جامعة المنوفية
MENOUFIA UNIVERSITY
FACULTY OF COMPUTERS AND INFORMATION

Fourth Year (Second Semester)

CS Dept., (CS 436)

Natural Language Processing NLP

Lecture Three

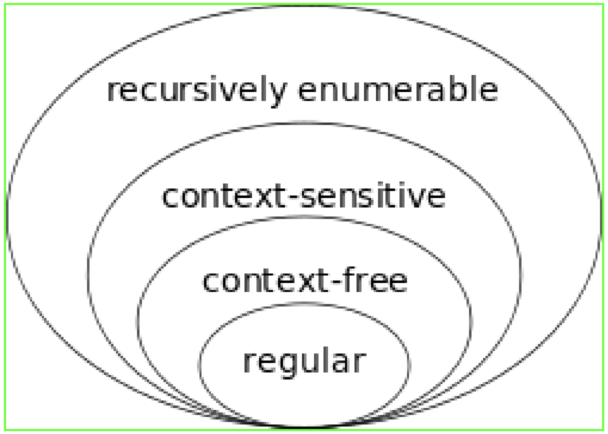
Dr. Hamdy M. Mousa

Automata and Languages

- An **automaton is an abstract model of a computer which** reads an **input string, and changes its internal state** depending on the **current input** symbol.
  - It can either **accept or reject the input string.**
- **Every automaton defines a language** (the set of strings it accepts).
- Different automata define different classes of language:
  - Finite-state automata define regular languages
  - Pushdown automata define context-free languages
  - Turing machines define recursively enumerable languages

Context-free languages

- A language is said to be **context-free** if it is generated by a context-free grammar.
  - The right hand side of the production rules in context free grammars are:
    - unrestricted and
    - can be any combination of terminals and non terminals.
    - All rules are one-to-one, one-to-many, or one-to-none.
- **Regular languages** are subsets of context free languages.
- **Regular language** (also called a rational language) is a formal language that can be expressed using a regular expression.

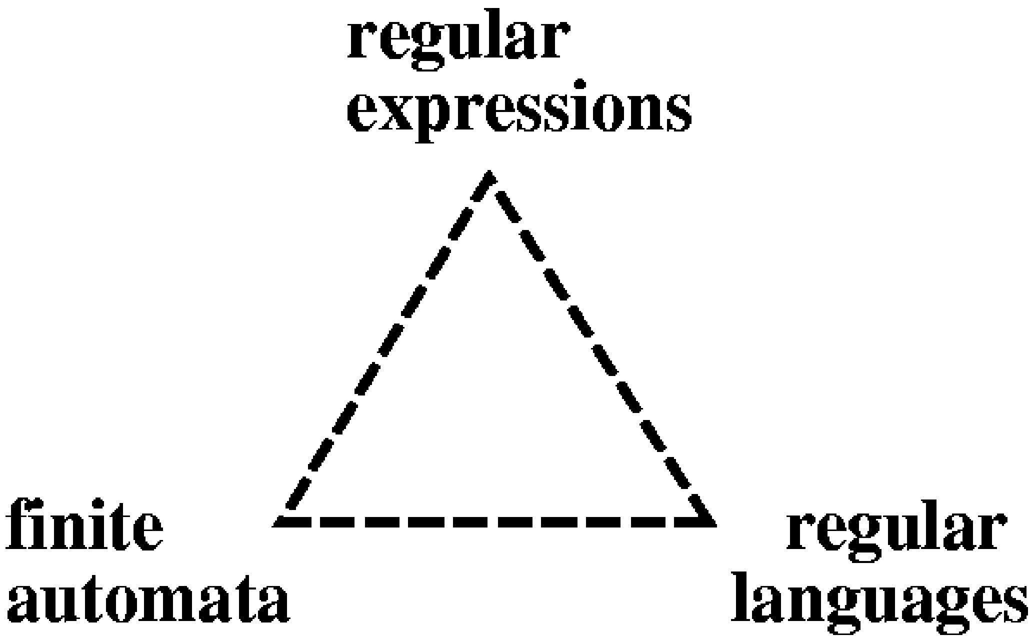


FINITE-STATE AUTOMATA

- The **regular expression** is more than just a convenient meta-language for text searching.
  - ***First***, regular expression is one way of **describing a finite-state automaton** (FSA).
    - Any regular expression can be implemented as a finite-state automaton.
    - Any finite-state automaton can be described with a regular expression.
  - ***Second***, 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 described regular languages.

FINITE-STATE AUTOMATA

- The relationship between finite automata, regular expressions, and regular languages



Question

Question: Which one of the following languages over the alphabet {0,1} is described by the regular expression?

$$(0+1)^*0(0+1)^*0(0+1)^*$$

(A) The set of all strings containing the substring 00.

(B) The set of all strings containing at most two 0's.

(C) The set of all strings containing at least two 0's.

(D) The set of all strings that begin and end with either 0 or

Question: Which of the following languages is generated by given grammar?

$$S \rightarrow aS \mid bS \mid \epsilon$$

(A) $\{a^n b^m \mid n, m \geq 0\}$
(B) $\{w \in \{a,b\}^* \mid w$ has equal number of a's and b's$\}$
(C) $\{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\}$
(D) $\{a, b\}^*$

Question: The regular expression $0^*(10^*)^*$ denotes the same set as:

(A) $(1^*0)^*1^*$
(B) $0 + (0 + 10)^*$
(C) $(0 + 1)^* 10(0 + 1)^*$
(D) none of these

FSA to Recognize

- Let's begin with the “sheep language”.
- we defined the sheep language as any string from the following (infinite) set:

baa!

baaa!

baaaa!

