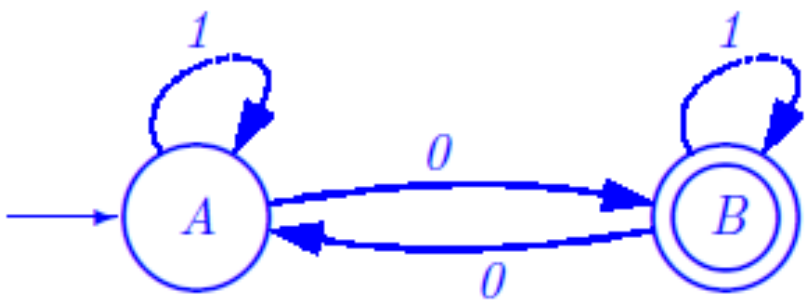
DFA

Example

Show a finite state machine, in either state graph or table form, for each of the following languages (in each case the input alphabet is {0,1}):

1. *Strings containing an odd number of zeros*

Solution for #1

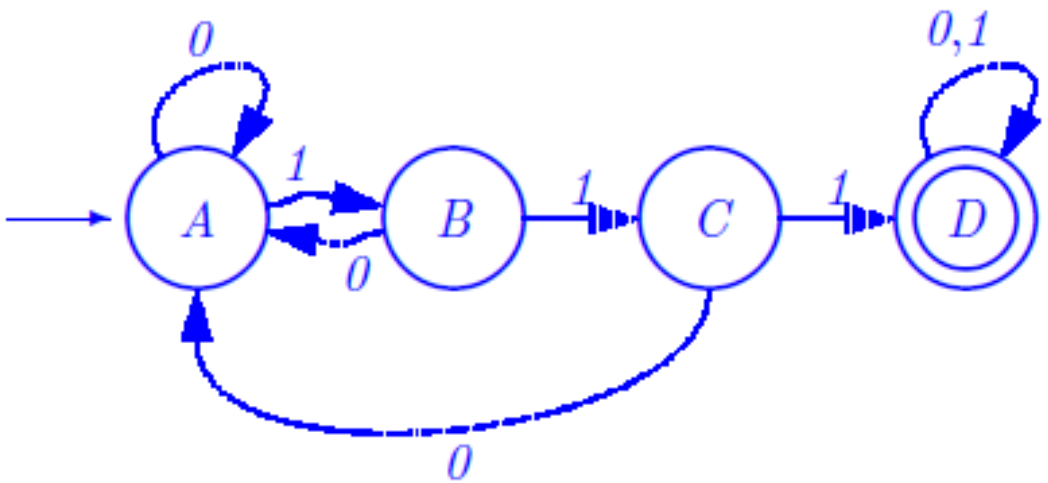


	0	1
A	B	A
*B	A	B

Example

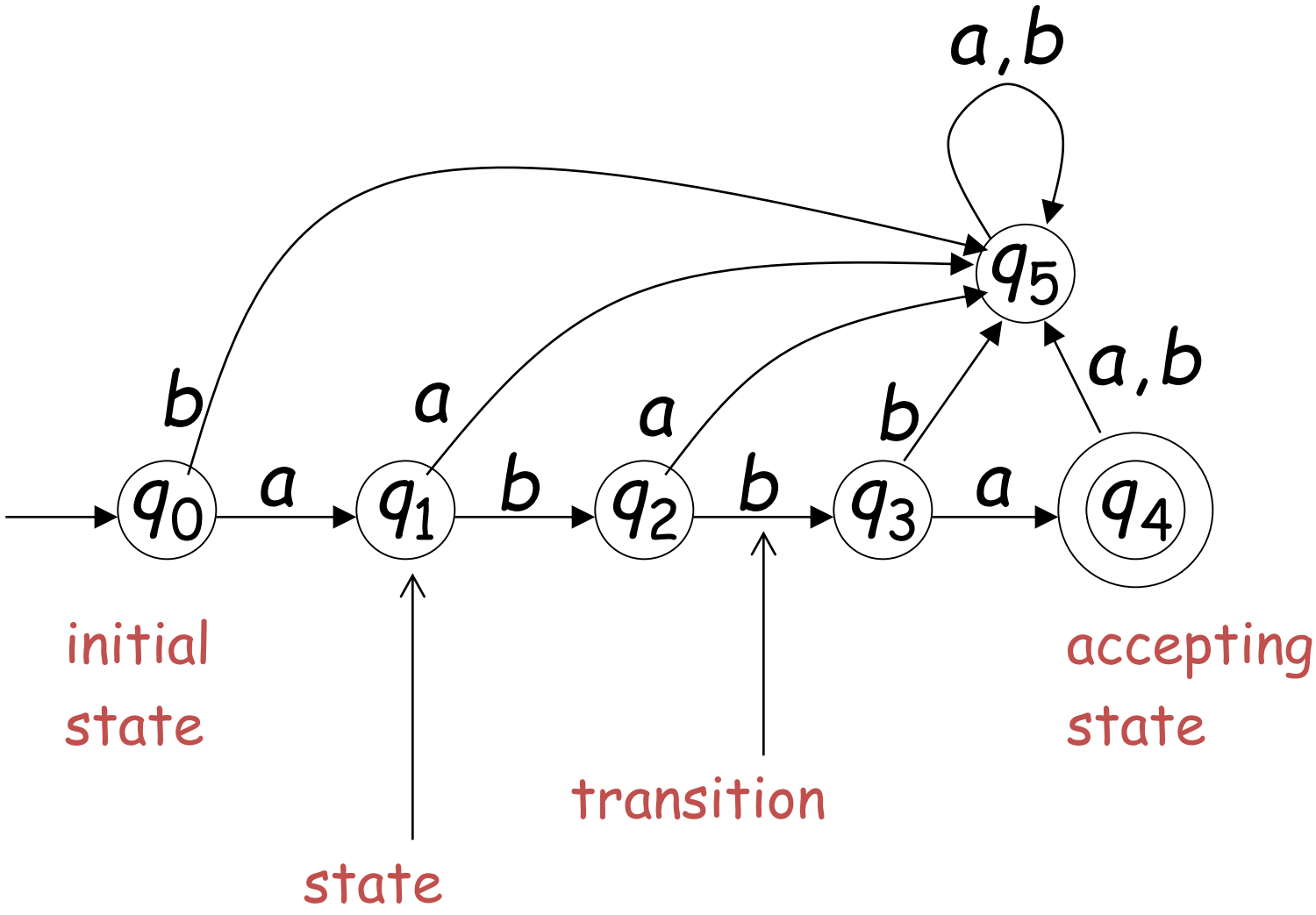
2. Strings containing three consecutive ones

