Writing a Lexical Analyzer in Haskell

Today

- (Finish up last Thursday) User-defined datatypes
- (Finish up last Thursday) Lexicographical analysis for punctuation and keywords in Haskell
- Regular languages and lexicographical analysis part I

This week

- HW2: Due tonight
- PA1: It is due in 6 days!
- PA2 has been posted. We are starting to cover concepts needed for PA2.

User-defined Datatypes in Haskell

Kindof like enumerate types but can have fields

```
data Bool = False | True
data Shape = Point | Rect Int Int Int Int | Circle Int
```

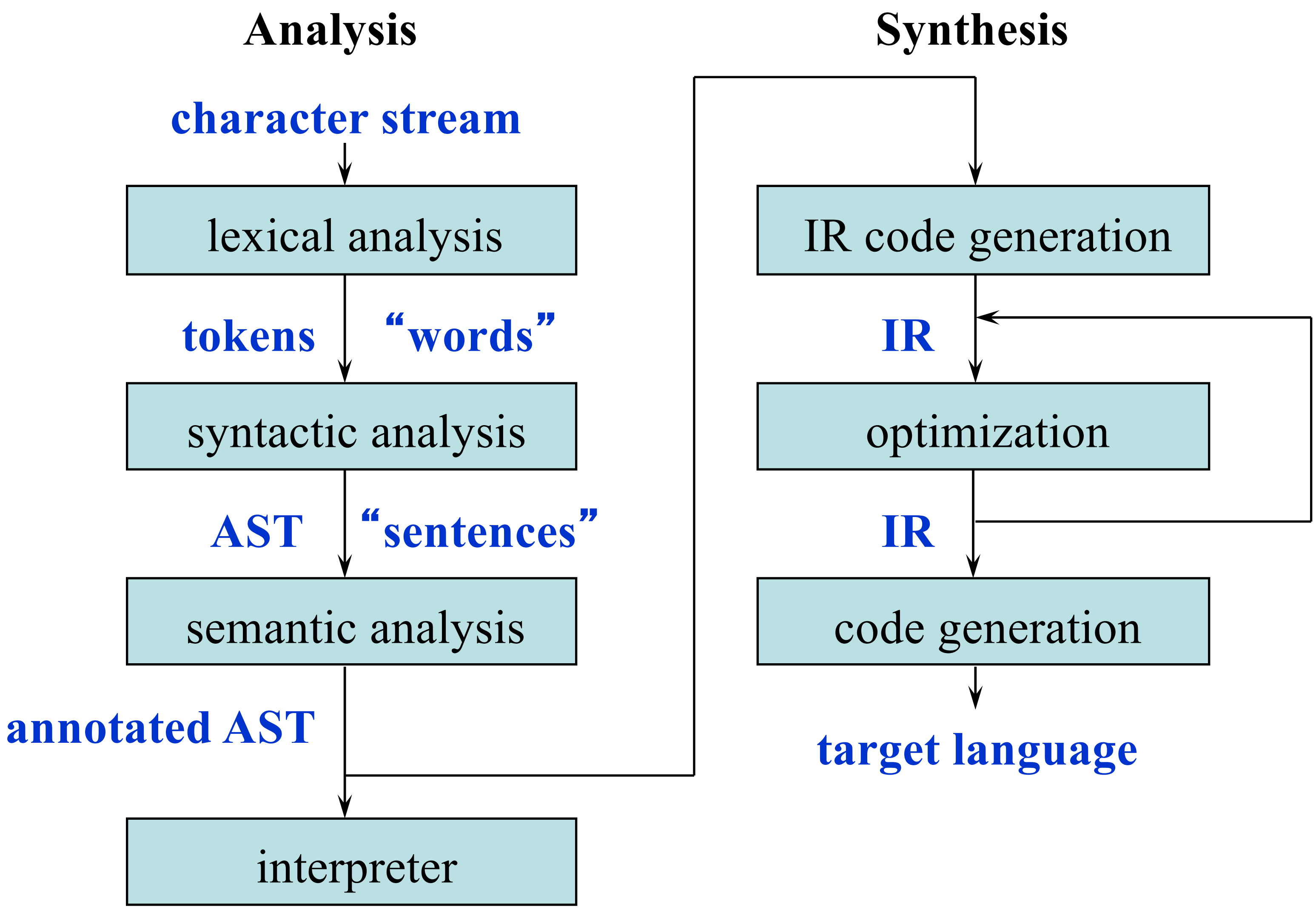
Can derive handy properties

```
data Color = Blue | Red | Yellow deriving (Show)
main = print Yellow
data Color = Blue | Red | Yellow deriving (Show,Eq)
if (Yellow==Blue) then ... else ...
```

Constructors can be used in pattern matching

```
foo :: Shape -> String
foo Point = "Point"
foo Rect p1 p2 p3 p4 = "Rect " ++ (show p1) ++ ...
```

Structure of a Typical Compiler



Tokens for Example MeggyJava program

```
import meggy.Meggy;

class PA3Flower {
public static void main(String[] whatever){
       {
  // Upper left petal, clockwise
  Meggy.setPixel( (byte)2, (byte)4, Meggy.Color.VIOLET );
  Meggy.setPixel( (byte)2, (byte)1, Meggy.Color.VIOLET);
  …
 }
}
```

Tokens: TokenImportKW, TokenMeggyKW, TokenSemi, TokenClassKW, TokenID "PA3Flower", TokenLBrace, …

Some Lexical Analysis with Haskell (why is this broken?)

```
module Lexer where

import Data.Char  -- needed for isSpace function

data Token
  = TokenIfKW
  | TokenComma
  -- TODO: constructors for all other tokens
  deriving (Show,Eq)

lexer :: String -> [Token]
lexer [] = []
lexer (‘i’:’f’:rest) = TokenIfKW : lexer rest
-- TODO: patterns for other keyword and punctuation tokens
lexer (c:rest) = if isSpace c then lexer rest else lexer (c:rest)
```

General Approach for Lexical Analysis

Regular Languages

Finite State Machines

- –DFAs: Deterministic Finite Automata
- –Complications when doing lexical analysis
- – NFAs: Non Deterministic Finite State Automata

From Regular Expressions to NFAs

From NFAs to DFAs

About The Slides on Languages and Finite Automata

Slides Originally Developed by Prof. Costas Busch (2004)

- Many thanks to Prof. Busch for developing the original slide set.

Adapted with permission by Prof. Dan Massey (Spring 2007)

- Subsequent modifications, many thanks to Prof. Massey for CS 301 slides

Adapted with permission by Prof. Michelle Strout (Spring 2011)

- Adapted for use in CS 453

Adapted by Wim Bohm(added regular expr → NFA → DFA, Spr2012)

Added slides from Profs. Christian Colberg and Saumya Debray (Fall 2016)

Languages

A language is a set of strings

(sometimes called sentences)

String: A finite sequence of letters

Examples: "cat", "dog", "house", …

Defined over a fixed alphabet:

$$\Sigma = \{a, b, c, \ldots, z\}$$

Empty String

A string with no letters: ε

Observations: $$|\varepsilon| = 0$$

$$\varepsilon w = w\varepsilon = w$$

$$\varepsilon abba = abba\varepsilon = abba$$

Regular Expressions

Regular expressions describe regular languages

You have probably seen them in OSs / editors

Example: $(a \mid (b)(c))*$

describes the language

$$L((a \mid (b)(c))*) = \{\varepsilon, a, bc, aa, abc, bca, \ldots\}$$

Recursive Definition for Specifying Regular Expressions

Primitive regular expressions: $\emptyset, \ \varepsilon, \ \alpha$

where $\alpha \in \Sigma$, *some alphabet*

Given regular expressions r_1 *and* r_2

$$\left.\begin{array}{l} r_1 \mid r_2 \\ r_1 \ r_2 \\ r_1 \ * \\ (r_1) \end{array}\right\} \text{Are regular expressions}$$

Regular operators

choice: $A \mid B$ — a string from L(A) or from L(B)

concatenation: $A\ B$ — a string from L(A) followed by a string from L(B)

repetition: A^* — 0 or more concatenations of strings from L(A)

A^+ — 1 or more

grouping: (A)

Concatenation has precedence over choice: A|B C vs. (A|B)C

More syntactic sugar, used in scanner generators:

- [abc] means a or b or c
- [\t\n] means tab, newline, or space
- [a-z] means a,b,c, …, or z

Example Regular Expressions and Regular Definitions

Regular definition:

name : regular expression

name can then be used in other regular expressions

Keywords "print", "while"

Operations: "+", "-", "*"

Identifiers:

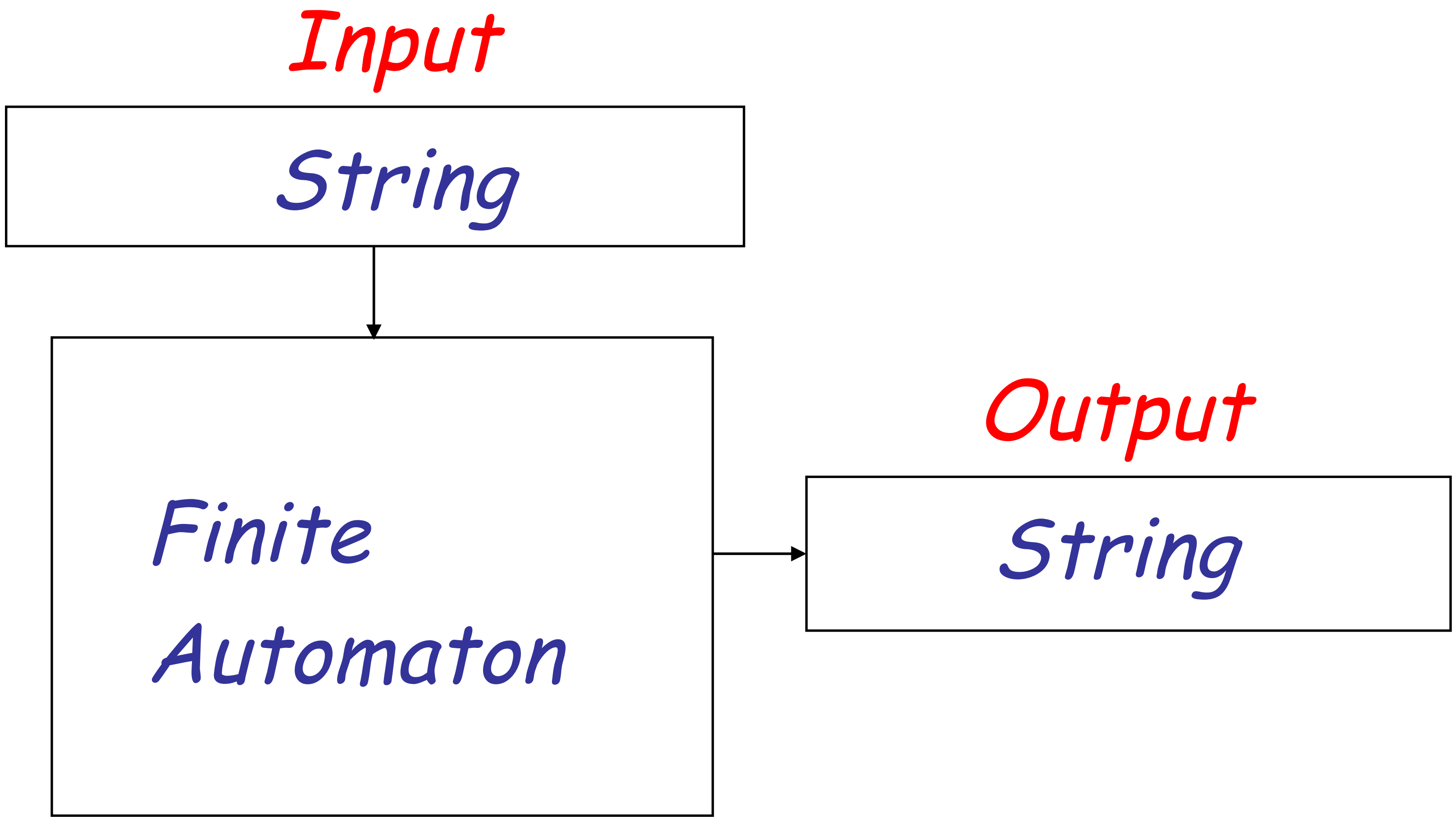
let : [a-zA-Z] // chose from a to z or A to Z

dig : [0-9]

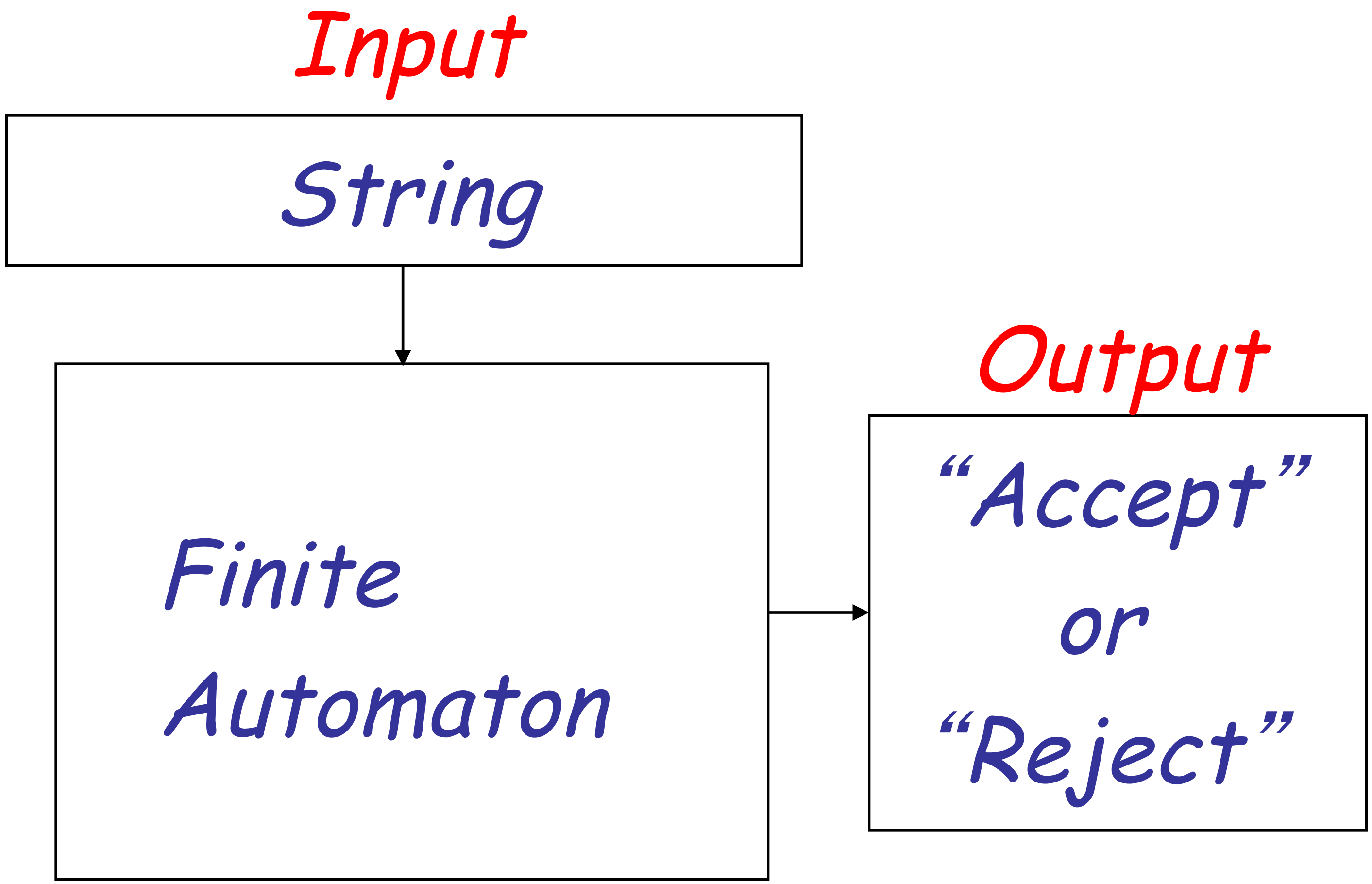
id : let (let | dig)*

Numbers: $\mathbf{dig^{+} = dig\ dig^{*}}$

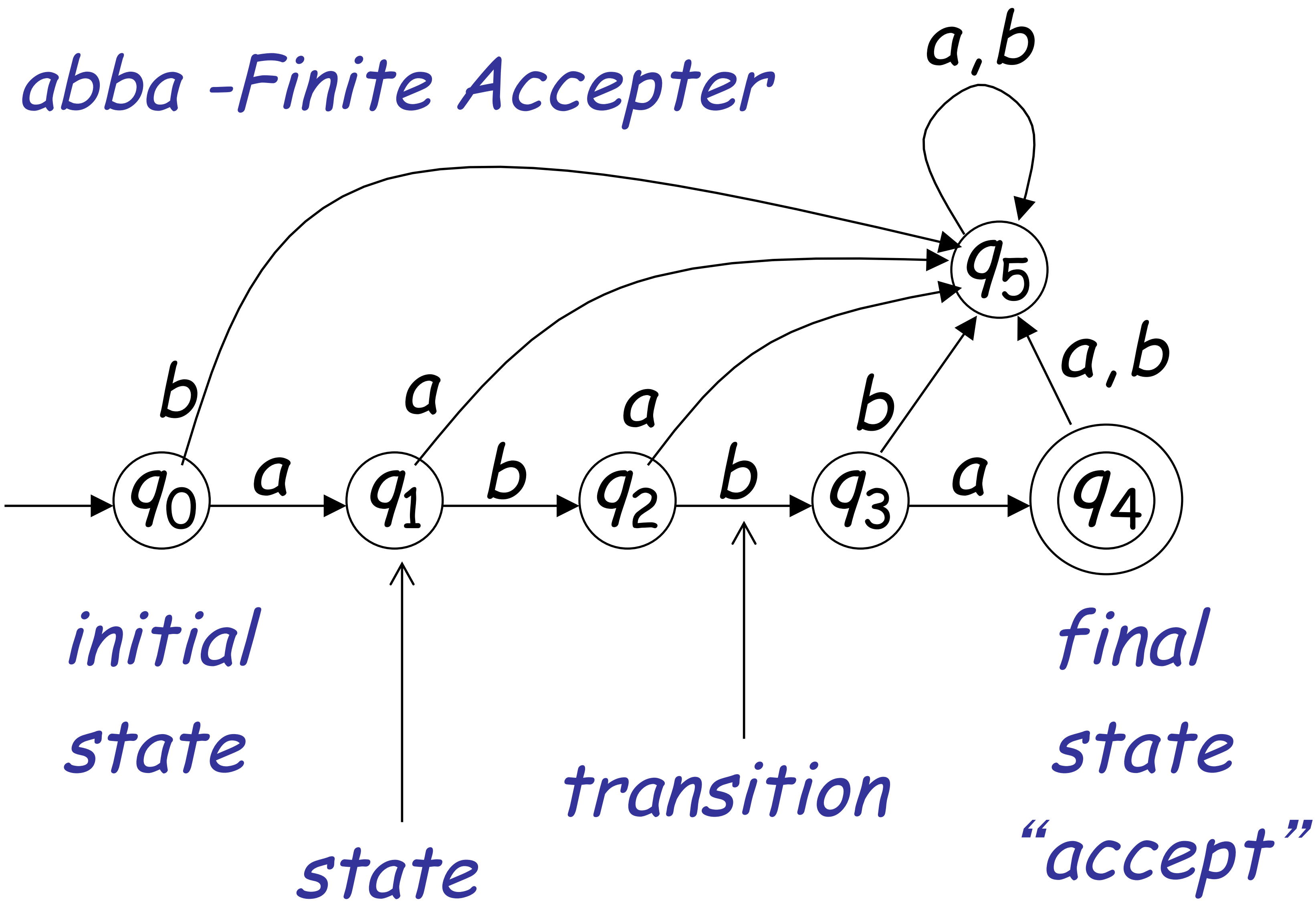
Finite Automaton, or Finite State Machine (FSM)



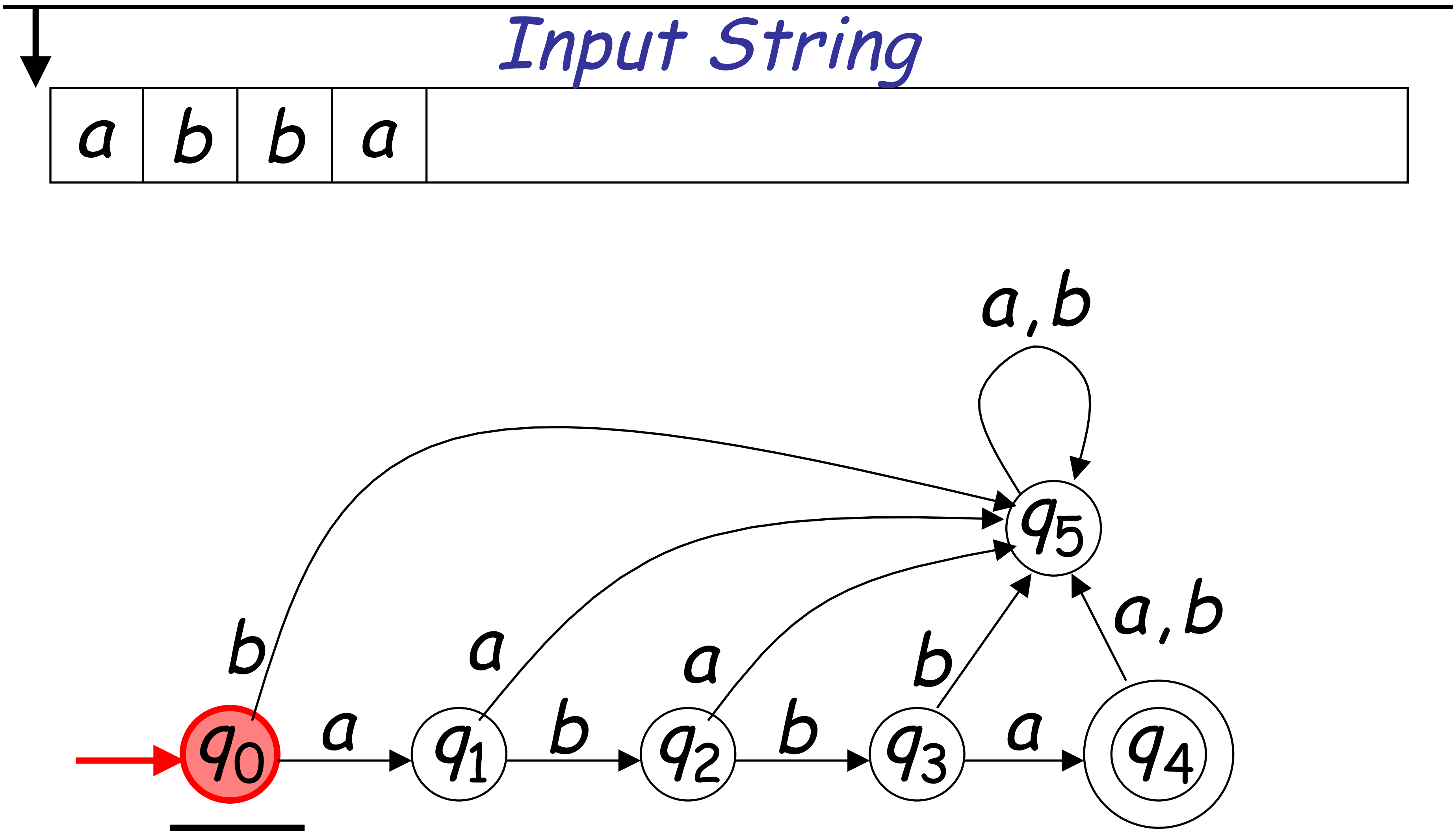
Finite State Machine



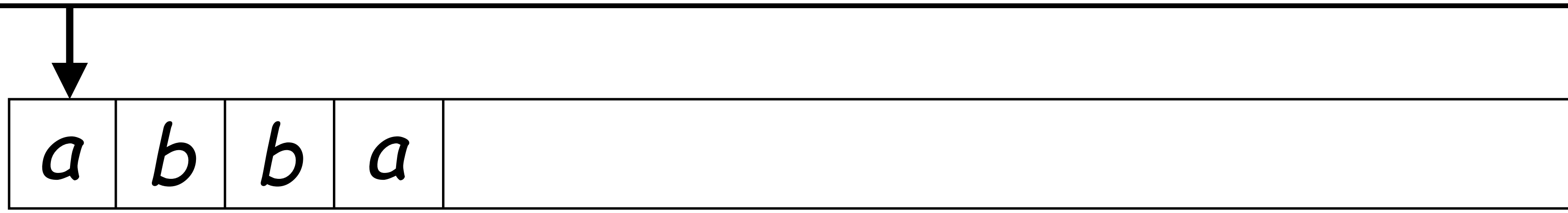
State Transition Graph

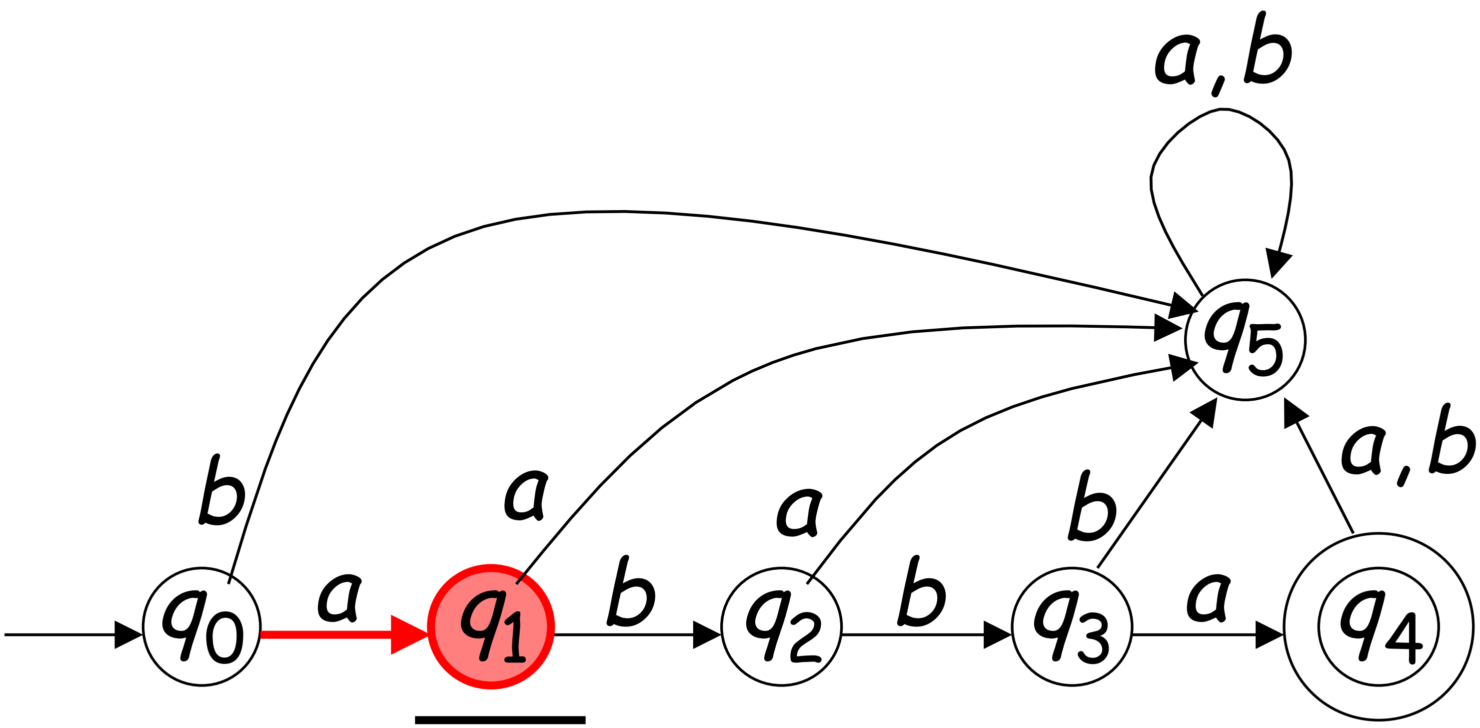


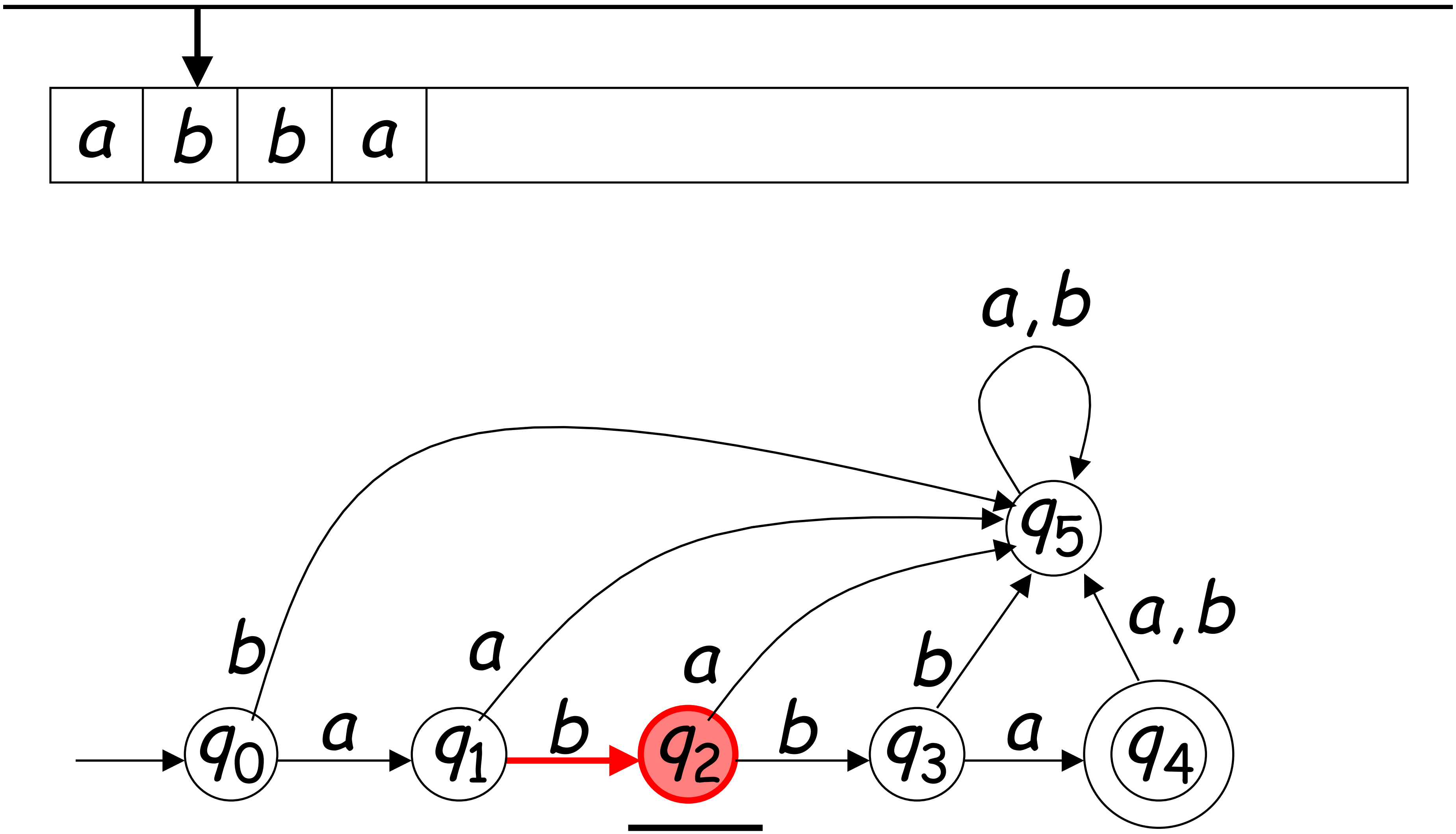
Initial Configuration

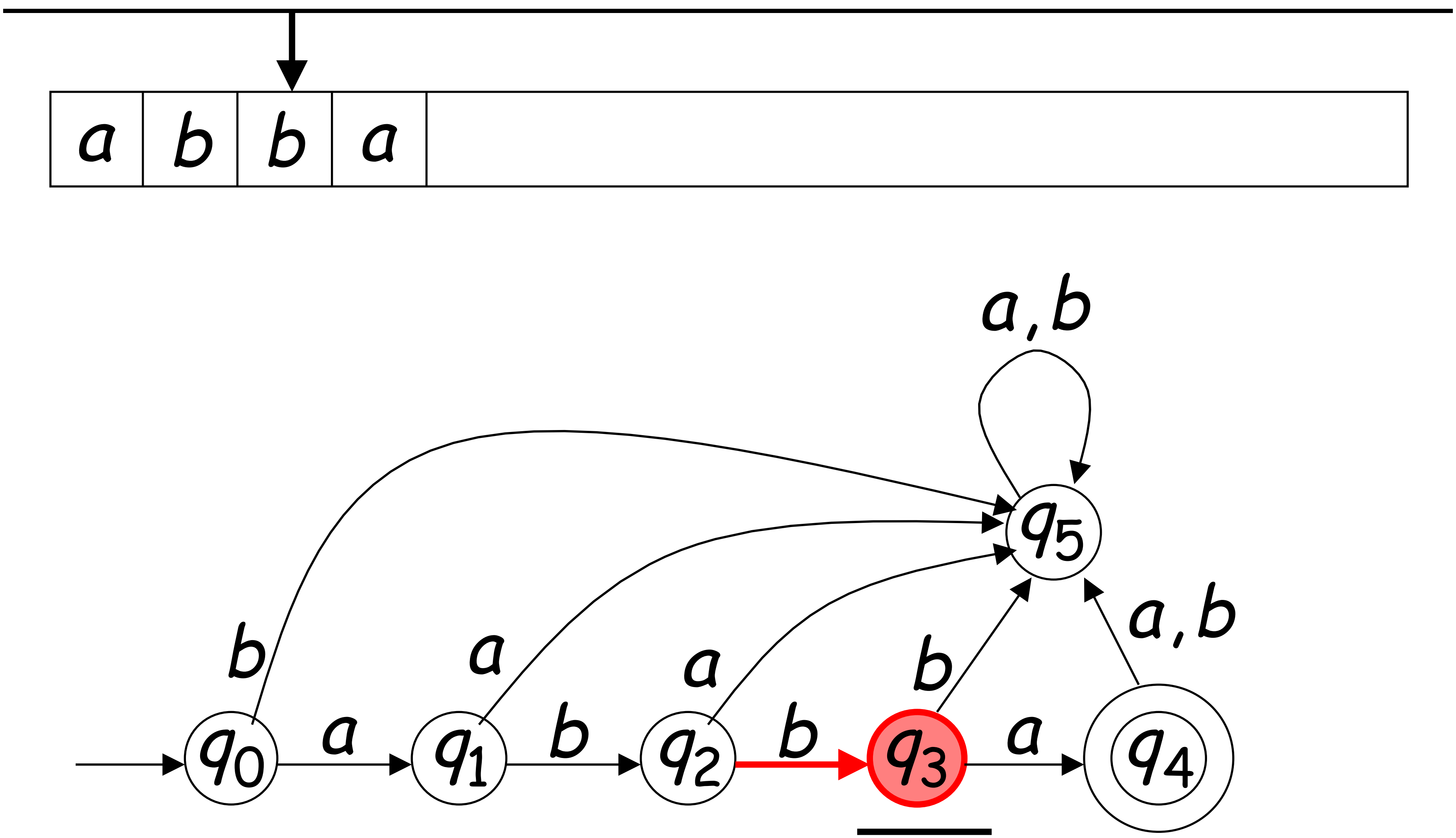


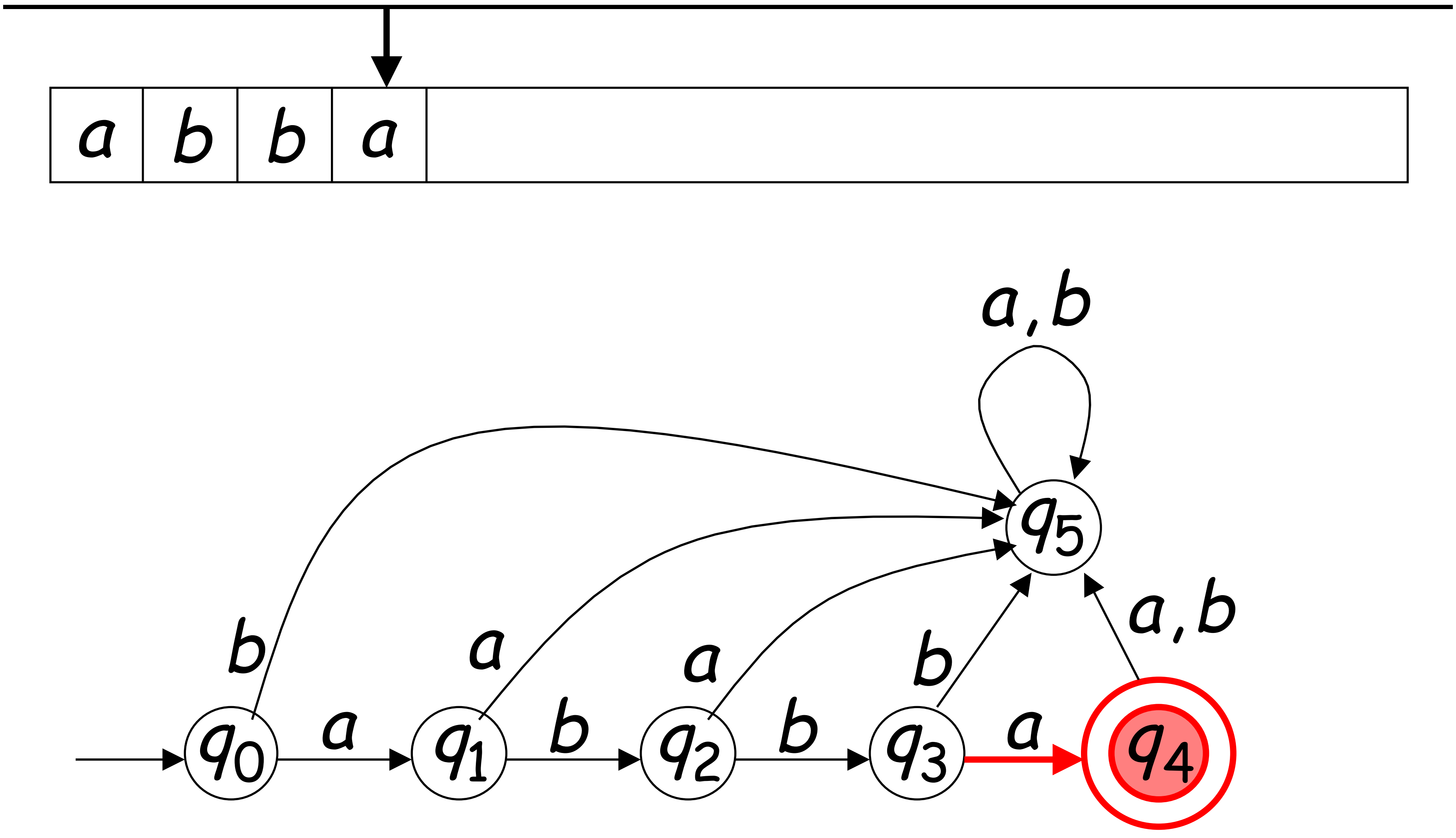
Reading the Input



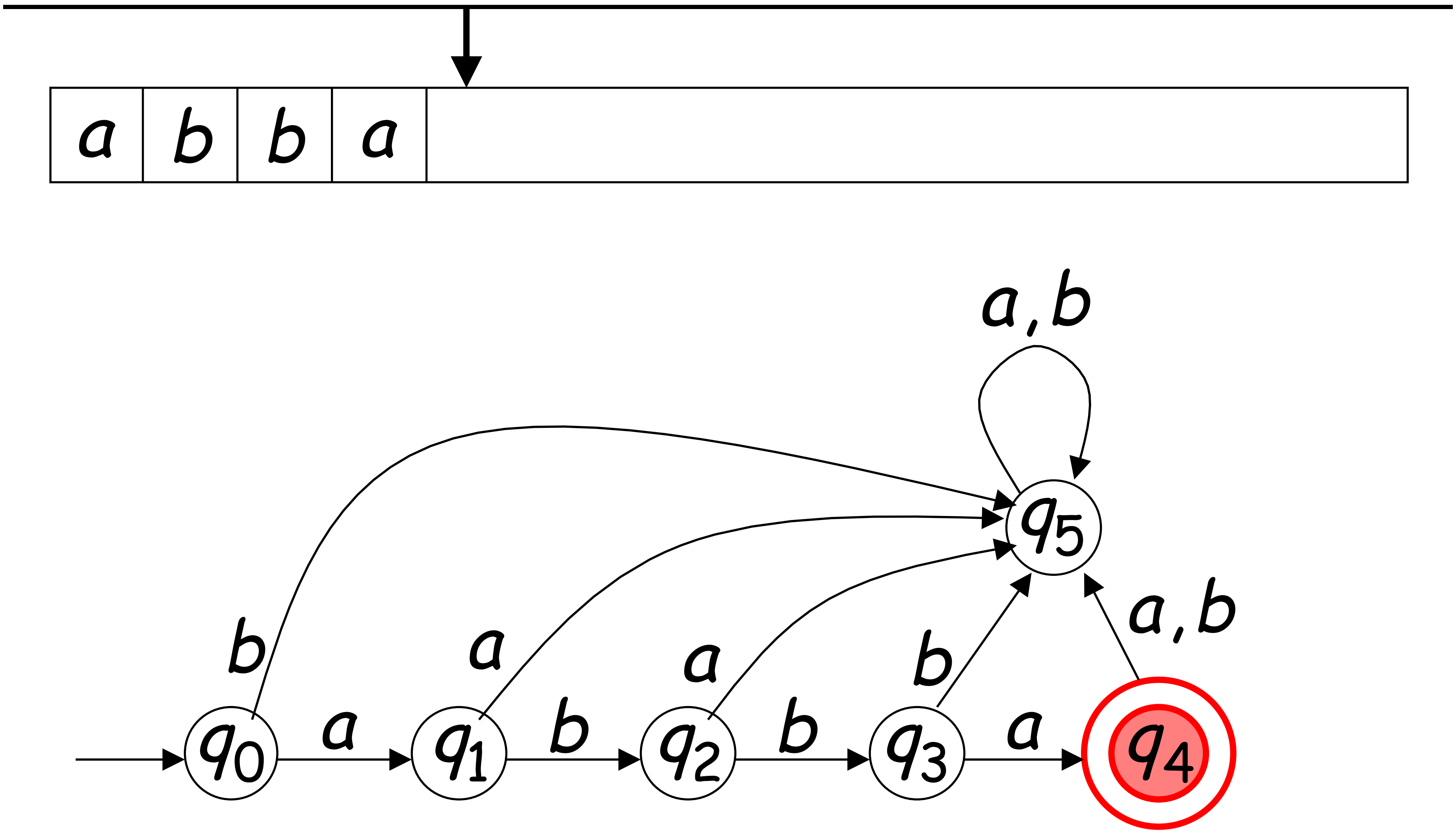



a
b
b
a
a,b
q5
b
a
a
b
a,b
q0
a
q1
b
q2
b
q3
a
q4


a
b
b
a
a,b
b
a
a
b
a,b
a
b
b
a
q_0
q_1
q_2
q_3
q_4
q_5


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

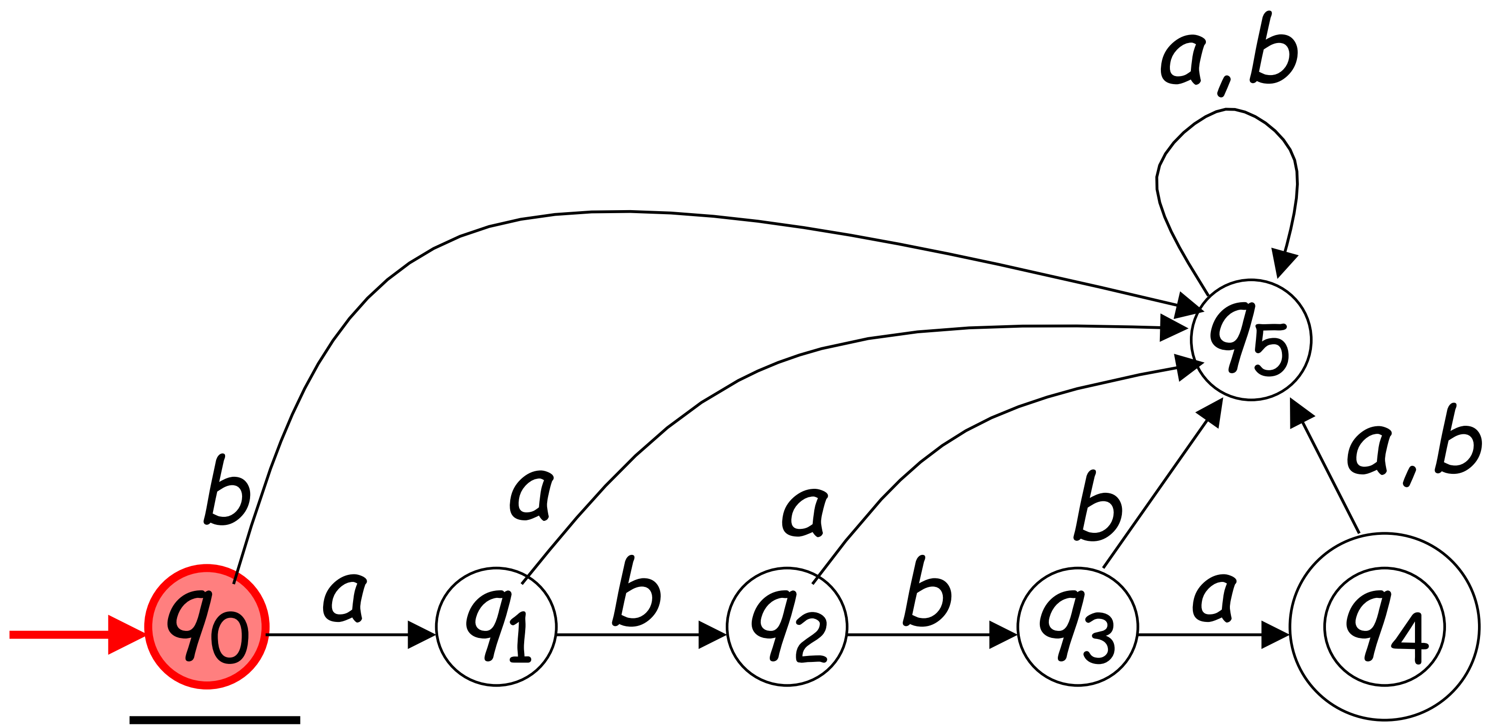
Input finished

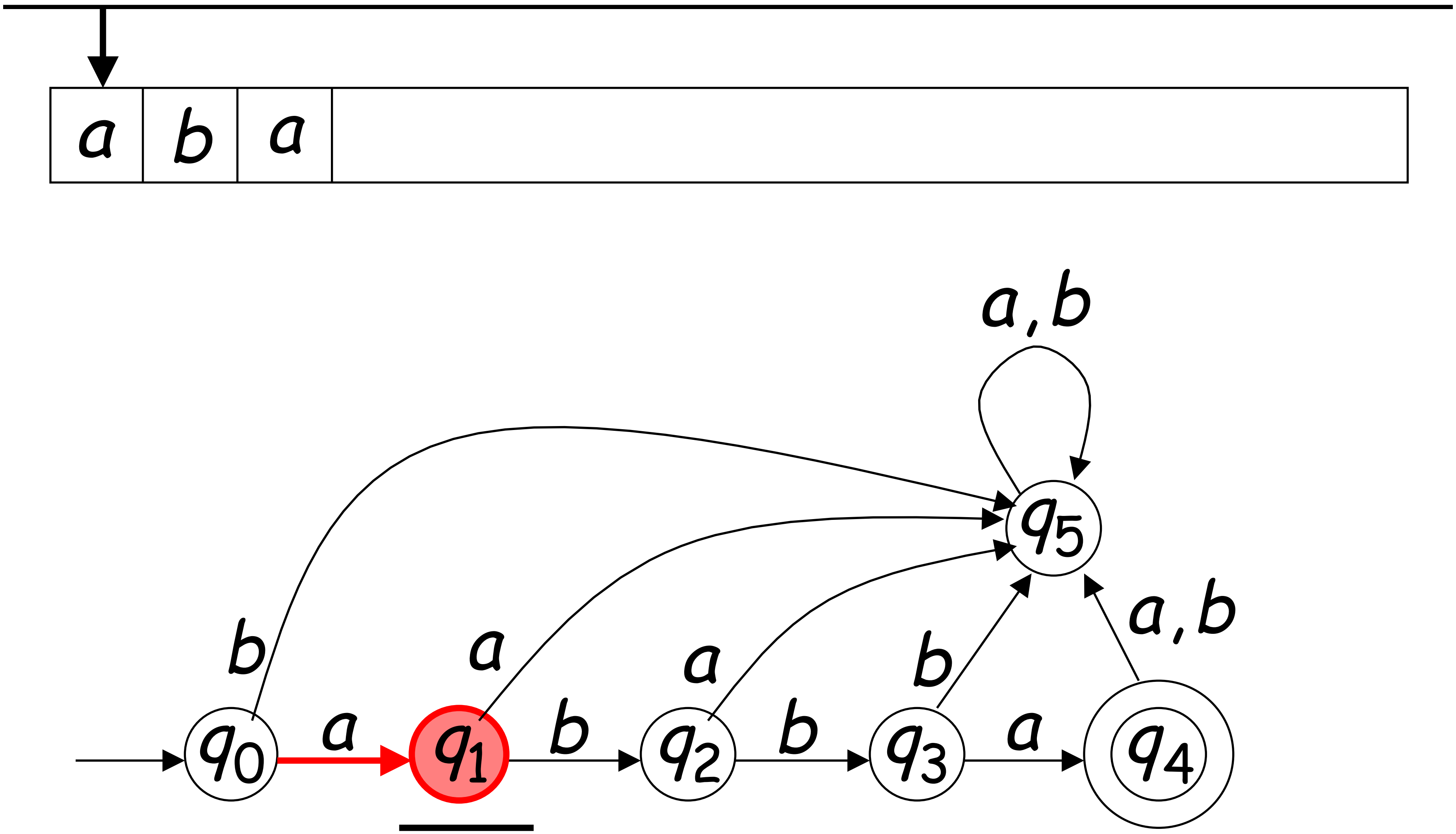


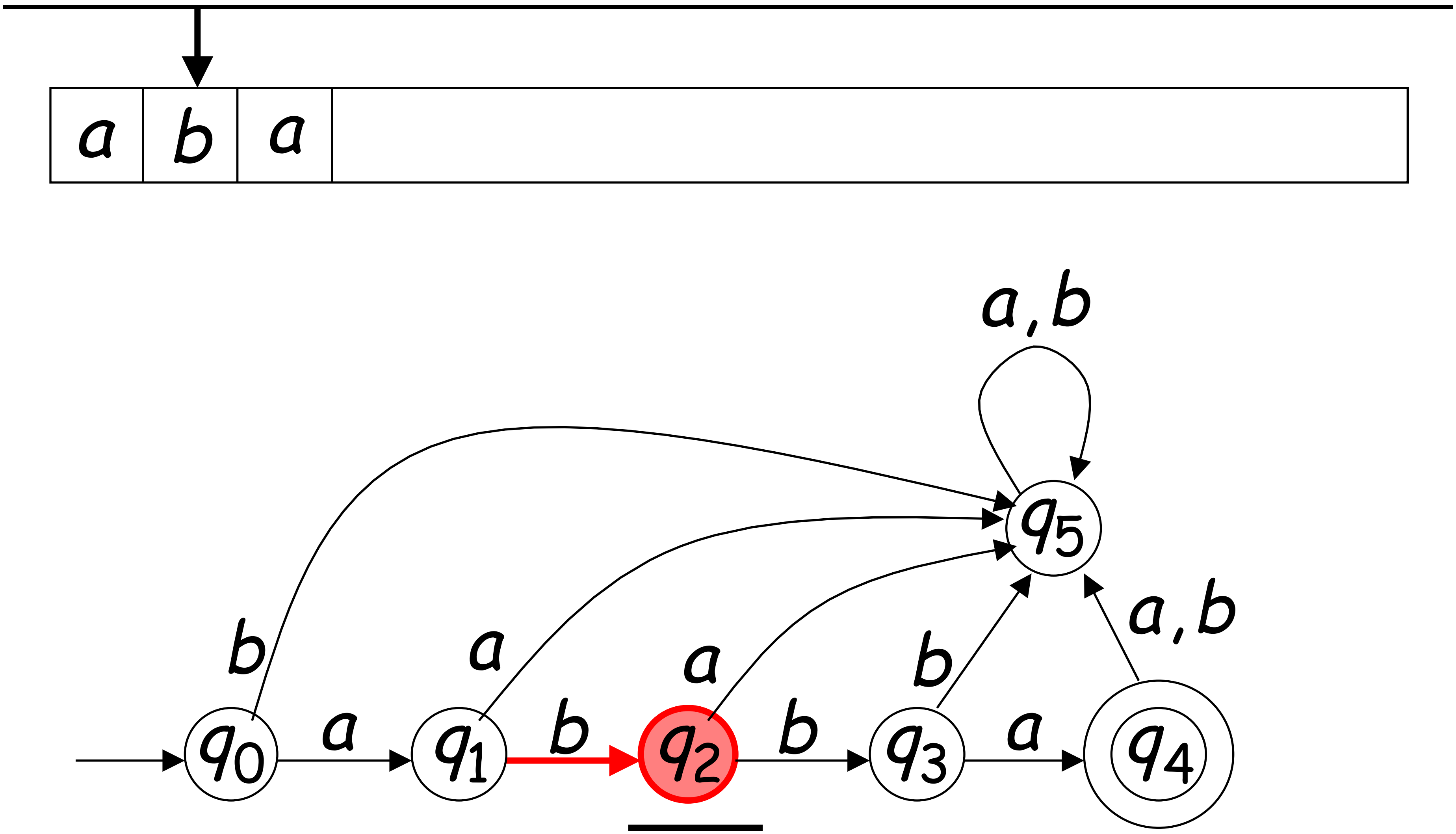
Output: "accept"