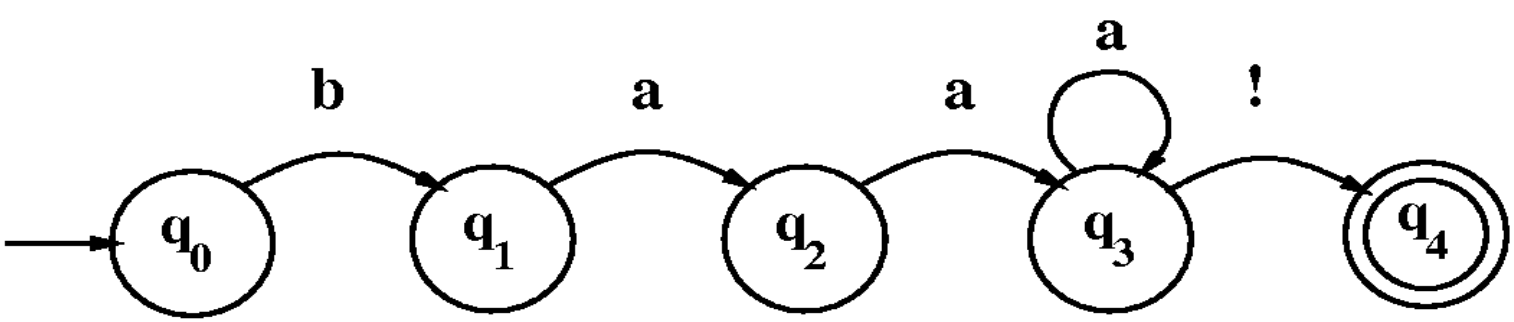
baaaaa!

baaaaaa!

....

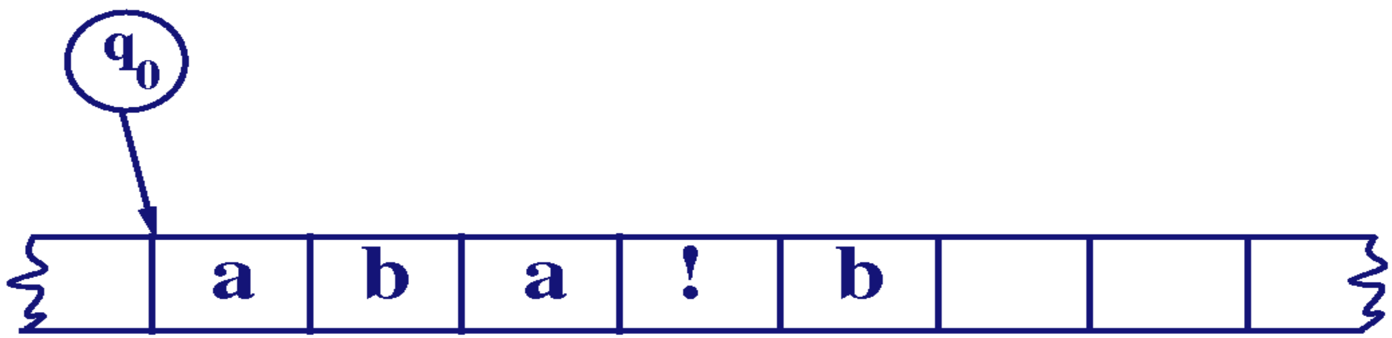
The regular expression for this kind of 'sheep talk' is:

/baa+ !/.



Finite automaton (Finite-State Automaton)

- The FSA can be used for **recognizing** (accepting) strings in the following way.
  - First, think of the input as being written on a long **tape** broken up into cells, with one symbol written in each cell of the tape.



- If the machine never gets to the final state,
  - Either because it runs out of input, or it gets some input that doesn't match an arc, or if it just happens to get stuck in some non-final state,
- we say the **machine rejects** or fails to accept an input.

State-Transition Table

- We can also represent an automaton with a state-transition table.
  - As in the graph notation, the state-transition table represents:
    - the start state,
    - the accepting states,
    - what transitions leave each state with which symbols.

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

Finite Automaton

- Finite automaton is defined by the following 5 parameters:
  - Q: a finite set of N states q_0, q_1, ……,q_N
  - $\sum$: a finite input alphabet of symbols
  - q_0: the start state
  - F: the set of final states, $F \subseteq Q$
  - δ (q, i): the transition function or transition matrix between states.
  - Given a state $q \in Q$ and an input symbol $i \in \sum$, $\delta(q, i)$ returns a new state $q' \in Q$. δ is thus a relation from Q x $\sum$ to Q;

- **<u>For the sheeptalk</u>**
  - automaton $Q = \{q_0,q_1,q_2,q_3,q_4\}$,
  - $\sum = \{a, b, !\}$, $F = \{q_4\}$, and
  - $\delta(q, i)$ is defined by the transition table

Recognition

- The process of determining if a string should be accepted by a machine
- The process of determining if a string is in the language we’re defining with the machine
- The process of determining if a regular expression matches a string

Recognition

- At the start state, Examine the current input (tape)
- Consult the transition table
- Go to the next state and update the tape pointer
- Repeat until you run out of tape

Deterministic Recognition Algorithm

- D-RECOGNIZE begins by initializing the variables index and current-state to the beginning of the tape and the machine's initial state. D-RECOGNIZE then enters a loop that drives the rest of the algorithm.

```
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
    elsif transition-table[current-state,tape[index]] is empty then
        return reject
    else
        current-state ← transition-table[current-state,tape[index]]
        index ← index + 1
  end
```

D-Recognize

1. Index the tape to the beginning and the machine to the initial state.
2. First check to see if you have any **more** input
   - If no and you're in a final state, ACCEPT
   - If no and you're in a non-final state, REJECT
3. If you have **more input check,** what state you're in by consulting the transition table.
   - The index of the **Current State** tells you what row in the table to consult.
   - The index on the tape symbol tells you what column to consult in the table.

Loop through until no more input then go back to 2.

Formal Language

- Even if the automaton had allowed an initial **a** it would have certainly failed on **c**, (since c isn't even in the sheeptalk alphabet !).
  - We can think of these **'empty' elements in the table** as if they all pointed at one 'empty' state, which we might call the **fail** state or **sink** state.
- **Formal language:** A model which can both **generate and recognize** all and only the strings of a formal language.
- Formal language is a set of strings,
  - each string composed of symbols from a finite symbol-set called an alphabet.