Solution for #2



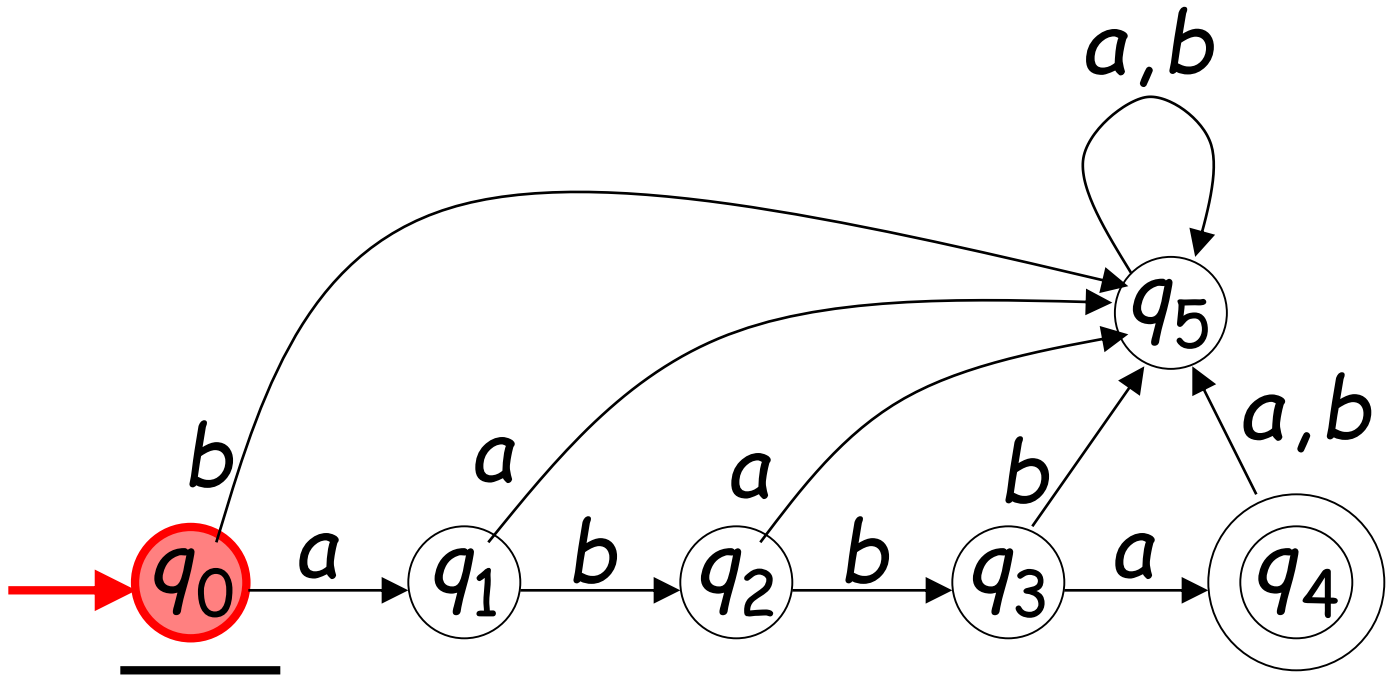
	0	1
A	A	B
B	A	C
C	A	D
*D	D	D

Example

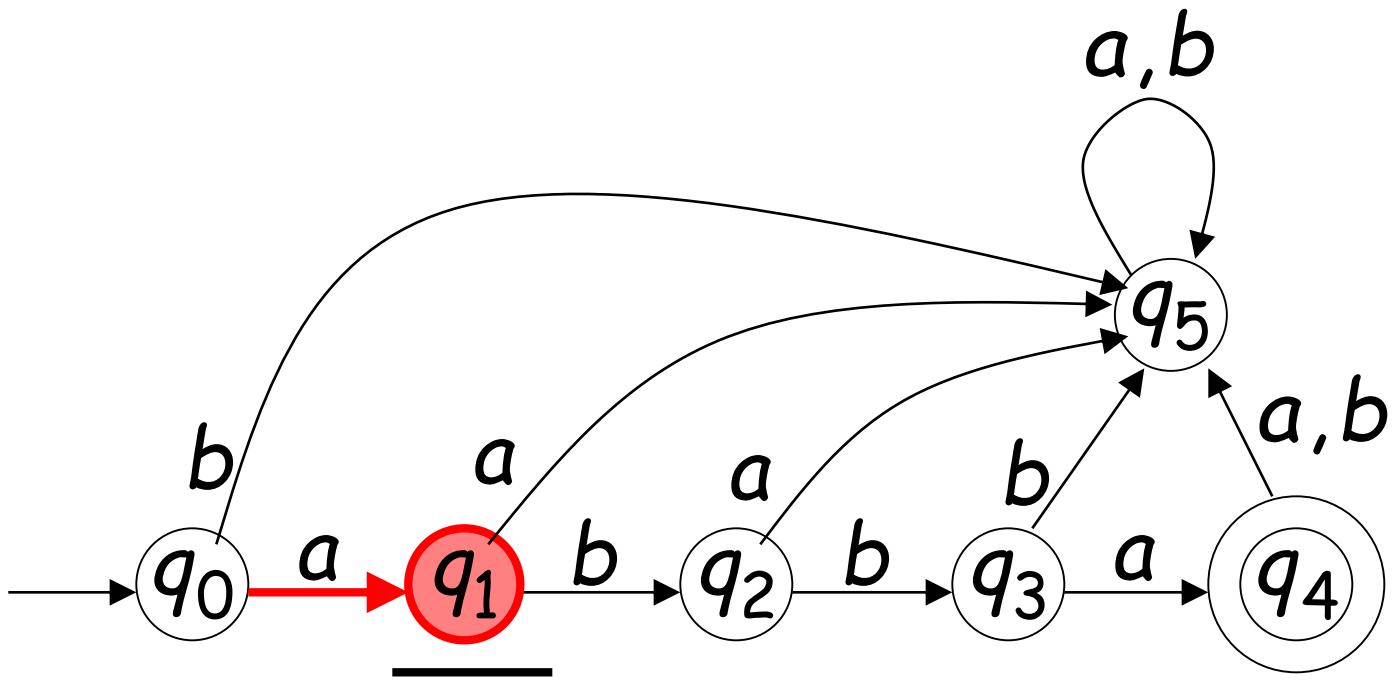

a, b
b
a
a
b
a, b
a
b
b
a
q_0
q_1
q_2
q_3
q_4
q_5
initial state
state
transition
accepting state

