Automata, Transducers, and Hidden Markov Models

Natalie Parde, Ph.D.

Department of Computer Science

University of Illinois at Chicago

CS 421: Natural Language Processing

Fall 2019

Many slides adapted from Jurafsky and Martin (https://web.stanford.edu/~jurafsky/slp3/) and Universiteit Utrecht's NLP course (http://www.phil.uu.nl/tst/2012/Slides/SLP_Lecture2.pdf).

What are finite state automata?

- **Computational models that can generate regular languages** (such as those specified by a regular expression)
- Also used in other NLP applications that function by **transitioning between finite states**
  - Dialogue systems
  - Morphological parsing
- Singular: Finite State Automaton (FSA)
- Plural: Finite State Automata (FSAs)

Key Components

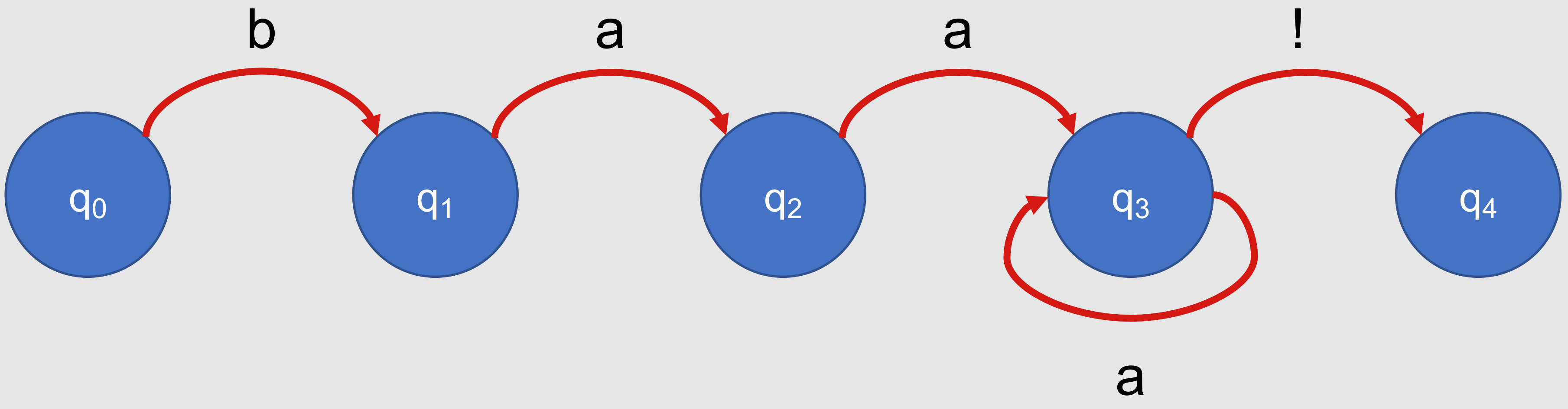
- Finite set of states
  - Start state
  - Final state
- Set of transitions from one state to another

How do FSAs work?

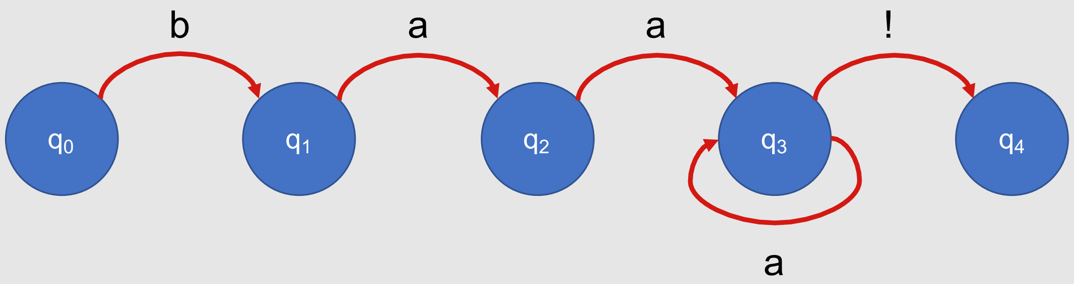
- For a given sequence of items (characters, words, etc.) to match, **begin in the start state**
- **If the next item** in the sequence **matches a state that can be transitioned to** from the current state, **go to that state**
- **Repeat**
  - If no transitions are possible, **stop**
  - If the state you stopped in is a final state, **accept the sequence**

FSAs are often represented graphically.

- Nodes = states
- Arcs = transitions



What do we know about this FSA?

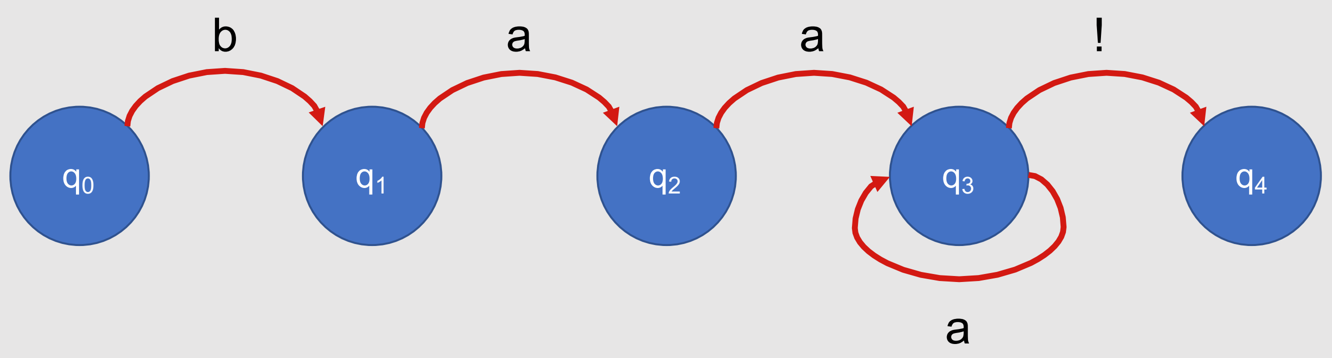


- Five states
  - q_0 is the start state
  - q_4 is the final (accept) state
- Five transitions
- Alphabet = {a, b, !}

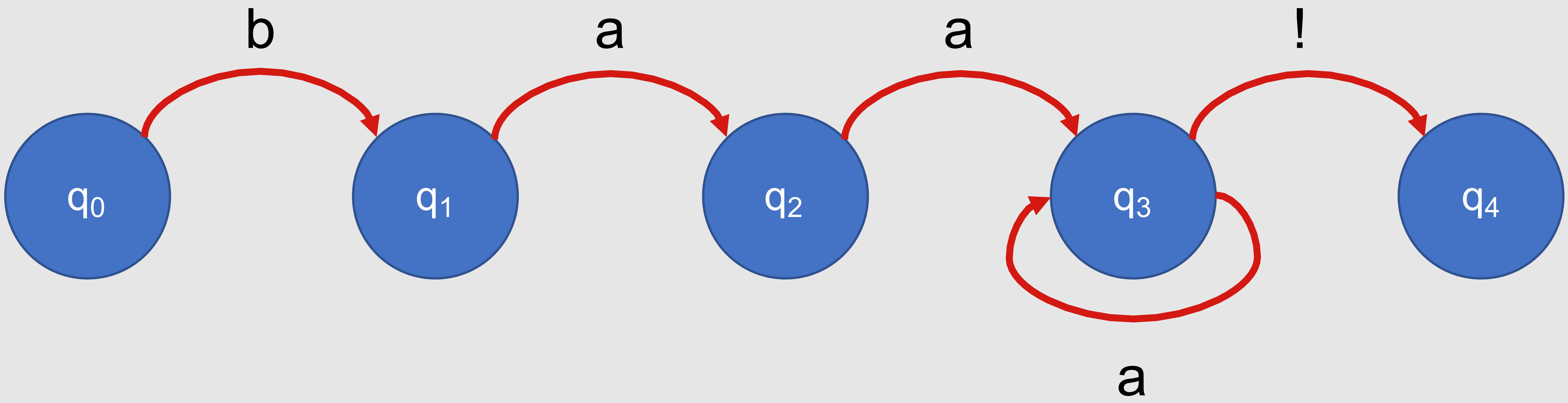
Which strings could this FSA match?

- baa!
- baaaa!
- ba!
- baaaaaaaa!
- baaaa
- baabaa!

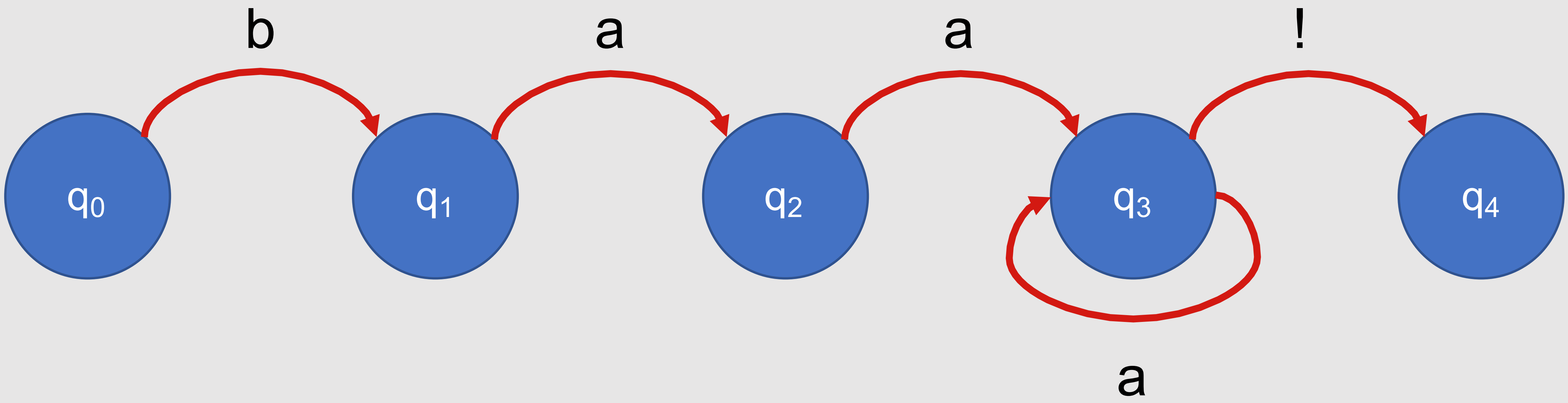
- https://www.google.com/search?q=timer



Regex that this FSA matches: baa+!

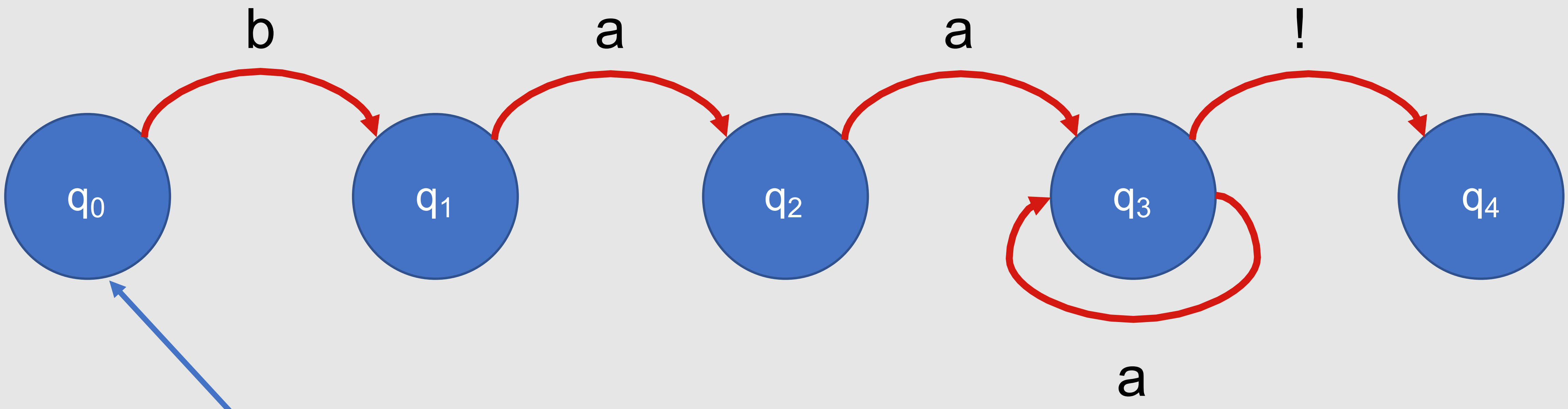


Regex that this FSA matches: baa+!



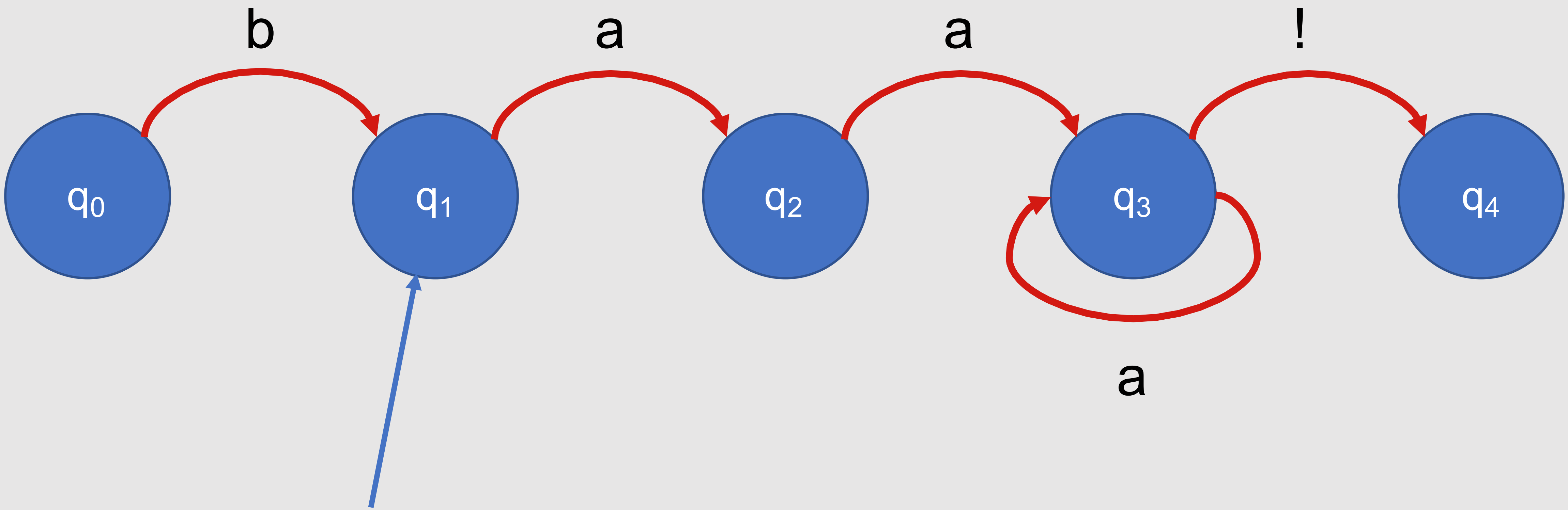
Test String: baa!