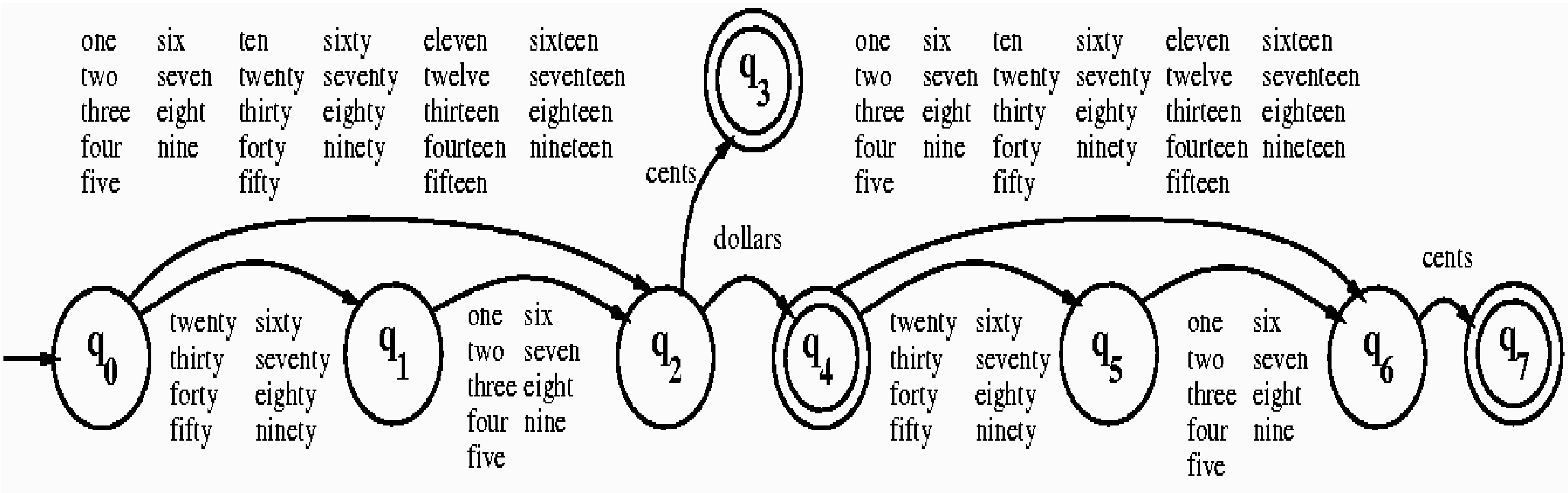
D-Recognize

- Deterministic means that at each point in processing there is always one unique thing to do (NO CHOICES)

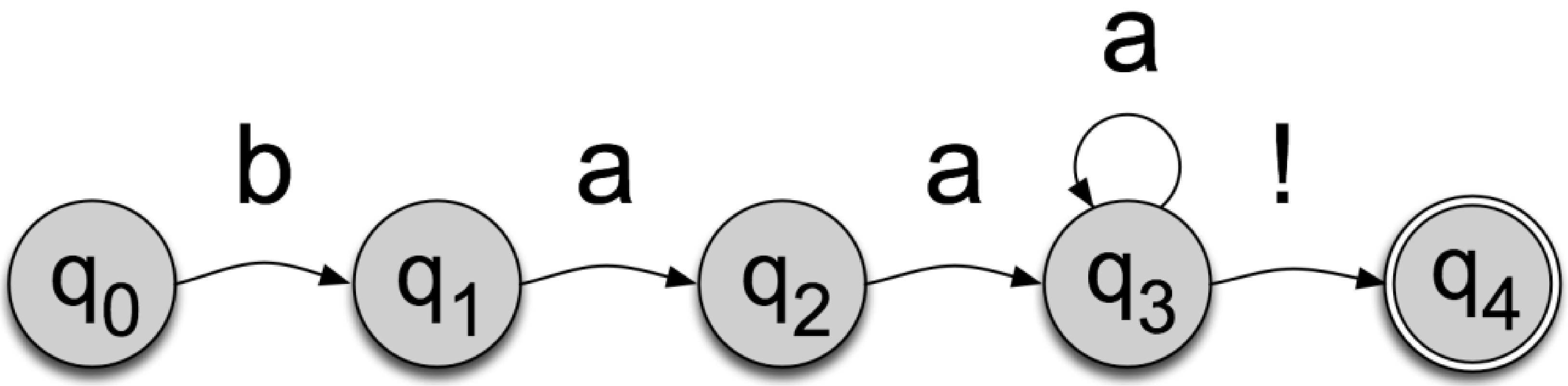
- D-recognize algorithm is a simple table-driven interpreter
- **D-Recognize is a deterministic algorithm**