Initial Configuration

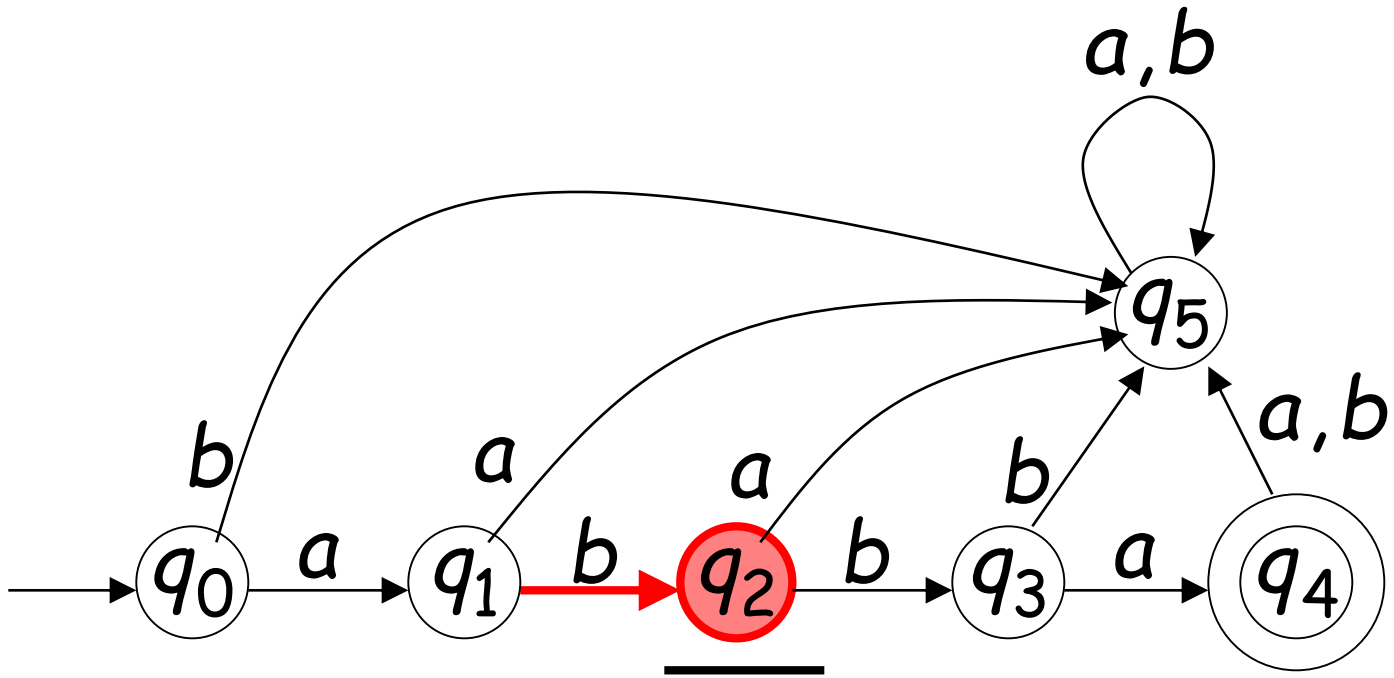
Input String



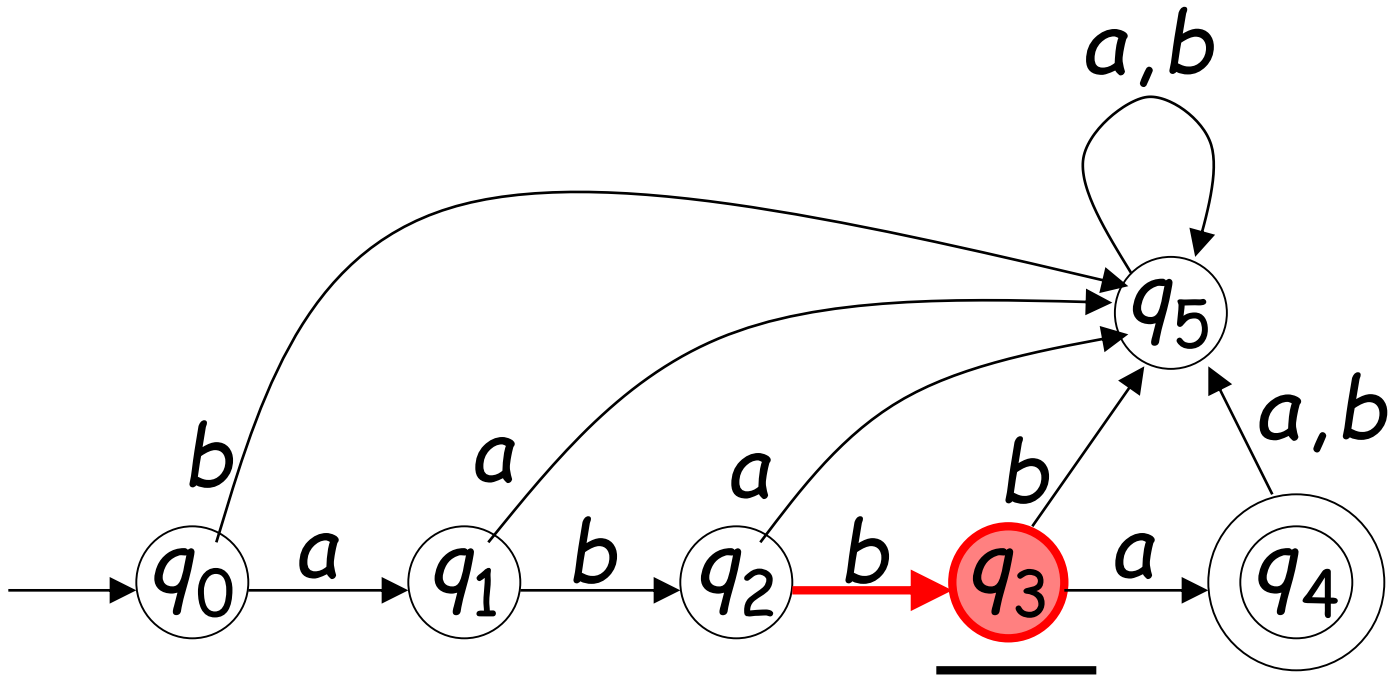
Reading the Input



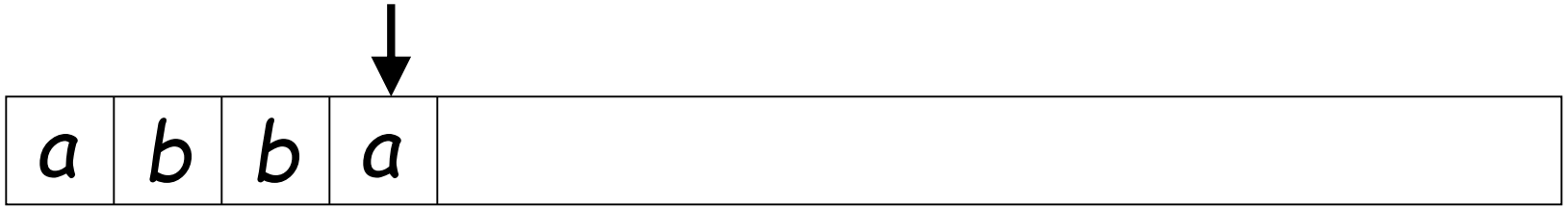
Reading the Input

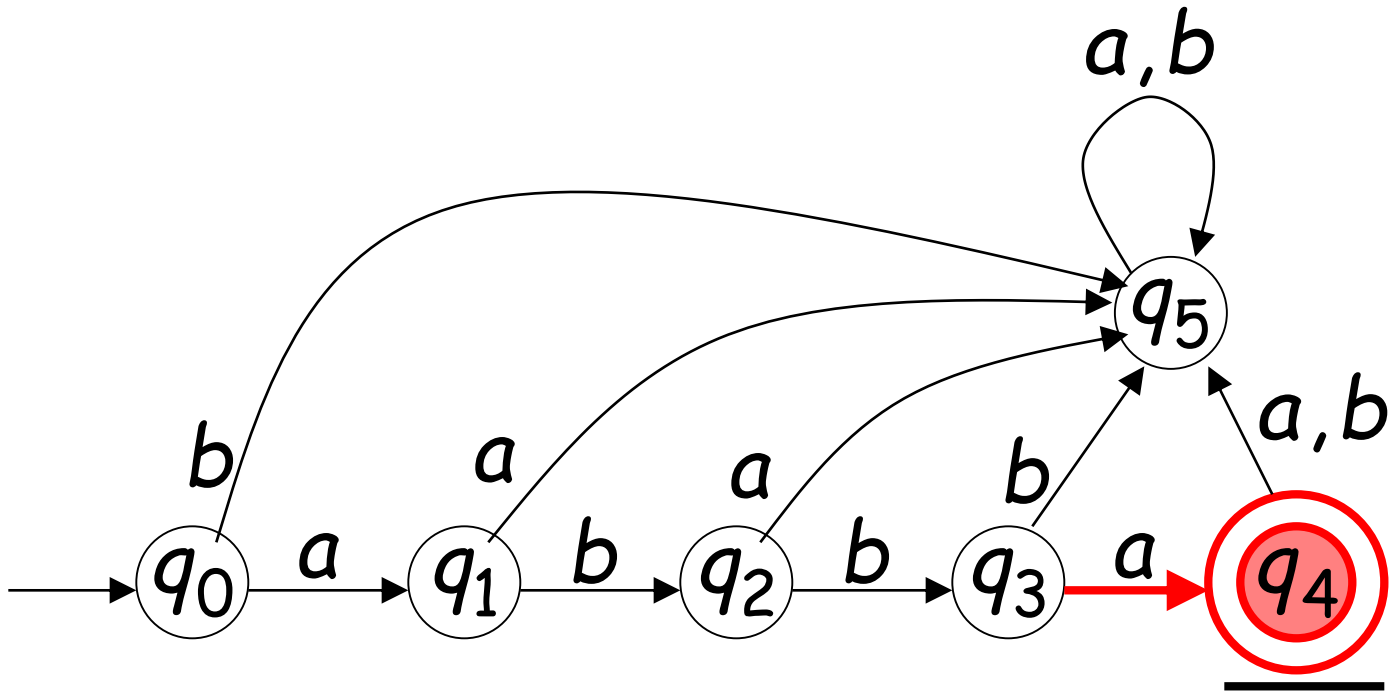


