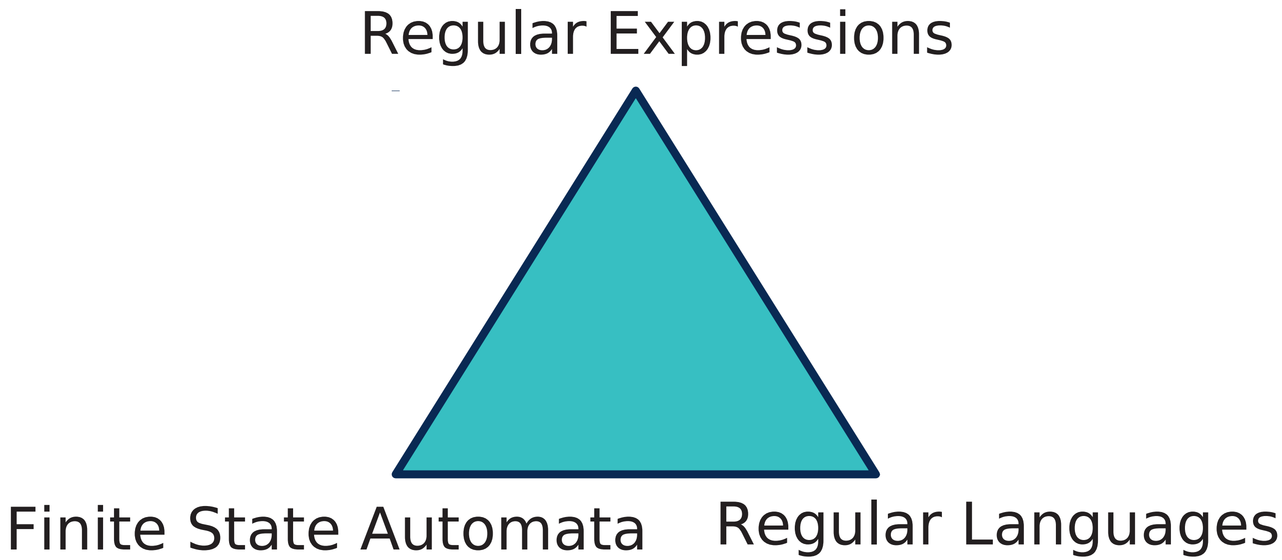
Agenda

- Finite-State Automata [FSA]
  - Deterministic FSA
- Formal languages
- Non-deterministic FSA
- Regular languages and FSA

Finite-State Automata

- A regular expression is one way of describing a finite-state automaton(FSA)
- Any regular expression can be implemented by as a finite-state automaton
- Symmetrically any FSA can be described with a regular expression
- A regular expression is one way of characterizing a particular kind of formal language called a regular language
- Both regular expressions and finite-state automata can be used to describe *regular languages*

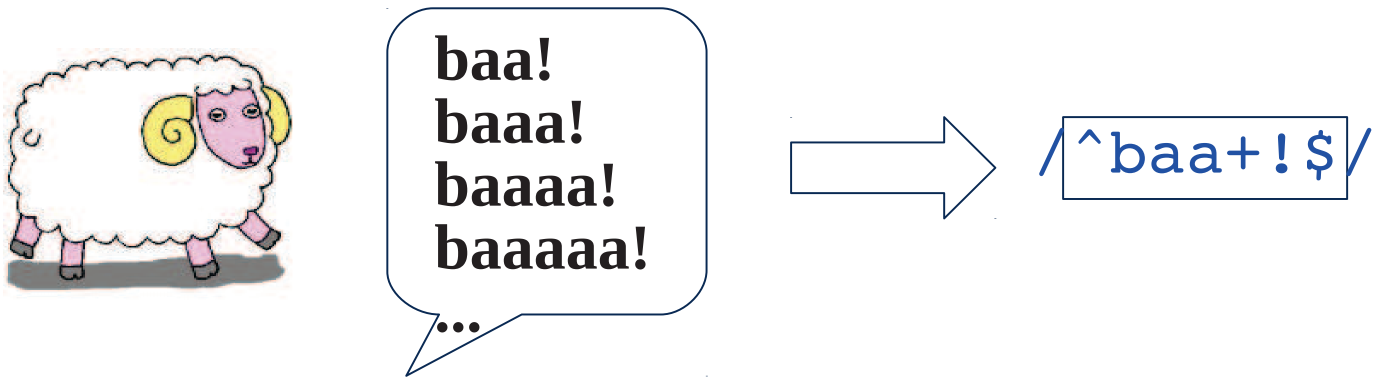
Finite-State Automata

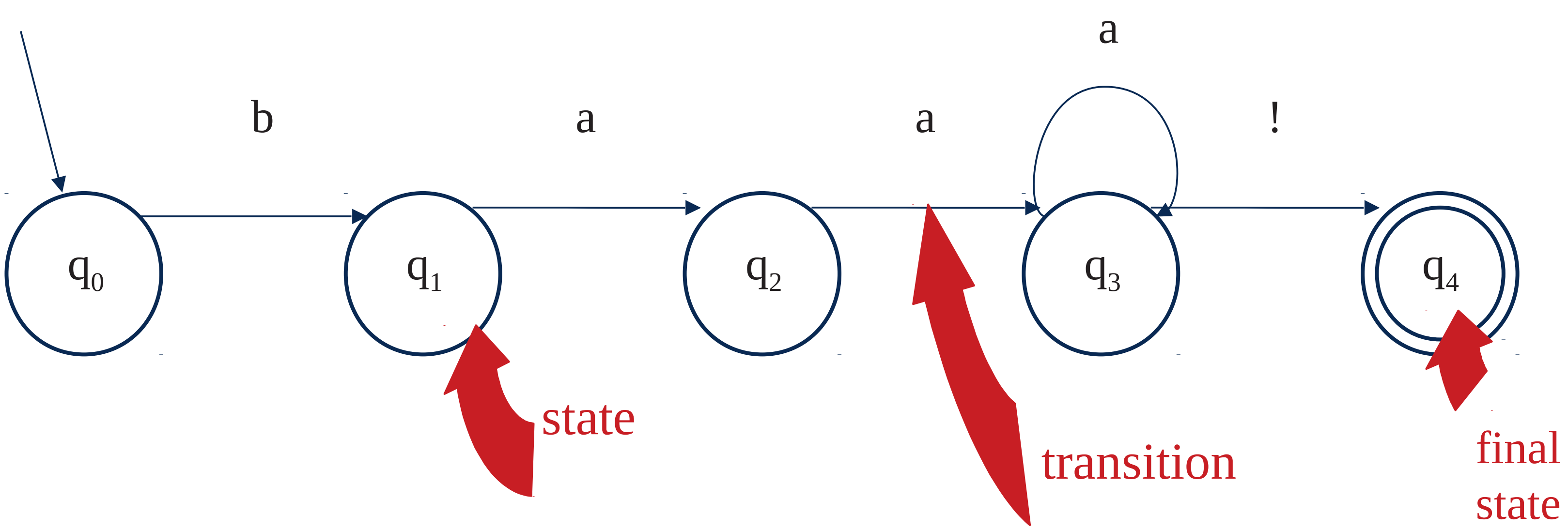


Finite-State Automata

- Sheep language: any string from the following (infinite) set: (baa!,baaa!....)
- The automaton (i.e., *machine, also called finite automaton, finite-state automaton, or FSA*) recognizes a set of strings
- Automaton is represented as a *directed graph*: a finite set of nodes together with a set of directed links between pairs of nodes called arcs
- Represent nodes with circles and arcs as with arrows
- States are represented by nodes in the graph
- State 0 is the start state, represented by the incoming arrow
- Final state or accepting state is represented by the double circle

Finite-State Automata





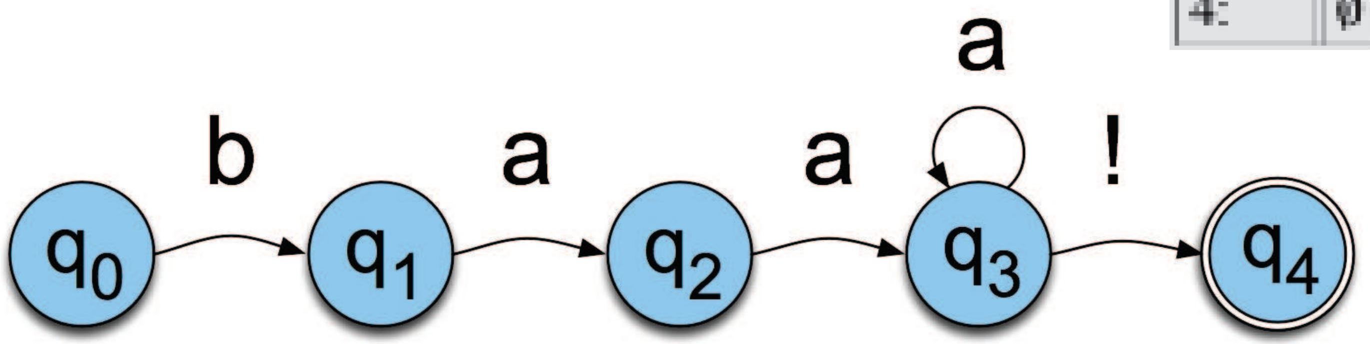
Finite-State Automata

- **FSA** is a 5-tuple consisting of
- **Q**: a finite set of N states $q_0,q_1,....q_N$
- **Σ :** a finite input alphabet of symbols
- **q0**: the start state
- **F**: a set of final states
- **δ(q,i)**: a transition function
- Given a state q and an input symbol i, δ(q,i) returns a new state q'

Sheep FSA

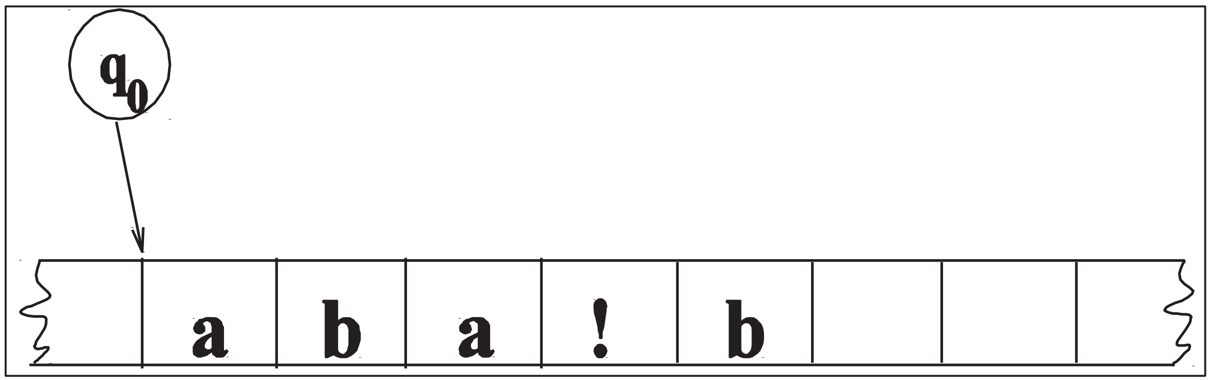
- We can say the following things about this machine
  - It has 5 states $Q:\{q_0,q_1,q_2,q_3,q_4\}$
  - b, a, and ! are in its alphabet Σ :(a,b,!)
  - q_0 is the start state
  - q_4 is an accept state
  - It has 5 transitions **$\delta(q,i)$**

	Input		
State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4:	∅	∅	∅



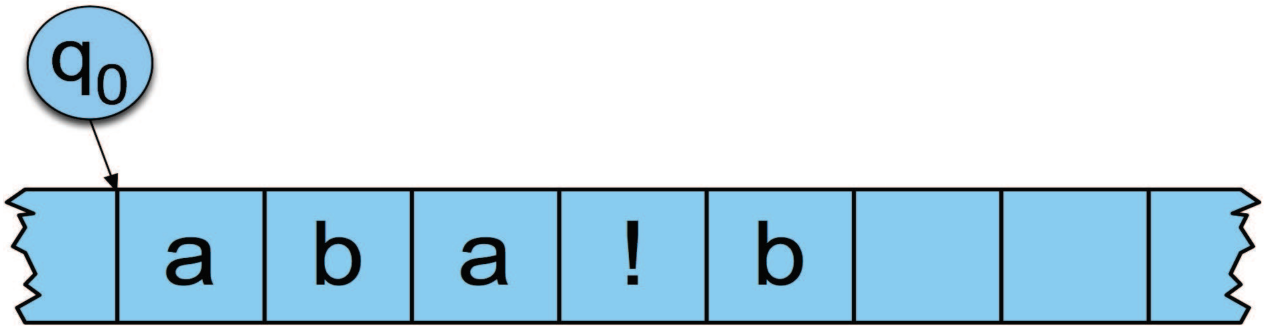
FSA: Recognition

- Recognition is the process of determining if a string should be accepted by a machine
- Or… it's the process of determining if a regular expression matches a string
- Traditionally, (Turing's idea) this process is depicted with a tape



FSA: Recognition

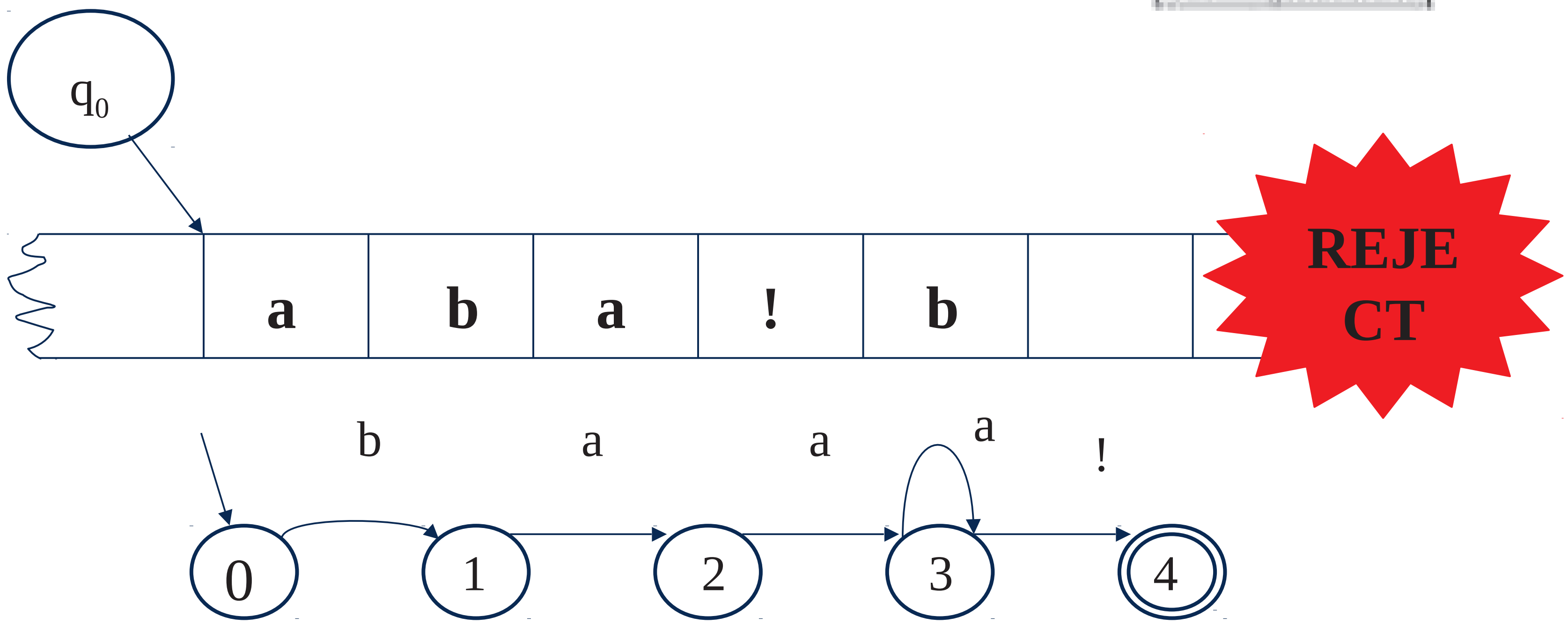
- Begin by start state
- Examine the current input
- Consult the table
- Go to a new state and update the tape pointer
- Until you run out of tape



FSA: Tracing a Rejection / fail state

- When it runs out of input or stuck with some non-final state
- When there is no legal transition for a given combination of state and input – fail state

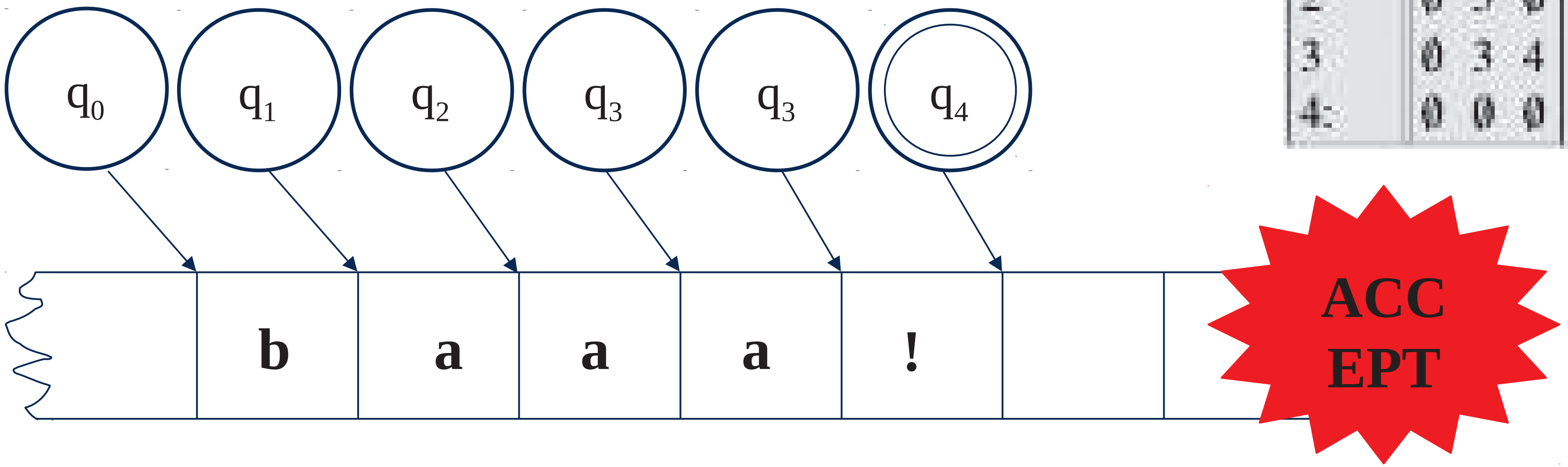
	Input		
State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4:	∅	∅	∅

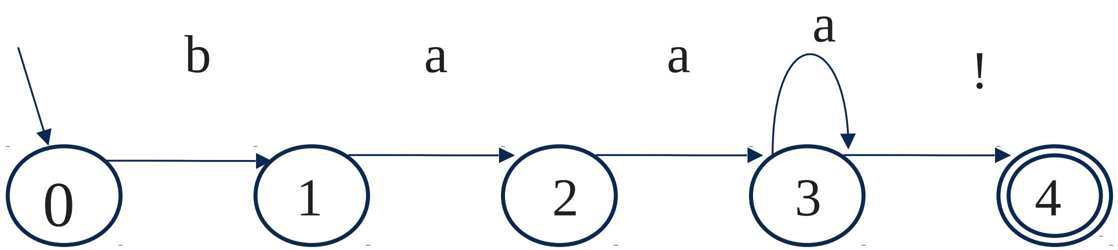


FSA: Tracing an Accept

- If reached end of input and in final state, then ACCEPT

	Input		
State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4:	∅	∅	∅





Deterministic Finite-State Automata

- The algorithm is called D-RECOGNIZE for deterministic recognizer
- A *deterministic algorithm* is one that has no choice points: the algorithm always knows what to do for any input
- D-RECOGNIZE takes as input a tape and an automaton and returns accept or reject
- Set the variable index to the beginning of the tape, and current-state to the machine's initial state
- First checks whether it has reached the end of its input, if so either accept or reject the input
- If input is left on the tape, look at the transition table to decide which state to move

Finite-State Automata

```
function D-RECOGNIZE(tape,machine) returns accept or reject

index <-- Beginning of tape
current-state <-- Initial state of machine
loop
if End of input has been reached then
    if current-state is an accept state then
        return accept
    else
        return reject
elseif transition-table[current-state,tape[index]] is empty then
    return reject
else
    current-state <-- transition-table[current-state,tape[index]]
    index <-- index + 1
end
```

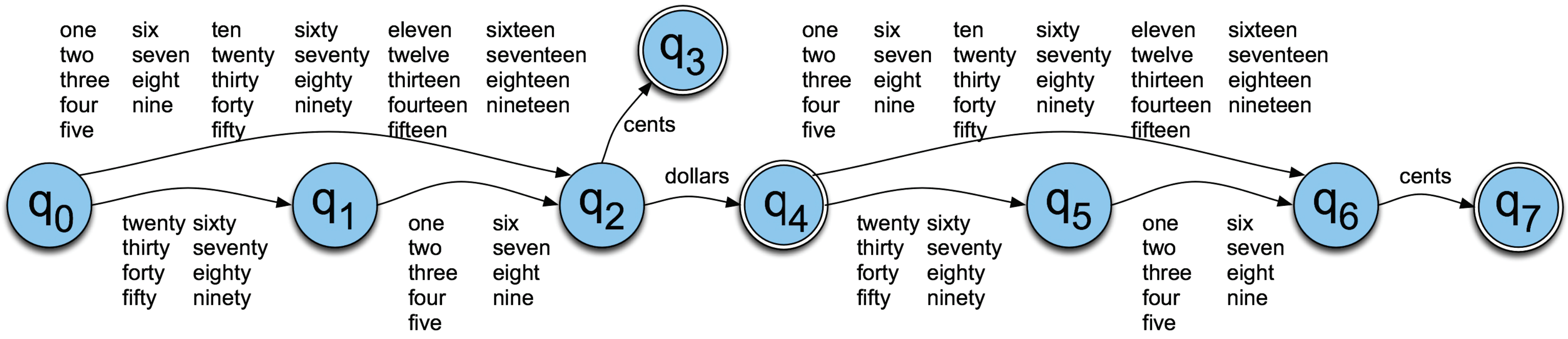
Formal Language

- A model that can both generate and recognize all and only the strings of a formal language acts as a definition of the formal language
- A formal language is a set of strings, each string composed of symbols from a finite symbol set.
- Given a model *m* (such as particular FSA), we can use L(*m*) to mean "the formal language characterized by *m*"
- So the formal language defined by sheeptalk automaton :
  - L(*m*) = {baa!,baaa!,baaaa!,baaaaa!.....}
- Often use formal language to model part of a natural language, such as parts of the phonology, morphology, or syntax

Formal Language

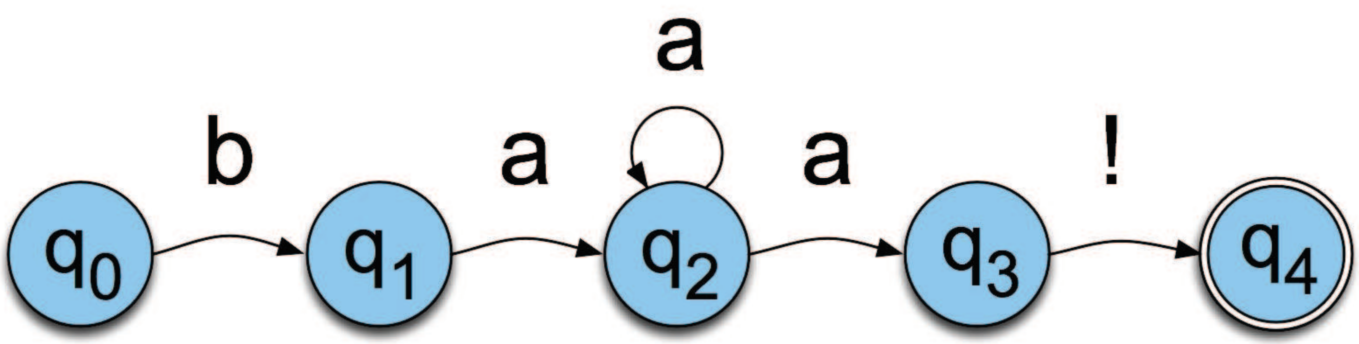
- The term *Generative* is based on the view that you can run the machine as a generator to get strings from the language
- FSAs can be viewed from two perspectives:
  - Acceptors that can tell you if a string is in the language
  - Generators to produce *all and only* the strings in the language

Dollars and Cents



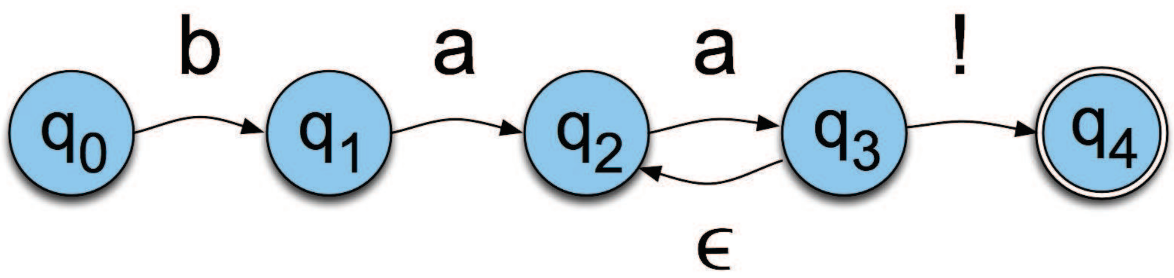
Non-deterministic FSA

- In state 2, if input is a, whether to remain in state 2 or go on to state 3
- Automata with decision points are called non-deterministic FSAs (or NFSAs)



Non-deterministic FSA

- Yet another technique
  - Epsilon transitions
  - Key point: these transitions do not examine the input
  - Move to state 2 without looking at the input or advance the input pointer – Non-determinism at state 3

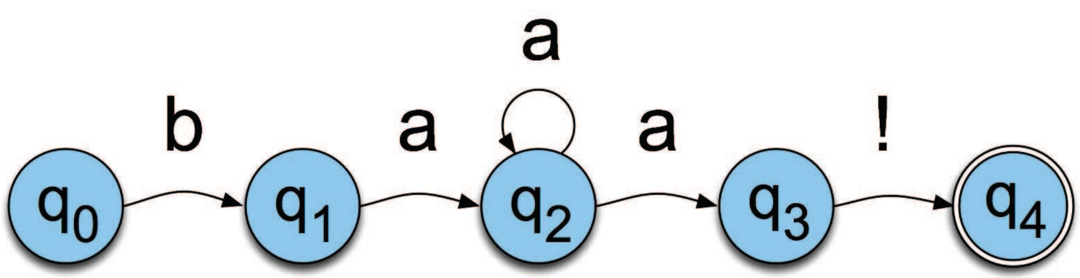


Non-deterministic FSA

- Since there is more than one choice at some point, there is a chance of wrong choice
- This problem of choice in non-deterministic can be solved by:
  - Backup: put a marker to mark where we are in the input, and what state the automaton was in. If wrong choice, back up and try another choice
  - Look-ahead: look ahead on the input to decide which path to take
  - Parallelism: look at every alternative path in parallel

Non-deterministic FSA

- Non-deterministic recognizer need to remember two things for each choice:
  - The state or node of machine and the corresponding position on the tape – search states
  - Search states are created for each of choices



	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Non-deterministic FSA

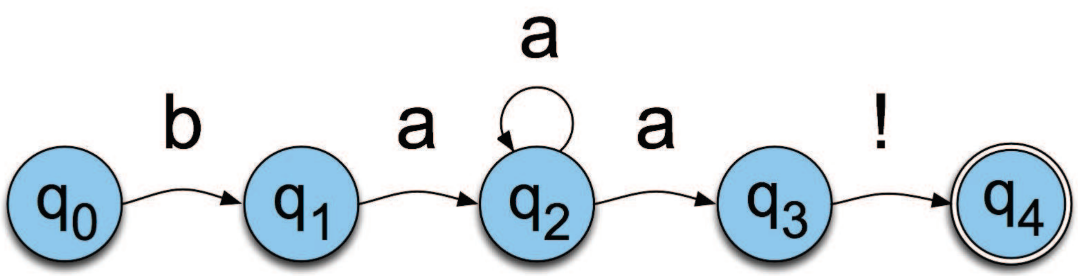
```
function ND-RECOGNIZE(tape,machine) returns accept or reject

agenda <-- {(Initial state of machine, beginning of tape) }
current-search-state <-- NEXT(agenda)
loop
if ACCEPT-STATE?(current-search-state) returns true then
        return accept
else
        agenda <-- agenda U GENERATE-NEW-STATES(current-search-state)

if agenda is empty then
        return reject
else
        current-search-state <-- NEXT(agenda)

end
```

Non-deterministic FSA

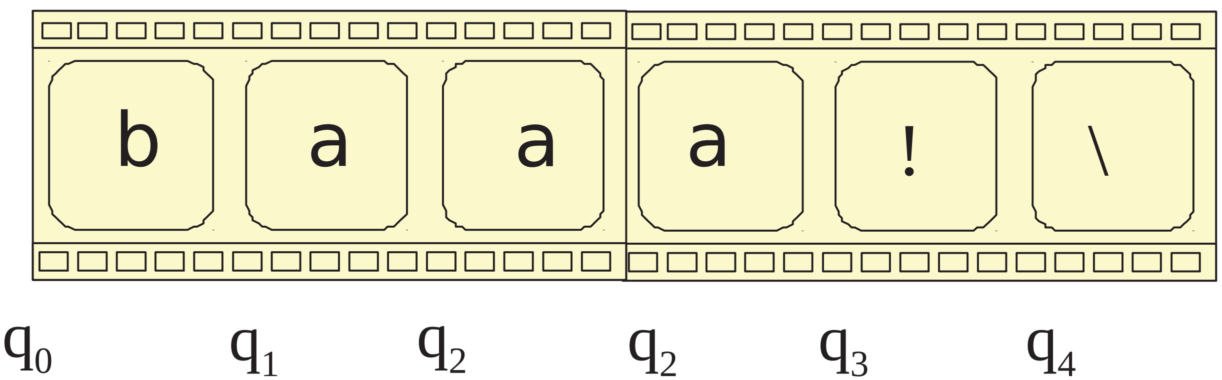
```
function GENERATE-NEW-STATES(current-state) returns a set of search-states

current-node <-- the node the current search-state is in
index <-- the point on the tape the current search-state is looking at
returns a list of search-states from transition tabel as follows:
(transition-table[current-node, ϵ], index) U
(transition-table[current-node,tape[index]], index + 1)
--------------------------------------------------------------------------------
function ACCEPT-STATE?(search-state) returns true or false

current-node <-- the node the search-state is in
index <-- the point on the tape search-state is looking at
if index is at the end of the tape and current-node is an accept state of machine then
      return true
else
      return false
```

Non-deterministic FSA

- Example



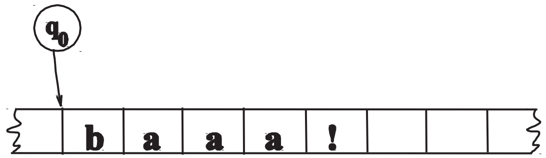
	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

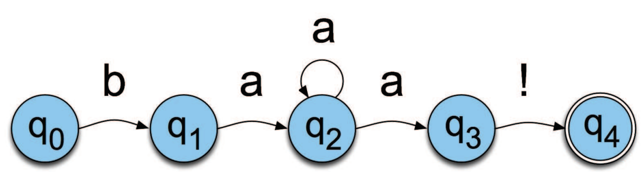


Non-deterministic FSA

- Example

1





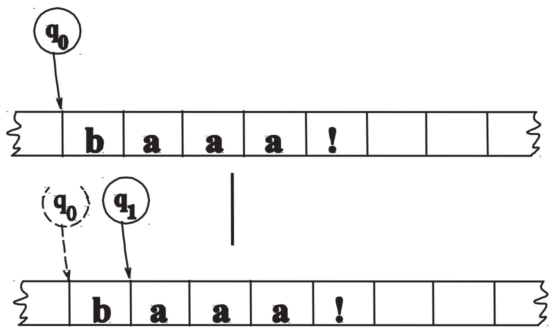
	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

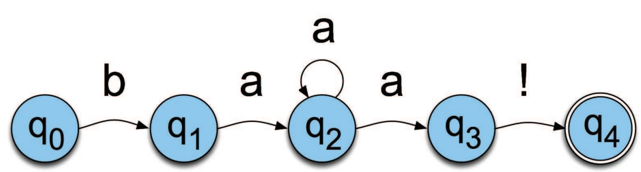
Non-deterministic FSA

- Example

1

2

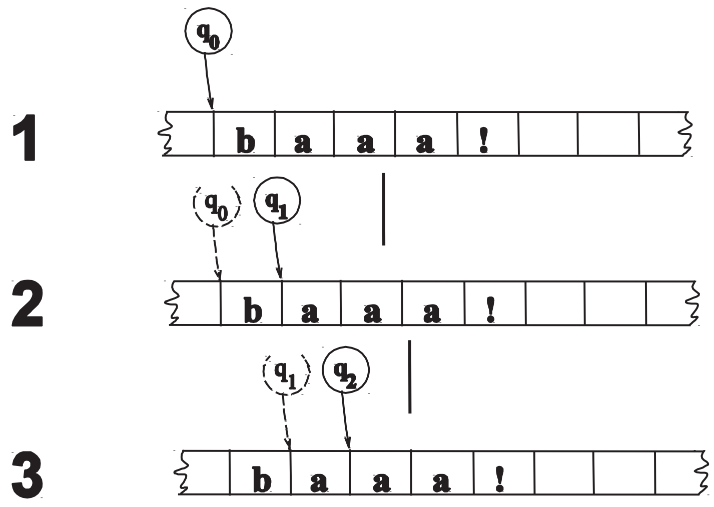


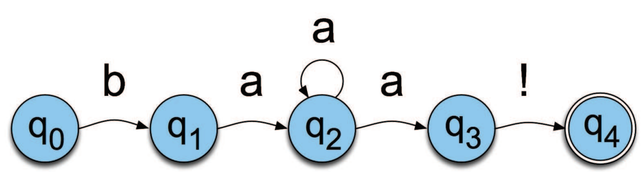


	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Non-deterministic FSA

- Example

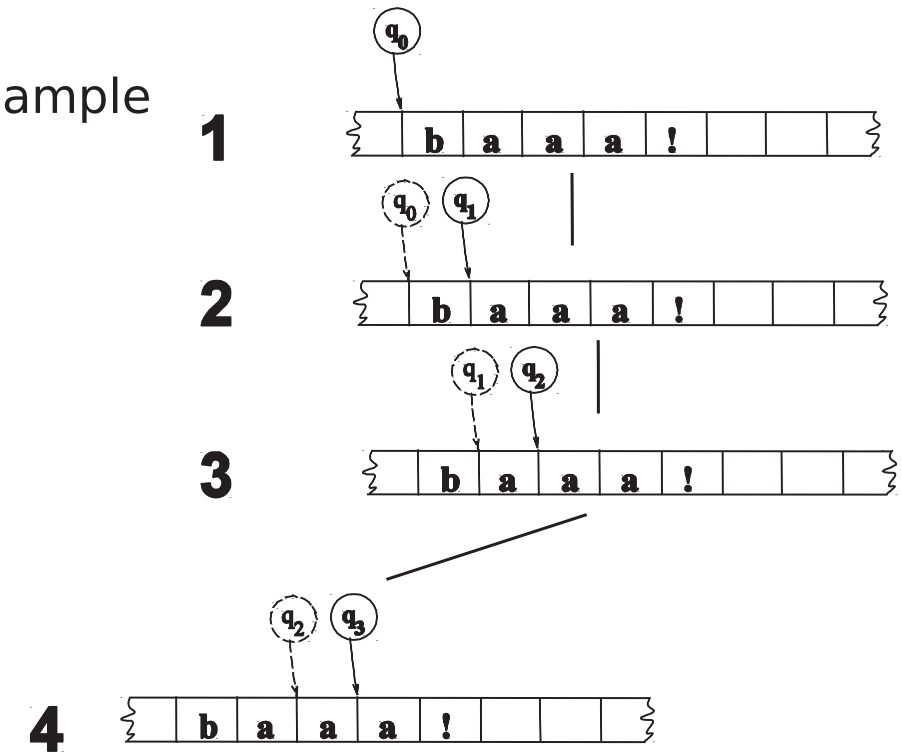


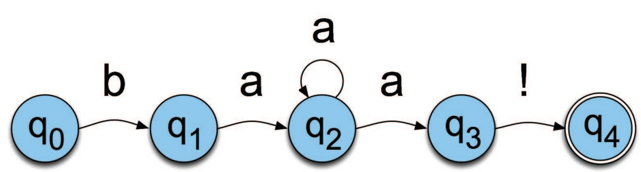


	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Non-deterministic FSA

- Example

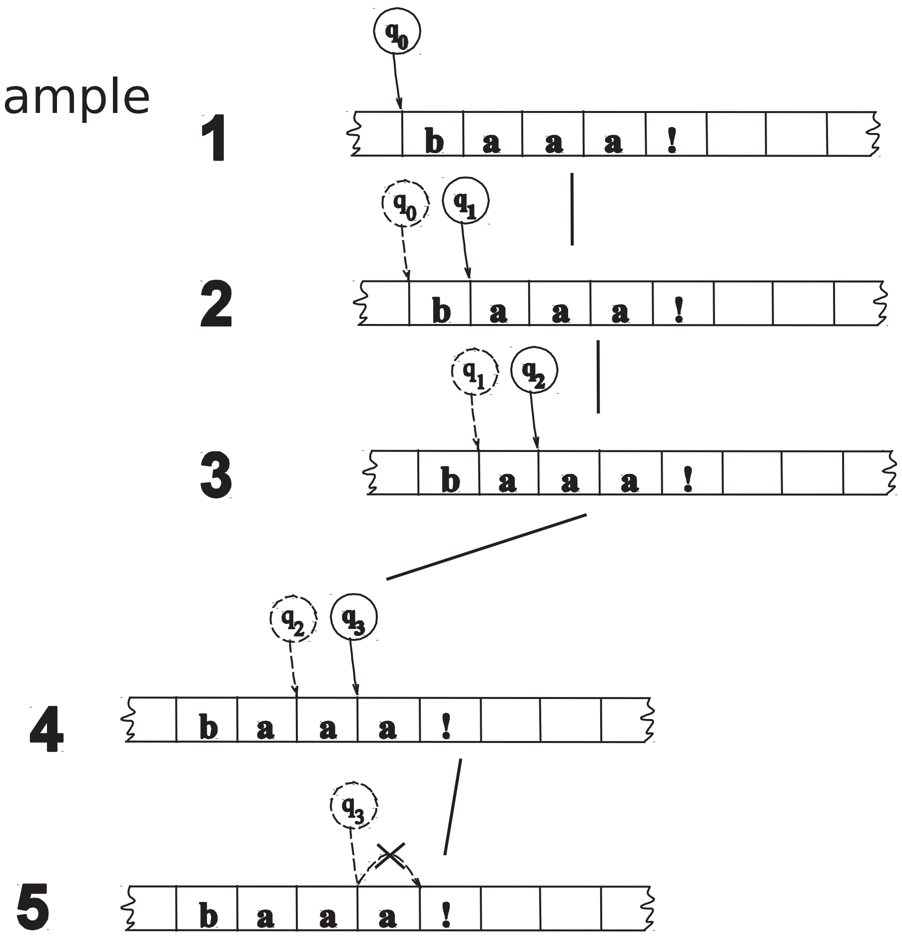


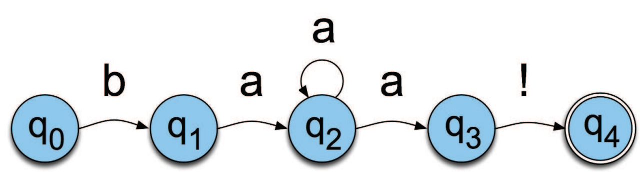


	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Non-deterministic FSA

- Example

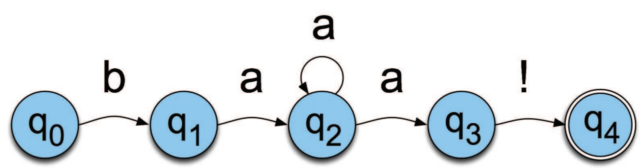


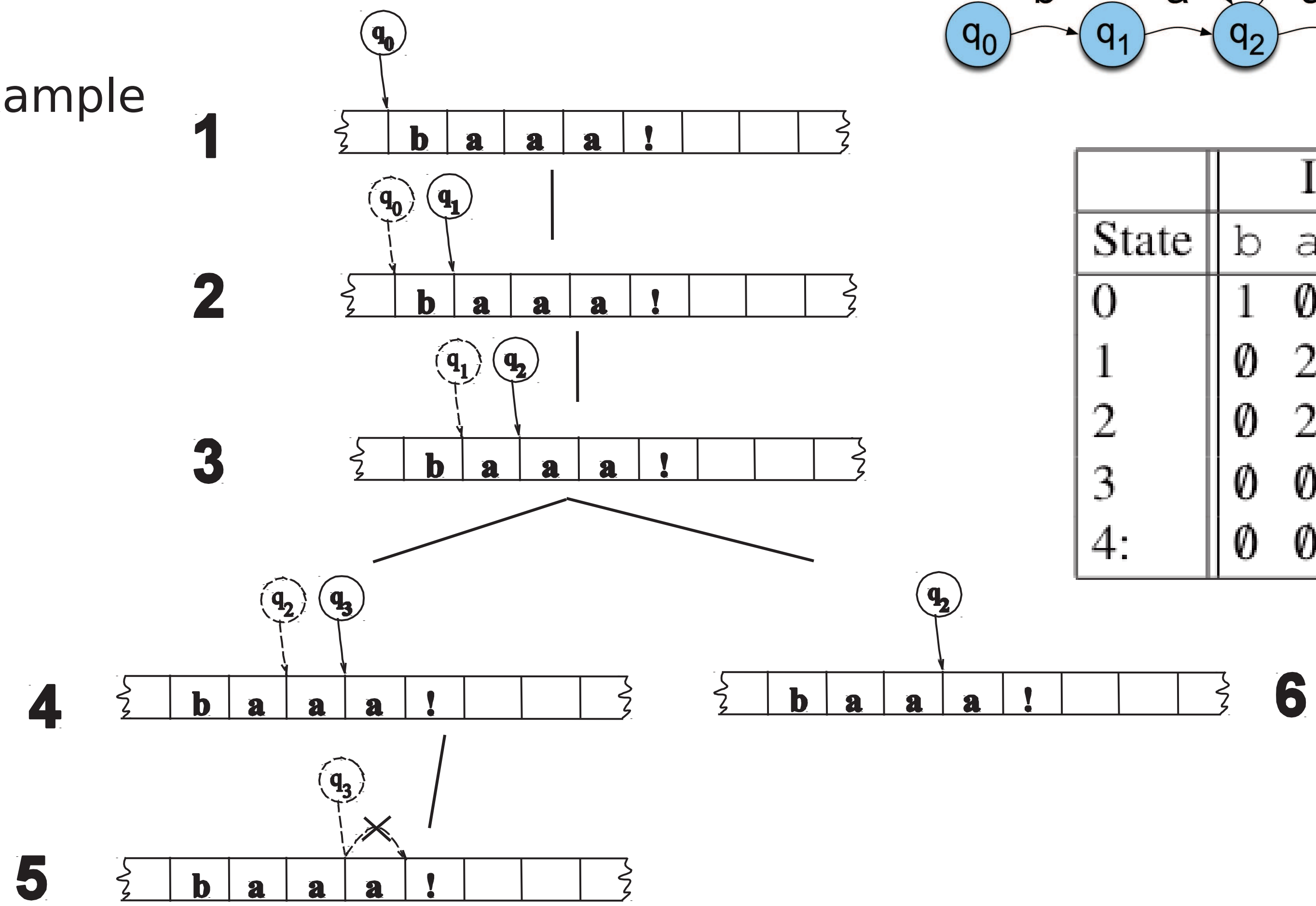


	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Non-deterministic FSA

- Example

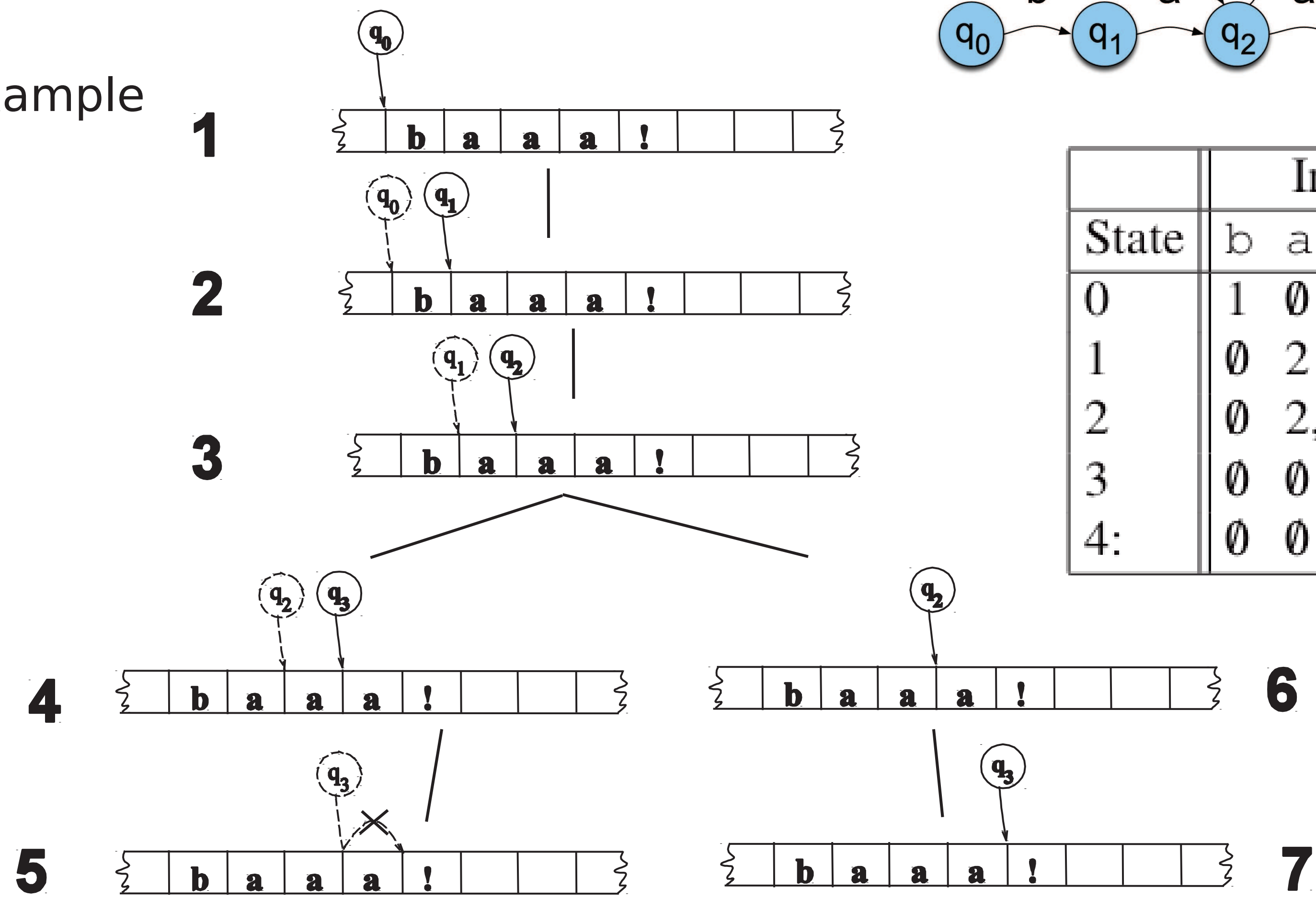


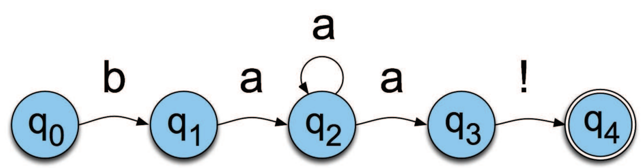


State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Non-deterministic FSA

- Example

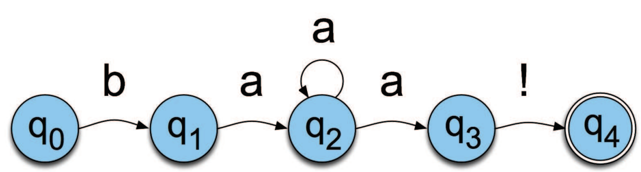




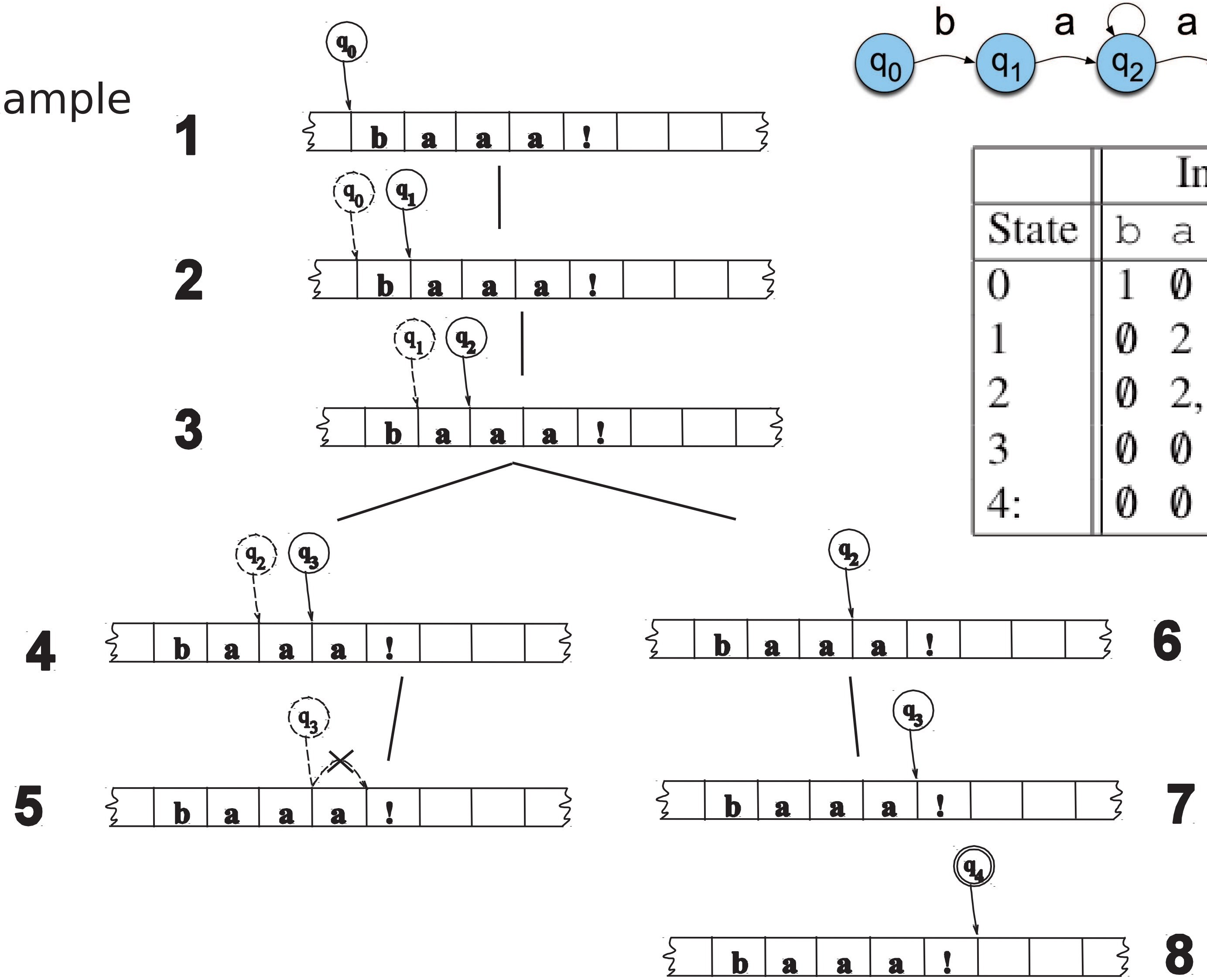
	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Non-deterministic FSA