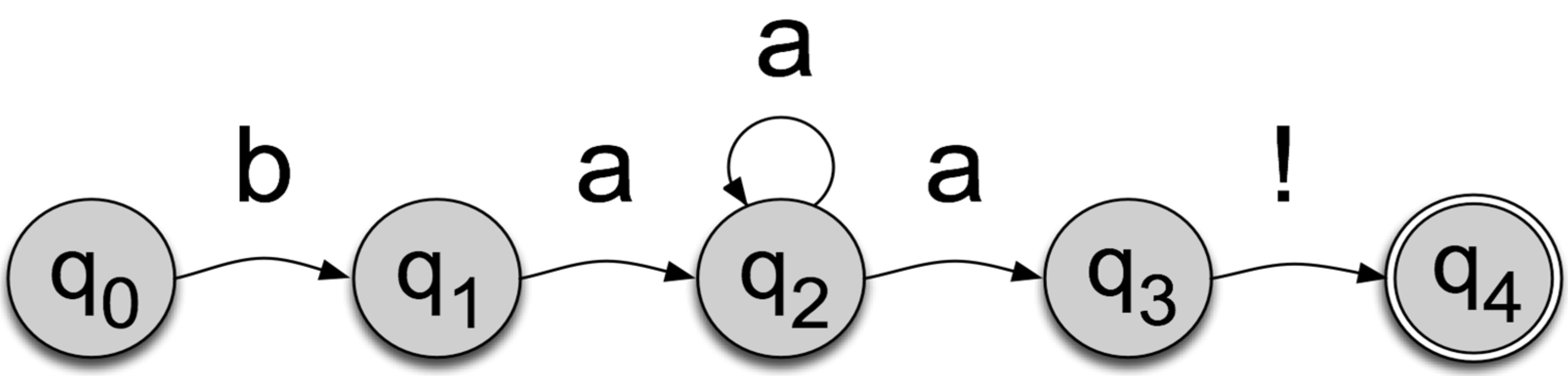
Example

- FSA for the simple dollars and cents

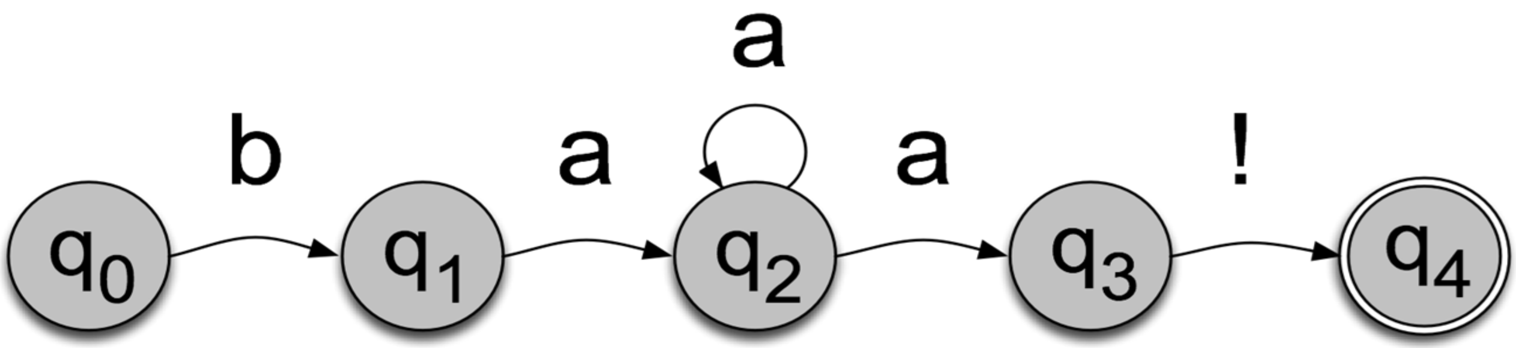


Difference Between These Two

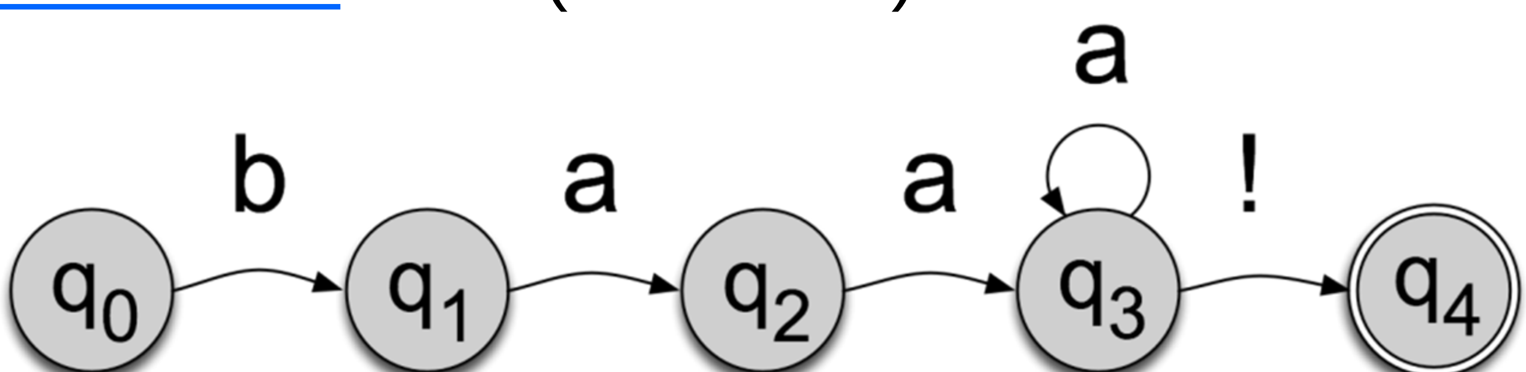




Non-deterministic FSA (NFSA)

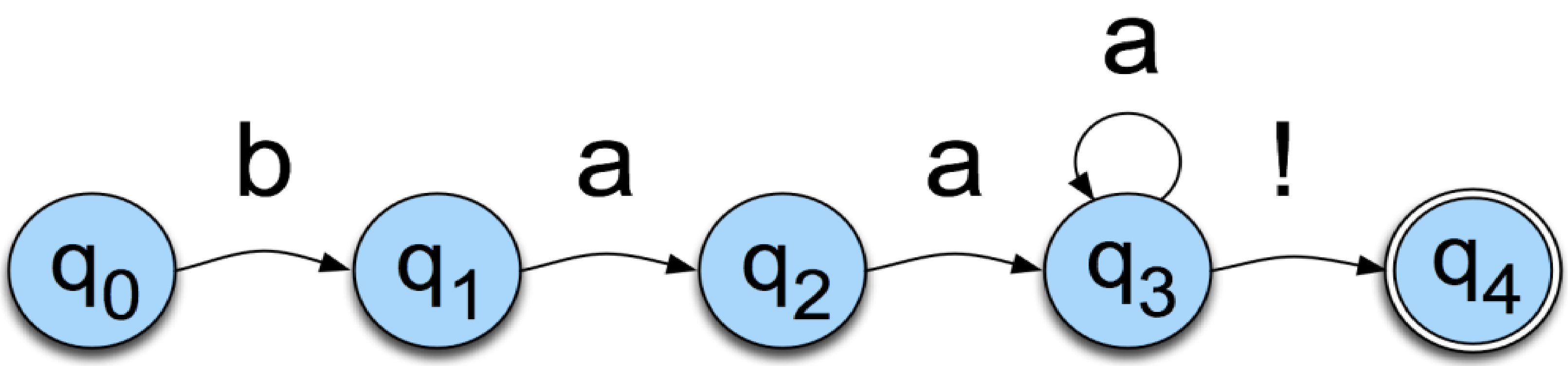


When we get to state 2, if we see an *a* we don't know whether to remain in state 2 or go on to state 3. Automata with decision points like this are called non-deterministic FSAs (or NFSAs).