Regex that this FSA matches: baa+!



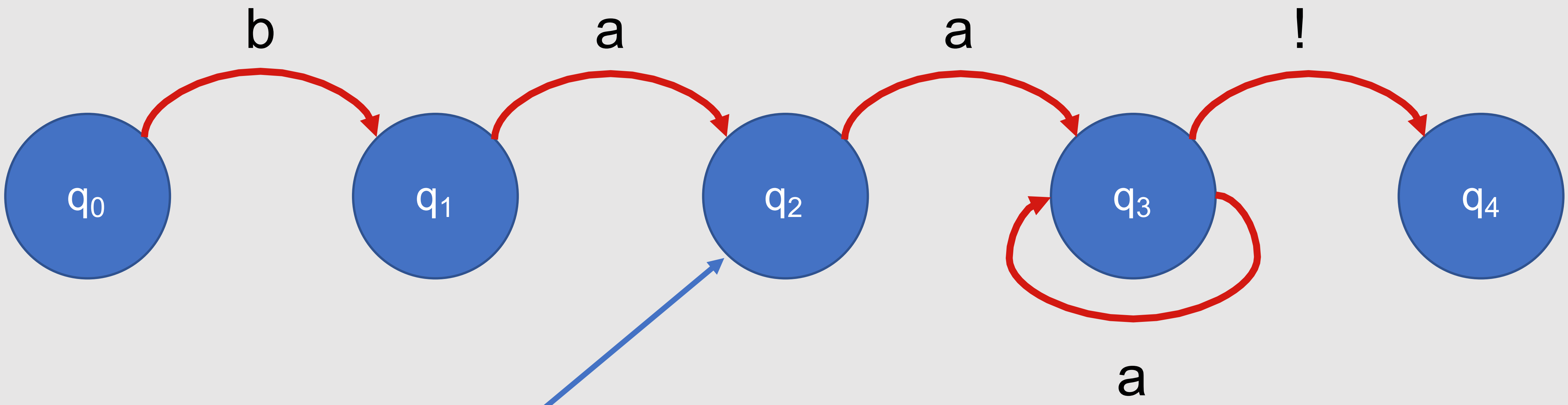
Test String: baa!

Regex that this FSA matches: baa+!



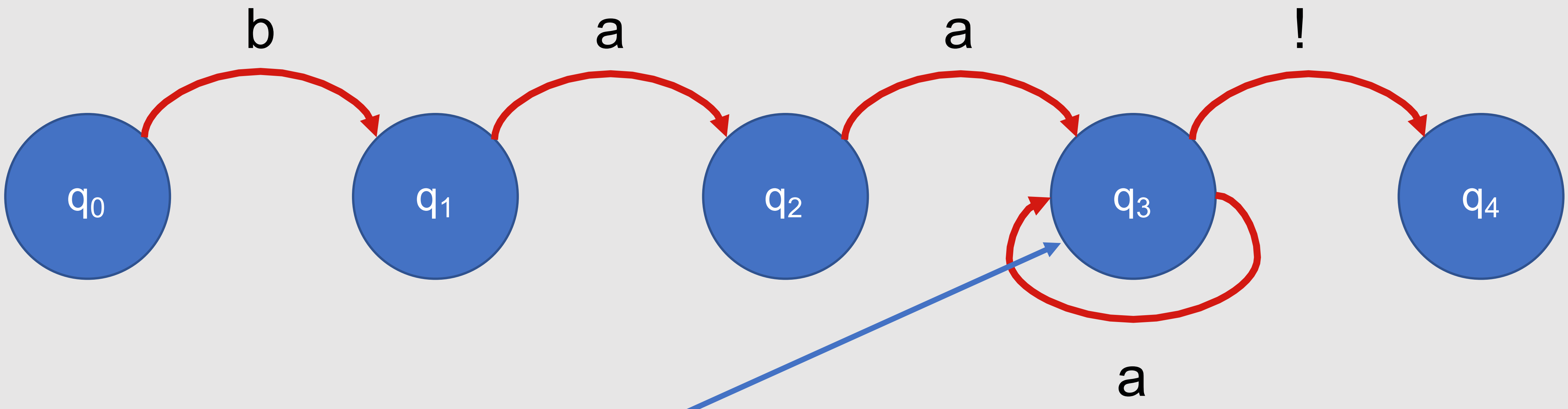
Test String: baa!

Regex that this FSA matches: baa+!



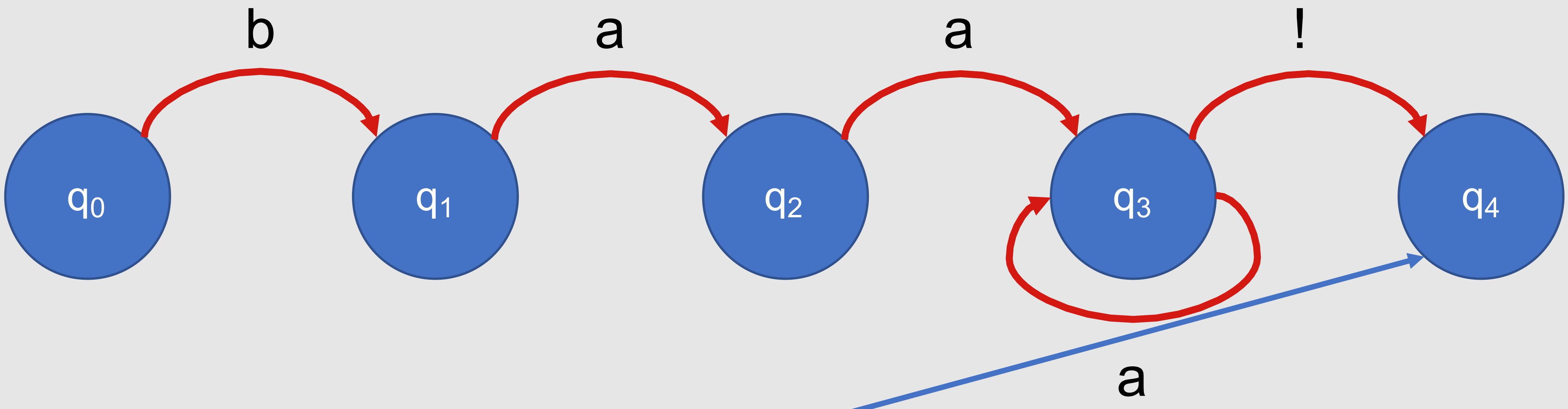
Test String: baa!

Regex that this FSA matches: baa+!



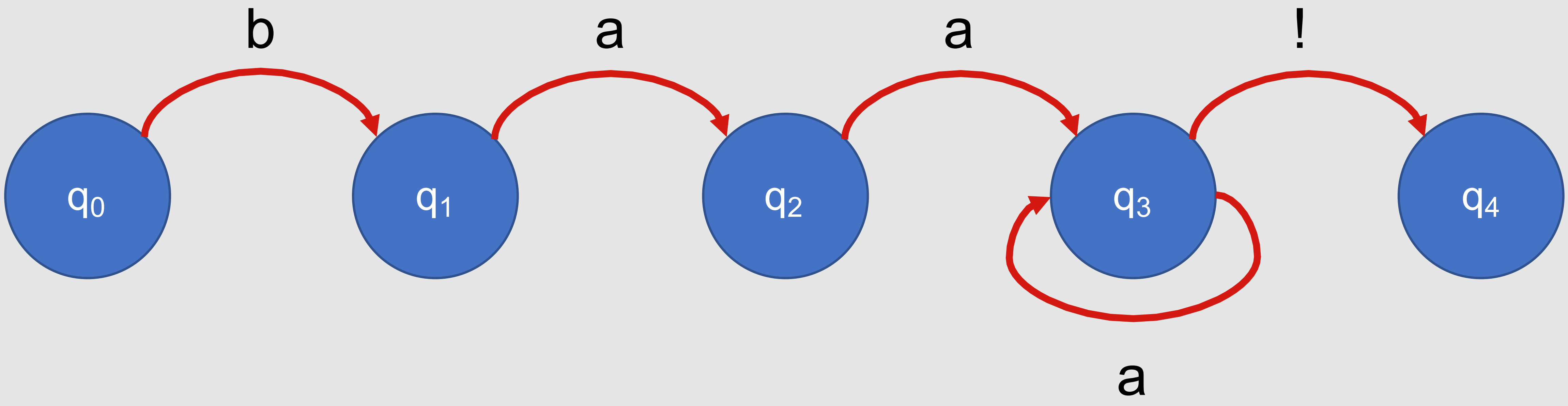
Test String: baa!

Regex that this FSA matches: baa+!



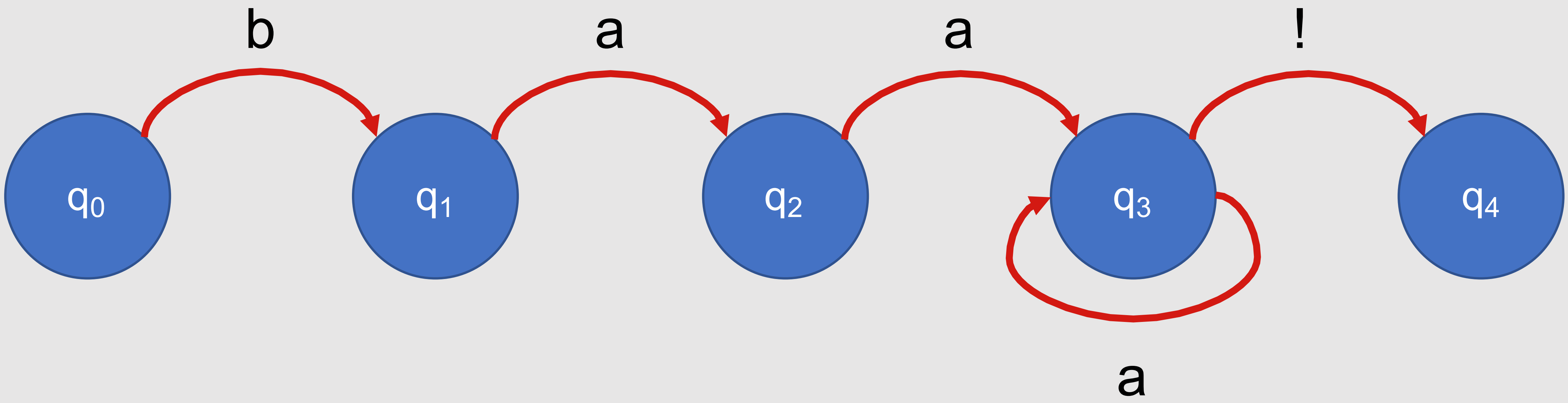
Test String: baa!

Regex that this FSA matches: baa+!



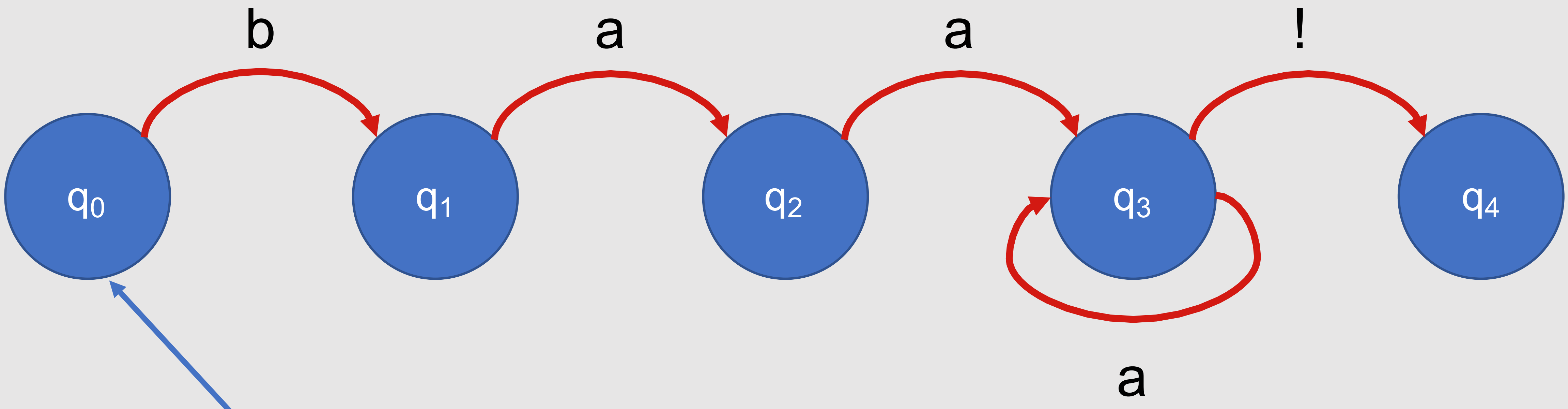
Test String: baa!

Regex that this FSA matches: baa+!



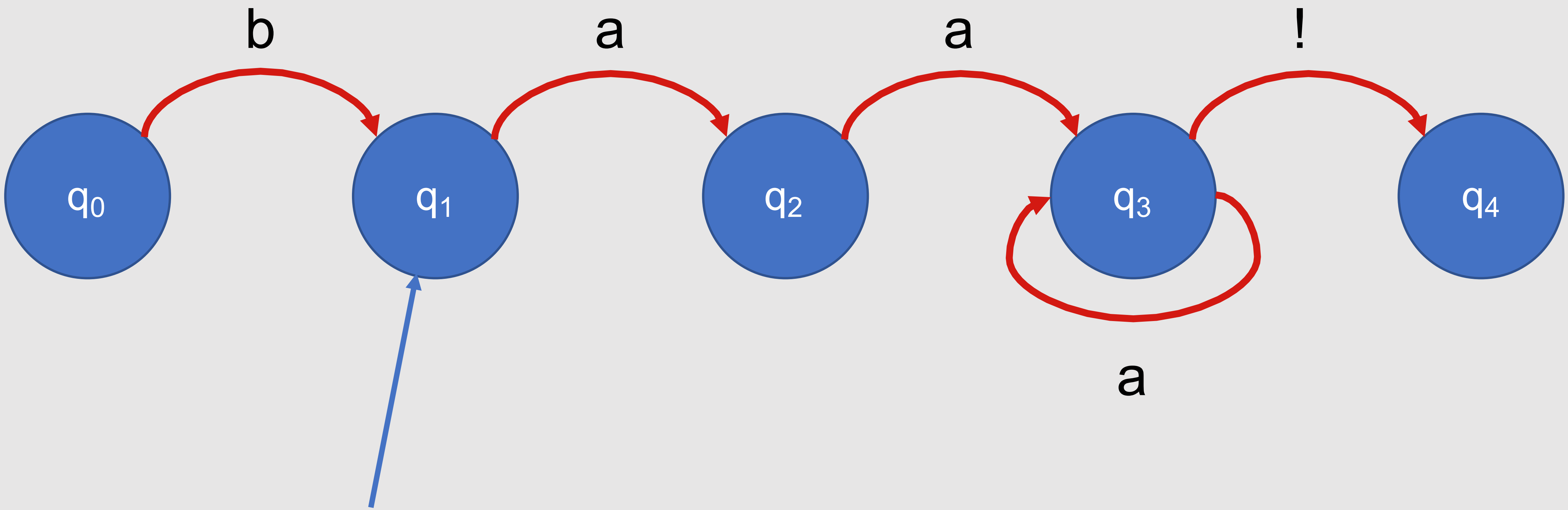
Test String: baabaa!