String Rejection

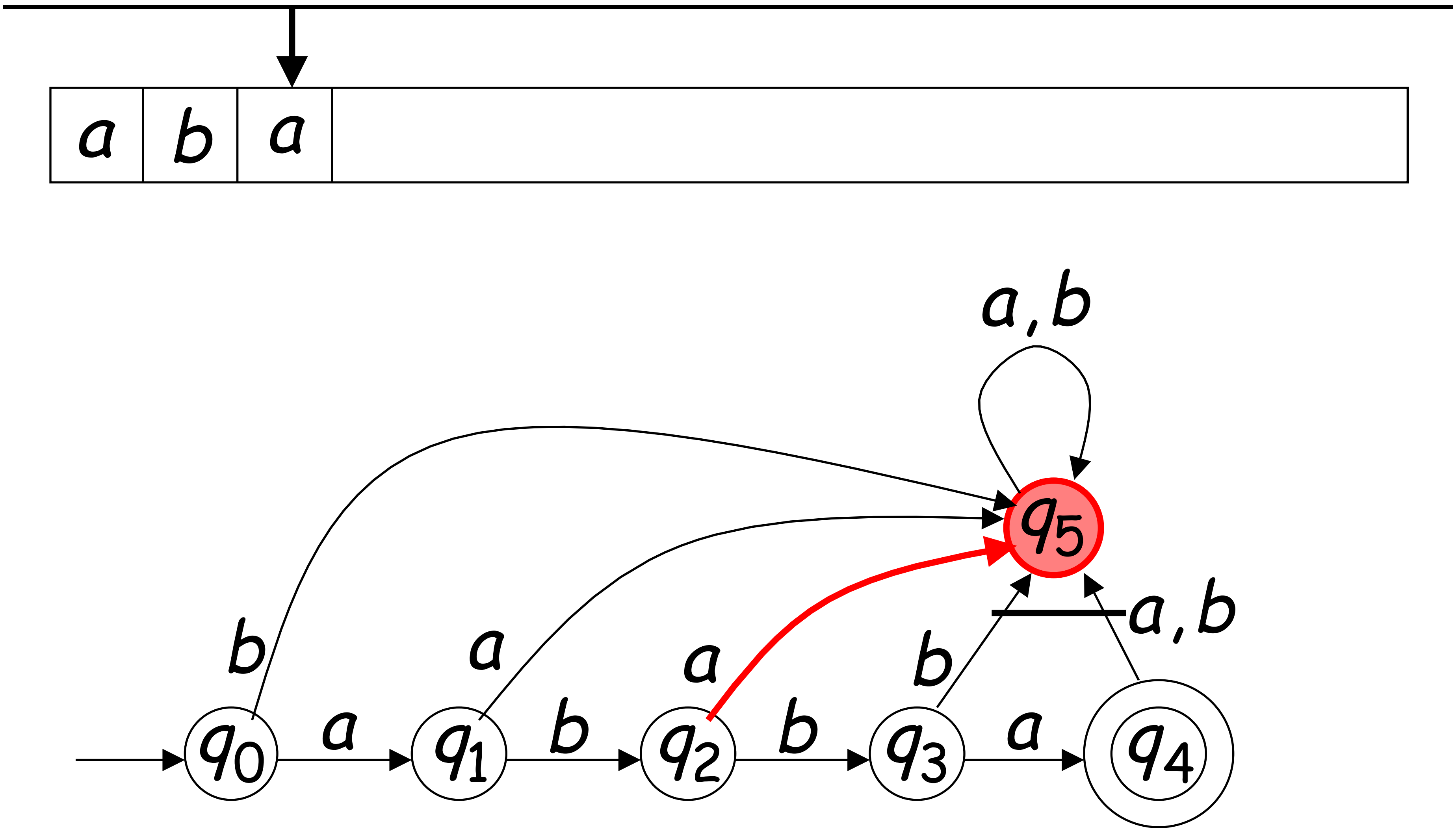
a	b	a	




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

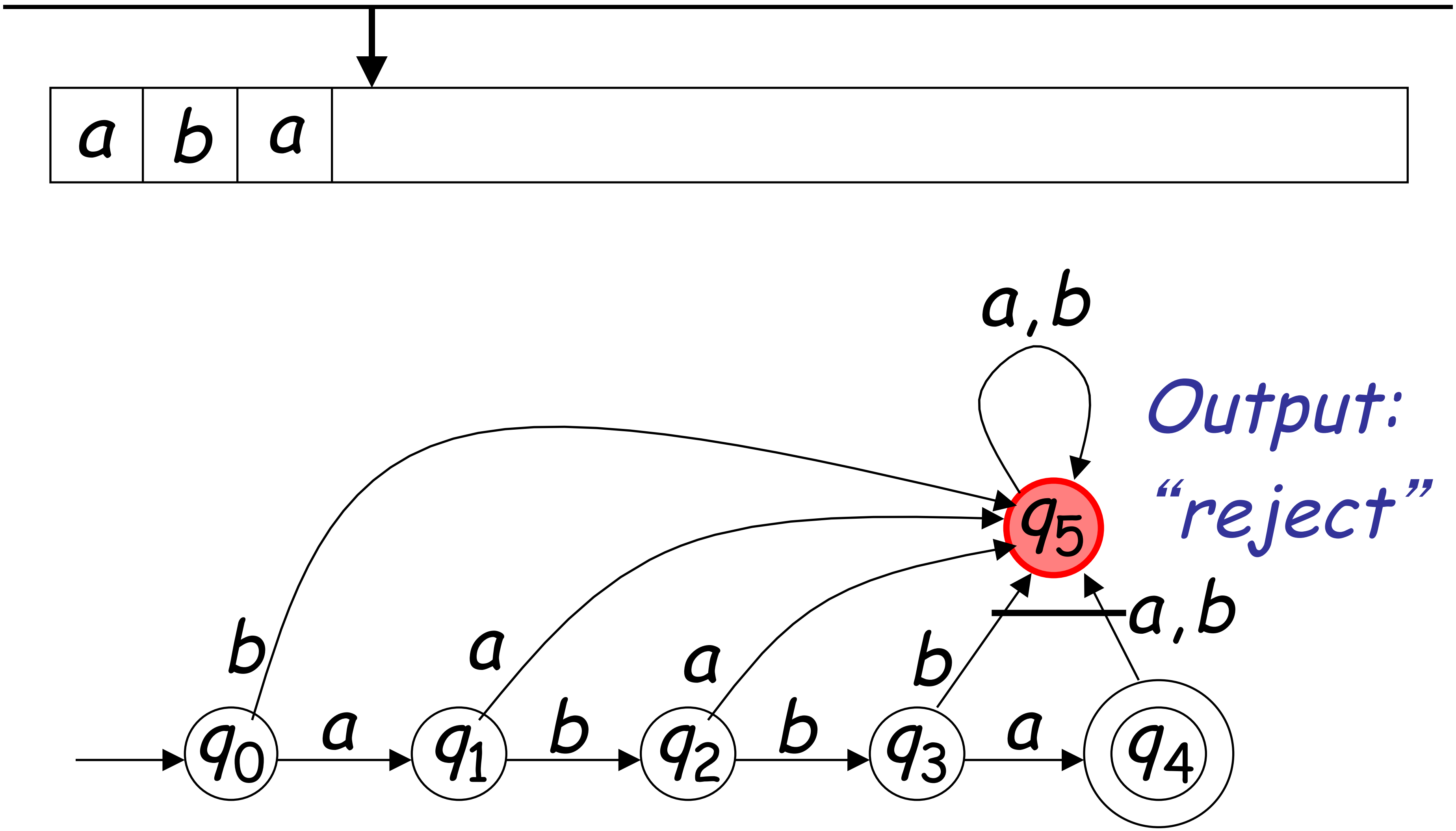


a
b
a
a,b
q5
a,b
b
a
a
b
q0
a
q1
b
q2
b
q3
a
q4



a
b
a
a,b
b
a
a
b
a,b
a
b
b
a
q_0
q_1
q_2
q_3
q_4
q_5

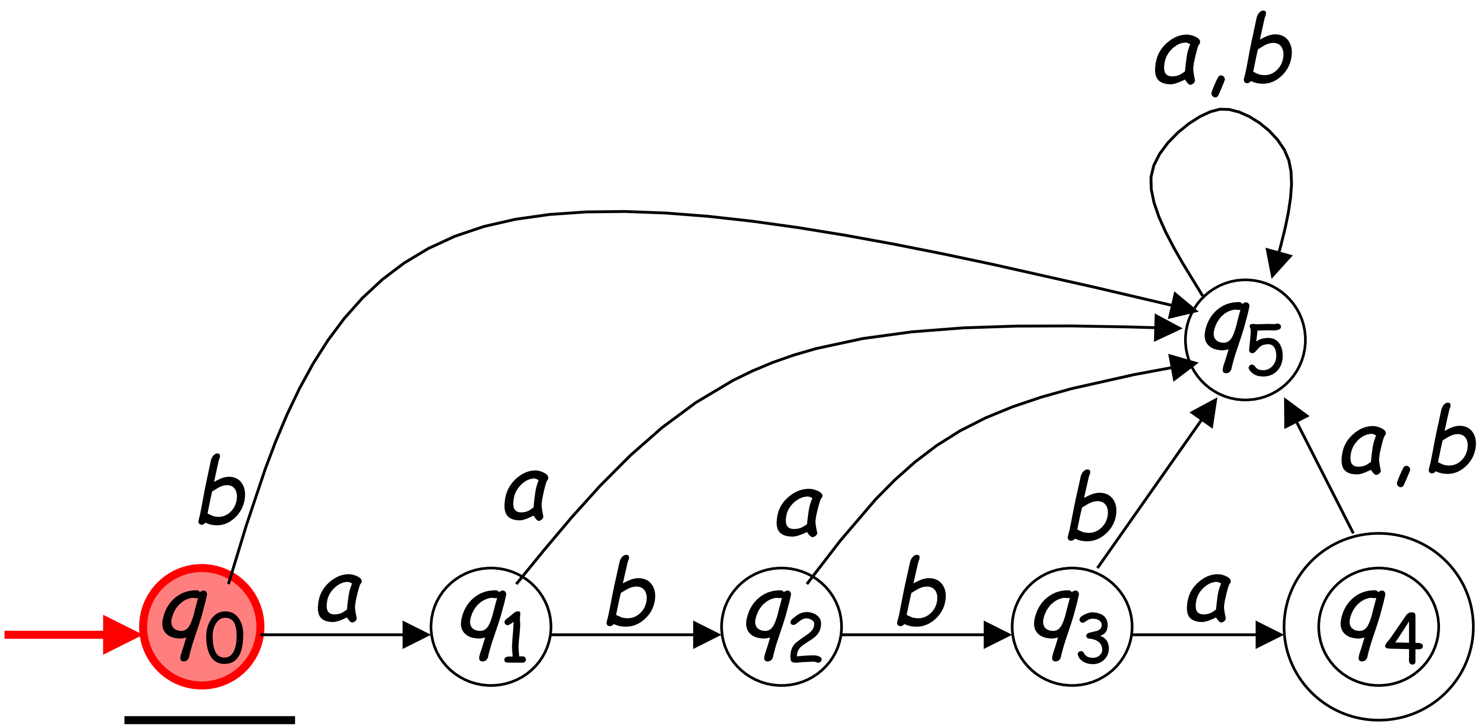
Input finished

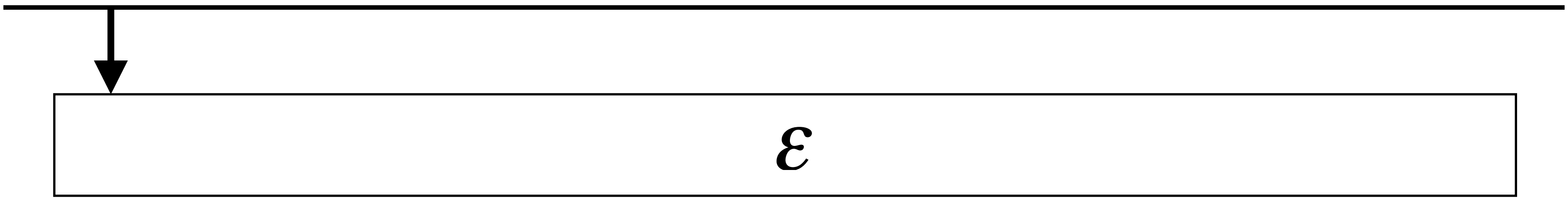


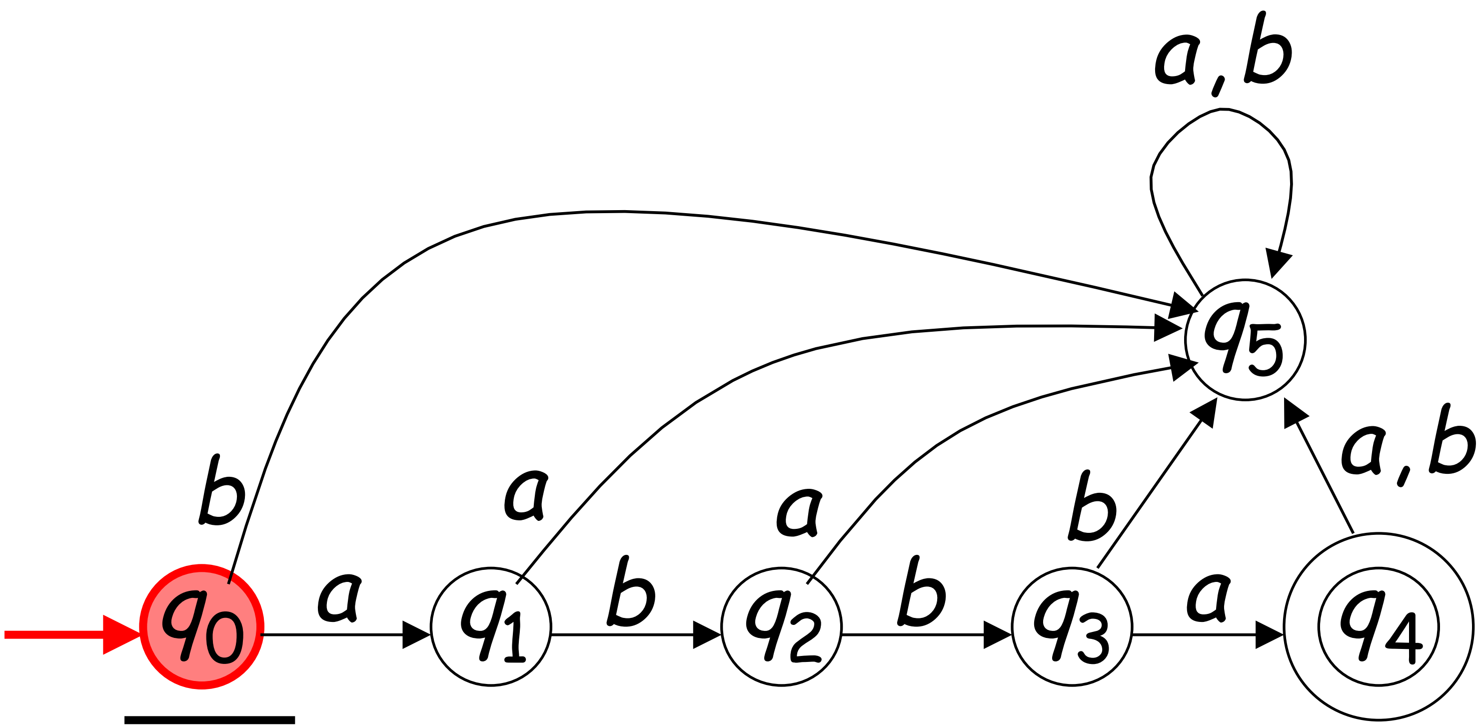
Output: "reject"

The Empty String

ε



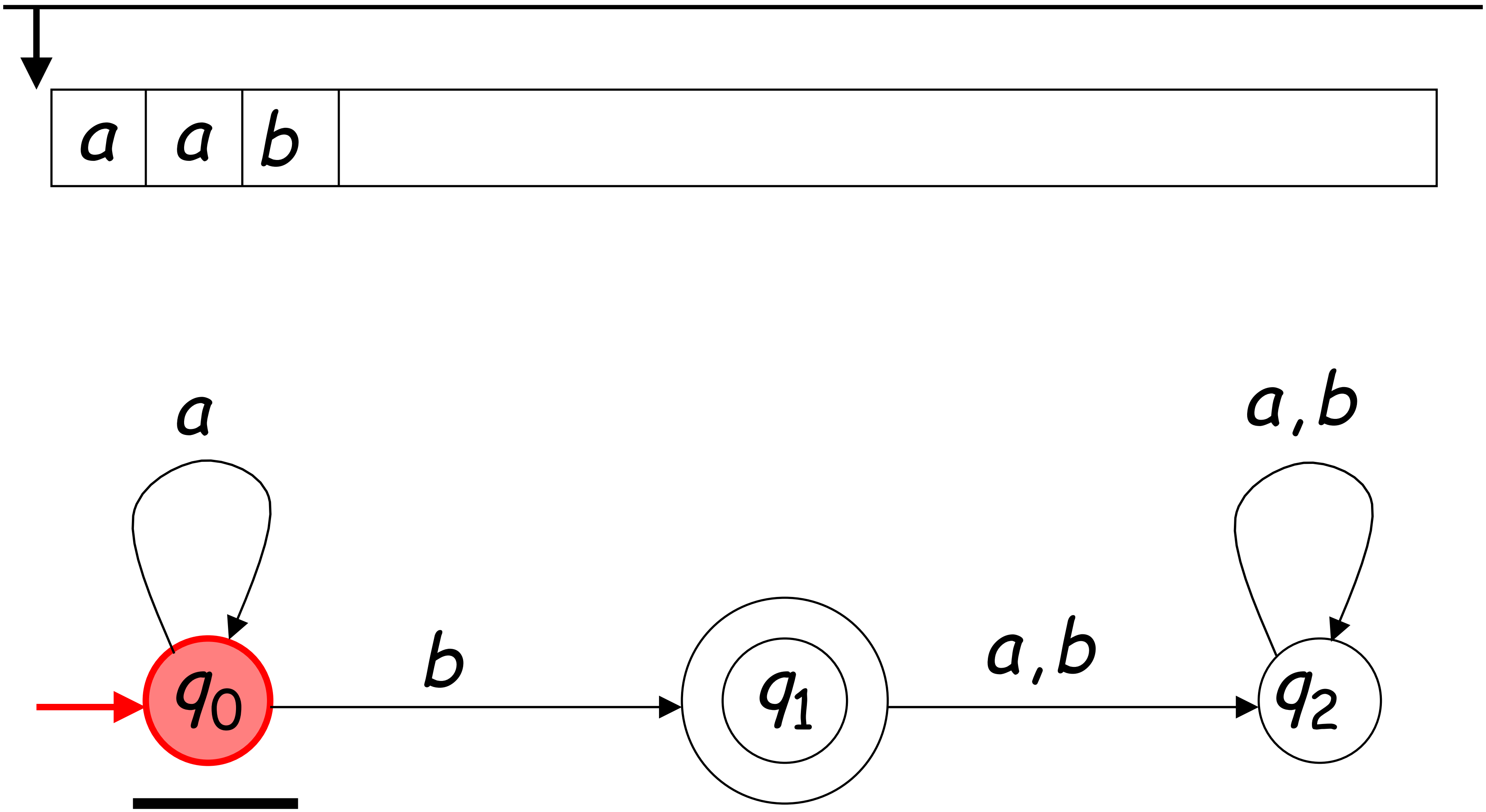


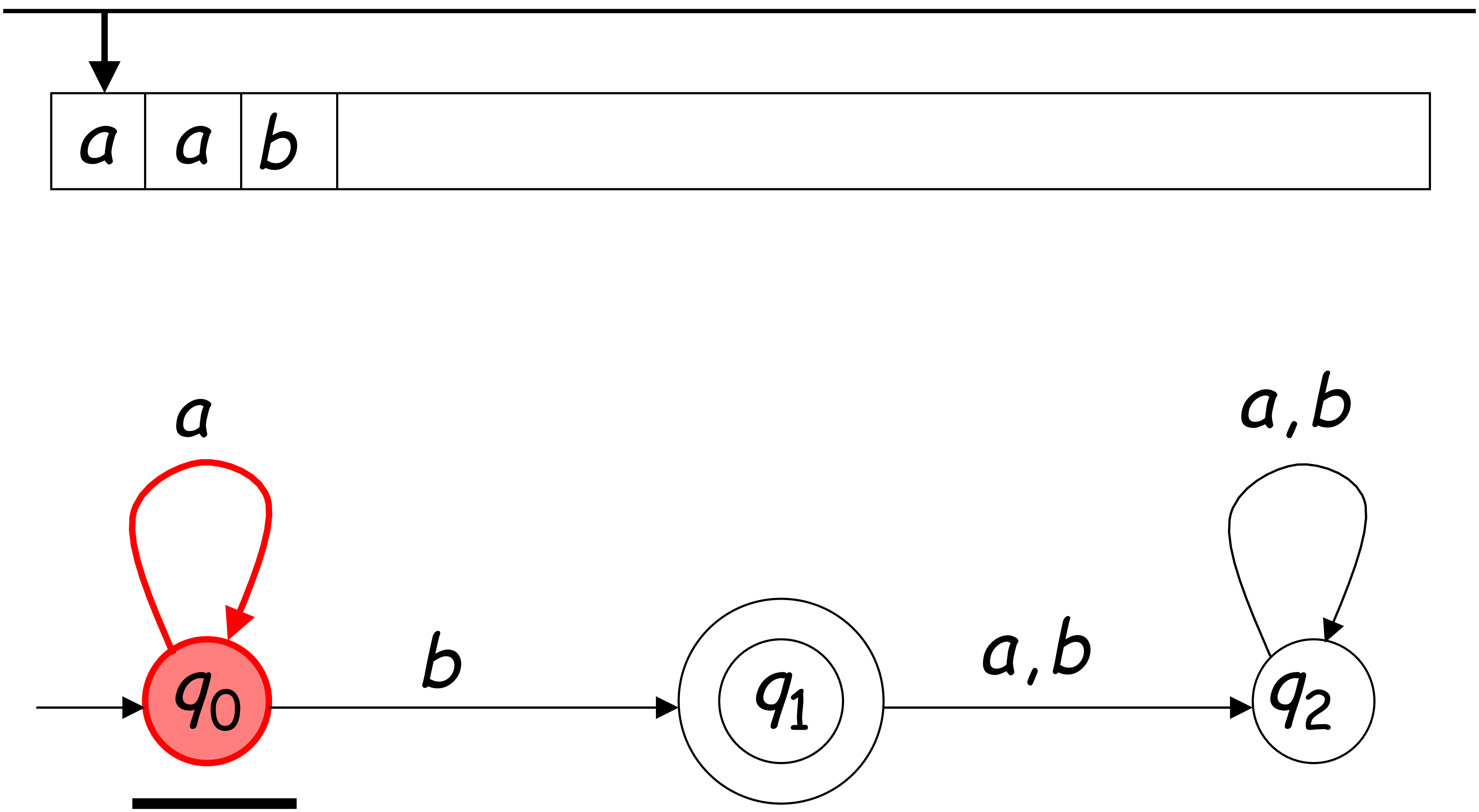


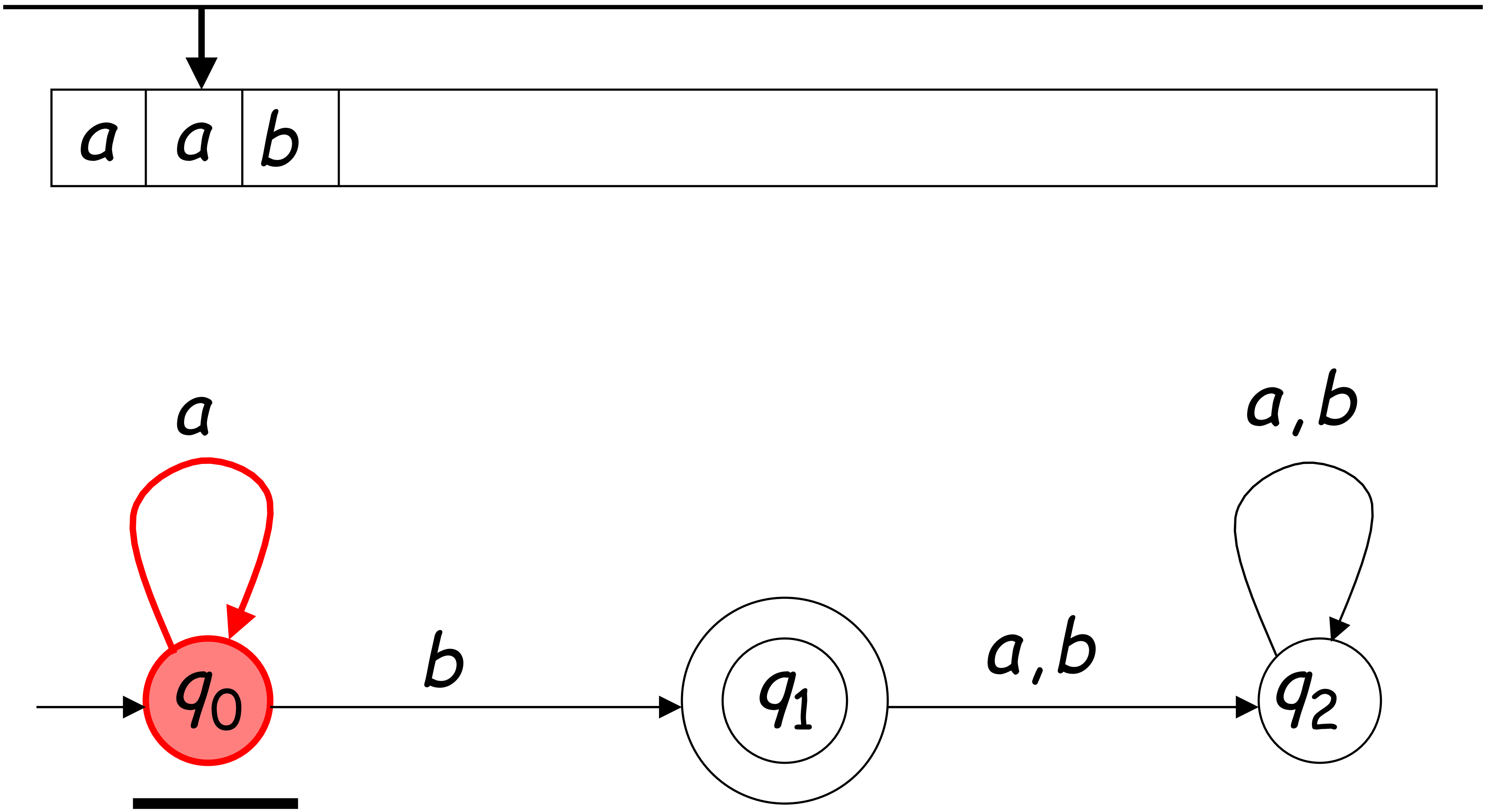
Output:
"reject"

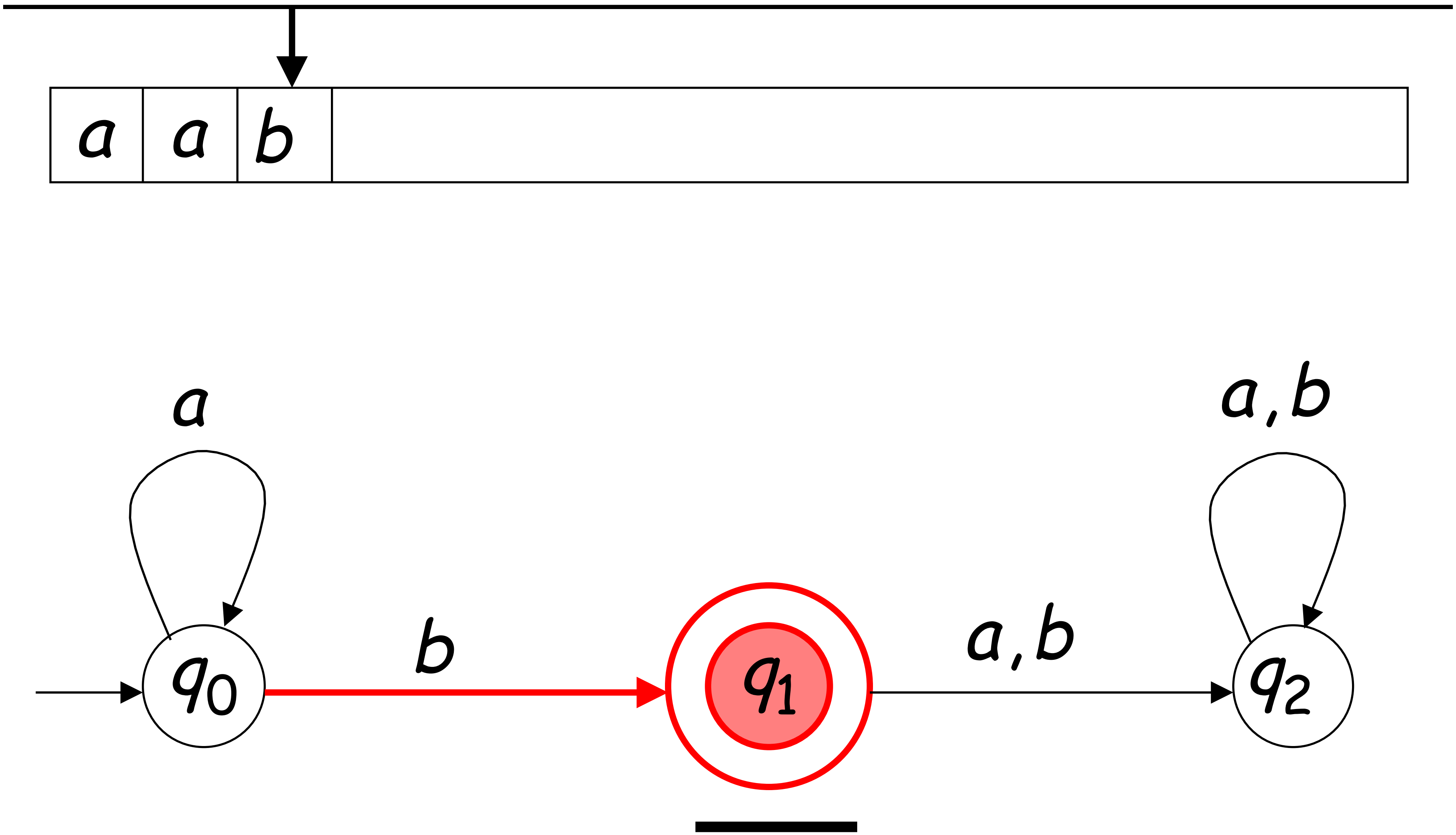
Would it be possible to accept the empty string?

Another Example

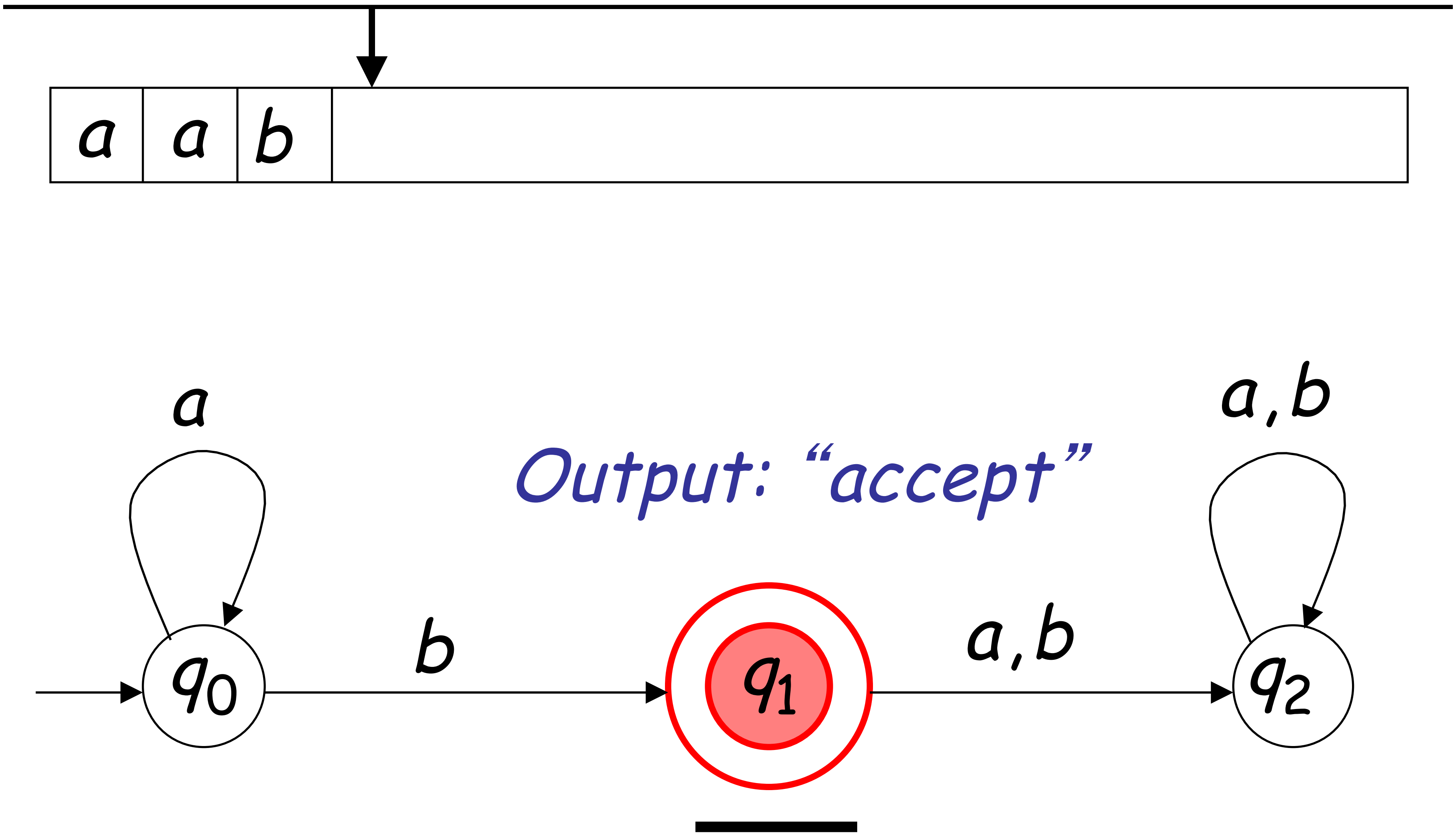



a
a
b
a
a,b
b
a,b
q0
q1
q2

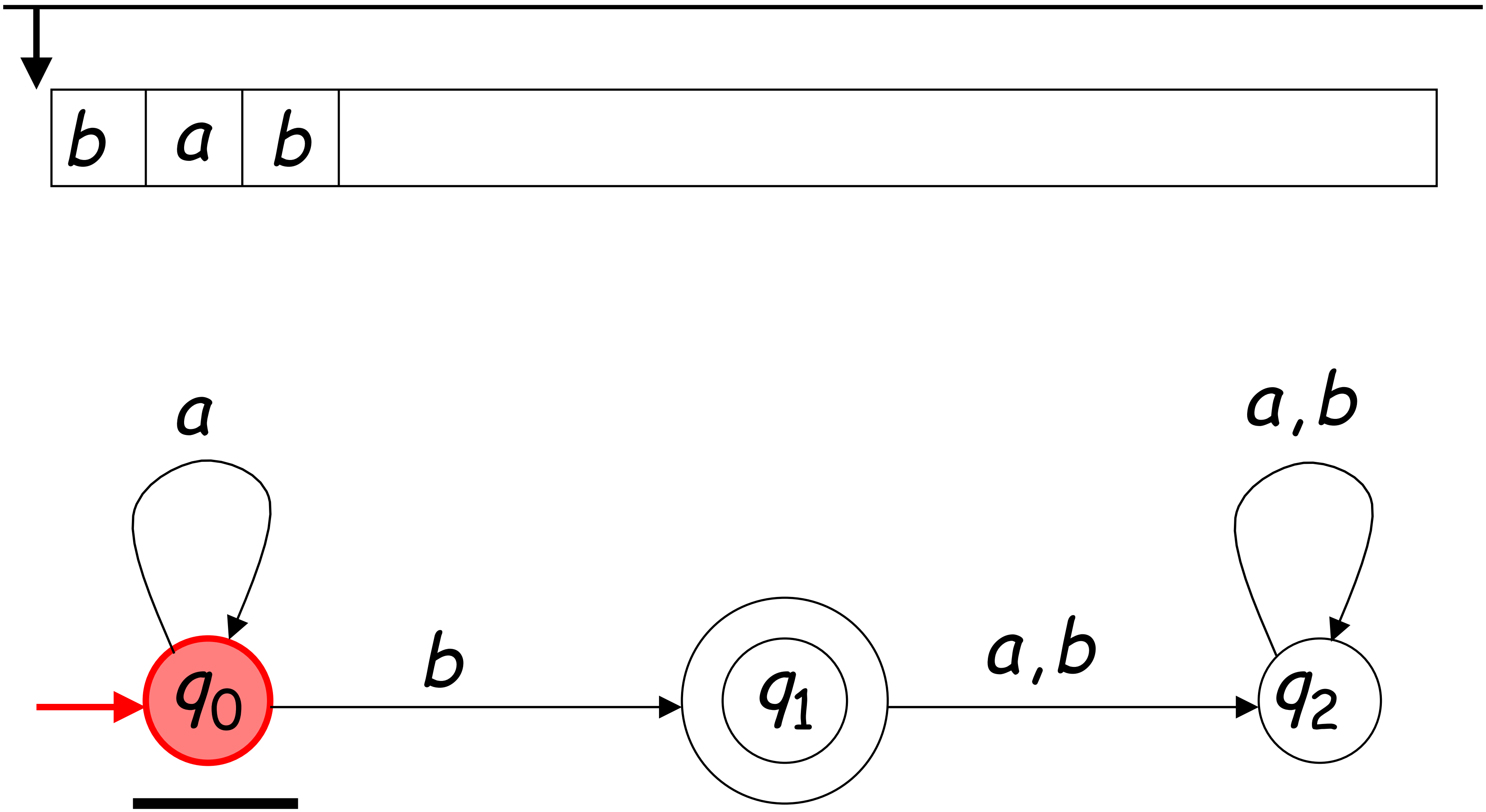

a
a
b
a
a,b
b
a,b
q_0
q_1
q_2

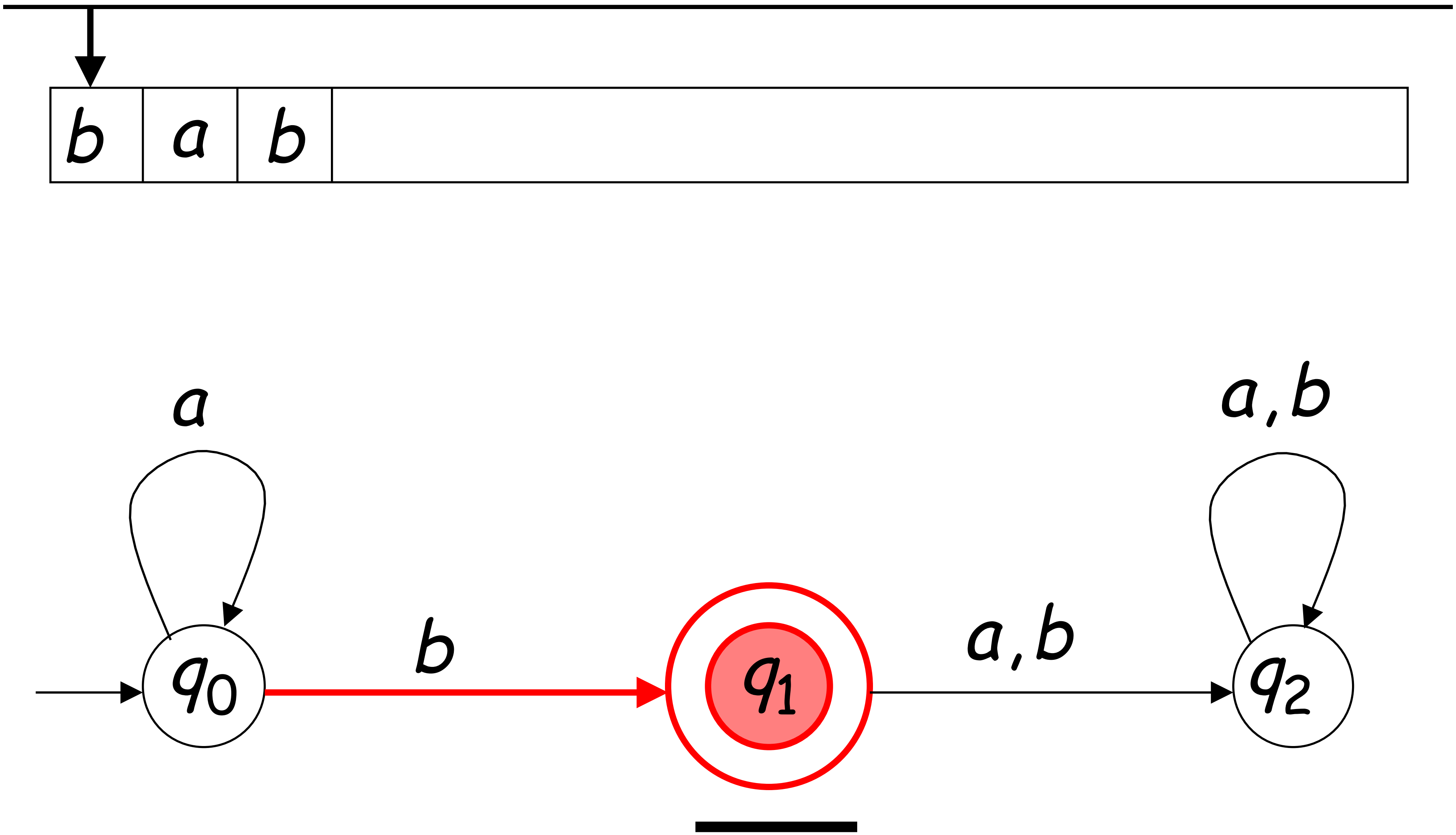

a
a
b
a
a,b
b
a,b
q_0
q_1
q_2

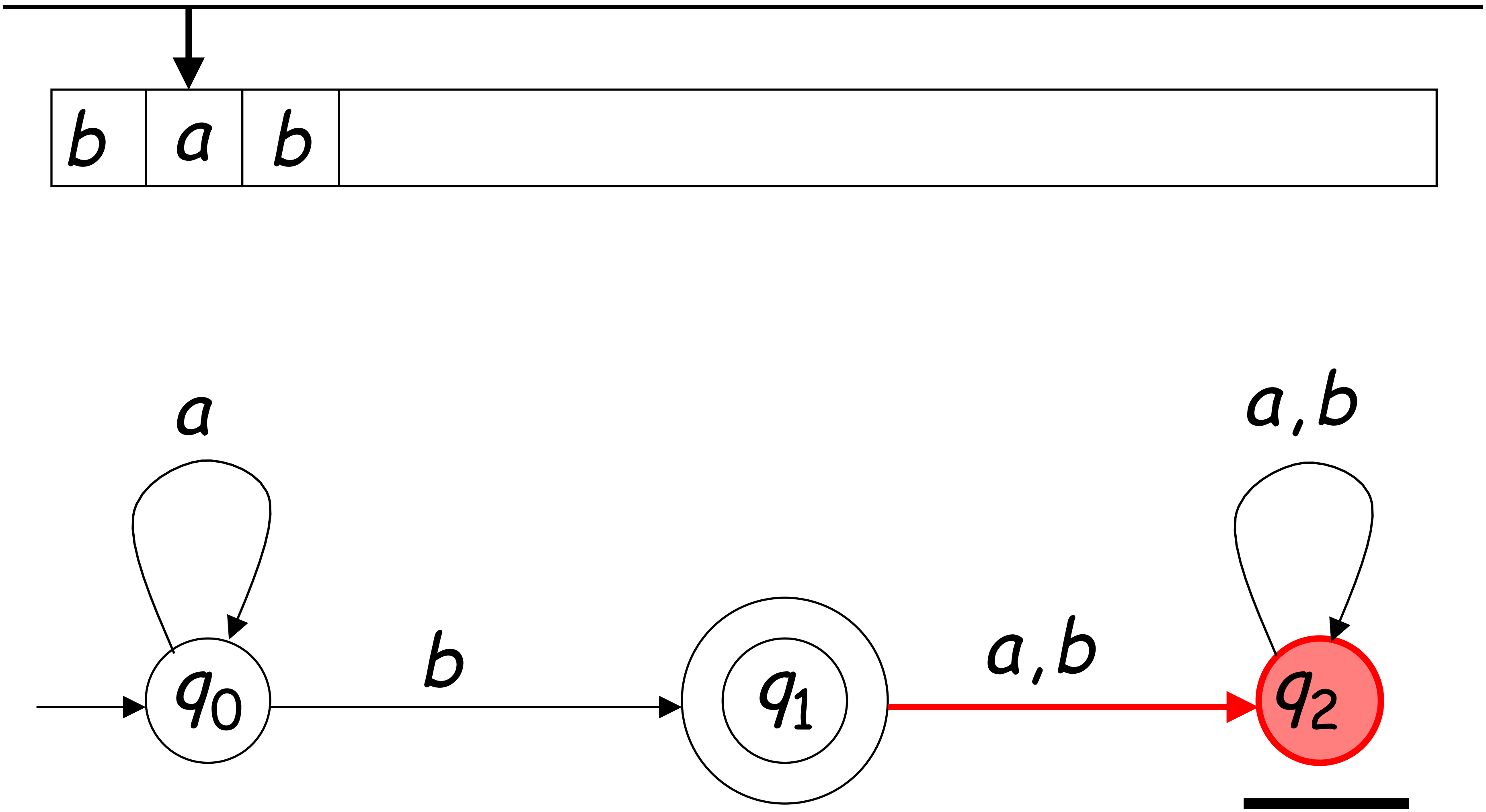
Input finished

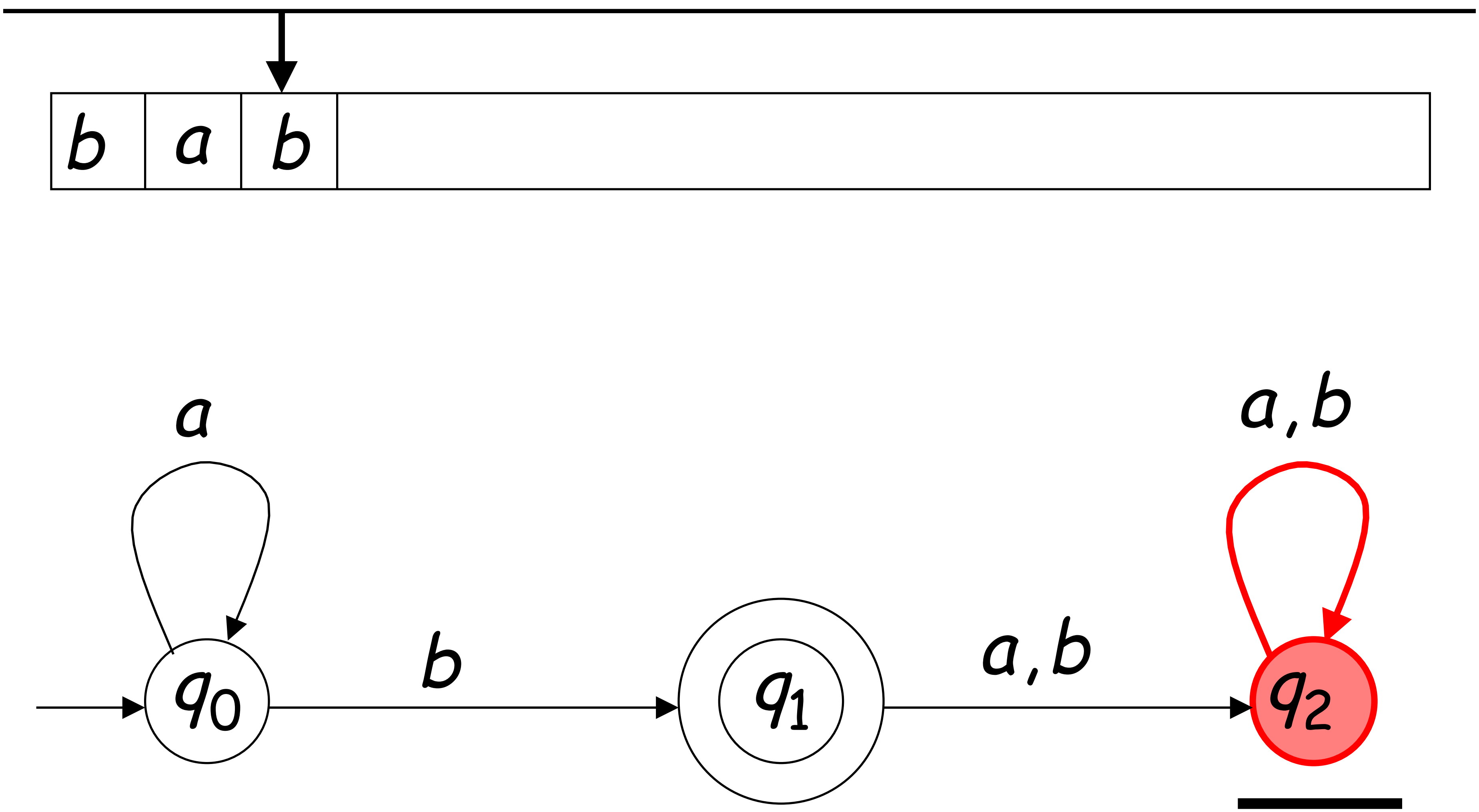


Rejection

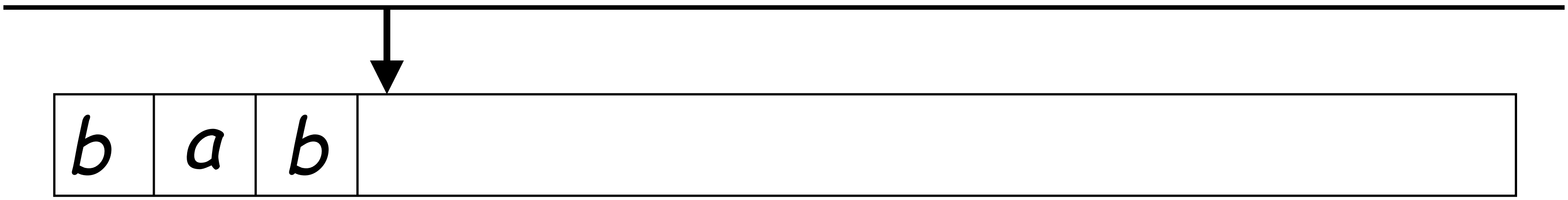



b
a
b
a
a,b
q0
b
q1
a,b
q2

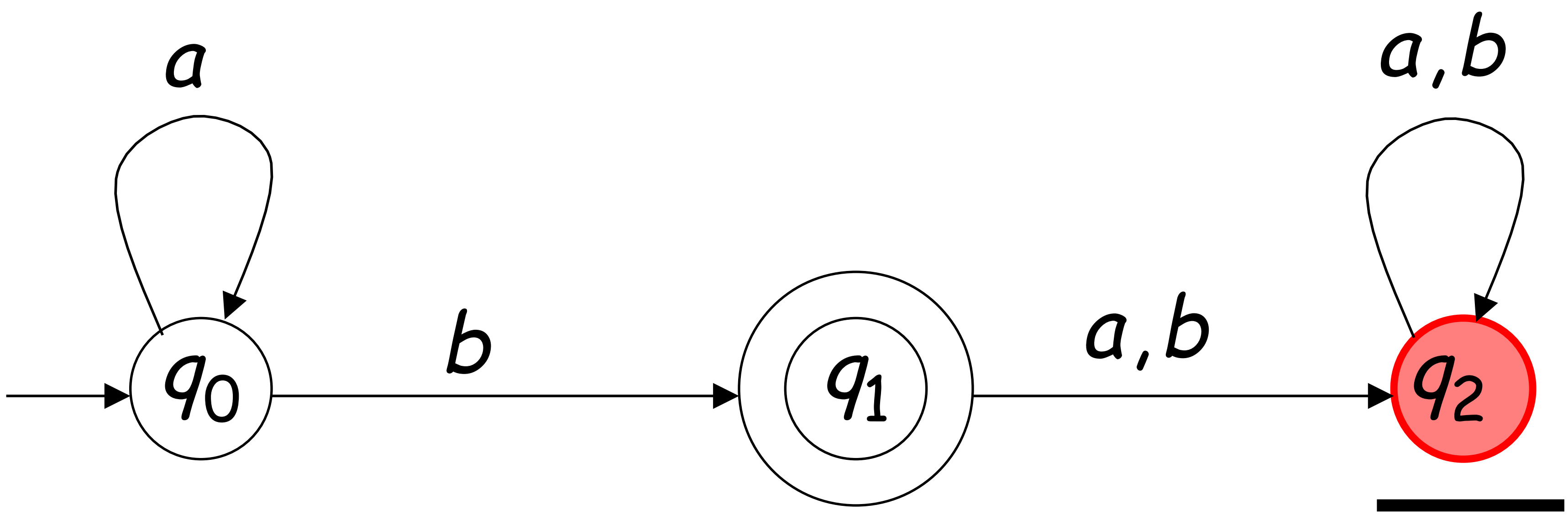

b
a
b
a
b
q_0
q_1
a,b
a,b
q_2


b
a
b
a
a,b
b
q_0
q_1
a,b
q_2

Input finished



Which strings are accepted?



Output: "reject"

Formalities

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q : set of states

Σ : input alphabet

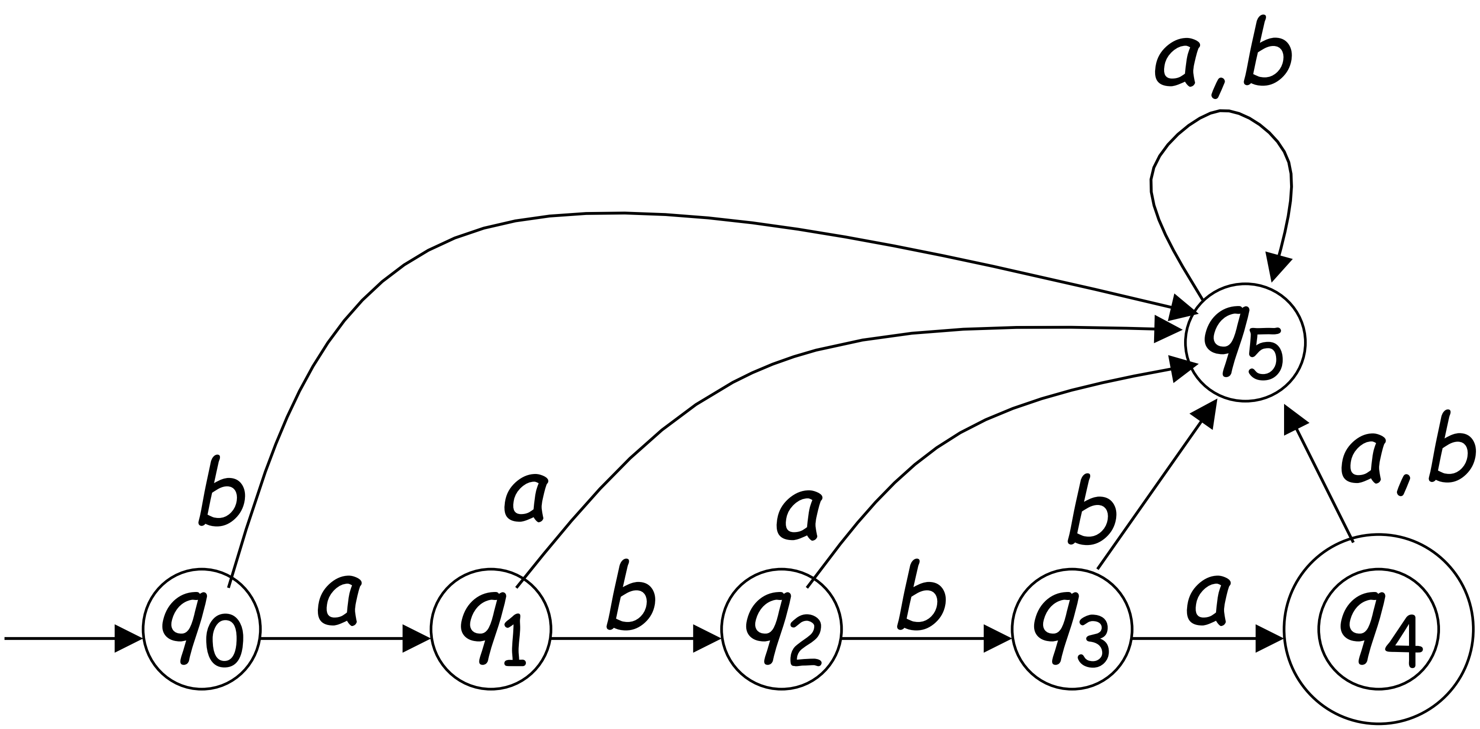
δ : transition function

q_0 : initial state

F : set of final (accepting) states

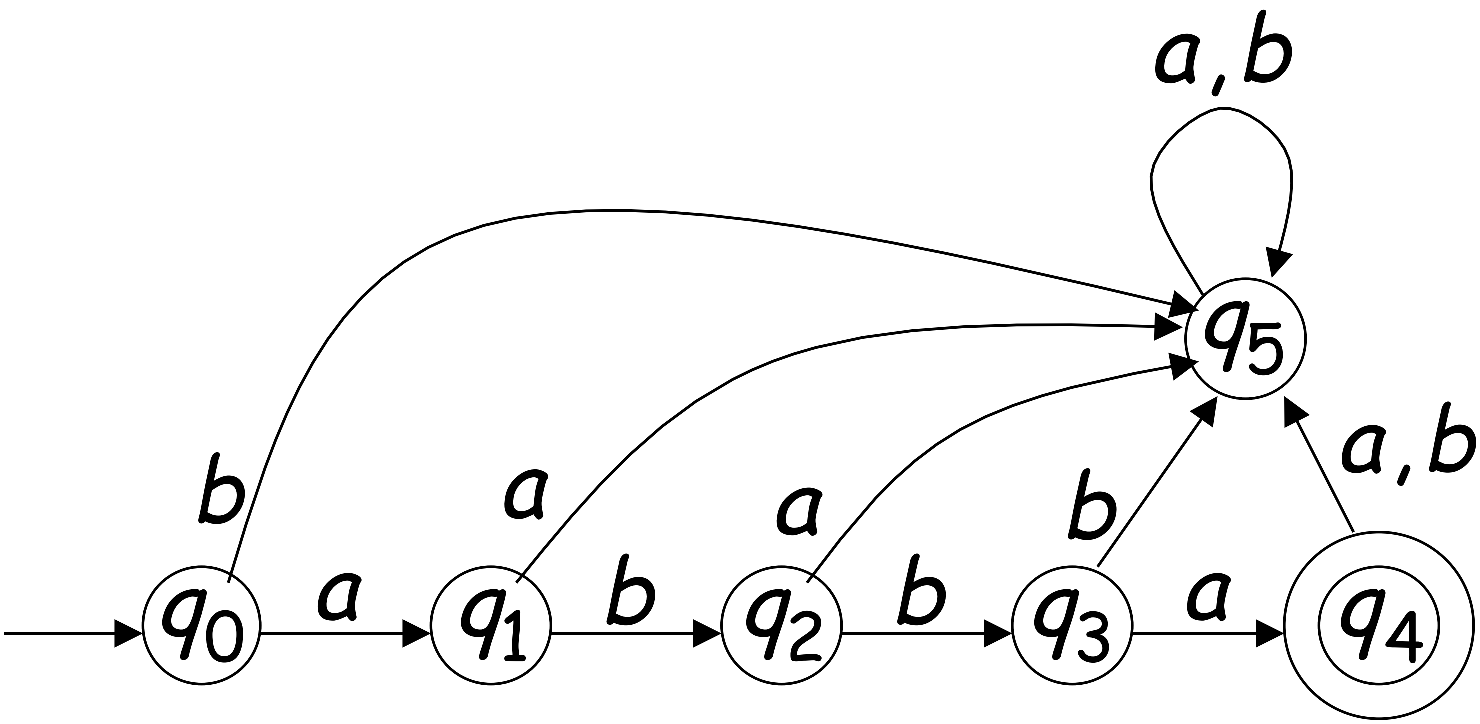
Input Alphabet Σ

$$\Sigma = \{a, b\}$$

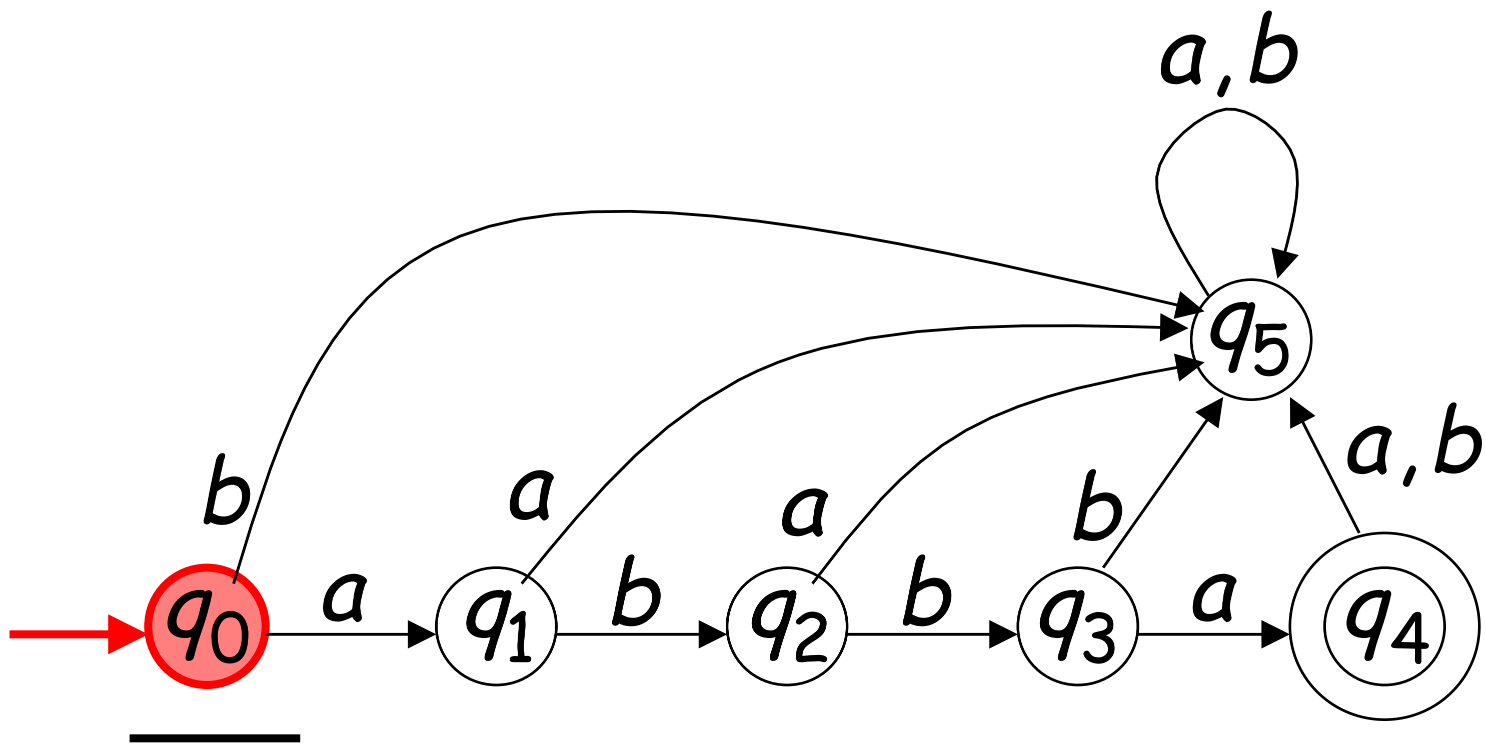


Set of States Q

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

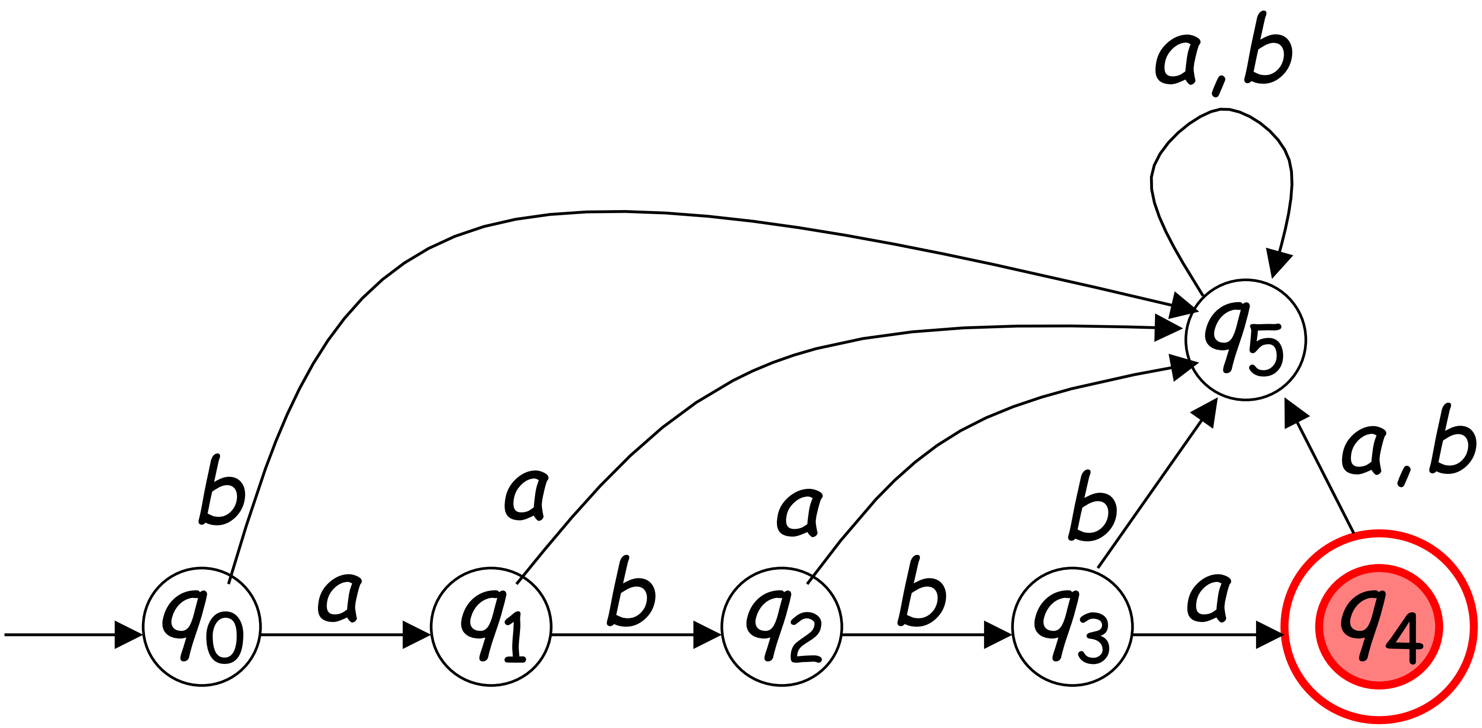


Initial State q_0



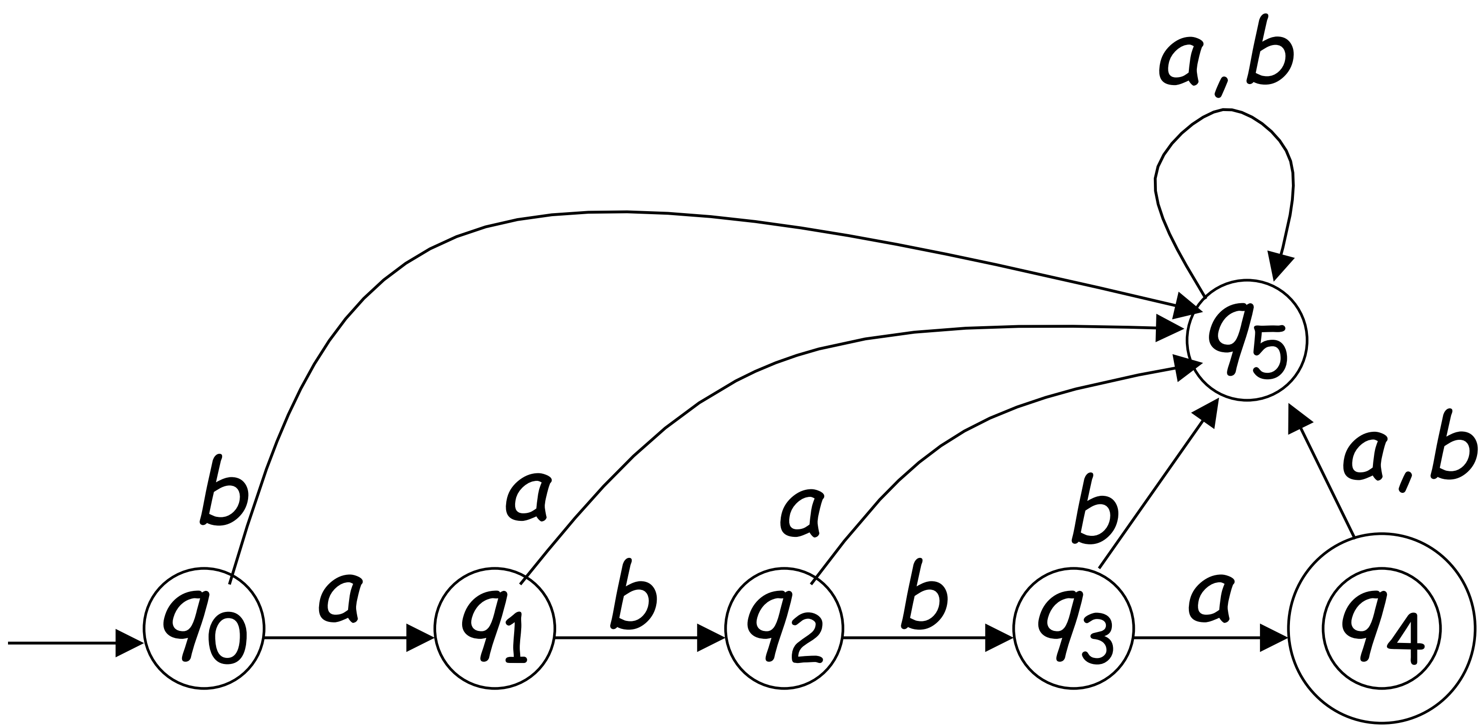
Set of Final States F

$$F = \{q_4\}$$

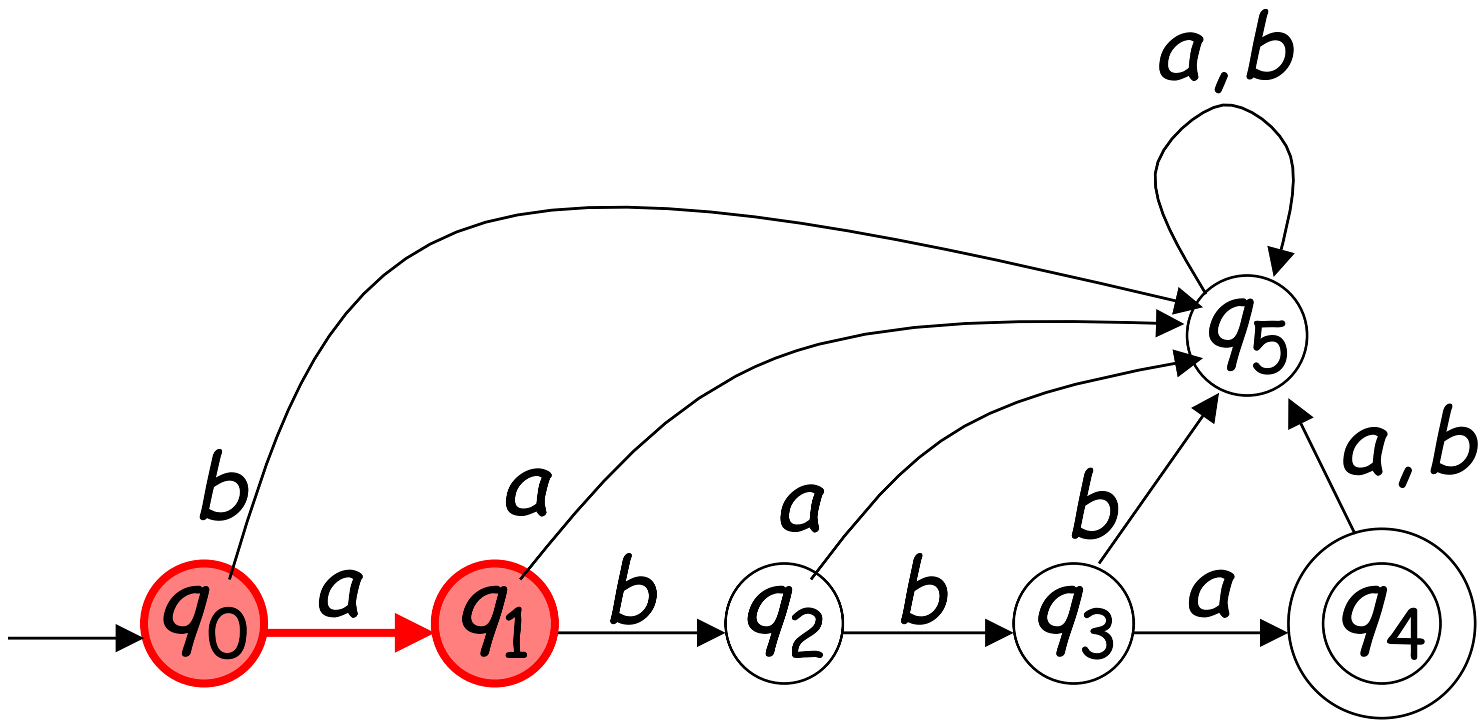


Transition Function δ

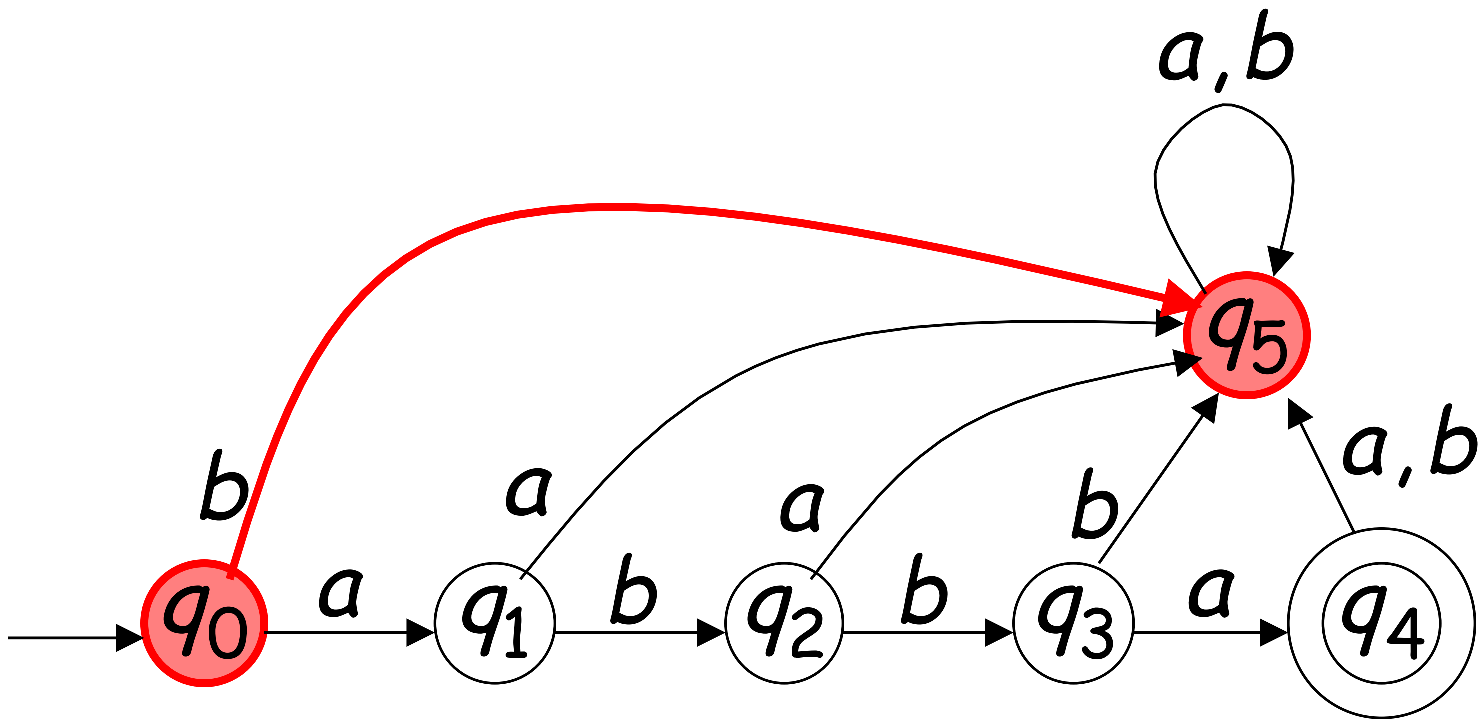
$$\delta : Q \times \Sigma \rightarrow Q$$



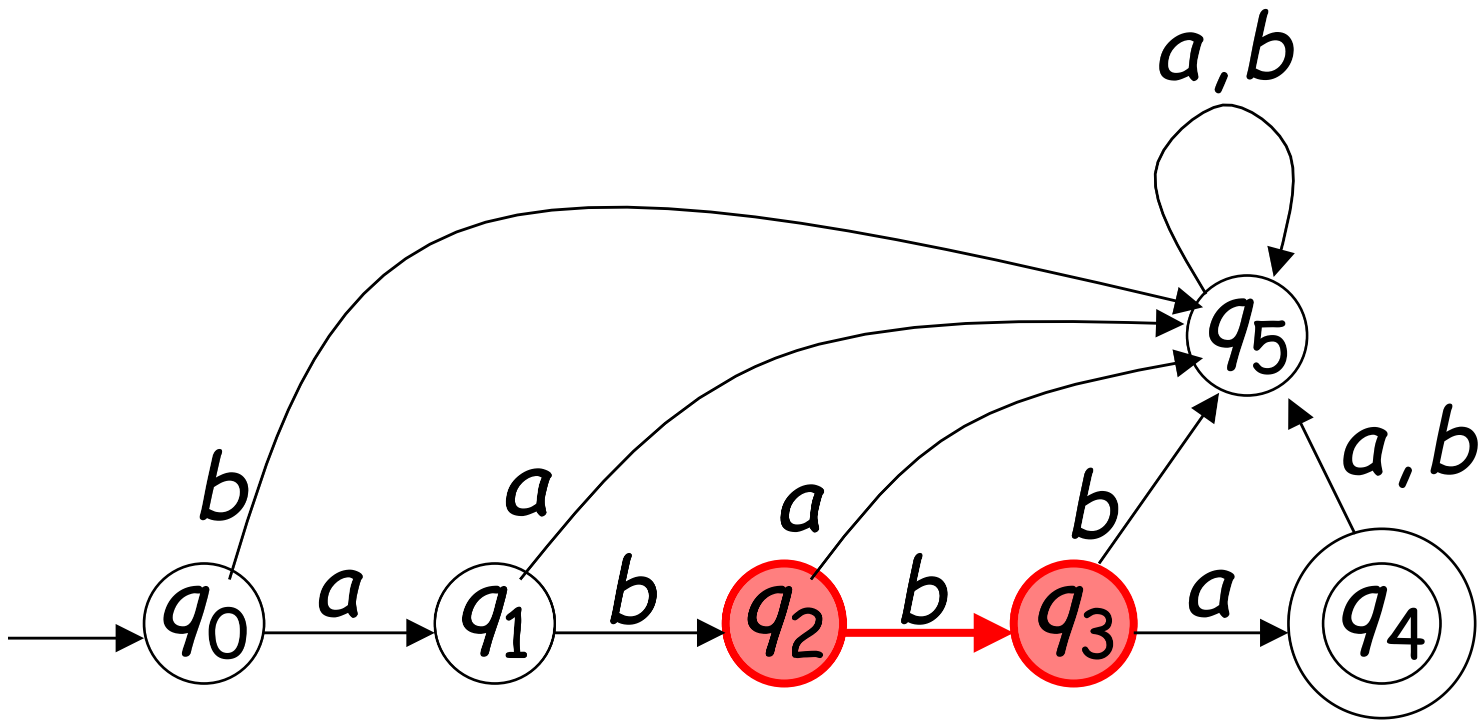
$$\delta(q_0, a) = q_1$$



$$\delta(q_0, b) = q_5$$

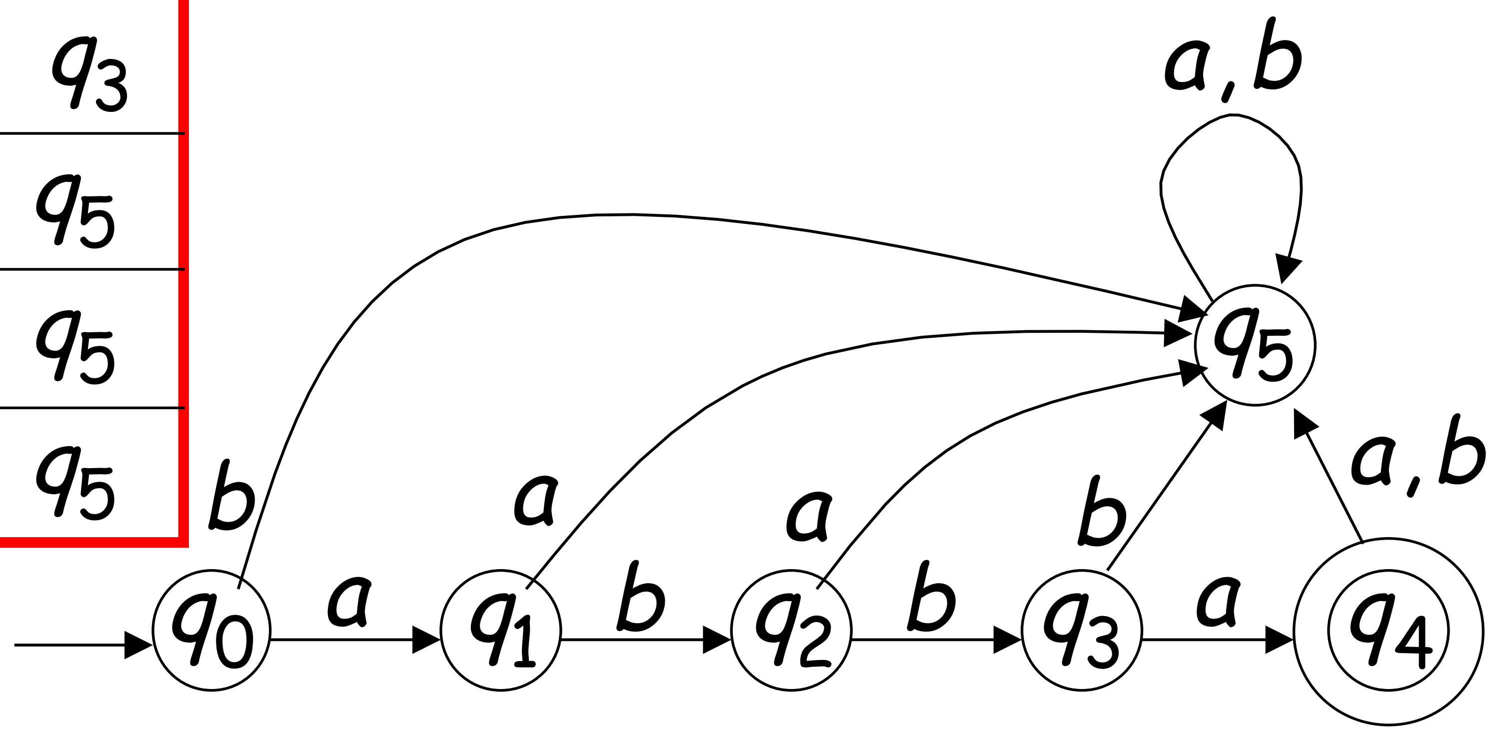


$$\delta(q_2, b) = q_3$$



Transition Function / Table δ

δ	a	b
q_0	q_1	q_5
q_1	q_5	q_2
q_2	q_5	q_3
q_3	q_4	q_5
q_4	q_5	q_5
q_5	q_5	q_5



Complications

1. "1234" is an NUMBER but what about the “123” in “1234” or the “23”, etc. Also, the scanner must recognize many tokens, not one, only stopping at end of file.

2. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID. We want to recognize IF.

3. We want to discard white space and comments.

4. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd”, we want to recognize a token, but add attributes to it.

Before Next Time

HW2: Due tonight!

PA1: It is due in 6 days. Should be almost done.

Read Chapters 2 and 3 in the online book.