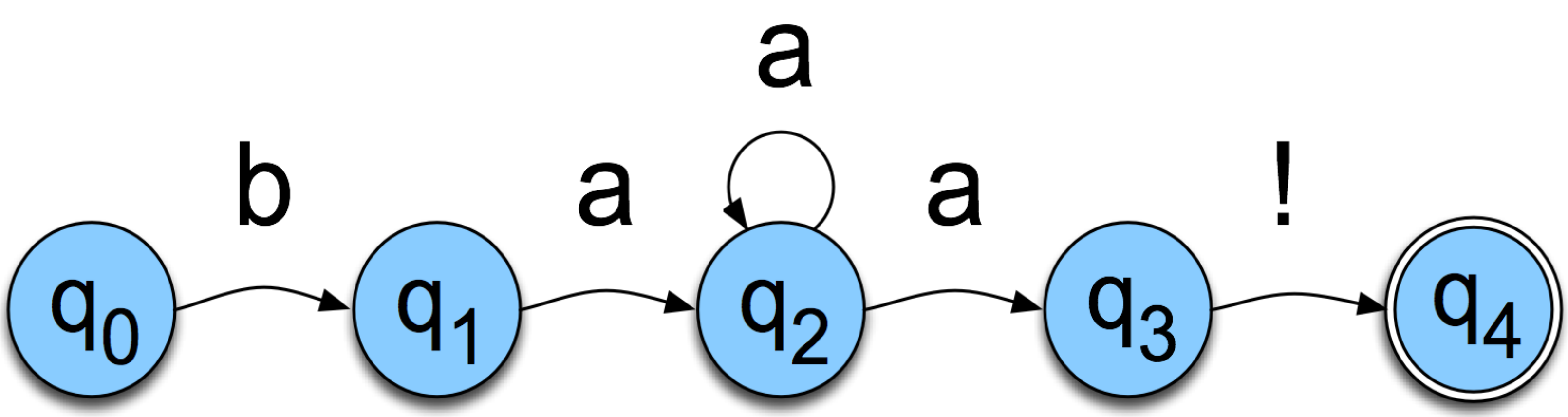
Determinism

- This automaton is deterministic
- If we're in any state, and we see a given input, there's only one place to go next (or else we fail).



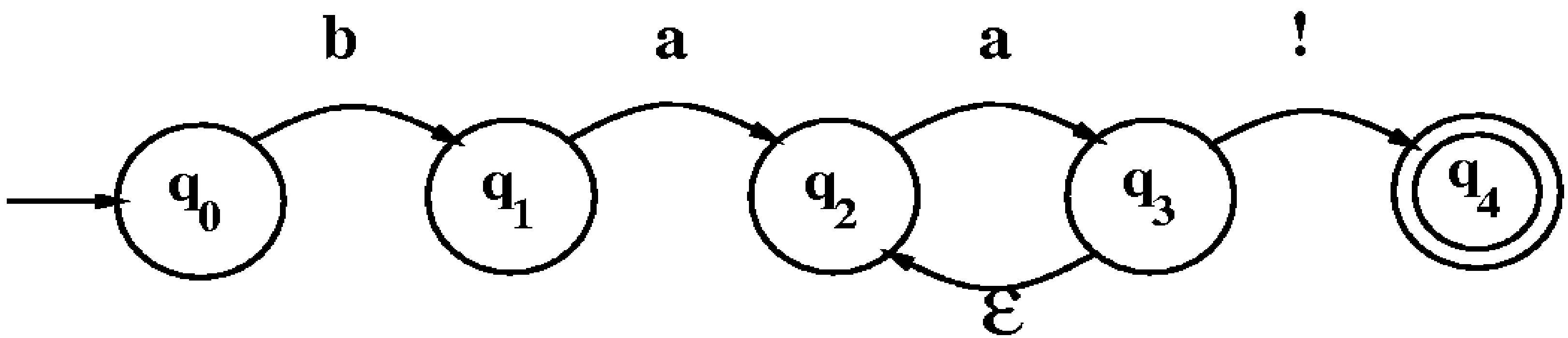
Non-Determinism

- This automaton is non-deterministic
- If we're in state 2, and we see an "a", we can go back to state 2, or move to state 3.



Another type: Non-deterministic FSA

- There is another common type of non-determinism, which can be caused by arcs that have **no symbols** on them (called ε-transitions).



We interpret this new arc as follows: if we are in state 3, we are allowed to move to *state 2 without looking at the input*, or advancing our input pointer.

- So this introduces another kind of non-determinism - we might not know whether to follow the epsilon-transition or the ! arc.

Why to use Non-determinism

- Non-deterministic machines can **be converted to** deterministic one with a fairy 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 and can't accept.

- Non-determinism doesn't get us more formal power and it causes headaches so why to use it?
  - More natural (understandable) solutions
  - Deterministic Machines are too big

NFSA to accept strings

- If we want to know whether a string is an instance of sheeptalk or not,
- If we use a non-deterministic machine to recognize it,
  - Follow the wrong arc and reject it when we should have accepted it.
  - This problem of choice in non-deterministic models will come up again and again as we build computational models, particularly for parsing.