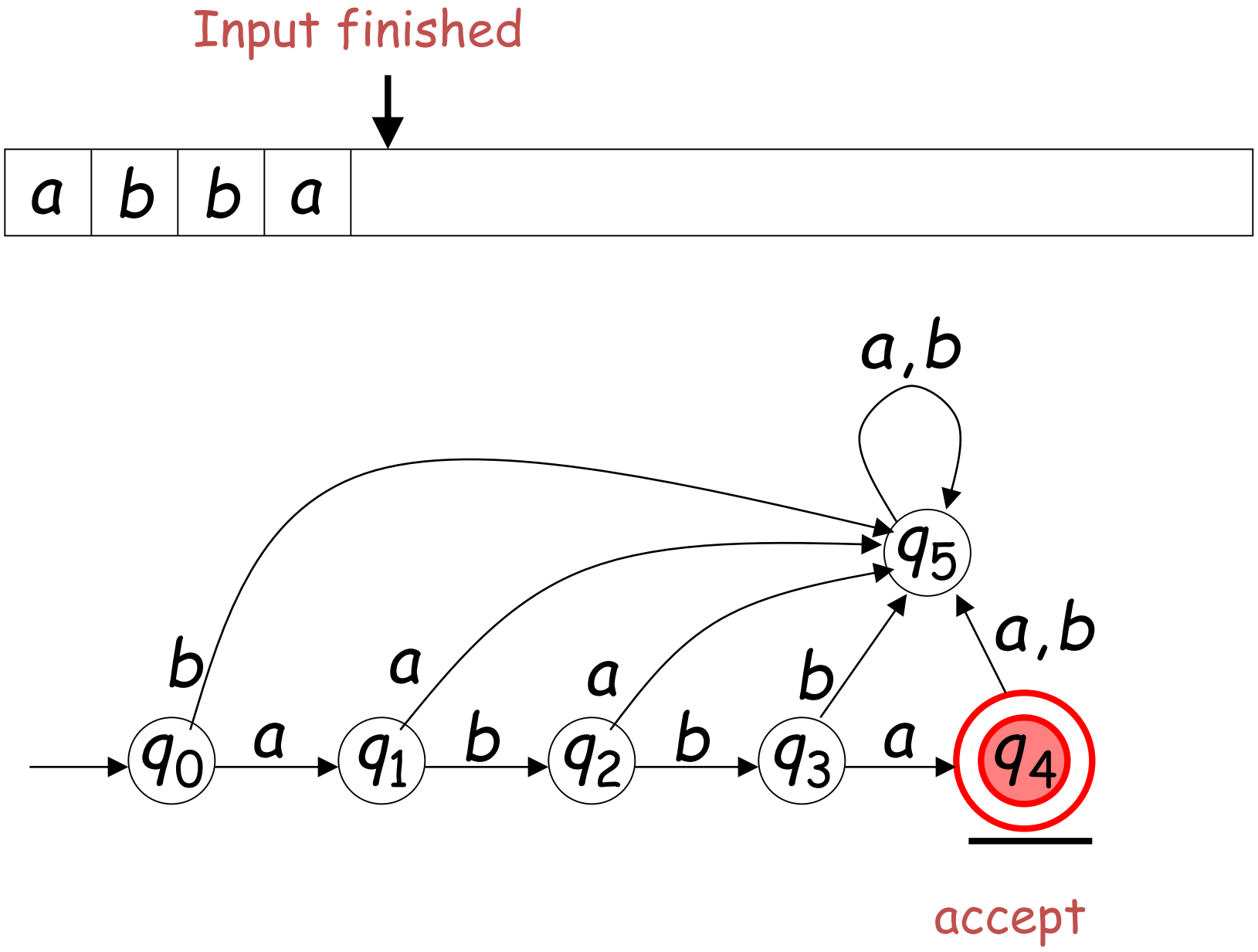
Reading the Input



Reading the Input



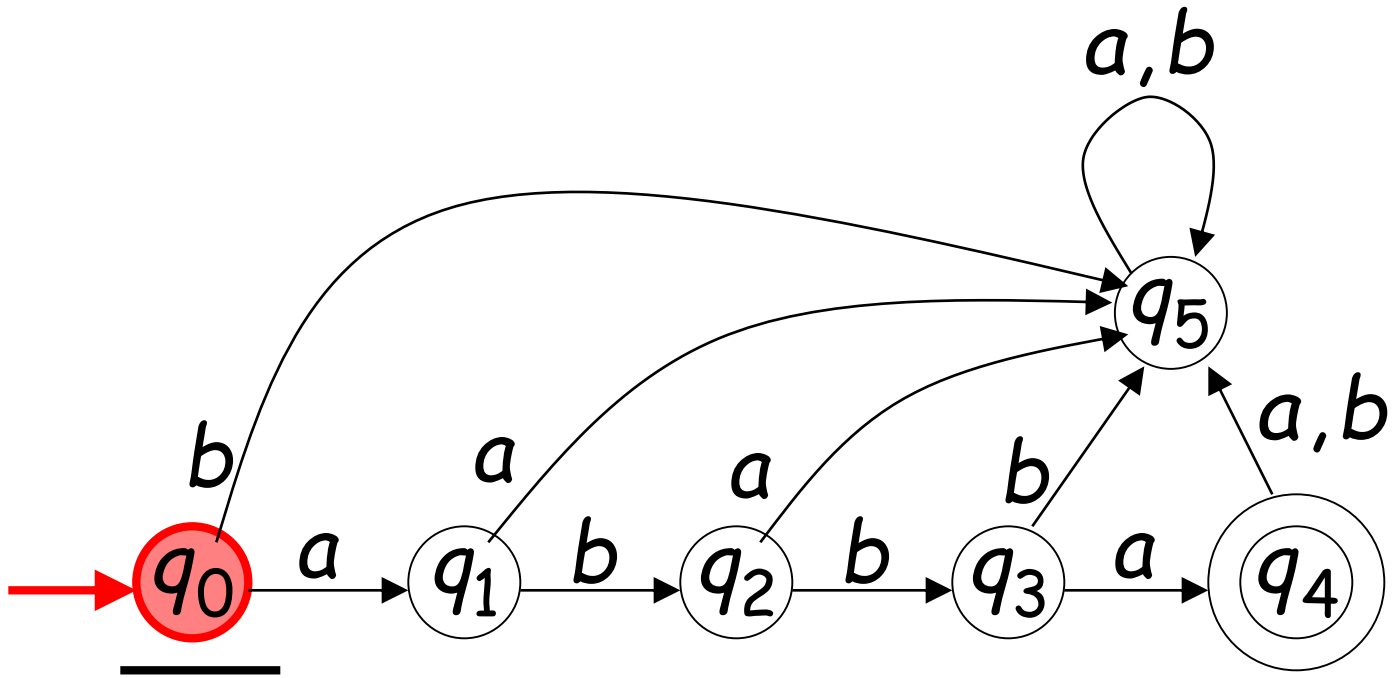


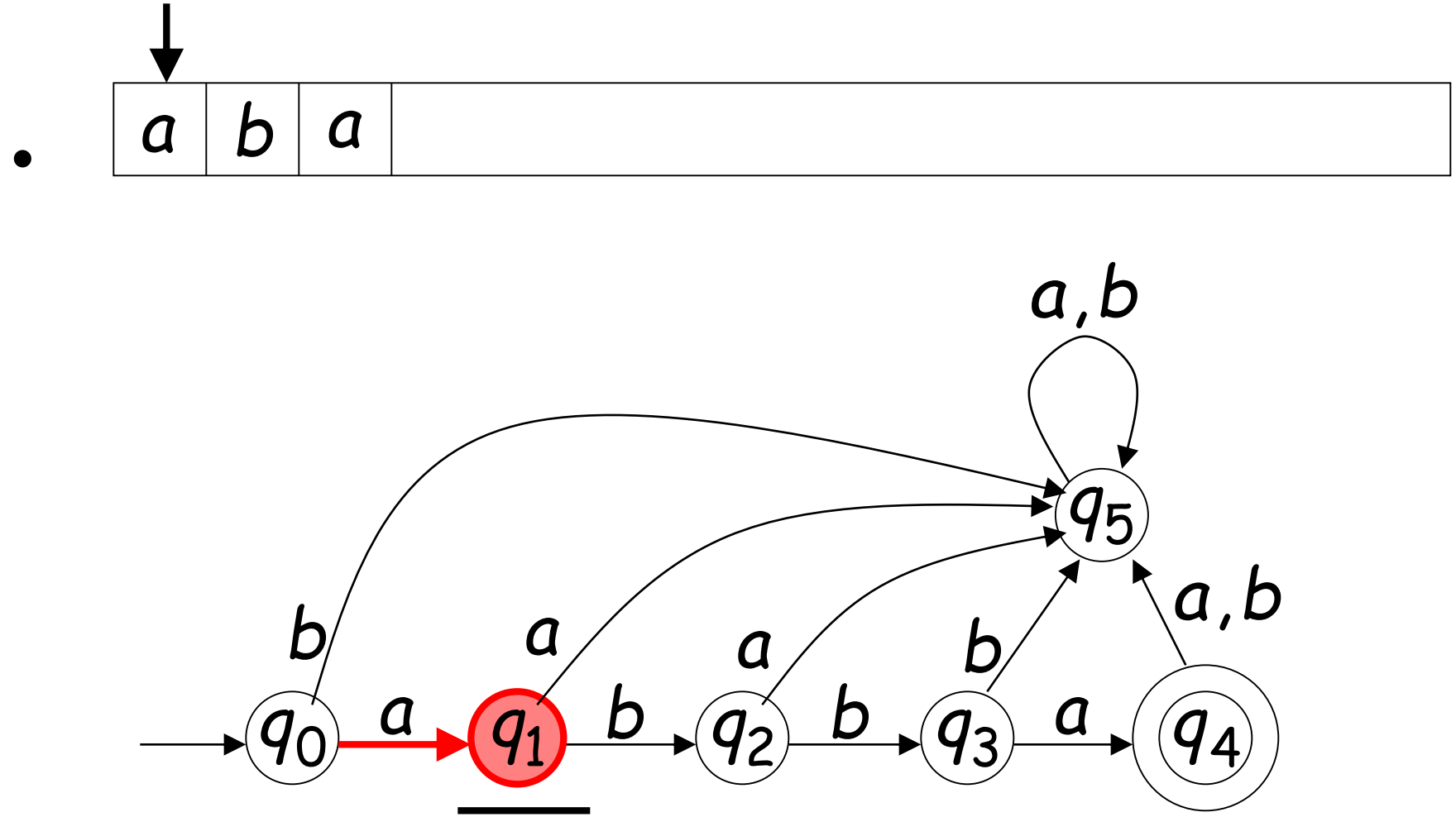

Input finished
a b b a
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4
accept

Another String

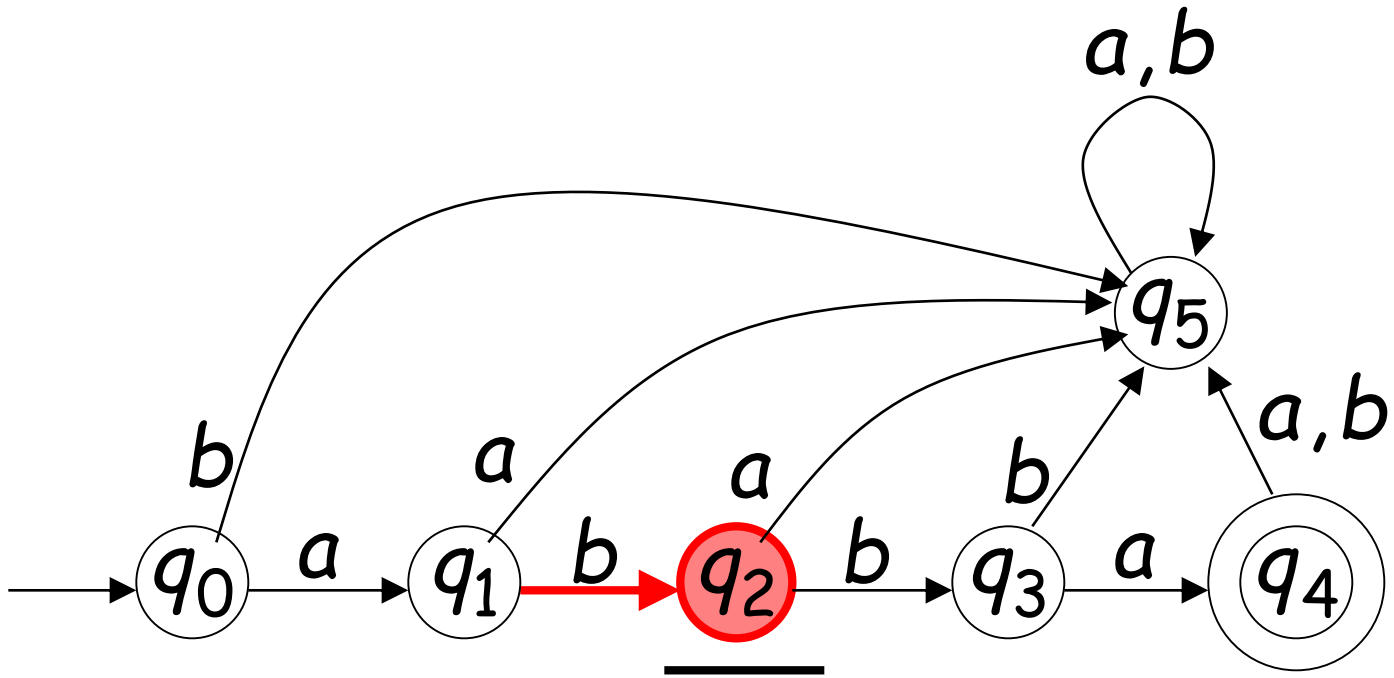
Rejection

a	b	a	

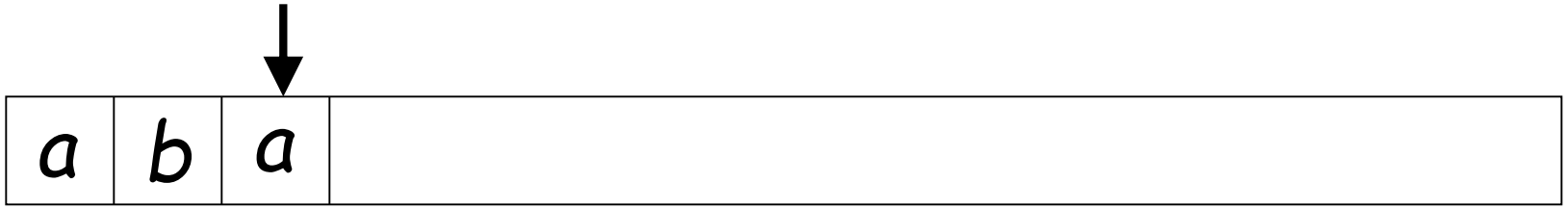


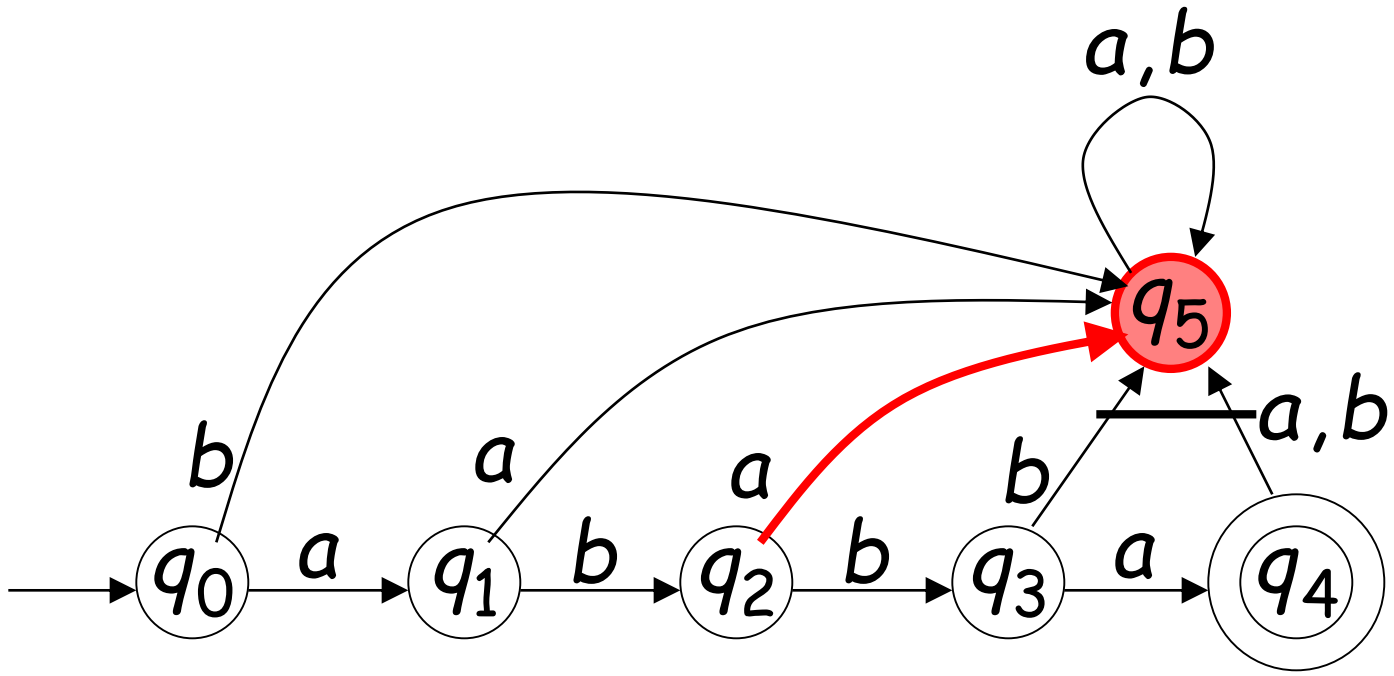

a
b
a
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4

Reading the Input

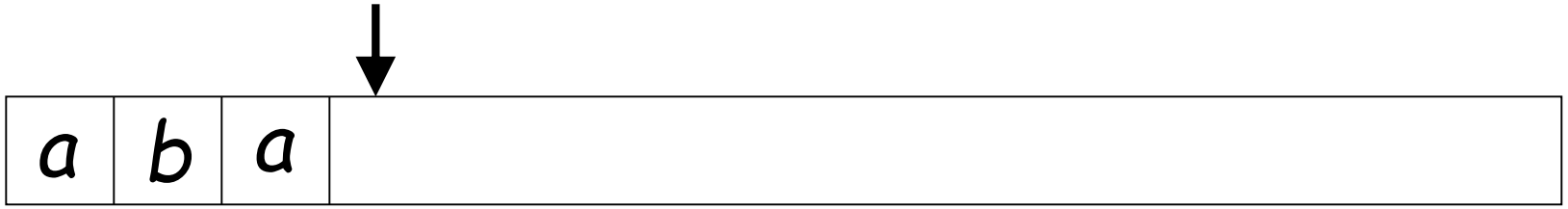


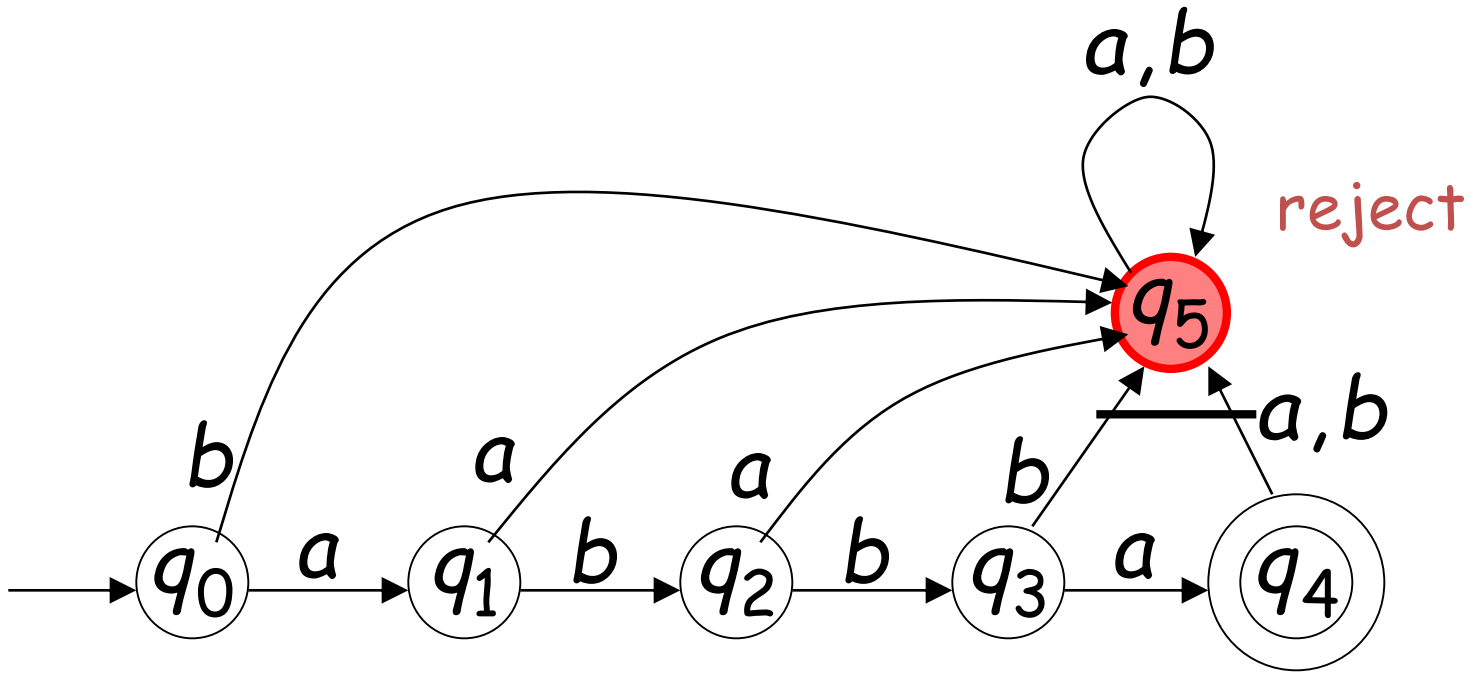
Reading the Input



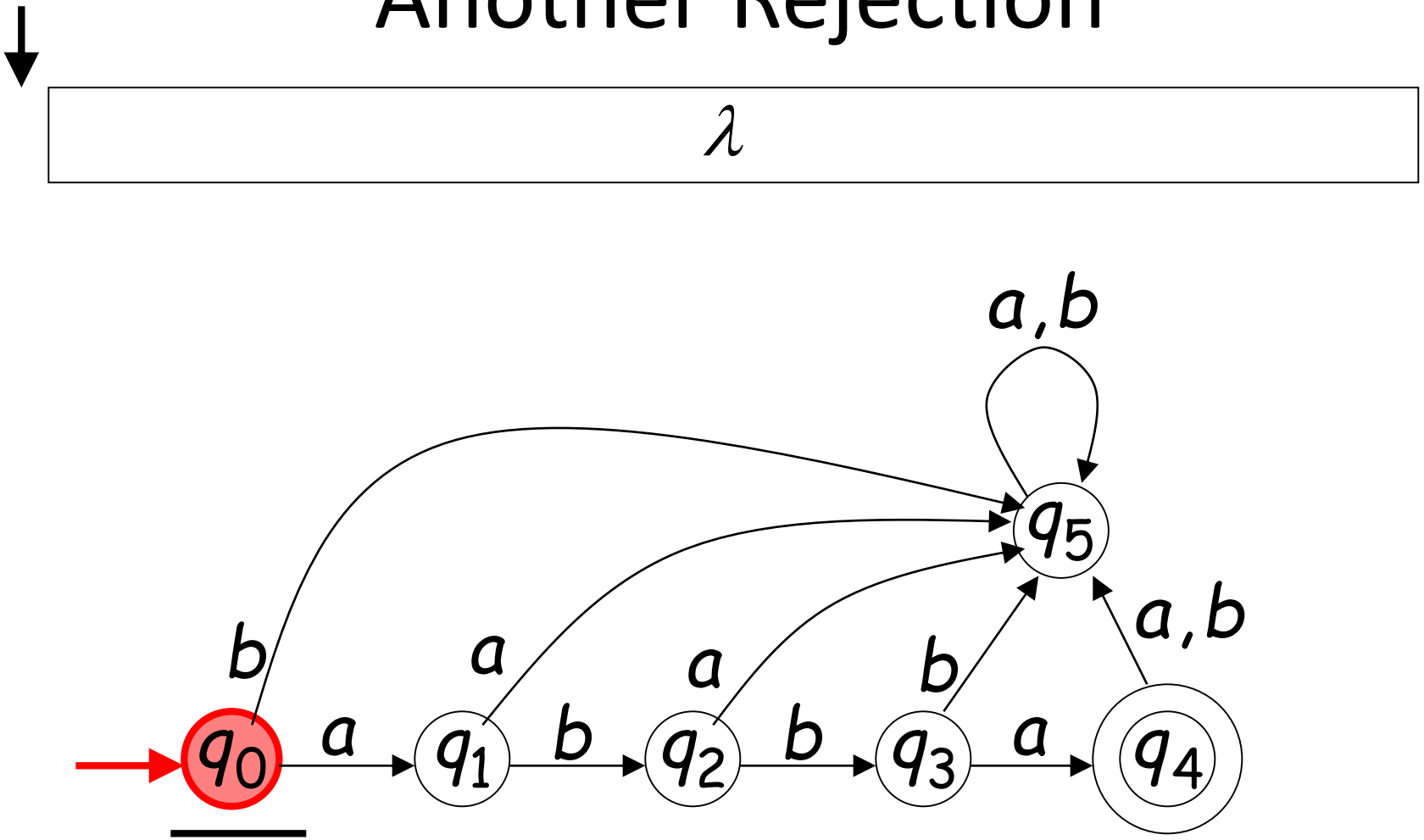


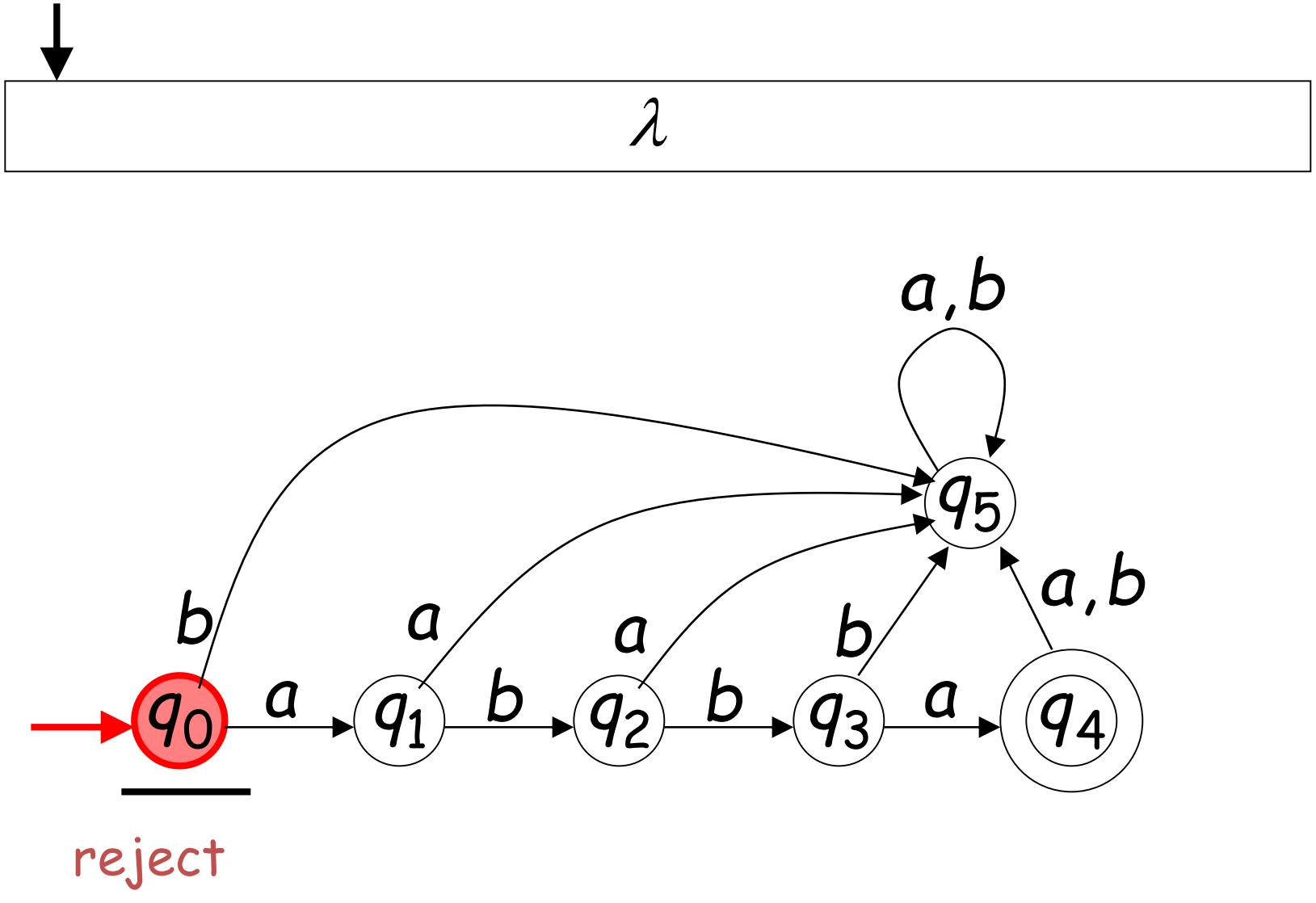
Input finished



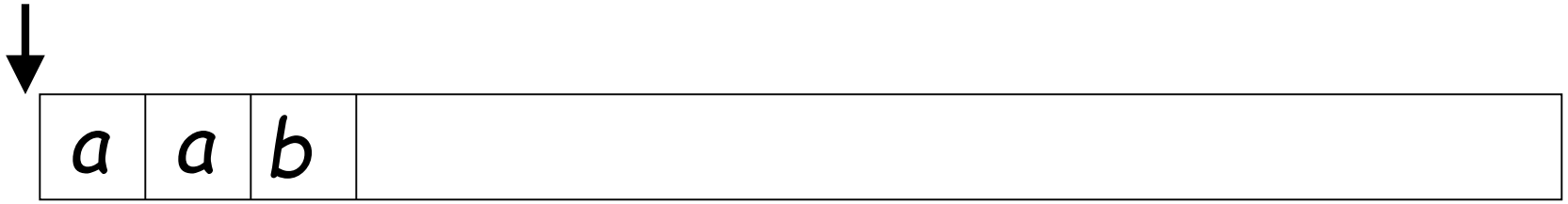


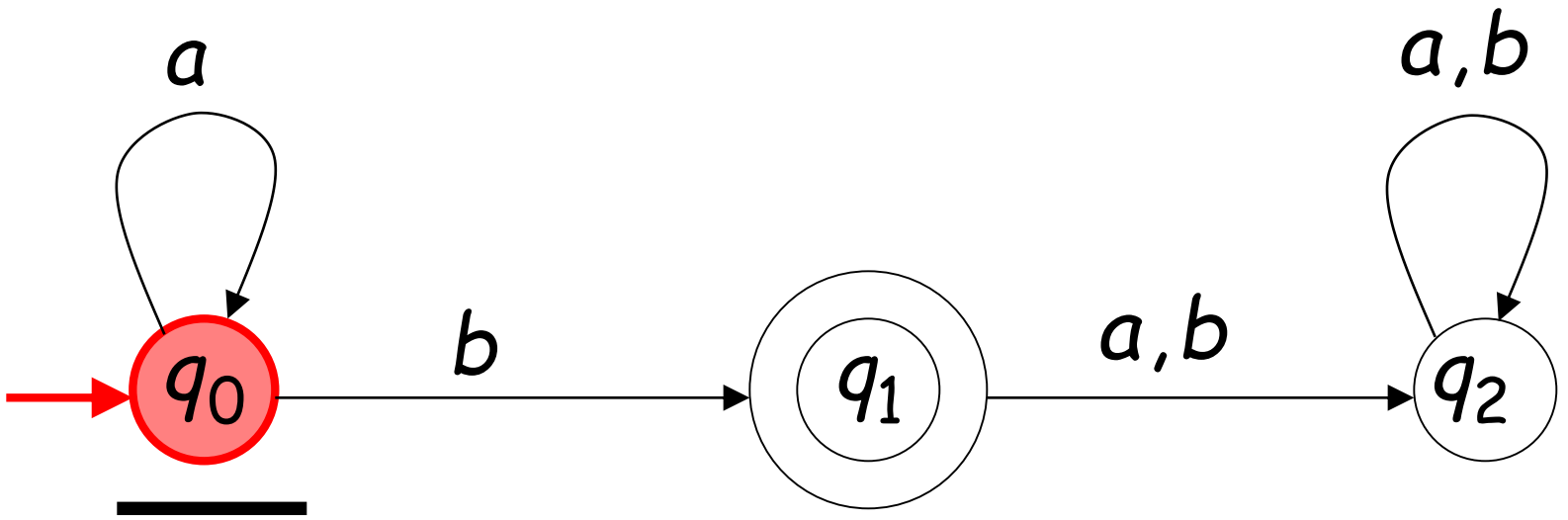
Another Rejection



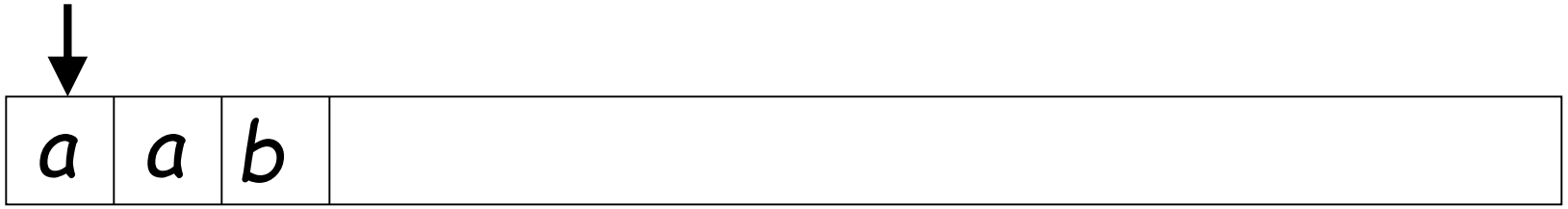

λ
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4
reject

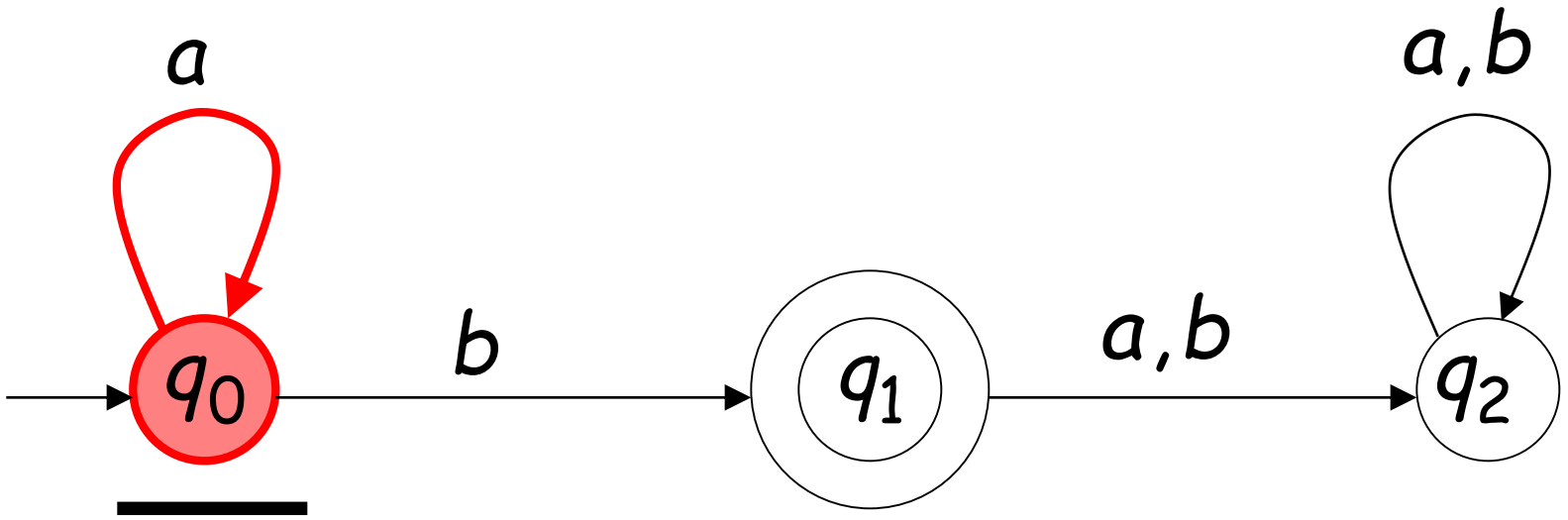
Another Example



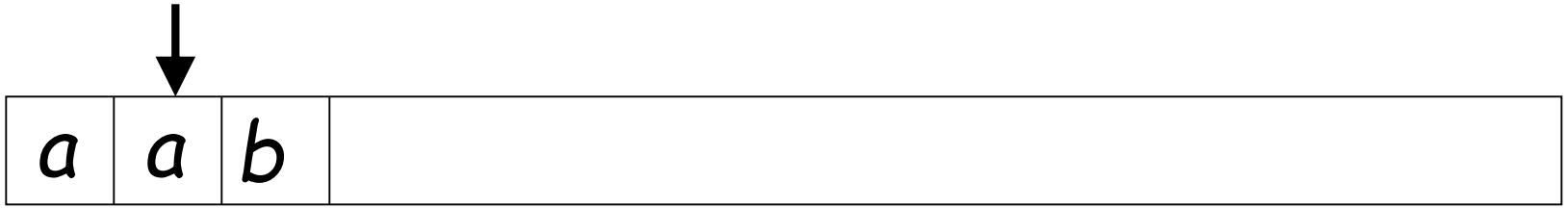


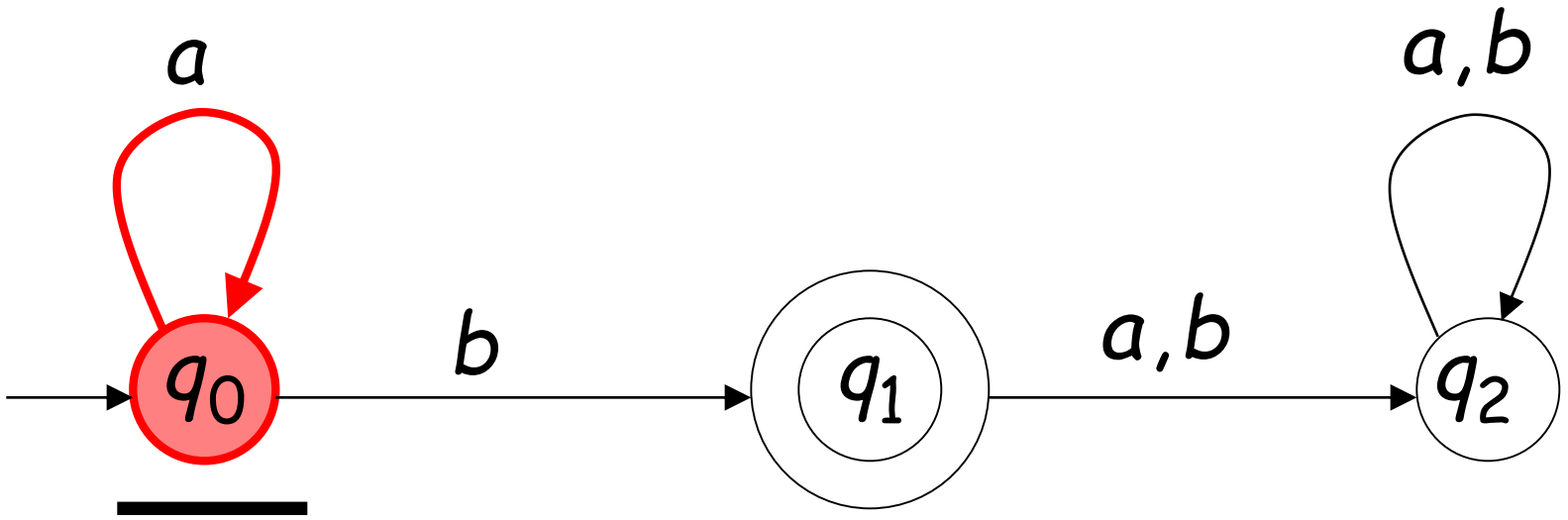
Reading the Input



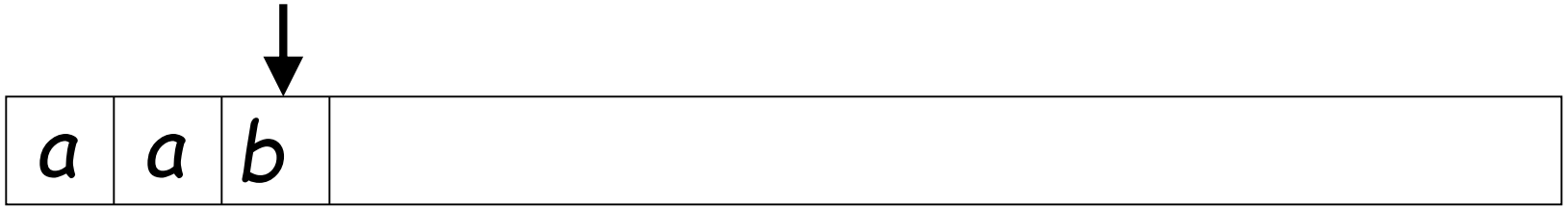


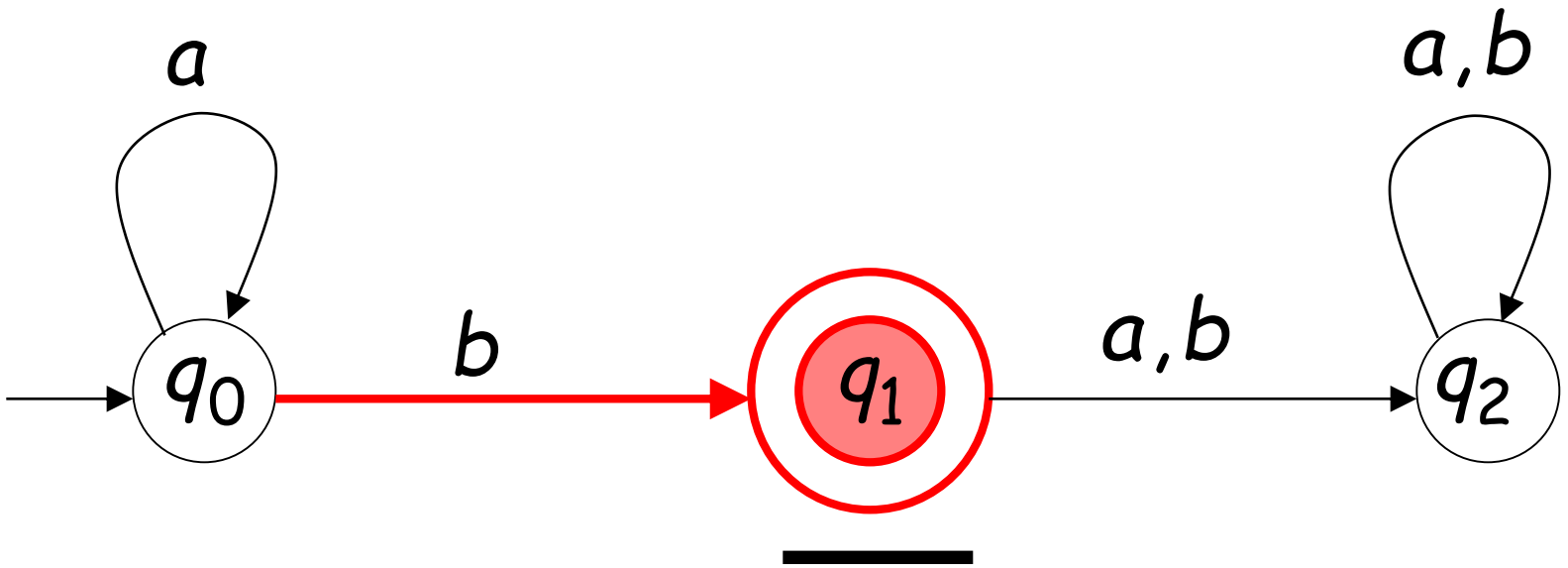
Reading the Input

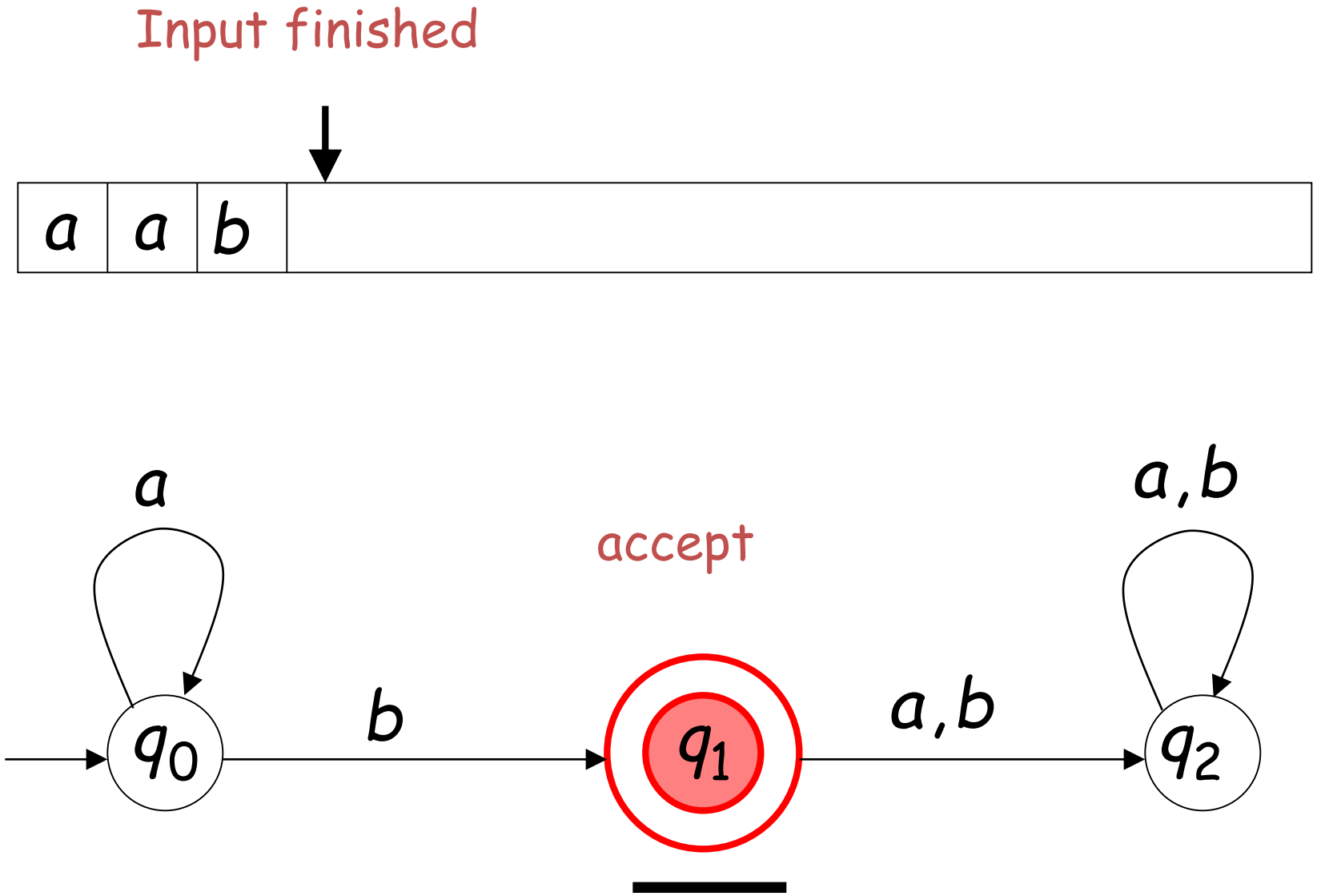




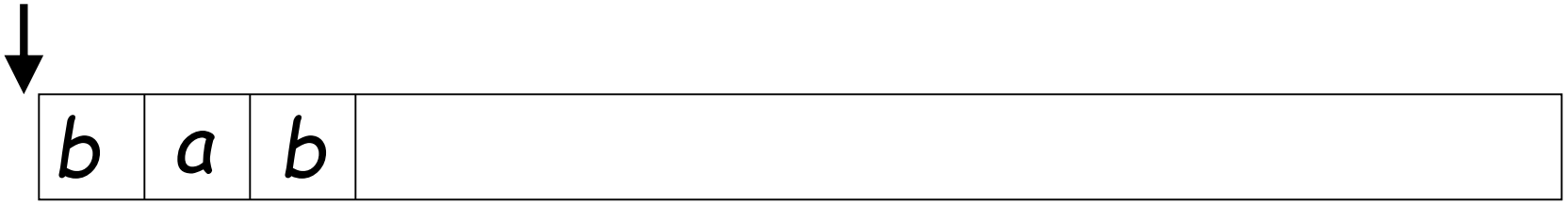
Reading the Input

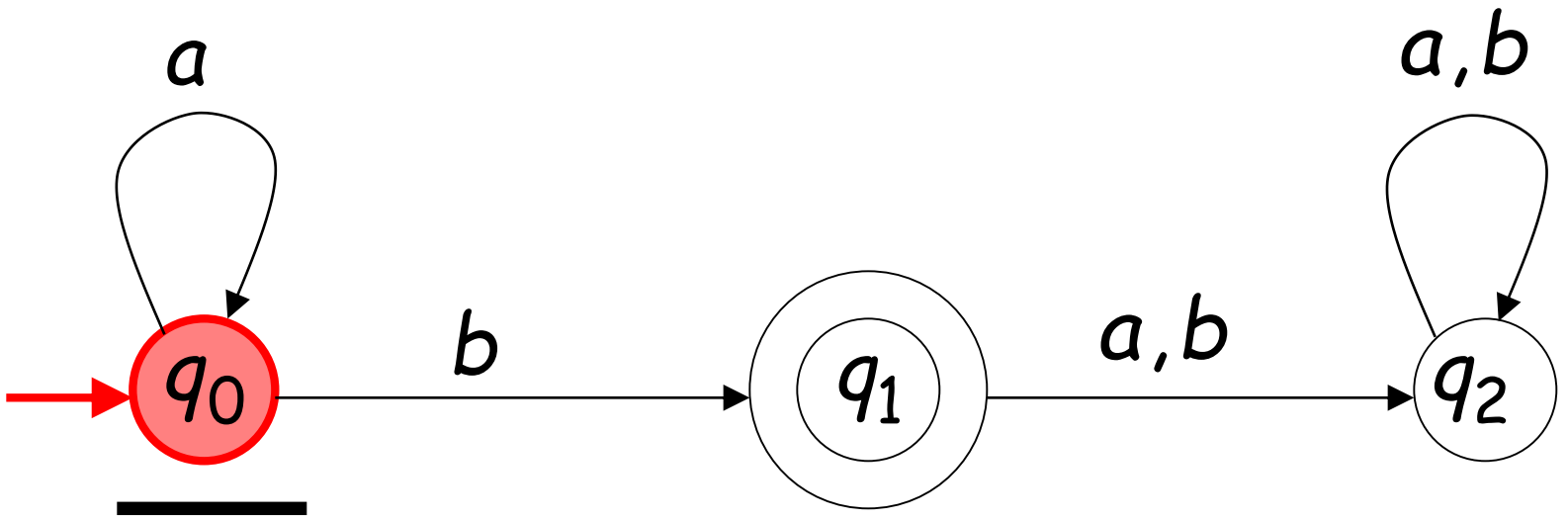




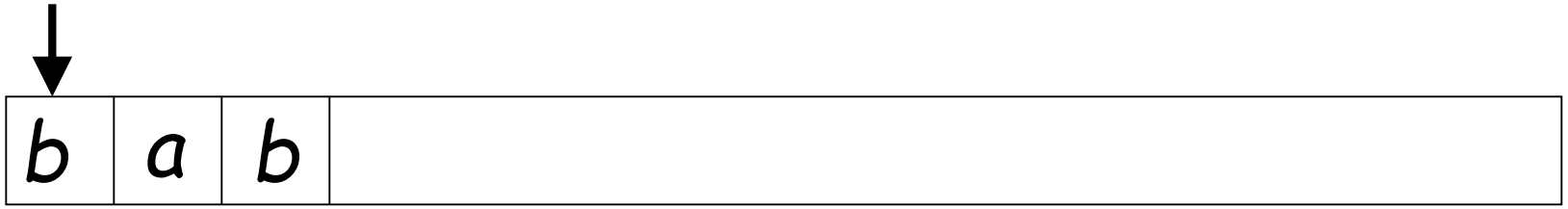

Input finished
a
a
b
a
accept
a,b
b
a,b
q_0
q_1
q_2

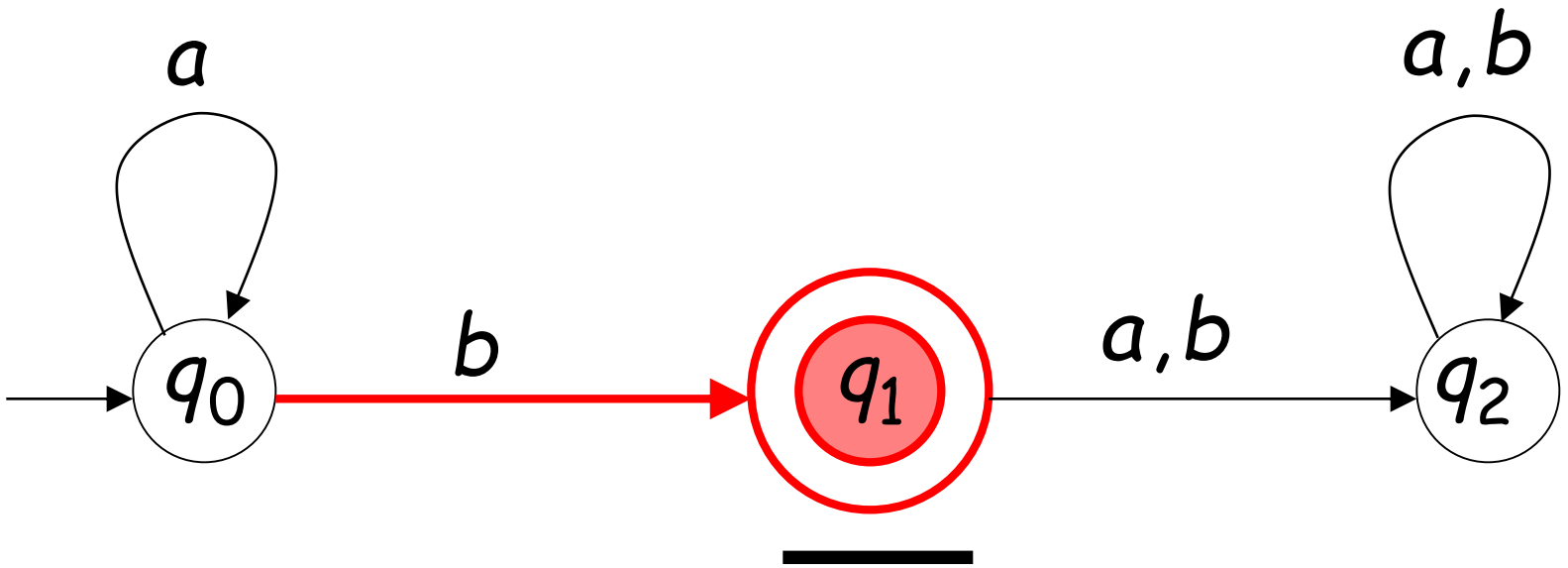
Rejection Example



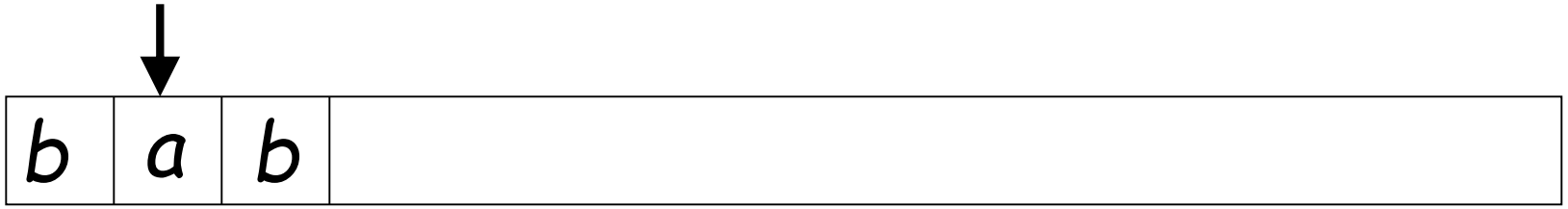


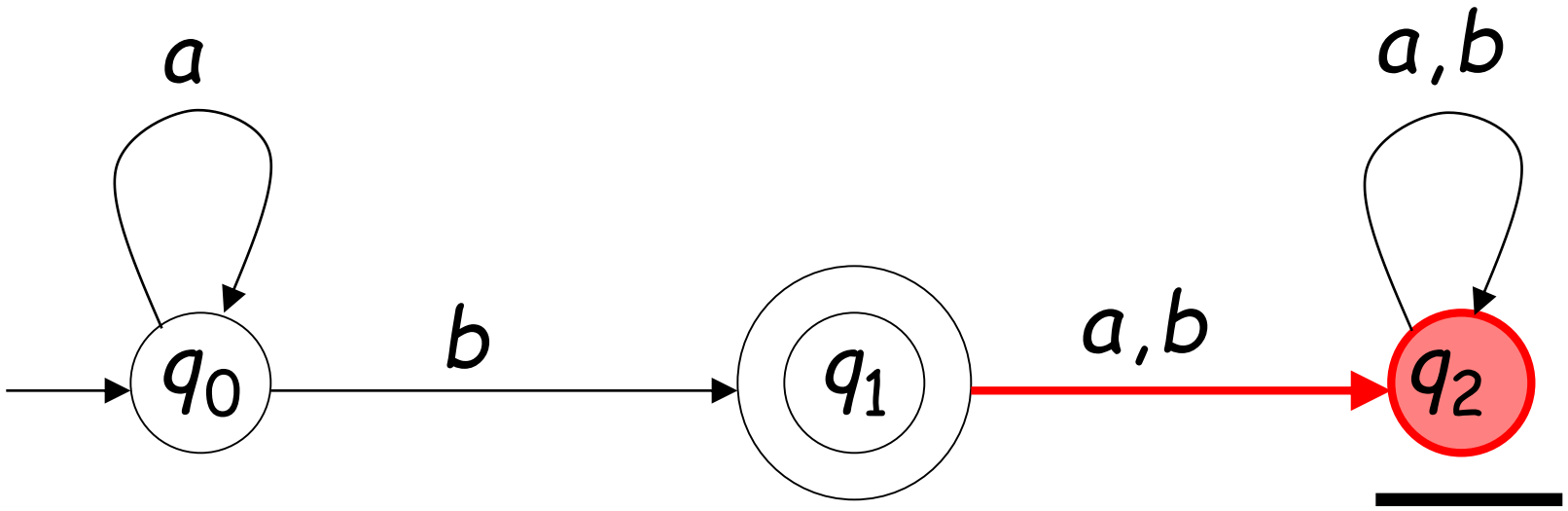
Reading the Input



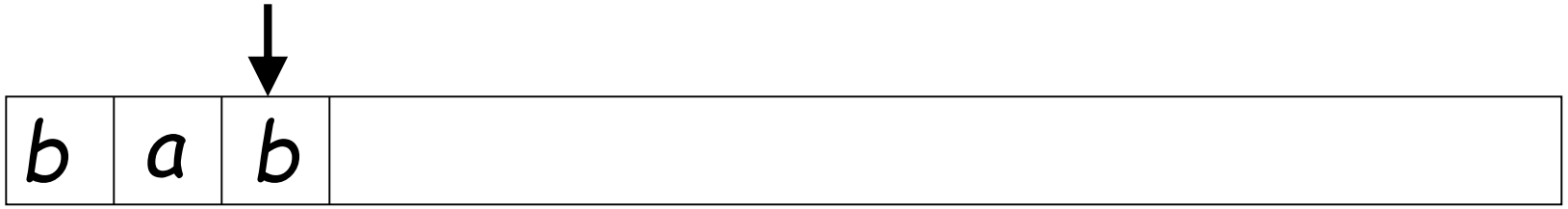


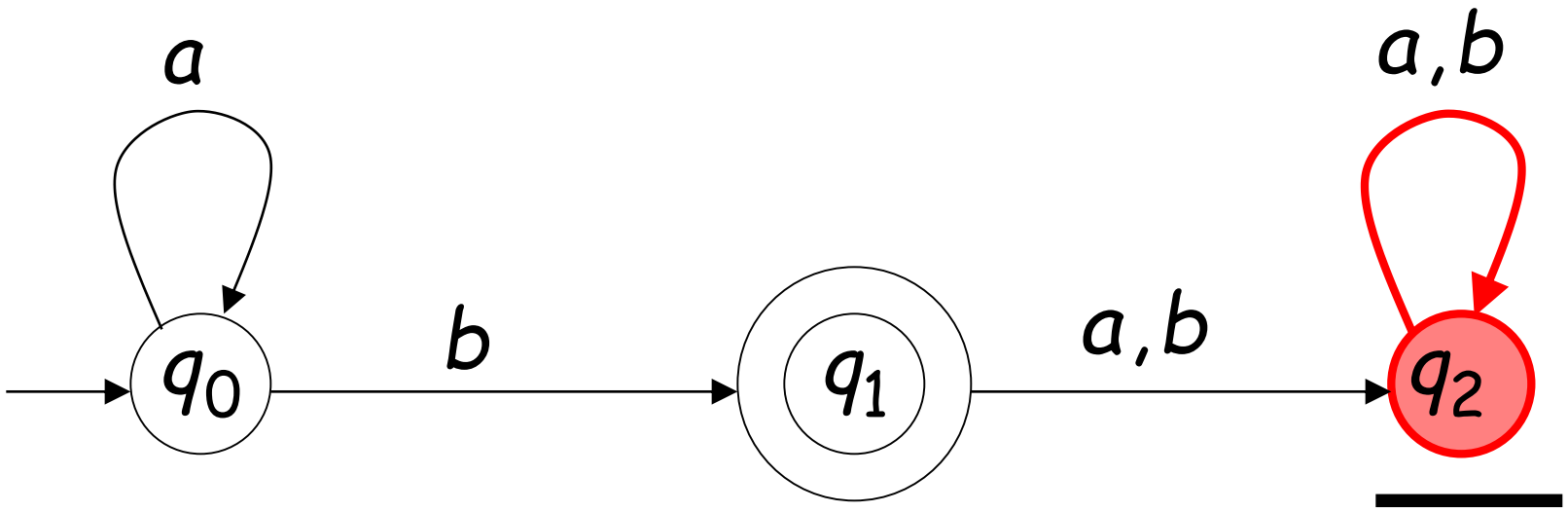
Reading the Input





Reading the Input





Input finished