Regex that this FSA matches: baa+!



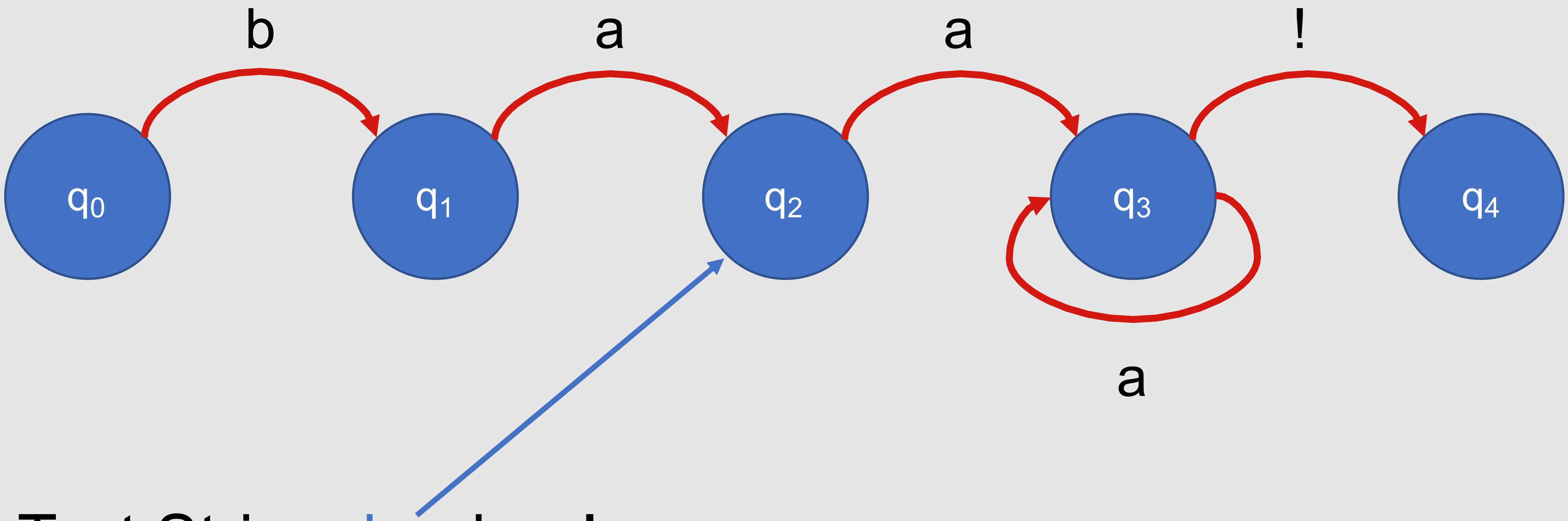
Test String: baabaa!

Regex that this FSA matches: baa+!



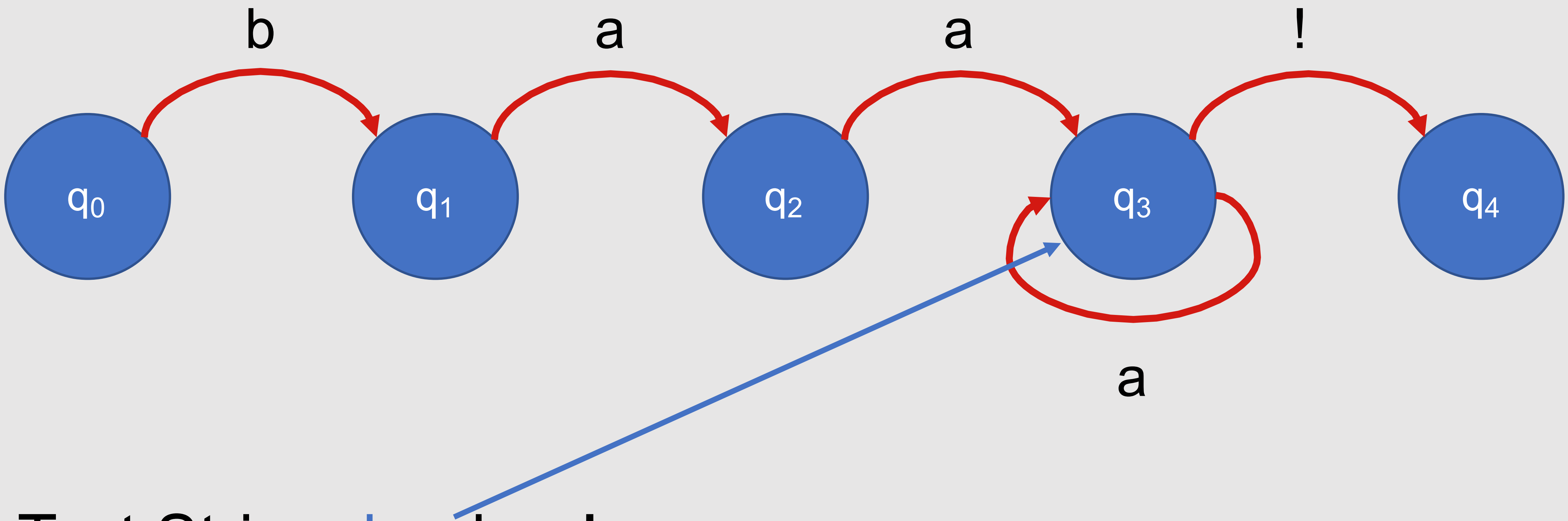
Test String: baabaa!

Regex that this FSA matches: baa+!



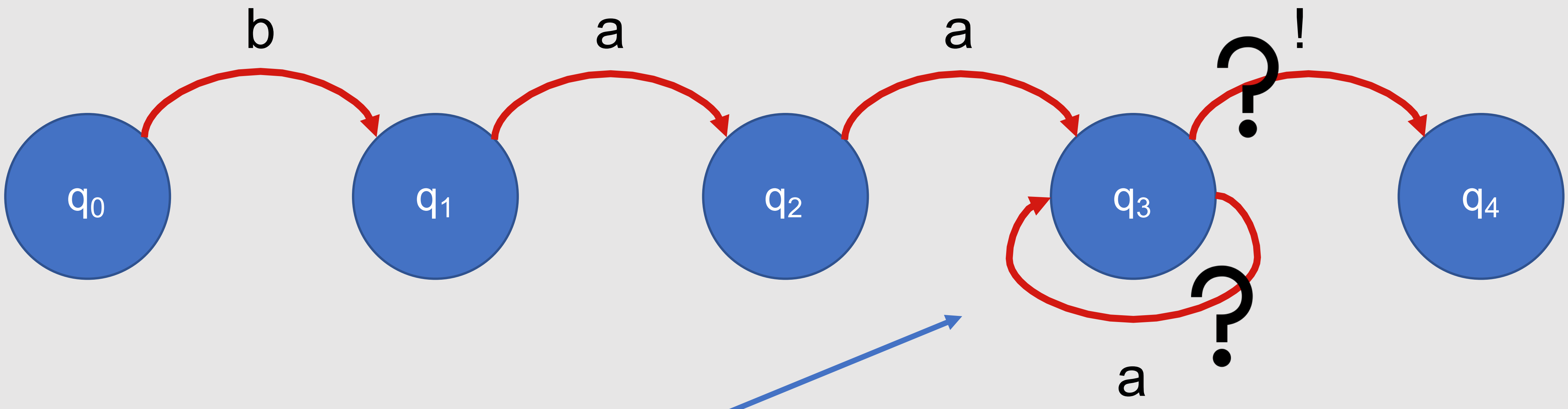
Test String: baabaa!

Regex that this FSA matches: baa+!



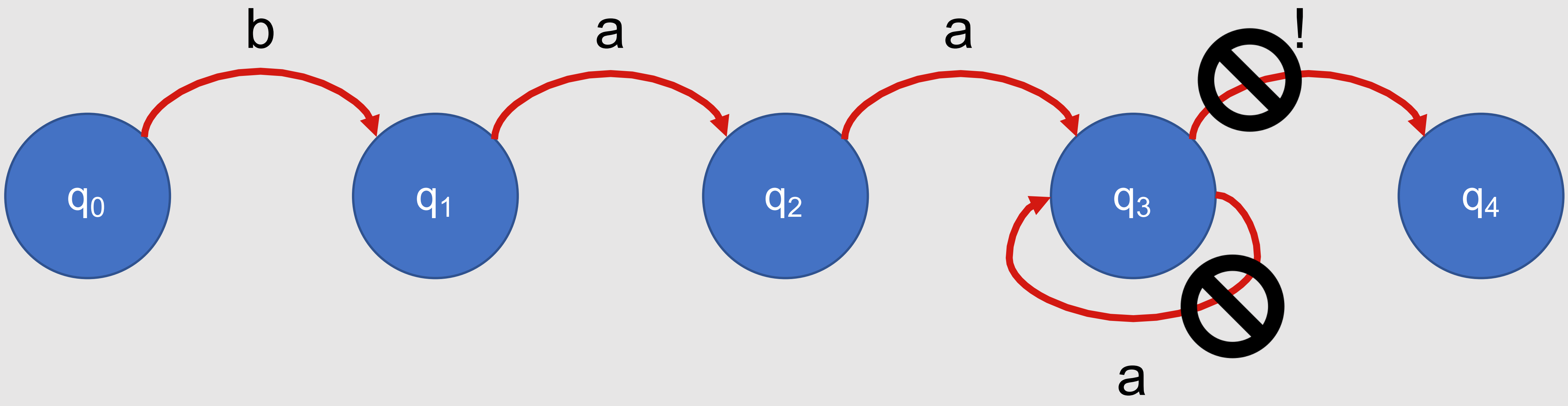
Test String: baabaa!

Regex that this FSA matches: baa+!



Test String: baabaa!

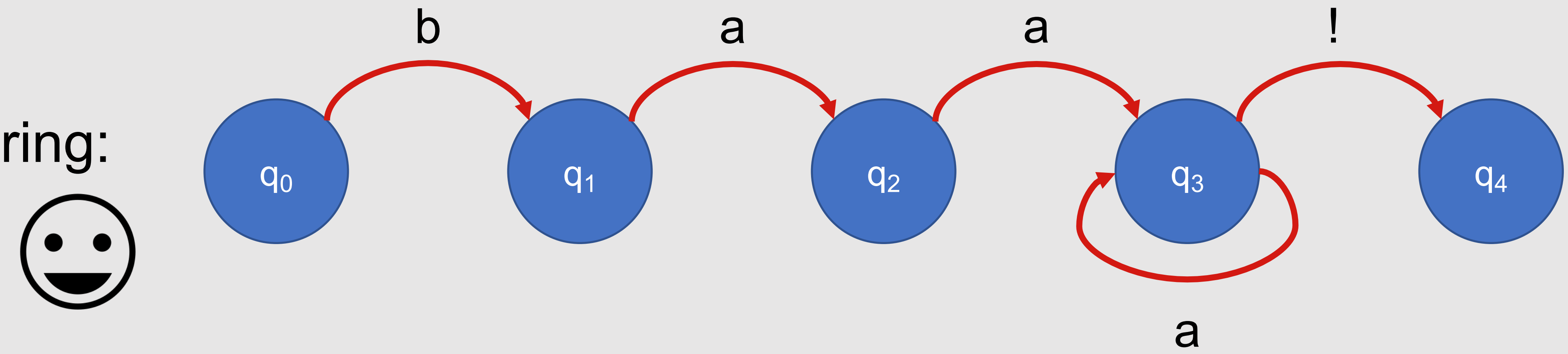
Regex that this FSA matches: baa+!



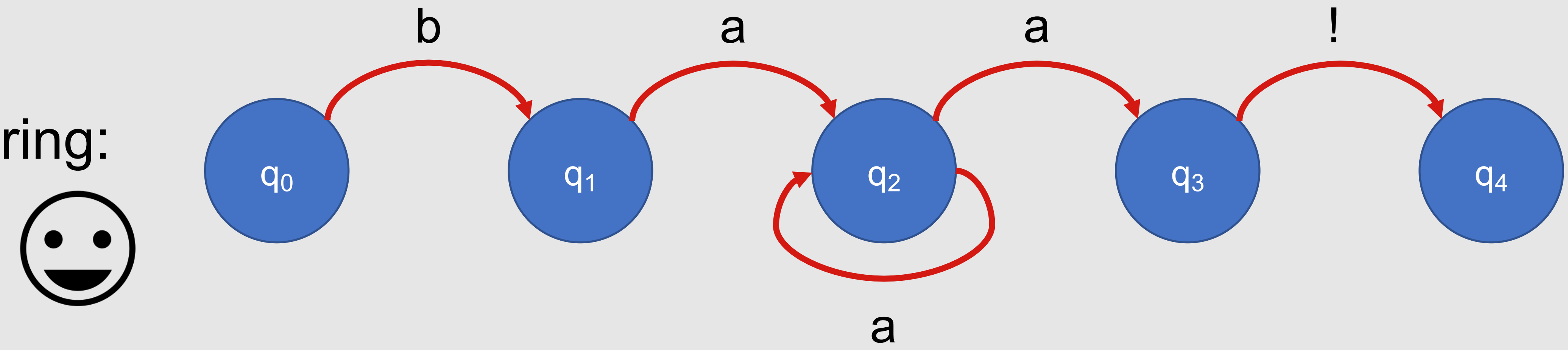
Test String: baabaa! ☹

Note: More than one FSA can correspond to the same regular language!

Test String:
baaa!



Test String:
baaa!



Formal Definition

- A finite state automaton can be specified by enumerating the following properties:
  - The set of states, ***Q***
  - A finite alphabet, ***Σ***
  - A start state, q_0
  - A set of accept/final states, $F \subseteq Q$
  - A transition function or transition matrix between states, **δ(*q*,*i*)**
- δ(q,i): Given a state q∈Q and input i∈Σ, **δ(*q*,*i*) returns a new state q'∈Q**.

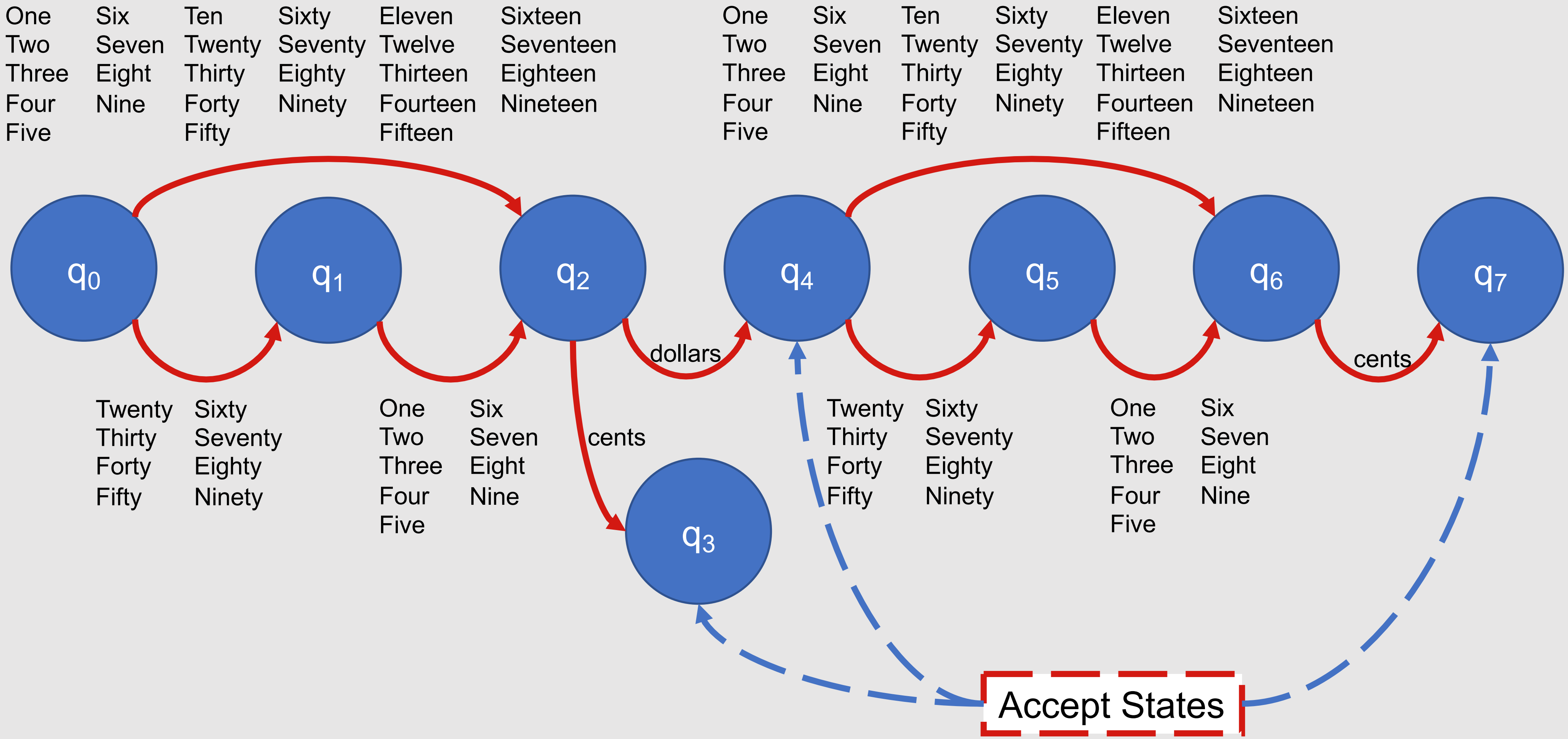
Alphabets

In the previous definition, alphabet does not necessarily mean [a-zA-Z]!

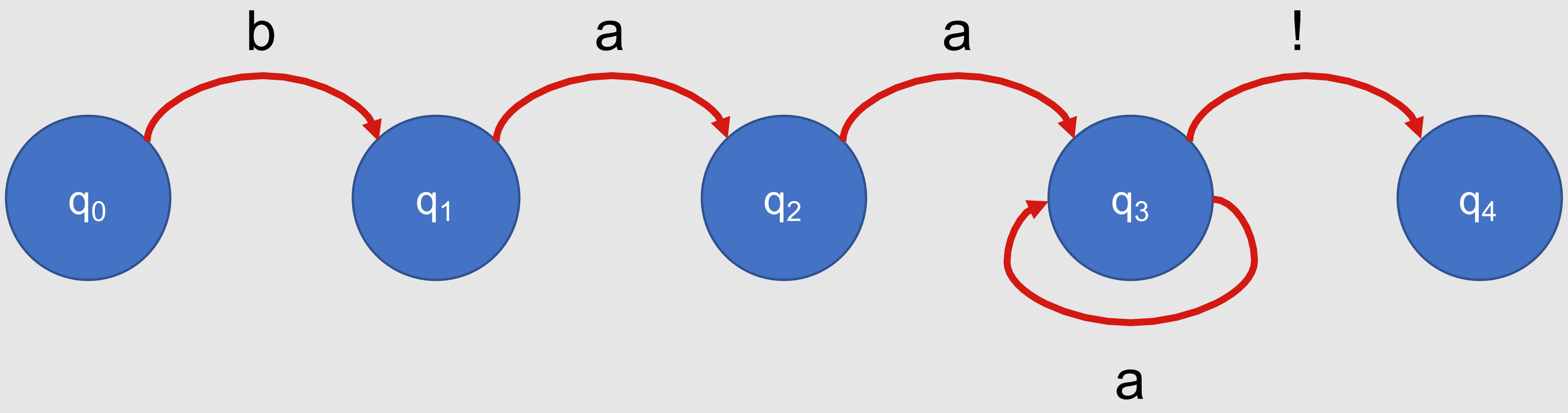
Alphabet = finite set of possible input symbols

An alphabet can be a subset of letters (e.g., {a, b}), a combination of letters and other characters (e.g., {a, b, !}), a subset of words (e.g., {lamb, sheep, baa!}), etc.

Example: FSA for Dollar Amounts



State transitions in FSAs can be represented using tables.



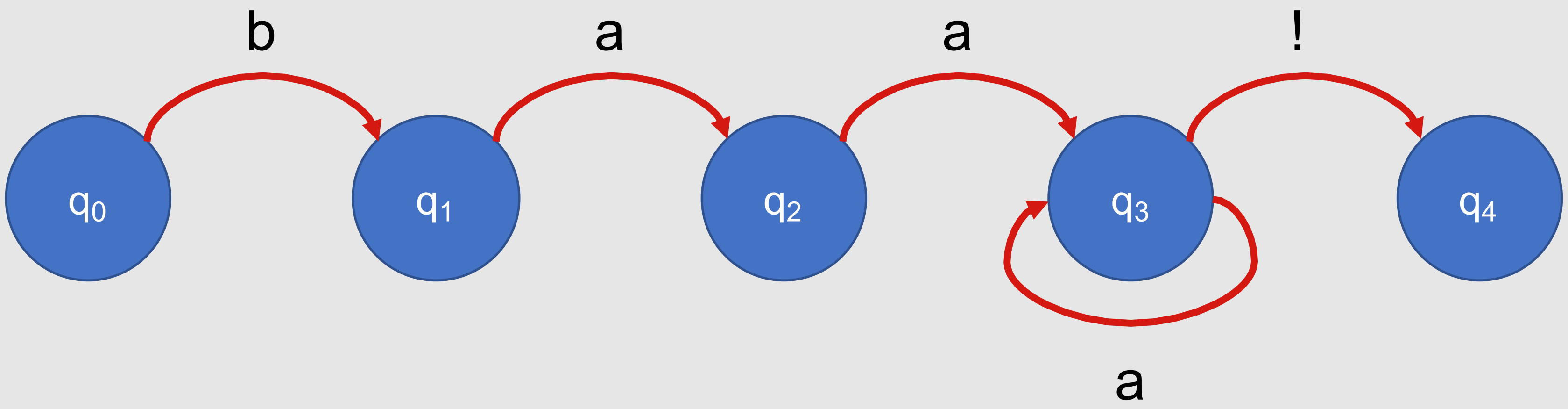
Next Item in Sequence

Currently in State

	b	a	!	<end>
q_0	q_1			
q_1				
q_2				
q_3				
q_4				

Go to State

State transitions in FSAs can be represented using tables.



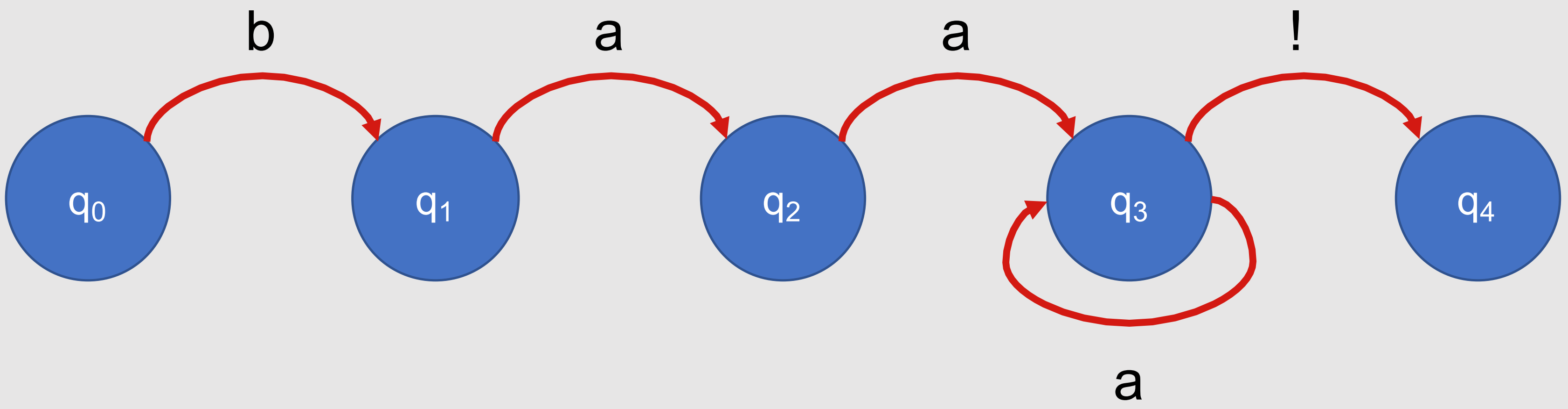
Next Item in Sequence

Currently in State

	b	**a**	**!**	**<end>**
q_0	q_1	☹	☹	☹
q_1				
q_2				
q_3				
q_4				

Go to State

State transitions in FSAs can be represented using tables.



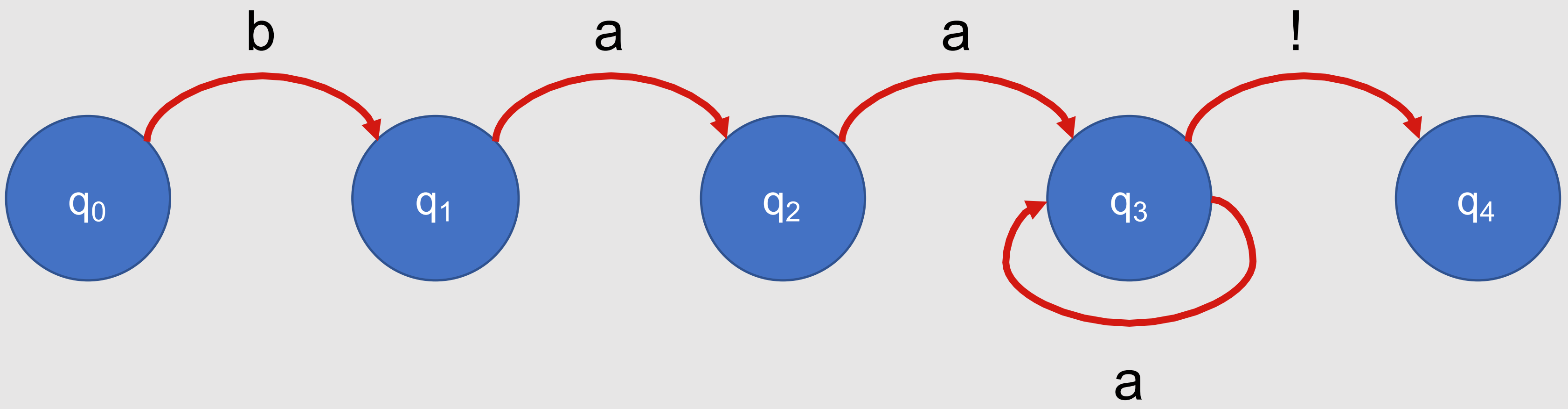
Next Item in Sequence

Currently in State

	b	a	!	<end>
q_0	q_1	☹	☹	☹
q_1	☹	q_2		
q_2				
q_3				
q_4				

Go to State

State transitions in FSAs can be represented using tables.



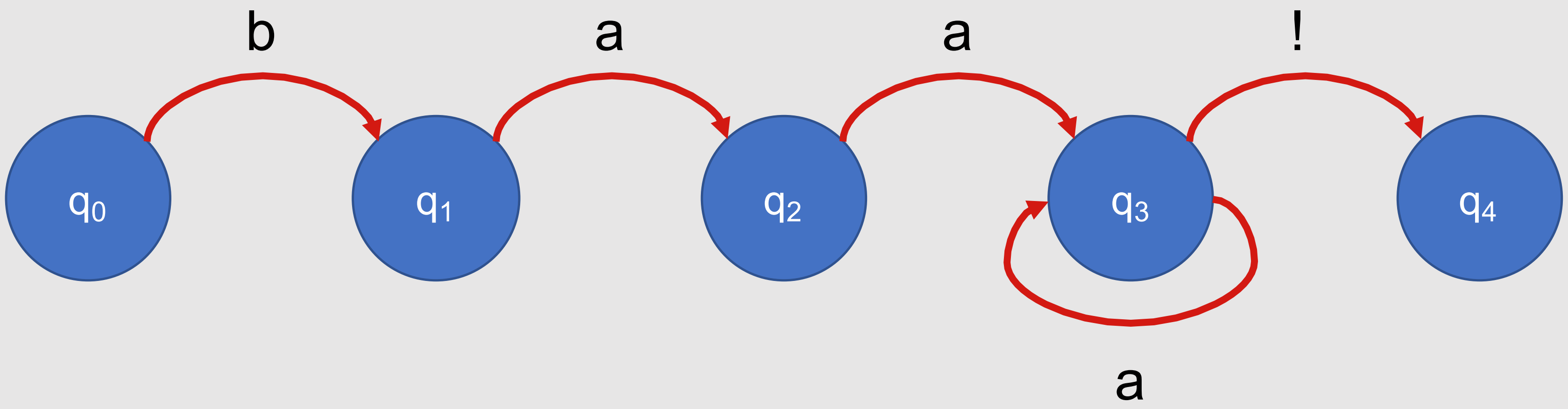
Next Item in Sequence

Currently in State

	b	a	!	<end>
q_0	q_1	☹	☹	☹
q_1	☹	q_2	☹	☹
q_2	☹	q_3		
q_3				
q_4				

Go to State

State transitions in FSAs can be represented using tables.



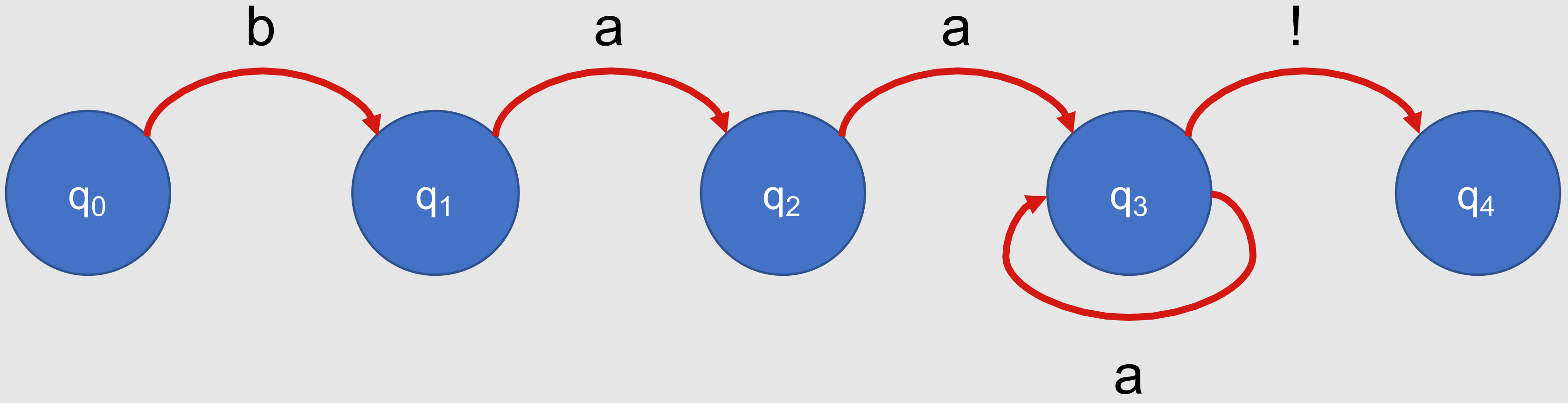
Next Item in Sequence

Currently in State

	b	a	!	<end>
q_0	q_1	☹	☹	☹
q_1	☹	q_2	☹	☹
q_2	☹	q_3	☹	☹
q_3	☹	q_3		
q_4				

Go to State

State transitions in FSAs can be represented using tables.



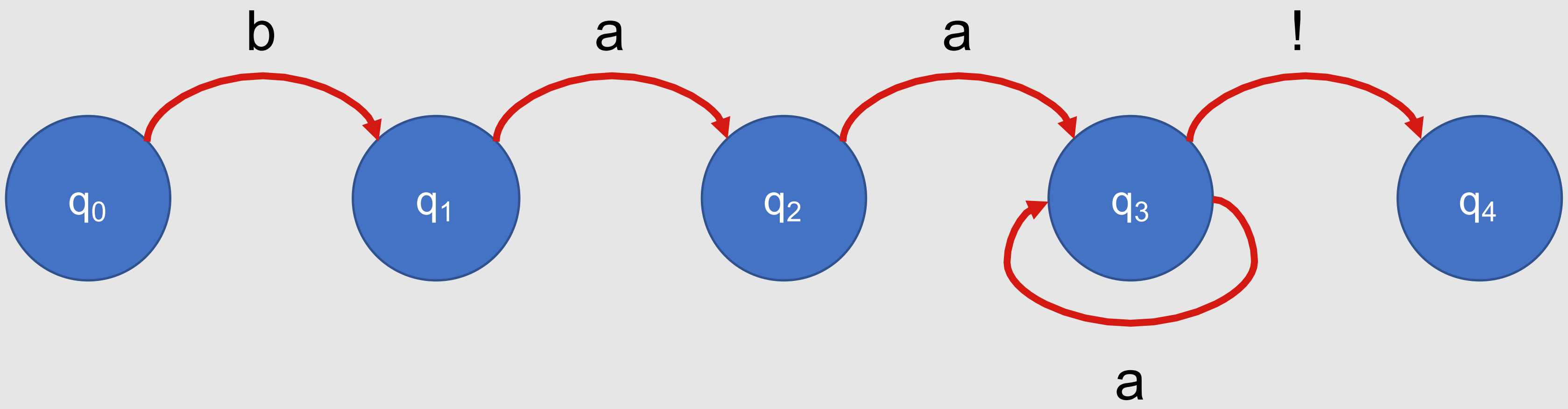
Next Item in Sequence

Currently in State

	b	a	!	<end>
q_0	q_1	☹	☹	☹
q_1	☹	q_2	☹	☹
q_2	☹	q_3	☹	☹
q_3	☹	q_3	q_4	
q_4				

Go to State

State transitions in FSAs can be represented using tables.



Next Item in Sequence

Currently in State

	b	a	!	<end>
q_0	q_1	☹	☹	☹
q_1	☹	q_2	☹	☹
q_2	☹	q_3	☹	☹
q_3	☹	q_3	q_4	☹
q_4	☹	☹	☹	☺

Accept!

State transition tables simplify the process of determining whether your input will be accepted by the FSA.

- For a given sequence of items to match, **begin in the start state** with the first item in the sequence
- **Consult the table** …is a transition to any other state permissible with the current item?
- If so, **move to the state indicated by the table**
- If you make it to the end of your sequence and to a final state, **accept**

Formal Algorithm

```
index ← beginning of sequence
current_state ← initial state of FSA
loop:
       if end of sequence has been reached:
              if current_state is an accept state:
                     return accept
              else:
                     return reject
       else if transition_table[current_state, sequence[index]] is empty:
              return reject
       else:
              current_state ← transition_table[current_state, sequence[index]]
              index ← index + 1
end
```

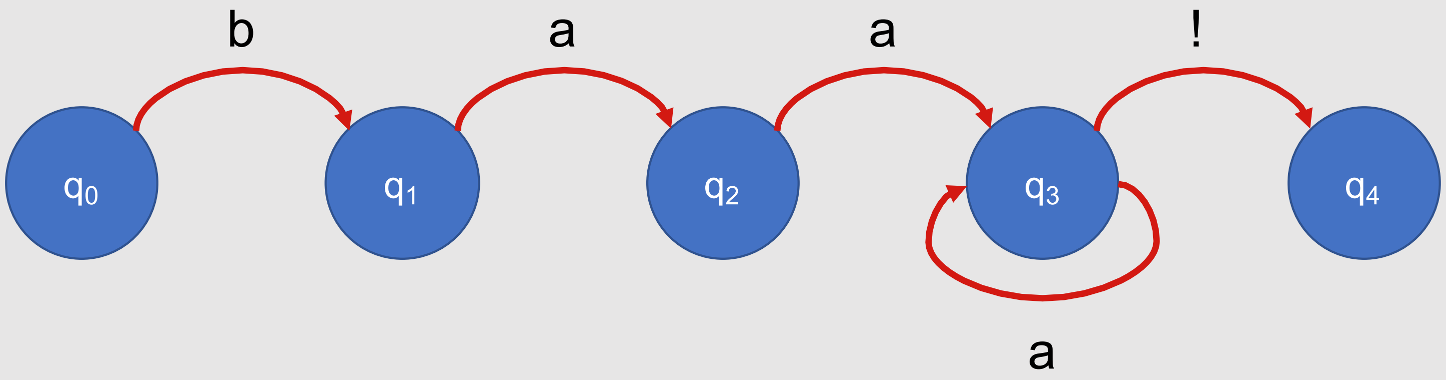
Deterministic vs. Non-Deterministic FSAs

Deterministic FSA: At each point in processing a sequence, there is one unique thing to do (no choices!)

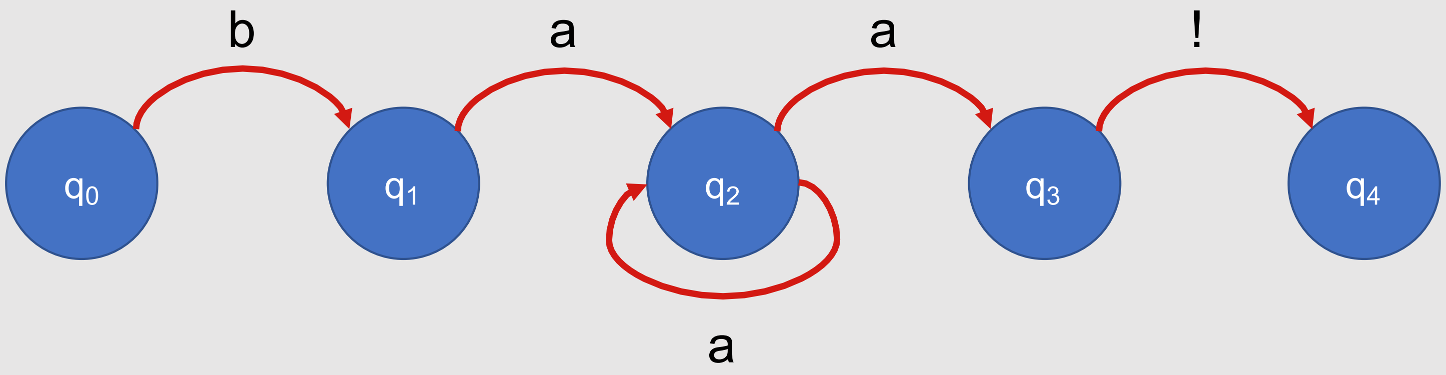
Non-Deterministic FSA: At one or more points in processing a sequence, there are multiple permissible next steps (choices!)

Deterministic or Non-Deterministic?

?

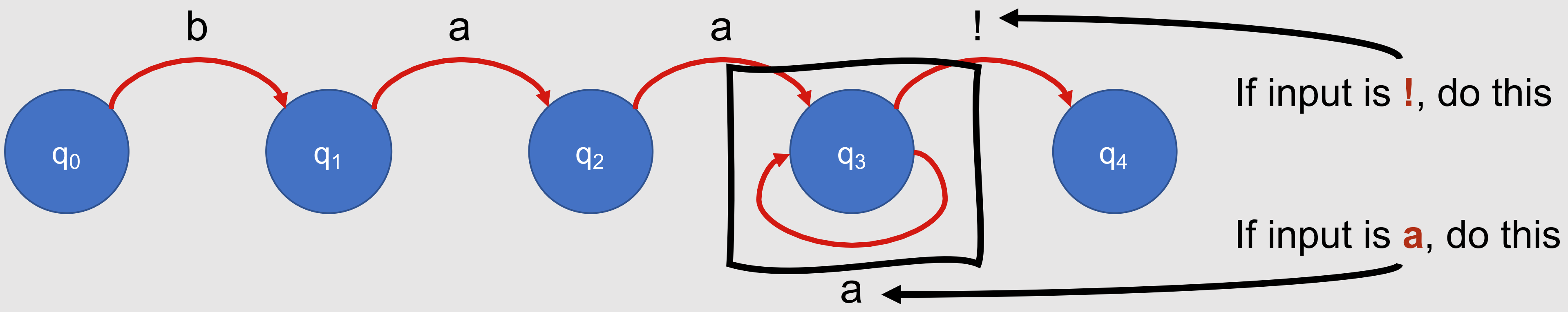


?

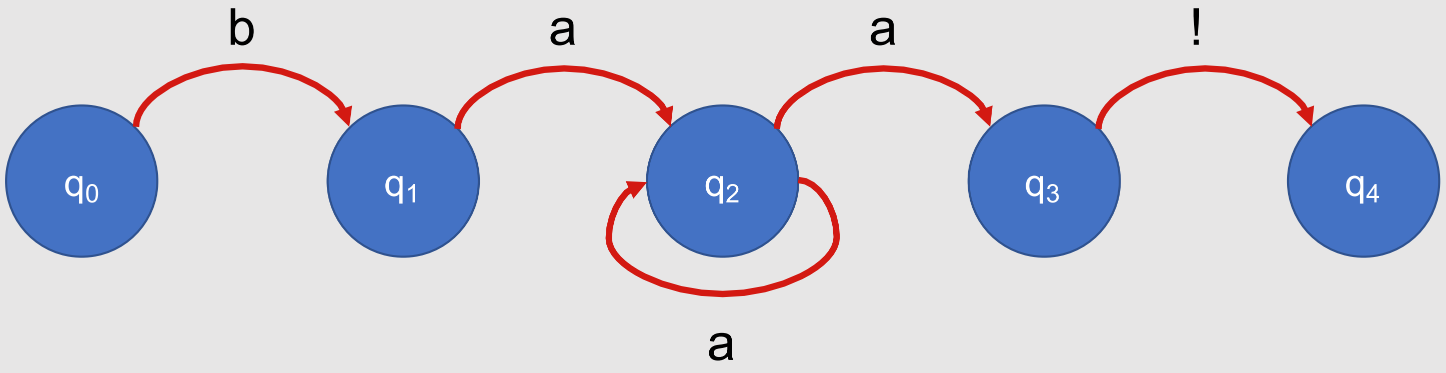


Deterministic or Non-Deterministic?

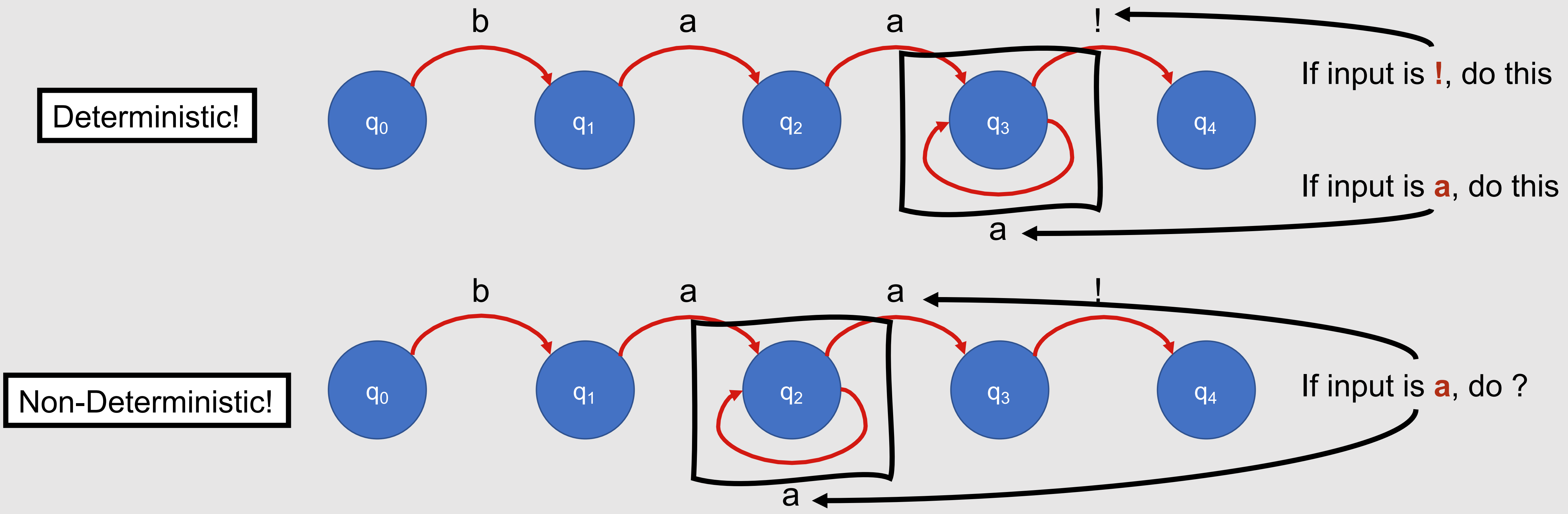
Deterministic!



?



Deterministic or Non-Deterministic?



Every non-deterministic FSA can be converted to a deterministic FSA.

- This means that both are equally powerful!
- Deterministic FSAs can accept as many languages as non-deterministic ones

Non-Deterministic FSAs: How to check for input acceptance?

- Two approaches:
  1. Convert the non-deterministic FSA to a deterministic FSA and then check that version
  2. Manage the process as a state-space search

Non-Deterministic FSA Search Assumptions

There exists at least one path through the FSA for an item that is part of the language defined by the machine

Not all paths directed through the FSA for an accept item lead to an accept state

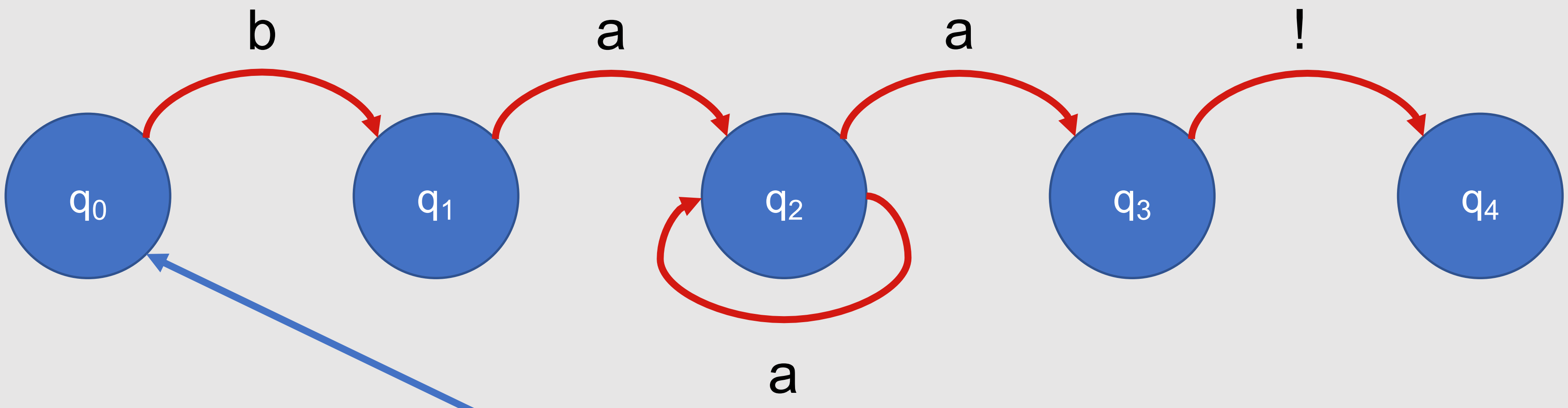
No paths through the FSA lead to an accept state for an item not in the language

Non-Deterministic FSA Search Assumptions

SUCCESS = PATH IS FOUND FOR A GIVEN ITEM THAT ENDS IN AN ACCEPT

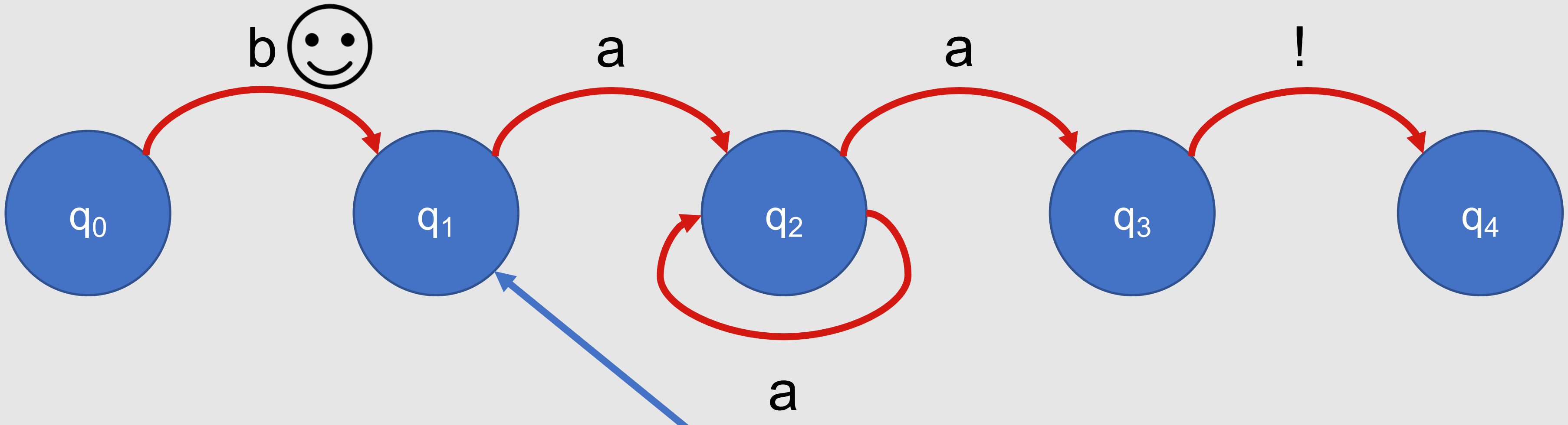
FAILURE = ALL POSSIBLE PATHS FOR A GIVEN ITEM LEAD TO FAILURE

Example: Non-Deterministic FSA Search



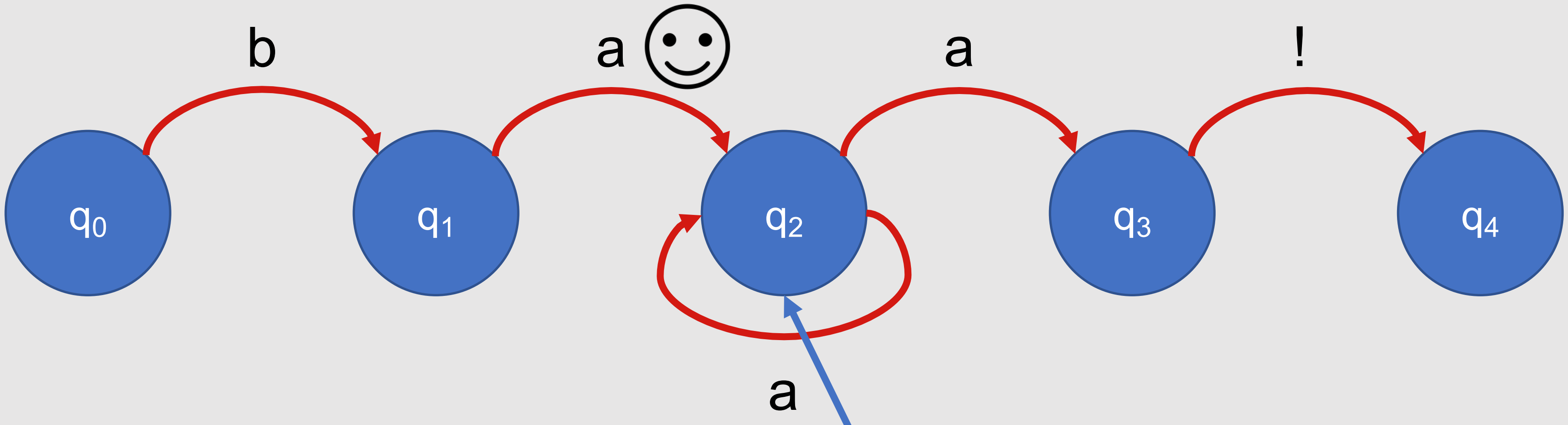
Test Input: baaa!

Example: Non-Deterministic FSA Search



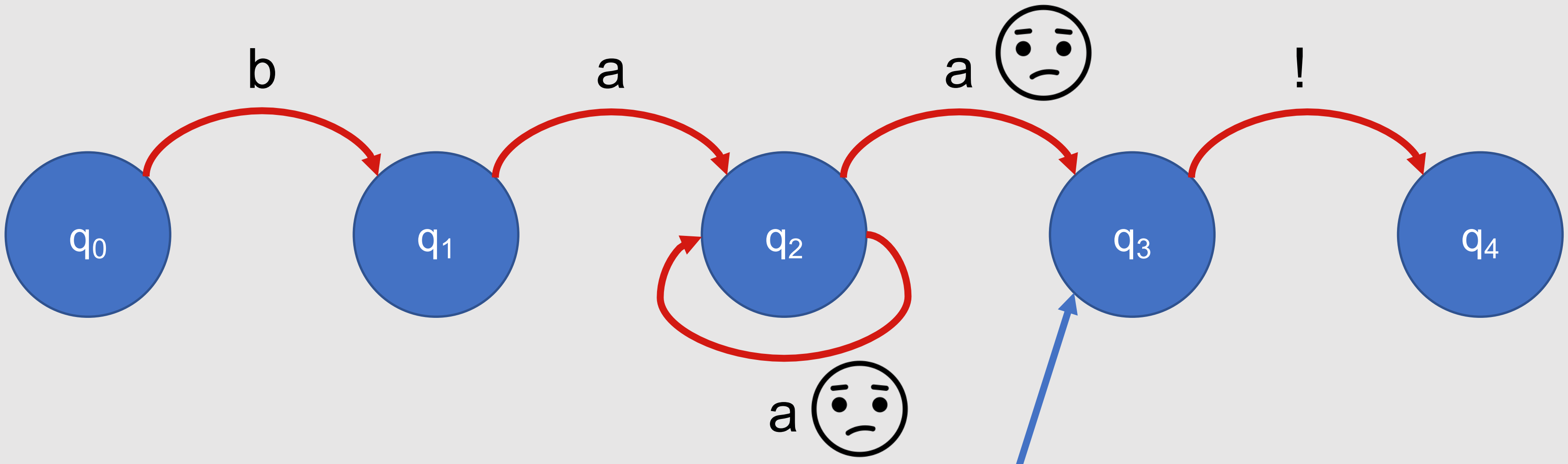
Test Input: **baaa!**

Example: Non-Deterministic FSA Search



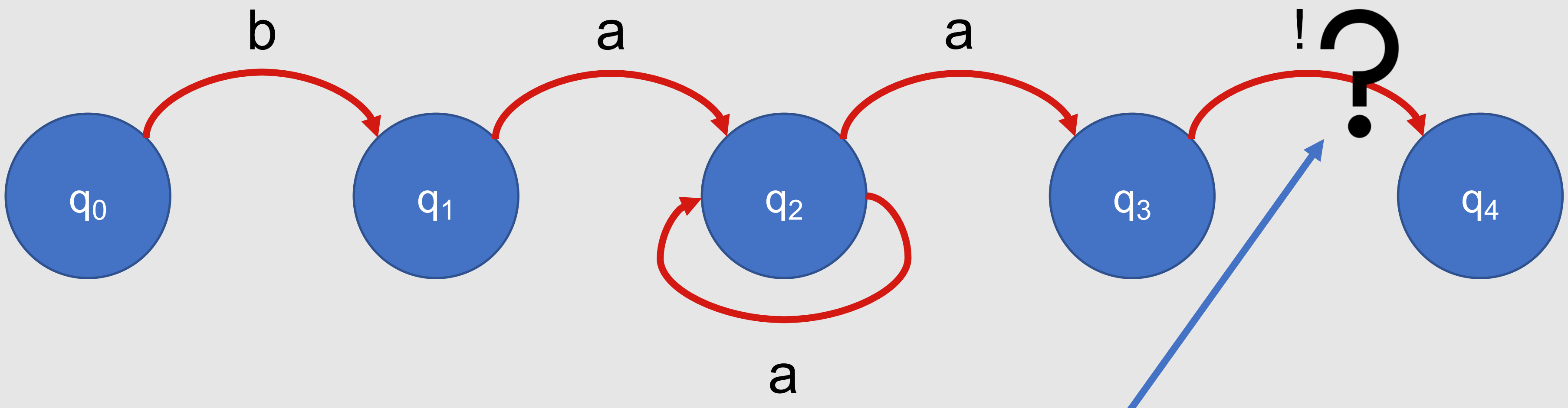
Test Input: baaa!