NFSA to accept strings

- There are three standard solutions to this problem:
  - **Backup:** At a choice point, we could put a marker to mark where we were in the input, and what state the automaton was in.
    - Then if it turns out that we took the wrong choice, we could back up and try another path.
  - **Look-ahead:** We could look ahead in the input to help us decide which path to take.
  - **Parallelism:** Whenever we come to a choice point, we could look at every alternative path in parallel.

NFSA Transition Table

- First, in order to represent nodes that have outgoing ε-transitions, we add a new ε-column to the transition table. If a node has an ε -transition, we list the destination node in the ε -column for that node's row.
- The second addition is needed to account for multiple transitions to different nodes from the same input symbol.

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

NFSA recognition algorithm

```
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 ∪ GENERATE-NEW-STATES(current-search-state)
    if agenda is empty then
       return reject
    else
       current-search-state ← NEXT(agenda)
  end

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
  return a list of search states from transition table as follows:
    (transition-table[current-node,ε], index)
    ∪
    (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

  current-node ← the node 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
```

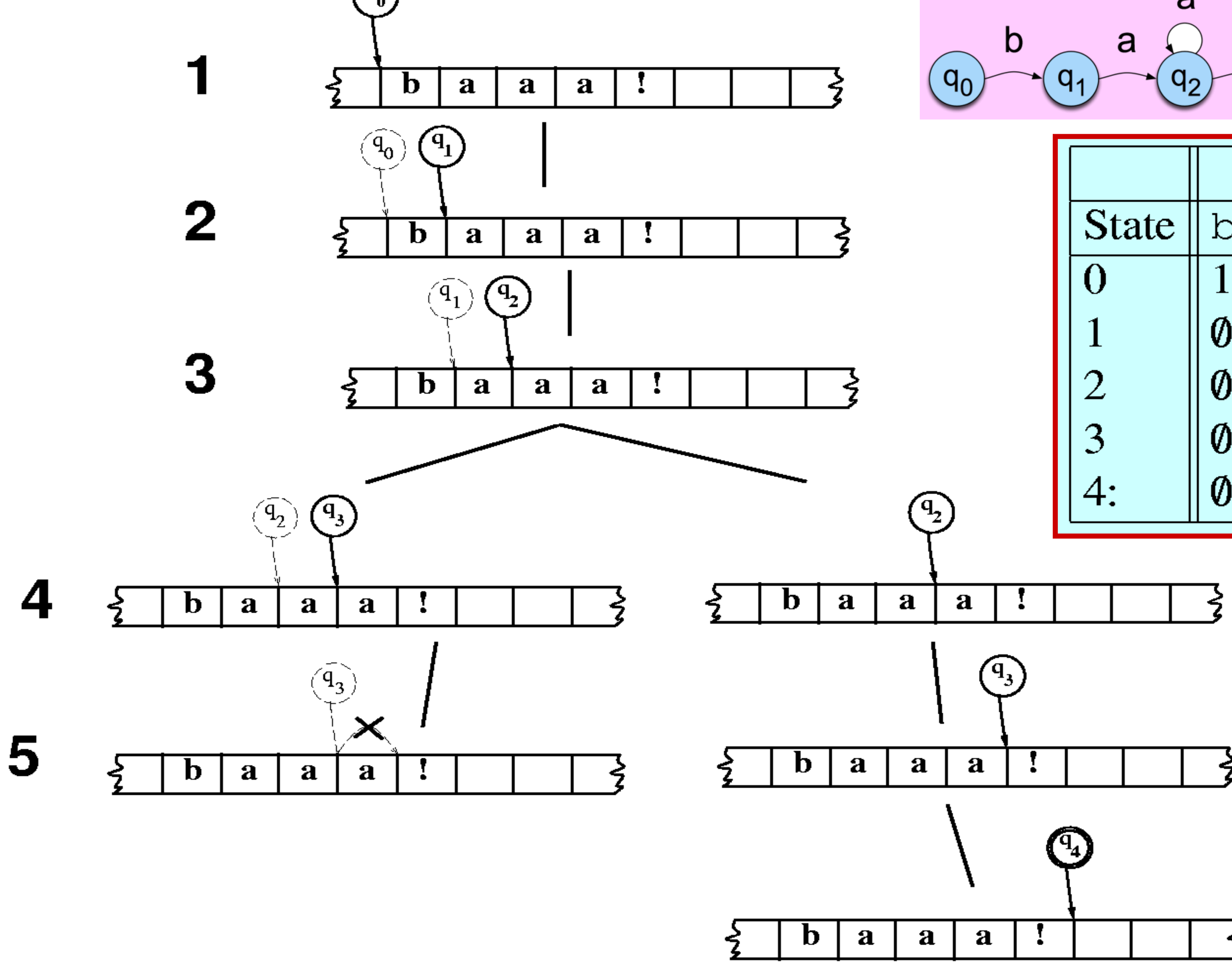
NFSA Recognition Algorithm

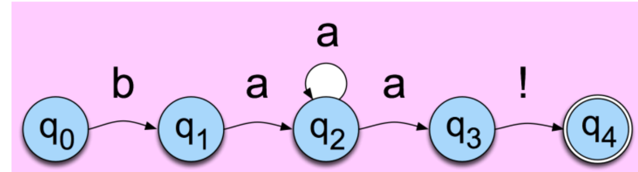
- The function ND-RECOGNIZE uses the variable agenda to keep track of all the currently unexplored choices generated during the course of processing.
- Each choice (search state) is a tuple consisting of a node (state) of the machine and a position on the tape.
- The variable current- search-state represents the branch choice being currently explored.

NFSA recognition algorithm

- It is important to understand why ND-RECOGNIZE returns a value of reject only when the agenda is found to be **empty.**
- Unlike D-RECOGNIZE, it does not return reject when it reaches the end of the tape in an non-accept machine-state or when it finds itself unable to advance the tape from some machine-state. This is because, in the non-deterministic case, such road-blocks only indicate failure down a given **path, not overall failure**.
- We can only be sure we can reject a string when **all possible choices** have been examined and found lacking.