- Example



	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

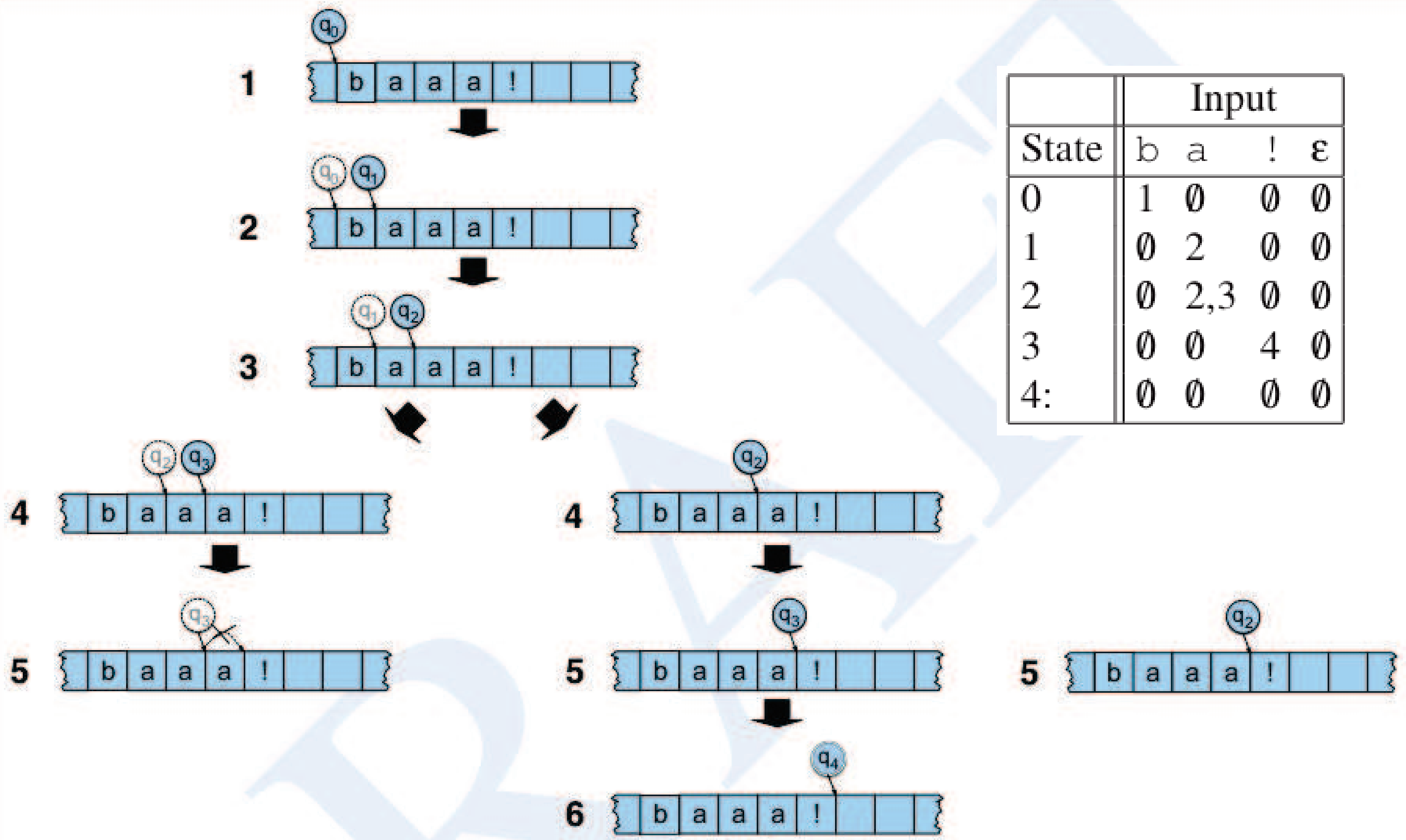


State-space Search Algorithm

- We model problem-solving as a search for a solution through a space of possible solutions.
- The space consists of states.
- States in the search space are pairings of tape positions and states in the machine.
- By keeping track of as yet unexplored states, a recognizer can systematically explore all the paths through the machine given an input.
- How the order in which the states in the space are considered?

State-space Search Algorithm

- The order in which a NFSA chooses the next state to explore on the agenda defines its search strategy
- Ordering of states leads to:
  - depth-first search or LIFO strategy
    - Consider the newly created state as the NEXT state
  - breadth-first search of FIFO strategy
    - Consider states in the order in which they are created



	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Figure 2.23 A breadth-first trace of FSA #1 on some sheeptalk.

Deterministic Vs Non-deterministic FSA

- For any NFSA, there is an equivalent DFSA.
- Non-deterministic machines can be converted to deterministic ones with a fairly simple construction.
- That means that they have the same power:
  - non-deterministic machines are not more powerful than deterministic ones in terms of the languages they can accept

Regular Languages and FSA

Regular Languages

- The alphabet Σ - the set of all symbols in the language and an empty string ε.
- The class of regular languages over Σ is then formally defined as:

1. $\emptyset$ is a regular language
2. $\forall a \in \Sigma \cup \varepsilon$, $\{a\}$ is a regular language
3. If L_1 and L_2 are regular languages, then so are:
   (a) $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$, the **concatenation** of L_1 and L_2
   (b) $L_1 \cup L_2$, the **union** or **disjunction** of L_1 and L_2
   (c) L_1^*, the **Kleene closure** of L_1

- Languages which meet the above properties are regular languages.
- Since regular languages are characterized by regular expressions, all regex operators can be implemented.

FSA

- Regular expressions are equivalent to finite-state automata.
- Proof contain two parts:
  - An automaton can be built for each regular language
  - Conversely a regular language can be built for each automaton
- Consider the regular expression ε , ∅ or any single symbol a

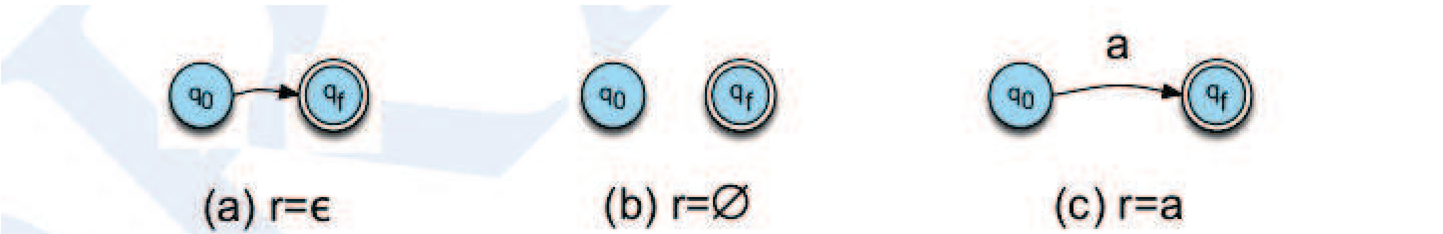


Figure 2.24 Automata for the base case (no operators) for the induction showing that any regular expression can be turned into an equivalent automaton.

FSA

- Primitive operations of regular expression can be imitated by an automaton:
- Concatenation: connect all the final states of FSA_1 to the initial state of FSA_2 by an ε-transition.
- Closure:
  - Create a new final and initial state
  - Connect original final states back to the initial states by ε-transitions
  - Put direct links between the new initial and final states by ε-transitions

FSA

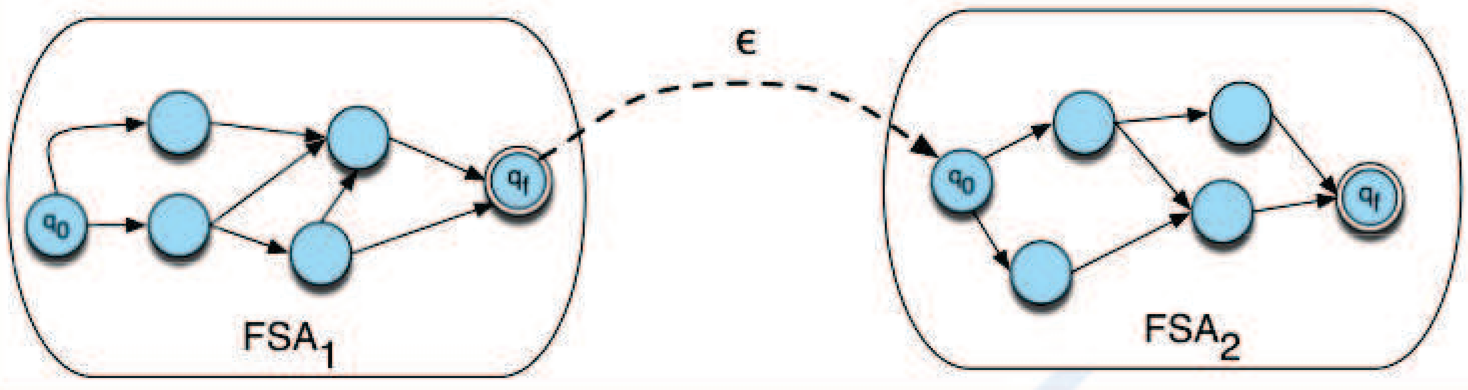


Figure 2.25 The concatenation of two FSAs.

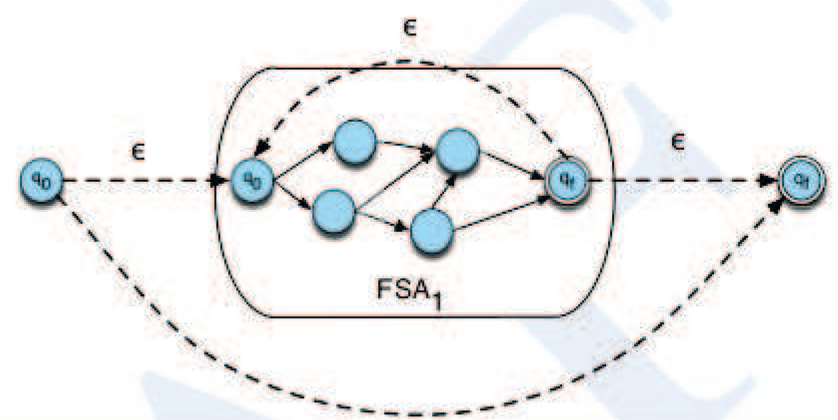


Figure 2.26 The closure (Kleene *) of an FSA.

FSA

- Union: Add a single new initial state and add new ε-transitions from it to former initial states of the two machines to be joined.

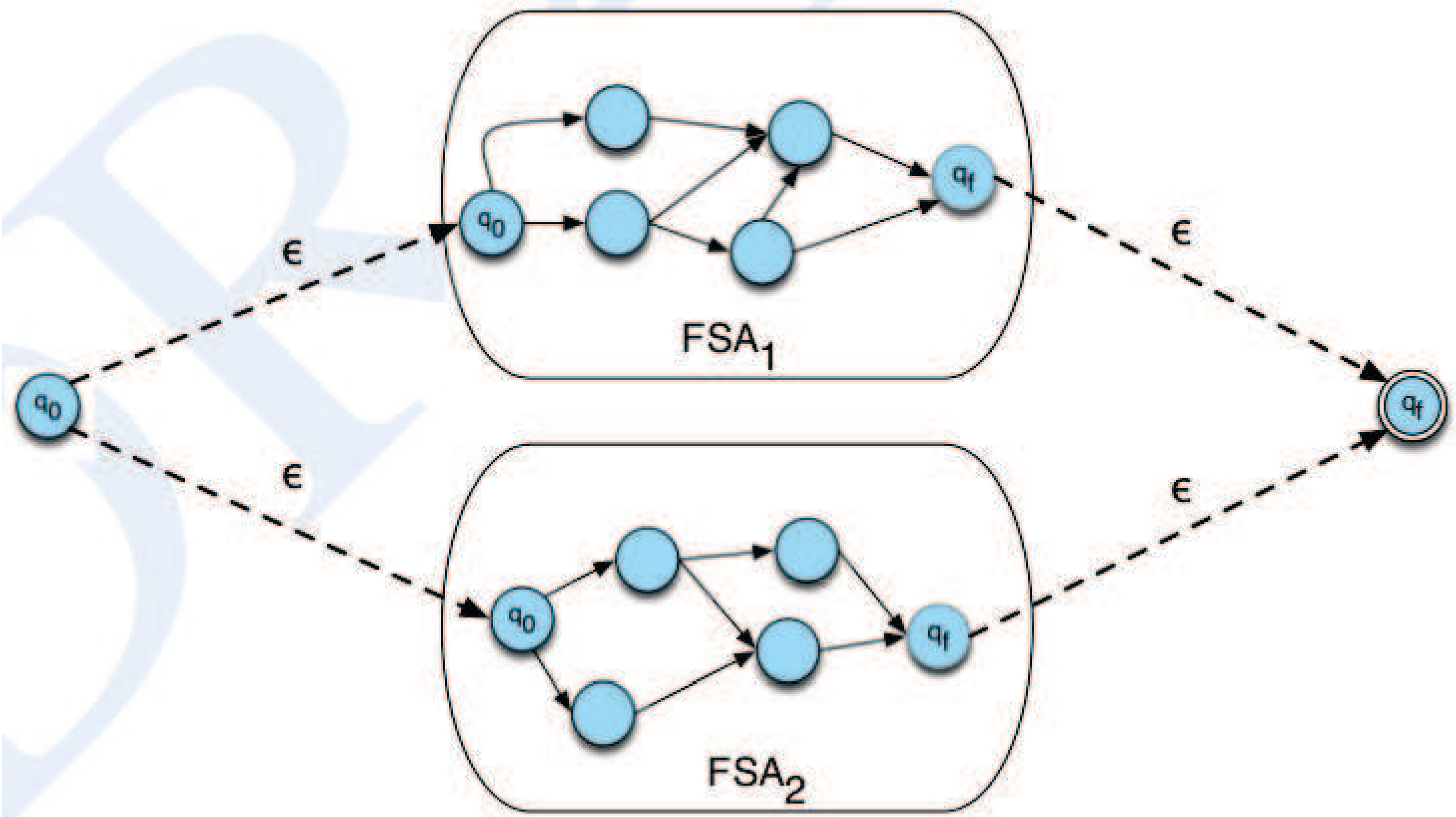


Figure 2.27 The union (|) of two FSAs.

THANK YOU