Example: Non-Deterministic FSA Search



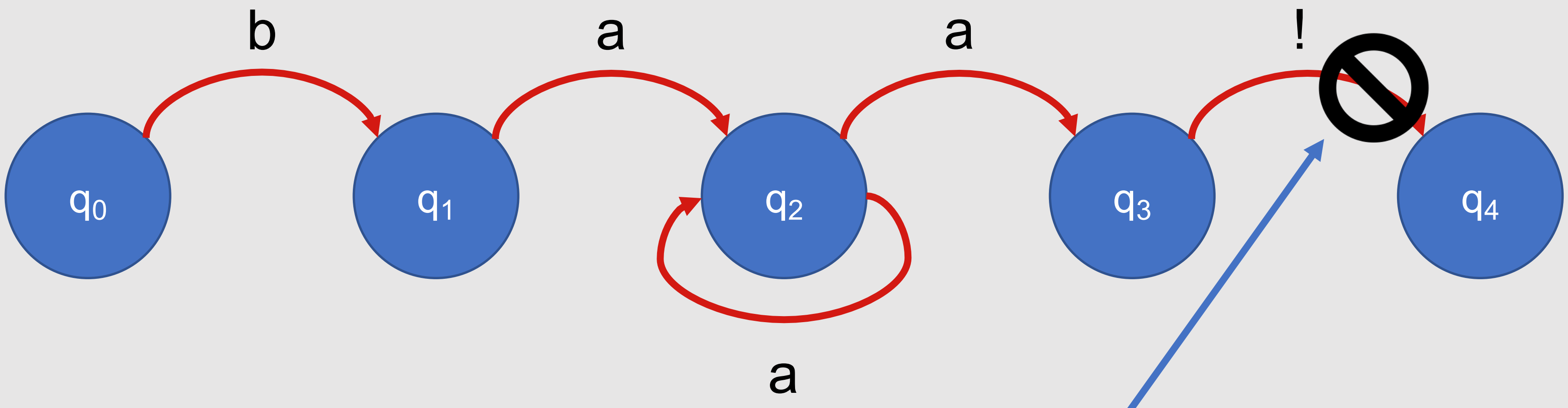
Test Input: baaa!

Example: Non-Deterministic FSA Search



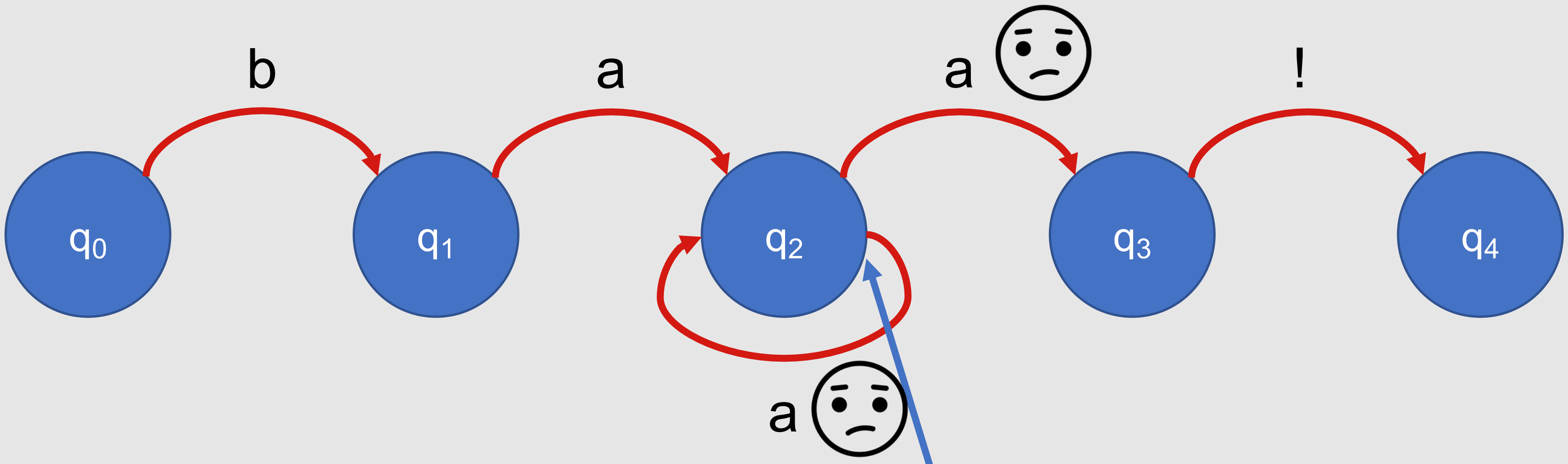
Test Input: baa**a**!

Example: Non-Deterministic FSA Search



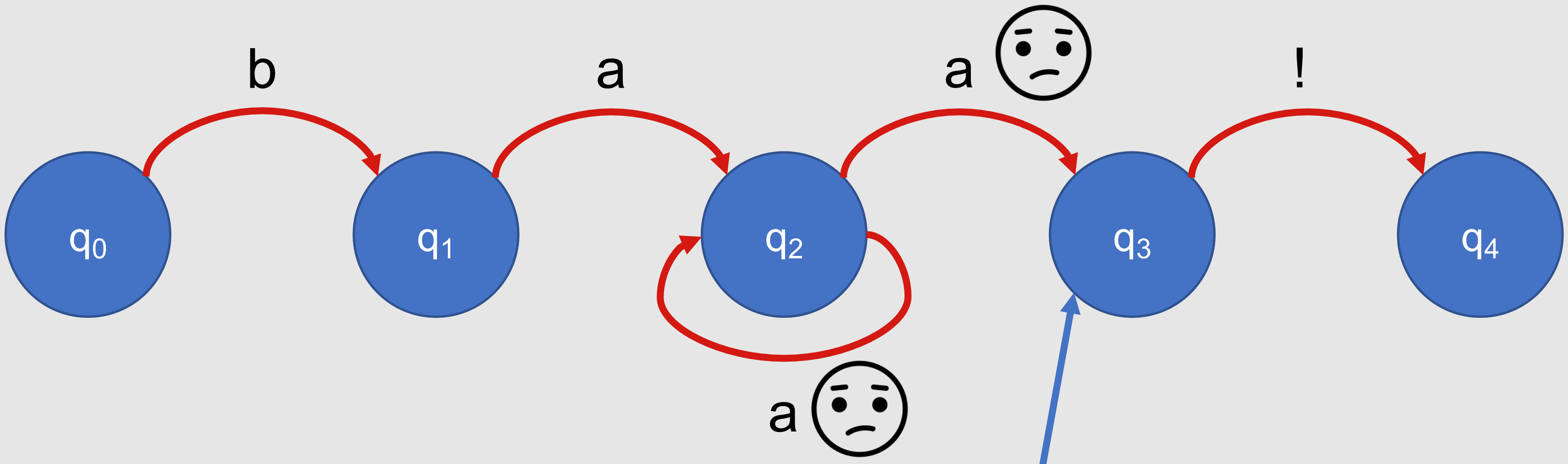
Test Input: baaa!

Example: Non-Deterministic FSA Search



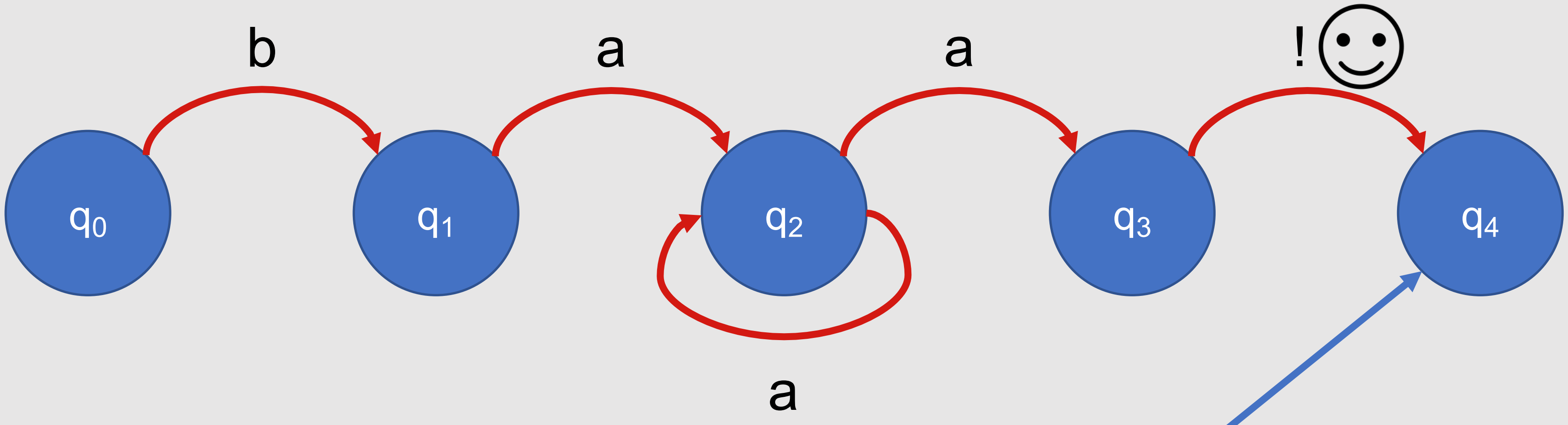
Test Input: baaa!

Example: Non-Deterministic FSA Search



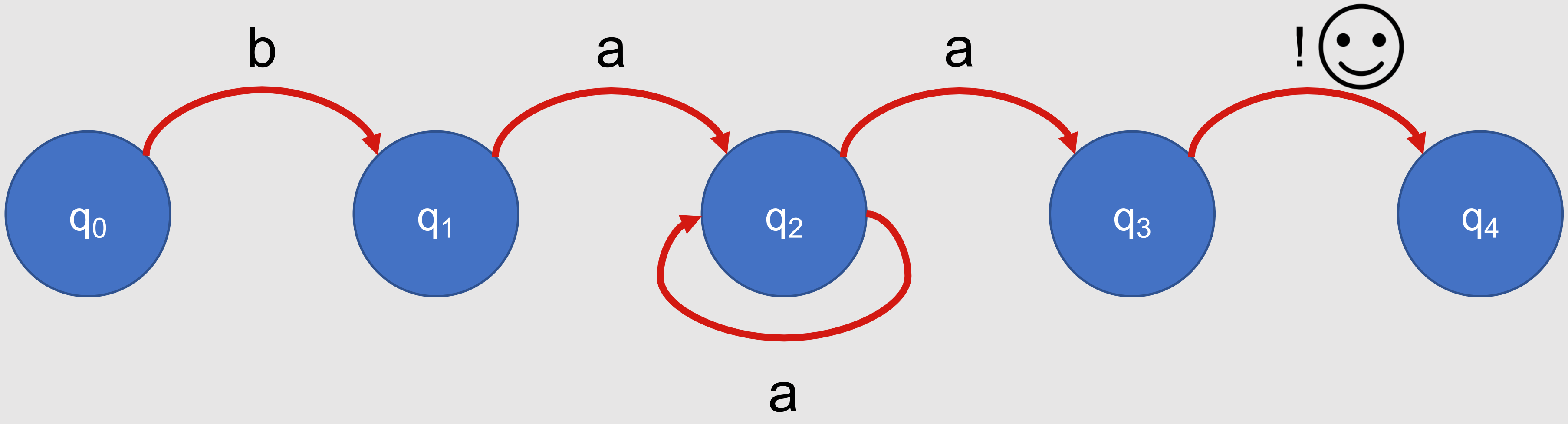
Test Input: baa**a**!

Example: Non-Deterministic FSA Search



Test Input: baaa!

Example: Non-Deterministic FSA Search



Test Input: baaa! 😀

Non-Deterministic FSA Search

- States in the search space are pairings of sequence indices and states in the FSA
- By keeping track of which states have and have not been explored, we can systematically explore all the paths through an FSA given an input

Compositional FSAs

- You can apply set operations to any FSA
  - Union
  - Concatenation
  - Negation
    - For non-deterministic FSAs, first convert to a deterministic FSA
  - Intersection
- To do so, you may need to utilize an ϵ transition
  - ϵ transition: Move from one state to another without consuming an item from the input sequence

Summary: Finite State Automata

- FSAs are computational models that describe regular languages
- To determine whether an input item is a member of an FSA's language, you can process it sequentially from the start to (hopefully) the final state
- State transitions in FSAs can be represented using tables
- FSAs can be either deterministic or non-deterministic

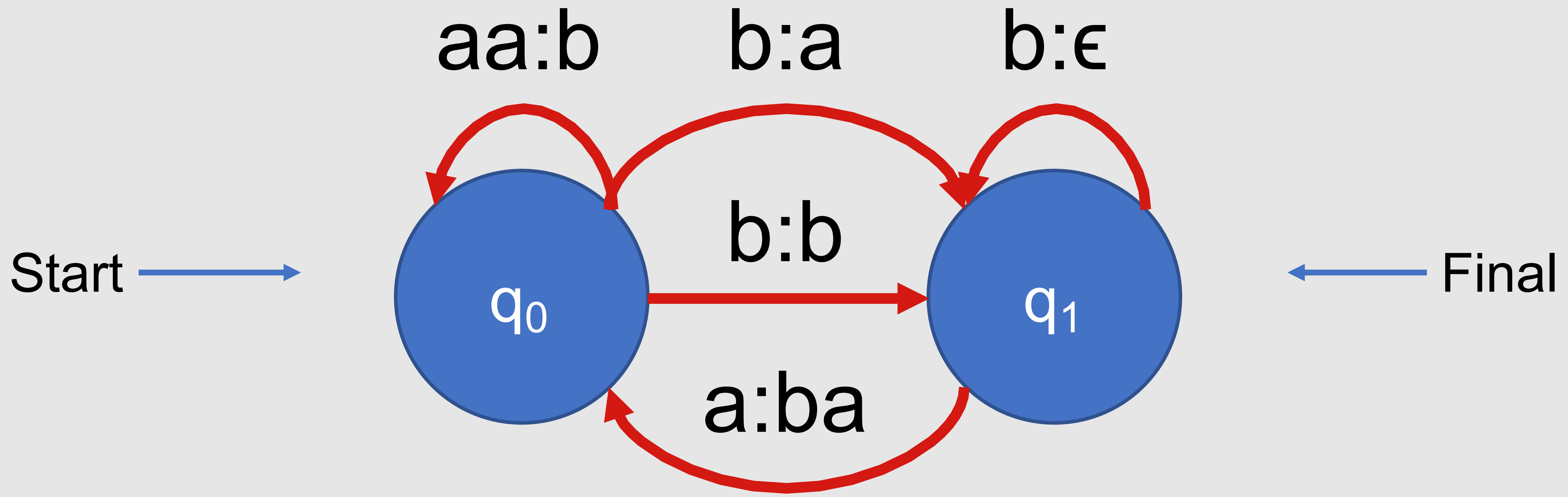
What are finite state transducers?

Finite State Transducer (FST): A type of FSA that describes mappings between two sets of items

This means that FSTs recognize or generate pairs of items

FSAs can be converted to FSTs by labeling each arc with two items (e.g., **a**:**b** for an input of **a** and and an output of **b**)

Example: Simple FST



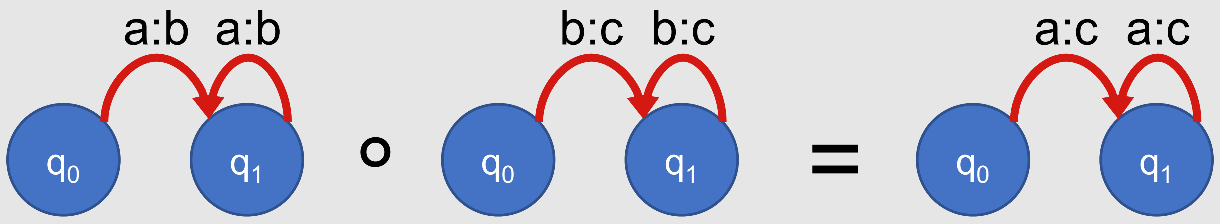
Formal Definition

- A finite state transducer can be specified by enumerating the following properties:
  - The set of states, ***Q***
  - A finite input alphabet, ***Σ***
  - A finite output alphabet, ***Δ***
  - A start state, q_0
  - A set of accept/final states, ***F⊆Q***
  - A transition function or transition matrix between states, **δ(*q*,*i*)**
  - An output function giving the set of possible outputs for each state and input, **σ(*q*,*i*)**
- δ(q,i): Given a state q∈Q and input i∈Σ, **δ(*q*,*i*) returns a new state q'∈Q**.

Formal Properties

Composition: Letting T_1 be an FST from I_1 to O_1 and letting T_2 be an FST from I_2 to O_2, the two FSTs can be composed such that the resulting FST maps directly from I_1 to O_2.

Inversion: Letting T be an FST that maps from I to O, its inversion (T^{-1}) will map from O to I.



Deterministic vs. Non-Deterministic FSTs

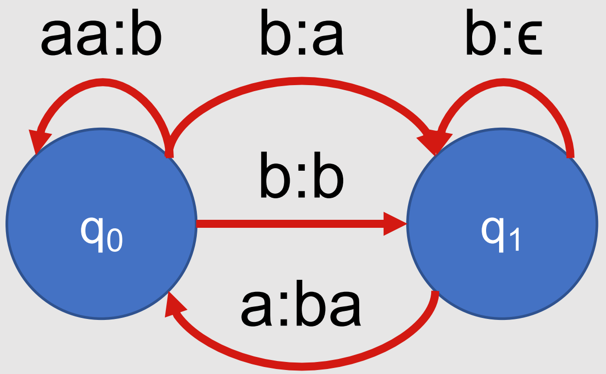
Just like FSAs, **FSTs can be non-deterministic** …one input can be translated to many possible outputs!

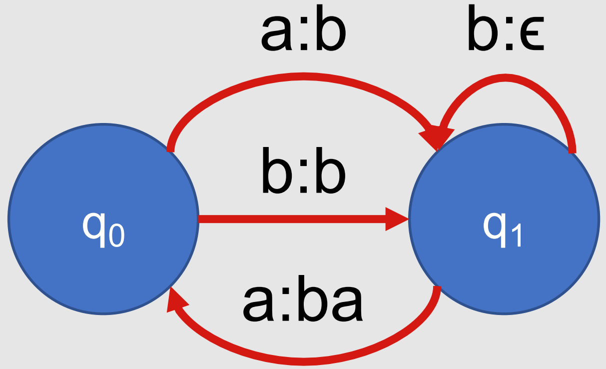
Unlike FSAs, **not all non-deterministic FSTs can be converted to deterministic FSTs**

FSTs with underlying deterministic FSAs (at any state, a given input maps to at most one transition out of the state) are called **sequential transducers**

Examples: Non-Deterministic and Sequential Transducers

Non-Deterministic





Sequential

Remember morphology?

- **Morphemes**:
  - Small meaningful units that make up words
  - **Stems**: The core meaning-bearing units
  - **Affixes**: Bits and pieces that adhere to stems and add additional information
    - -ed
    - -ing
    - -s
- Morphological parsing is a classic use case for FSTs

Morphological Parsing

- The task of recognizing the component morphemes of words (e.g., foxes → fox + es) and building structured representations of those components

Why is morphological parsing necessary?

Morphemes can be **productive**

- Example: -ing attaches to almost every verb, including brand new words
  - "Why are you Instagramming that?"

Some languages are very **morphologically complex**

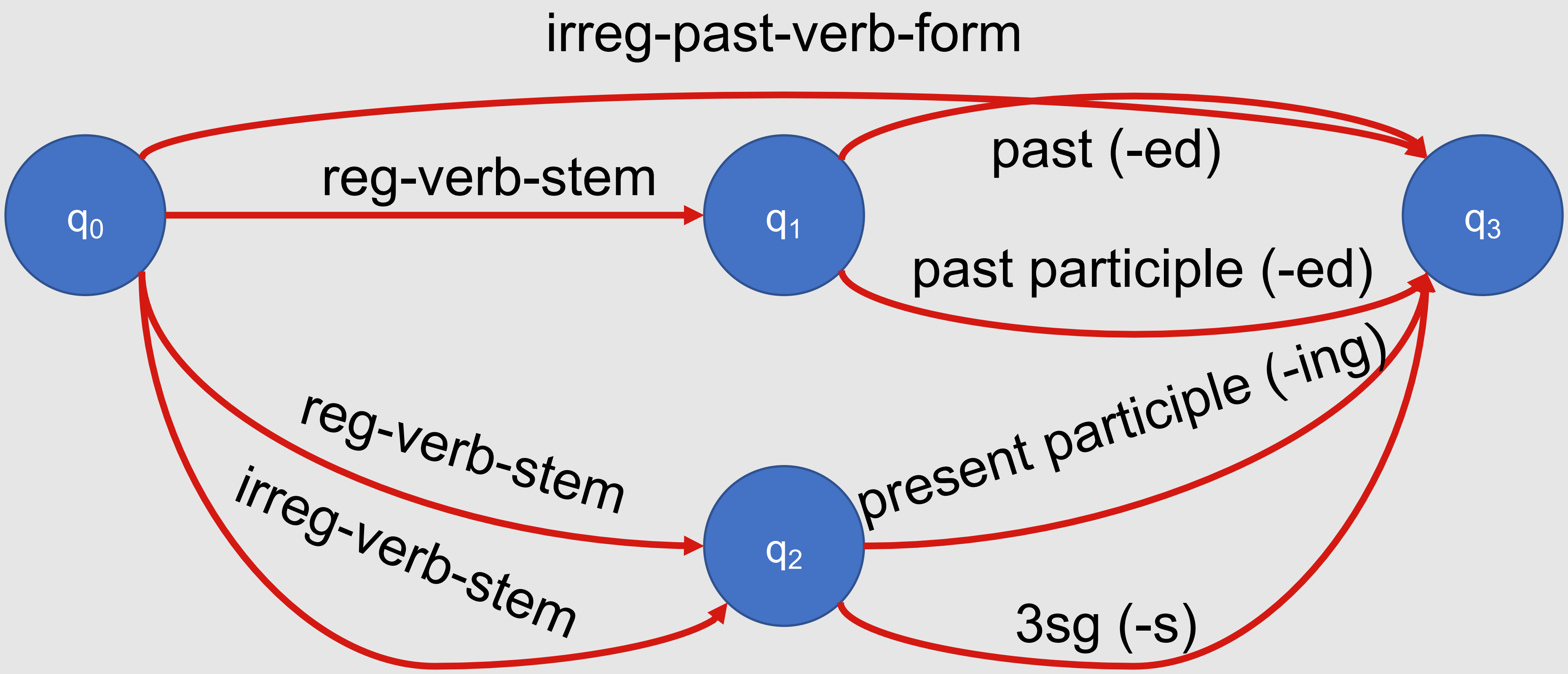
- Uygarlastiramadiklarimizdanmissinizcasina
  - Uygar 'civilized' + las 'become'
  - + tir 'cause' + ama 'not able'
  - + dik 'past' + lar 'plural'
  - + imiz 'p1pl' + dan 'abl'
  - + mis 'past' + siniz '2pl' + casina 'as if'

Finite State Morphological Parsing

Goal: Take input surface realizations and produce morphological parses as output

Surface Text	Morphological Parse
cats	cat +N +PL
cat	cat +N +SG
cities	city +N +PL
geese	goose +N +PL
goose	goose +N +SG
merging	merge +V +PresPart
caught	catch +V +Past

Example Morphological Lexicon

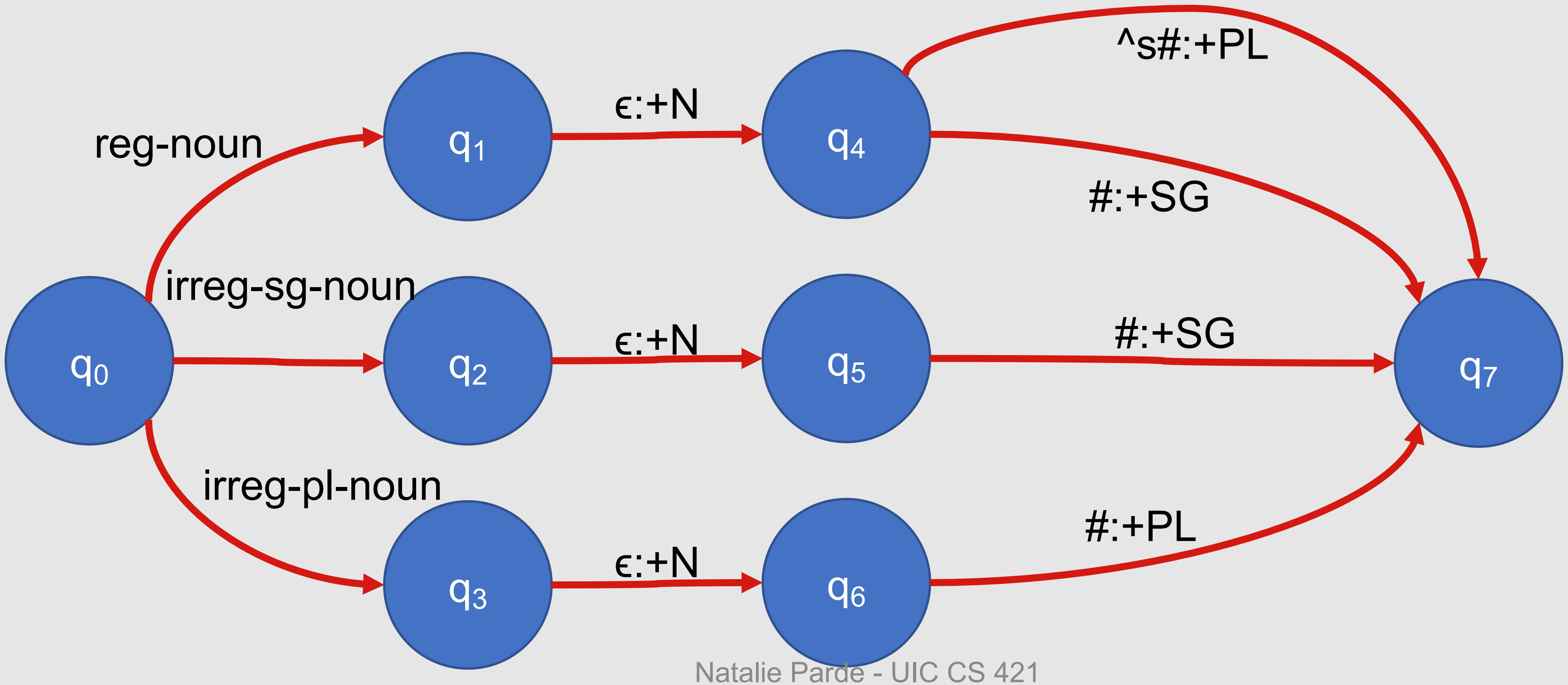


Finite State Morphological Parsing

- Two sets of items:
  - Surface form (input text)
  - Lexical form (morphological parse)

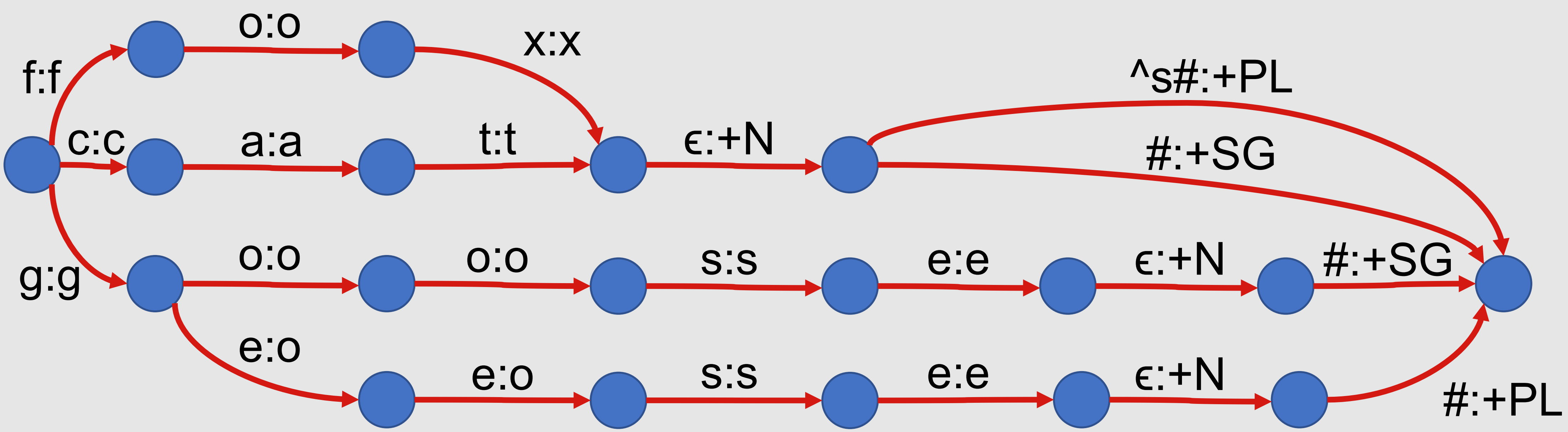
Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



Summary: Finite State Transducers

- FSTs are FSAs that describe mappings between two sets
- Although all non-deterministic FSAs can be converted to deterministic versions, all non-deterministic FSTs cannot
- FSTs with underlying deterministic FSAs are called sequential transducers
- FSTs are particularly useful for morphological parsing