Tracing the execution of NFSA





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

REGULAR LANGUAGES AND FSAs

- The class of languages that are definable by regular expressions is exactly the same as the class of languages that are characterizable by finite-state automata (deterministic or non-deterministic).
  - Because of this, we call these languages the regular languages.
- In order to give a formal definition of the class of regular languages, we need to refer back to two earlier concepts:
  - the alphabet Σ, which is the set of all symbols in the language, and
  - the empty string ε, which is conventionally not included in Σ.

REGULAR LANGUAGES AND FSAs

- In addition, we make reference to the empty set ϕ (which is distinct from ε).
- The class of regular languages (or regular sets) over $\sum$ is then formally as follows:

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

- All and only the sets of languages which meet the above properties are **regular languages**.

Regular languages

- All and only the sets of languages which meet the above properties are **regular languages**.
- **Three operations which define regular languages:**
  - **Concatenation,**
  - **Disjunction/union (also called '|'), and**
  - **Kleene closure.**

- For example all the counters (*,+, { n, m}) are just a special case of repetition.
- The square braces [ ] are a kind of disjunction (i.e. [ab] means "a or b", or the disjunction of a and b).

Regular languages

Regular languages are also closed under the following operations (where Σ^* means the infinite set of all possible strings formed from the alphabet Σ):

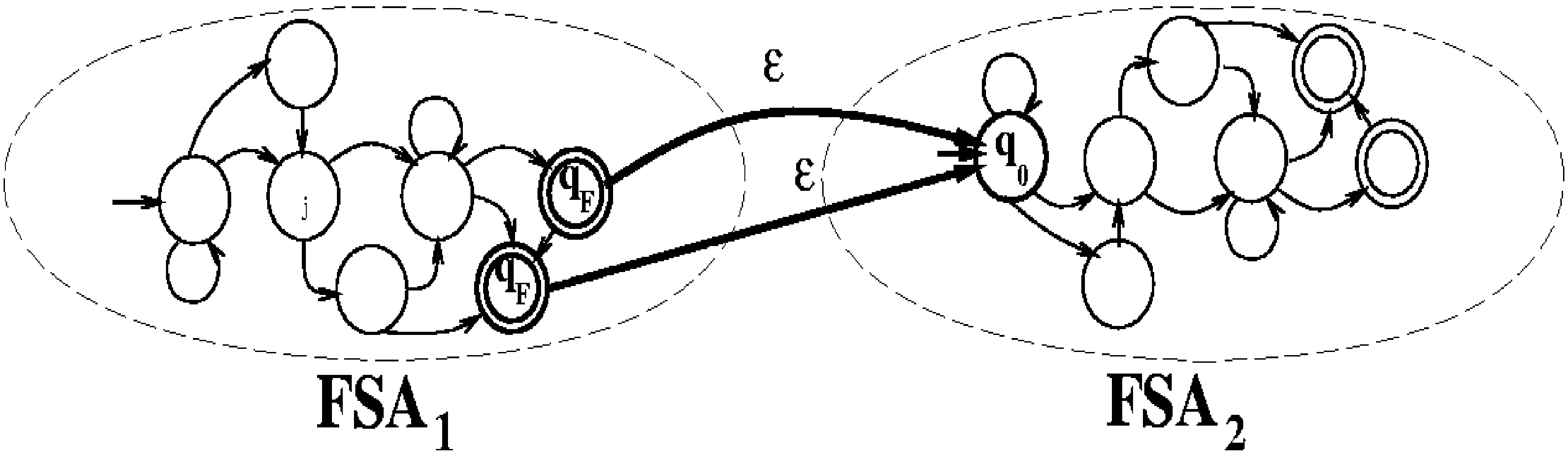
- intersection: if L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$, the language consisting of the set of strings that are in both L_1 and L_2.
- difference: if L_1 and L_2 are regular languages, then so is $L_1 - L_2$, the language consisting of the set of strings that are in L_1 but not L_2.
- complementation: If L_1 is a regular language, then so is $\Sigma^* - L_1$, the set of all possible strings that aren't in L_1
- reversal: If L_1 is a regular language, then so is L_1^R, the language consisting of the set of reversals of all the strings in L_1.

Primitive operations

- For the inductive step, the primitive operations of a regular expression (*concatenation, union, closure*) can be imitated by an automaton:

- **Concatenation:** We just string two FSAs next to each other by connecting all the final states of FSA_1 to the initial state of FSA_2 by an **ε**-transition.

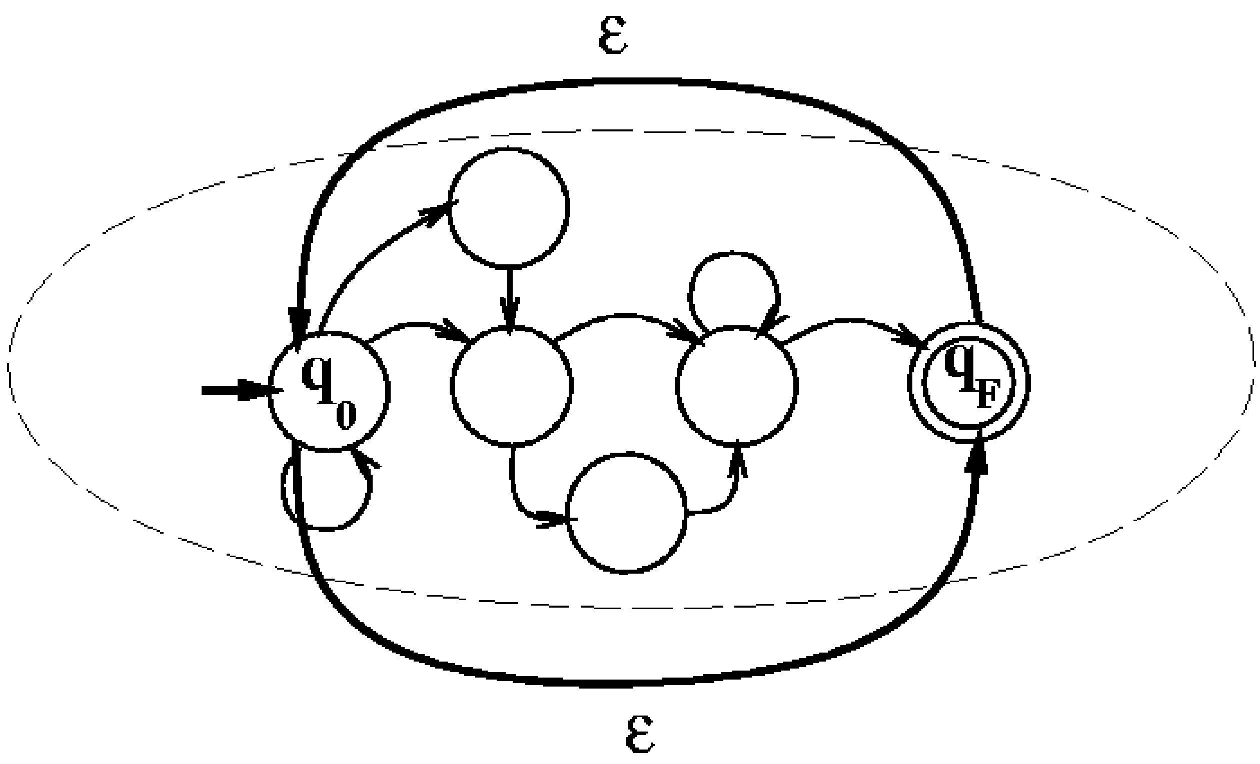
The concatenation of two FSAs



Primitive operations

- For the inductive step, the primitive operations of a regular expression (*concatenation, union, closure*) can be imitated by an automaton:
- **Closure:** We connect all the final states of the FSA back to the initial states by **ε**-transitions (***Kleene *),*** then put direct links between the initial and final states by **ε transitions** (this implements the possibly of having ***zero occurrences).***

The closure (Kleene *) of an FSA

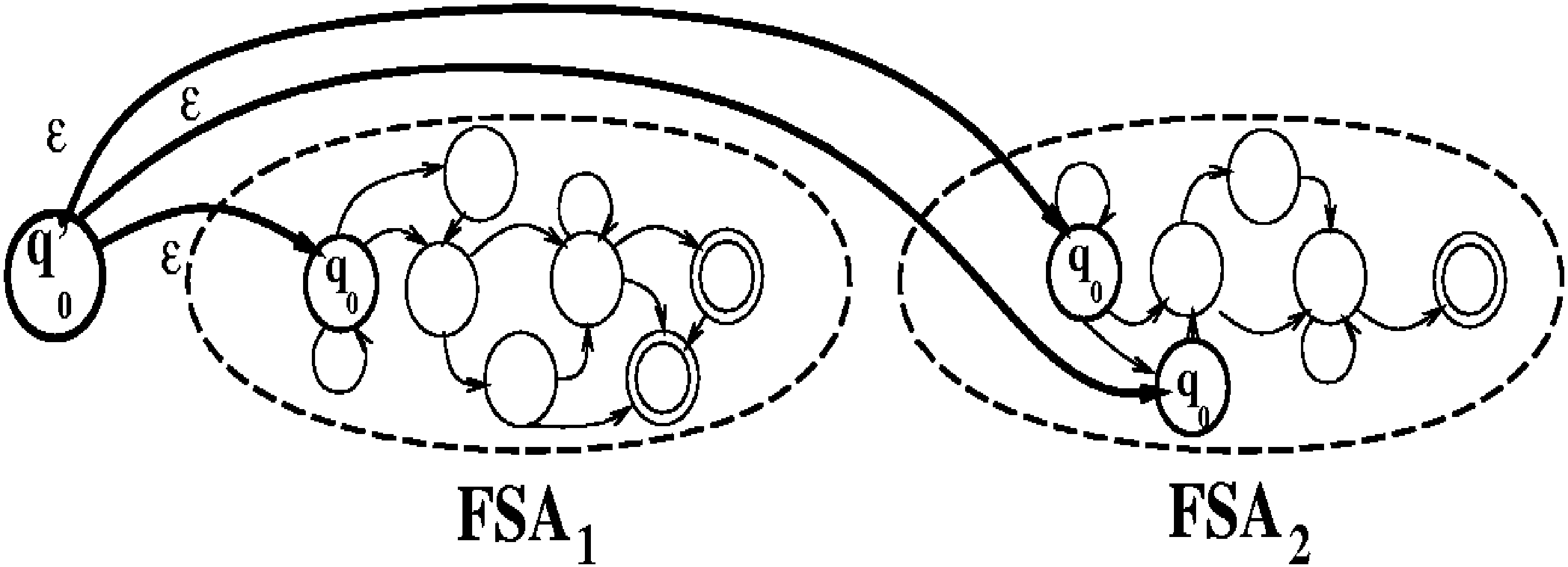


Primitive operations

- For the inductive step, the primitive operations of a regular expression (*concatenation, union, closure*) can be imitated by an automaton:

- **Union:** We add a single new initial state *q'_0, and add new transitions* from it to all the former initial states of the two machines to be joined.

The union (|) of two FSAs



Negation

- Construct a machine M_2 to accept all strings not accepted by machine M_1 and reject all the strings accepted by M_1
  - Invert all the accept and not accept states in M_1

Regular expressions and FSAs: Phone numbers

- Regular expression to validate phone numbers:
  - **(+44)(0)20-12341234, 02012341234, +44 (0) 1234-1234**
  - But not: **(44+)020-12341234, 12341234(+020)**

Regular expressions and FSAs: Phone numbers

- Regular expression to validate phone numbers:
  - **(+44)(0)20-12341234, 02012341234, +44 (0) 1234-1234**
  - But not: **(44+)020-12341234, 12341234(+020)**

  ^(\(?\+?[0-9]*\)?)?[0-9_\- \(\)]*$
- FSA: