Introduction to Languages and Computation

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Computational Models as Ideal Computers

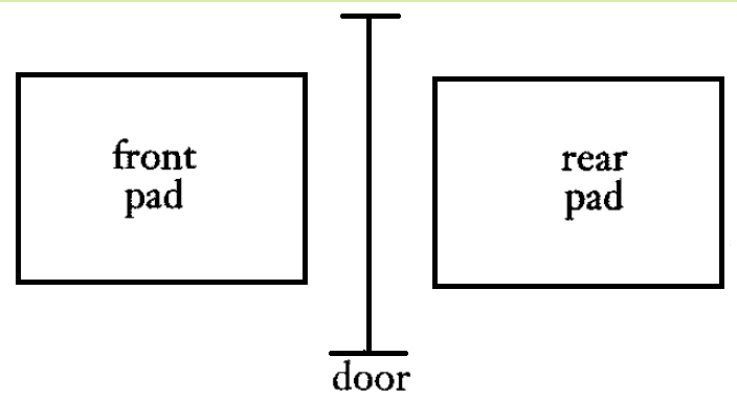
- We begin with the question: "What is a computer?"
- Real computers are too complicated to allow for a manageable mathematical theory.
- We replace a real computer by an idealized computer called a **computational model**.
- As with any scientific model, a computational model may be accurate in some ways but perhaps not in others.
- We will use several different computational models, depending on the features we want to focus on.
- We begin with the simplest model, called the finite state machine or finite automaton.

Subsection 1

Finite Automata

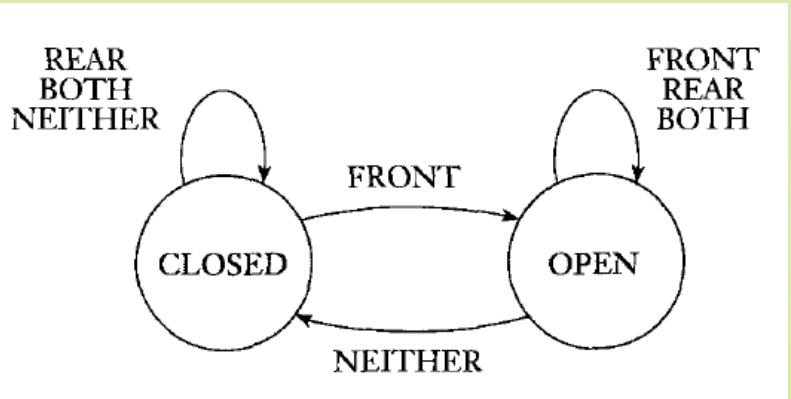
Finite Automata and Control

- Finite automata are good models for computers with an extremely limited amount of memory.
- Computers with such a small memory are still very useful.
- Such computers are very common and lie at the heart of various electromechanical devices.
  - The controller for an automatic door is an example.
    - An automatic door has a pad in front to detect the presence of a person about to walk through the doorway.
    - Another pad is located to the rear so that the controller can hold the door open long enough for the person to pass and also so that the door does not strike someone standing behind it as it opens.



Operation of the Door Controller

- The controller is in either of two states: "OPEN" or "CLOSED".
- There are four possible input conditions:
  - "FRONT" (a person is standing on the pad in front of the doorway);
  - "REAR" (a person is standing on the pad to the rear of the doorway);
  - "BOTH" (people are standing on both pads)
  - "NEITHER" (no one is standing on either pad).
- The controller moves from state to state, depending on the input:
  - When in the CLOSED state and receiving input NEITHER or REAR, it remains in the CLOSED state. In addition, if the input BOTH is received, it stays CLOSED. If the input FRONT arrives, it moves to the OPEN state.
  - In the OPEN state, if input FRONT, REAR, or BOTH is received, it remains in OPEN. If input NEITHER arrives, it returns to CLOSED.



Applications and Usefulness

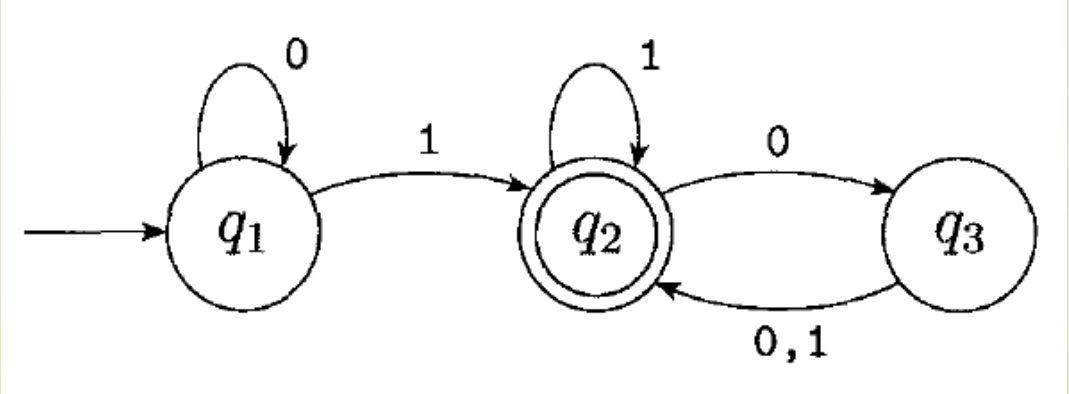
- The door controller may also be represented in tabular form:

	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

  It is a computer that has just a single bit of memory.
- Other common devices have controllers with larger memories.
  - In an elevator controller a state may represent the floor the elevator is on and the inputs might be the signals received from the buttons. This computer might need several bits to keep track of this information.
  - Controllers for various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators, are also examples of computers with limited memories.
- The design of such devices uses the methodology of finite automata.
- Studying finite automata
  - clarifies what they are and what they can and cannot do;
  - allows practicing with mathematical definitions, theorems, and proofs in a relatively simple setting.

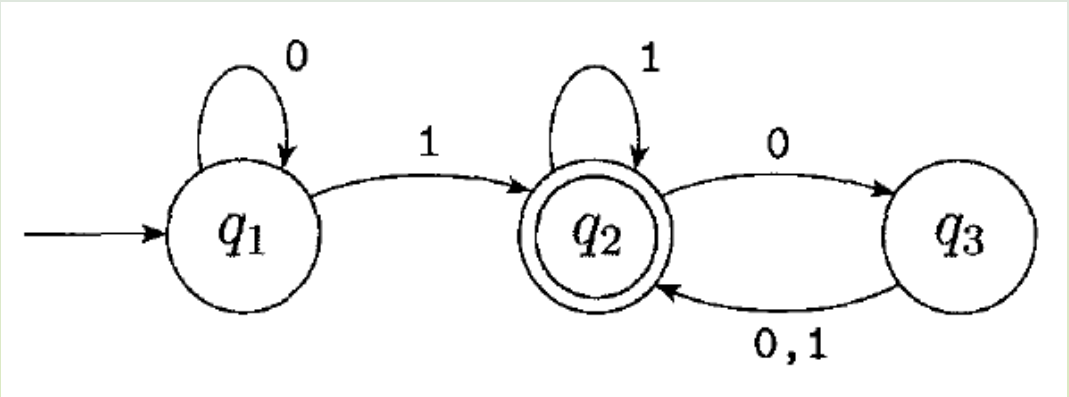
A Finite State Automaton M_1

- The following figure depicts a finite automaton M_1:



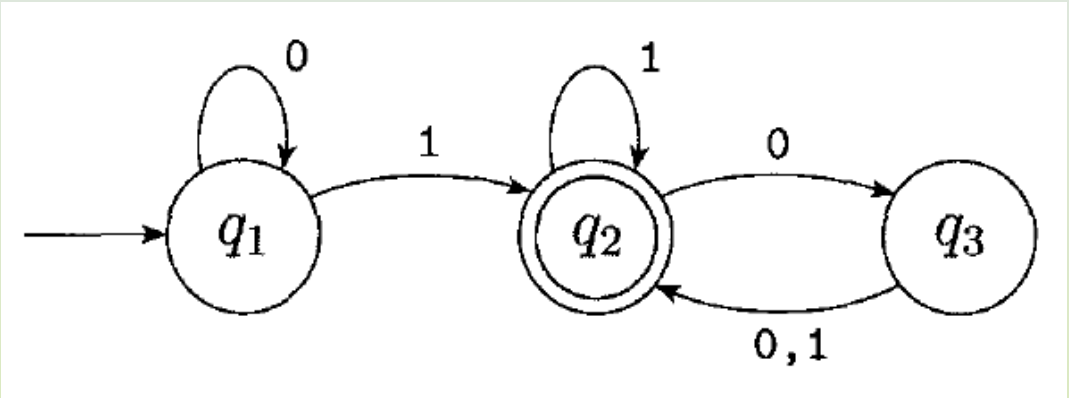
- It is called the **state diagram** of M_1.
- It has three **states** q_1, q_2 and q_3.
- The **start state** q_1 is indicated by the sourceless arrow pointing at it.
- The **accept state** q_2 is the one with a double circle.
- The arrows going from one state to another are called **transitions**.
- When this automaton receives an input string such as 1101, it processes that string and produces an output.
- The output is either **accept** or **reject**.

Operation of M_1



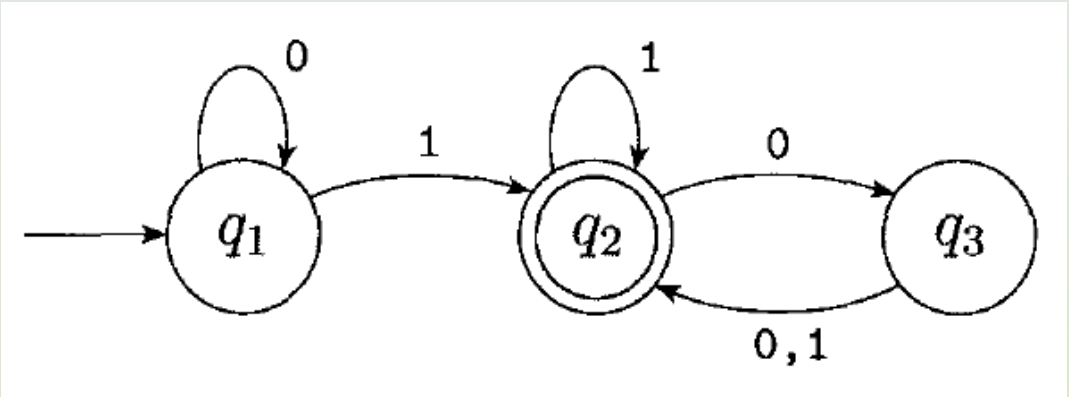
- The processing begins in M_1's start state.
- The automaton receives the symbols from the input string one by one from left to right.
- After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label.
- When it reads the last symbol, M_1 produces its output.
  - The output is accept if M_1 is now in an accept state.
  - It is reject if it is not.

An Example of the Operation of M_1



- Example: We feed 1101 to the machine M_1:
  1. Start in state q_1.
  2. Read 1, follow transition from q_1 to q_2.
  3. Read 1, follow transition from q_2 to q_2.
  4. Read 0, follow transition from q_2 to q_3.
  5. Read 1, follow transition from q_3 to q_2.
  6. Accept because M_1 is in an accept state q_2 at the end of the input.

Language Accepted by M_1



- Experimenting with this machine on a variety of input strings reveals that it accepts the strings $1, 01, 11$, and 0101010101.
- In fact, M_1 accepts any string that ends with a 1, as it goes to its accept state q_2 whenever it reads the symbol 1.
- It also accepts strings $100, 0100, 110000$, and 0101000000, and any string that ends with an even number of 0s following the last 1.
- It rejects other strings, such as $0, 10, 101000$.
- What is the language consisting of all strings that M_1 accepts?

Formal Definition: Raison d'être and Components

- We need a formal definition for finite automata:
  - A formal definition is precise. It resolves any uncertainties about what is allowed in a finite automaton.
  - A formal definition provides notation. Good notation helps us think and express thoughts clearly.
- A finite automaton has several parts.
  - It has a set of states and rules for going from one state to another, depending on the input symbol.
  - It has an input alphabet that indicates the allowed input symbols.
  - It has a start state and a set of accept states.
- Formally, a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states.
- We use a transition function, frequently denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol 1, the transition function imposes $\delta(x, 1) = y$.

Definition of Finite Automata

Definition (Finite Automaton)

A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of **states**,
2. Σ is a finite set called the **alphabet**,
3. $\delta : Q \times \Sigma \to Q$ is the **transition function**,
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the **set of accept states** or **final states**.

- We can use the notation of the formal definition to describe individual finite automata by specifying each of the five parts listed.
- Example: M_1 is described formally as $M_1 = (Q, \Sigma, \delta, q_1, F)$, where
  1. $Q = \{q_1, q_2, q_3\}$,
  2. $\Sigma = \{0, 1\}$,
  3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

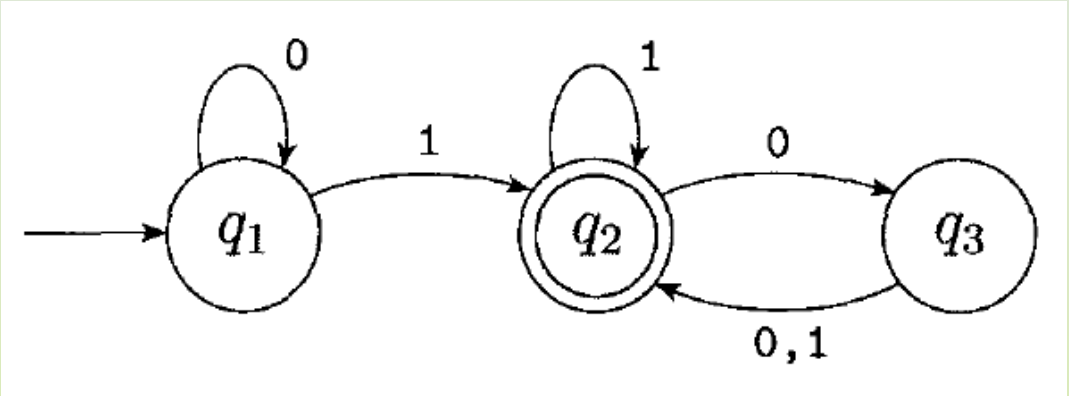
  4. q_1 is the start state, and
  5. $F = \{q_2\}$.

Language Recognized by a Finite Automaton

- If A is the set of all strings that machine M accepts, we say that A is the **language of machine** M and write $L(M) = A$.
- We say that M **recognizes** A or that M **accepts** A.
- Because the term accept has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term **recognize** for languages in order to avoid confusion. A machine may accept several strings, but it always recognizes only one language.
- If the machine accepts no strings, it still recognizes one language, the empty language $\emptyset$.

Example of Language Recognized by a Finite Automaton

- Consider again M_1:



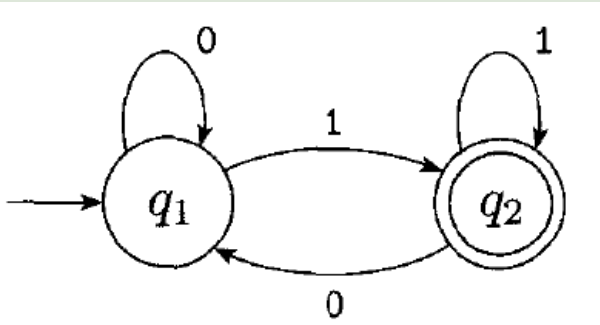
Let

$$A = \{w : w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}.$$

Then $L(M_1) = A$, or equivalently, M_1 recognizes A.

Another Example

- Consider now the finite automaton M_2.



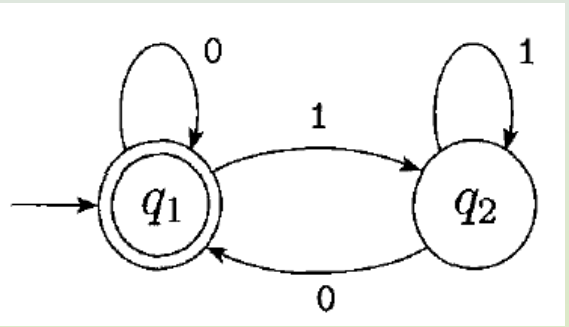
δ	0	1
q_1	q_1	q_2
q_2	q_1	q_2

- $M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$, with δ shown above.
- When we try the machine on some sample input strings, its method of functioning often becomes apparent.
  - On the sample string 1101 the machine M_2 starts in its start state q_1 and proceeds first to state q_2 after reading the first 1, and then to states q_2, q_1 and q_2 after reading $1, 0$ and 1. The string is accepted because q_2 is an accept state.
  - But string 110 leaves M_2 in state q_1, so it is rejected.

  After trying a few more examples, you would see that M_2 accepts all strings that end in a 1. Thus, $L(M_2) = \{w : w \text{ ends in a } 1\}$.

Machine M_3

- Consider the finite automaton M_3:

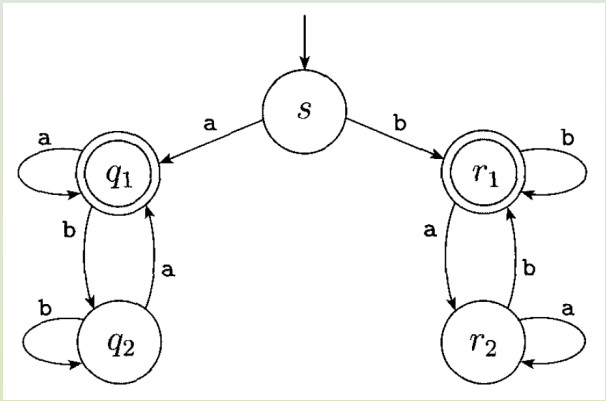


- Machine M_3 is similar to M_2 except for the accept state.
- As usual, the machine accepts all strings that leave it in an accept state when it has finished reading.
- Since the start state is also an accept state, M_3 accepts the empty string ε. As soon as a machine begins reading the empty string it is at the end, so, if the start state is an accept state, ε is accepted.
- In addition to the empty string, this machine accepts any string ending with a 0:

$$L(M_3) = \{w : w \text{ is the empty string } \varepsilon \text{ or ends in a } 0\}.$$

Machine M_4

- The following figure shows a five-state machine M_4:

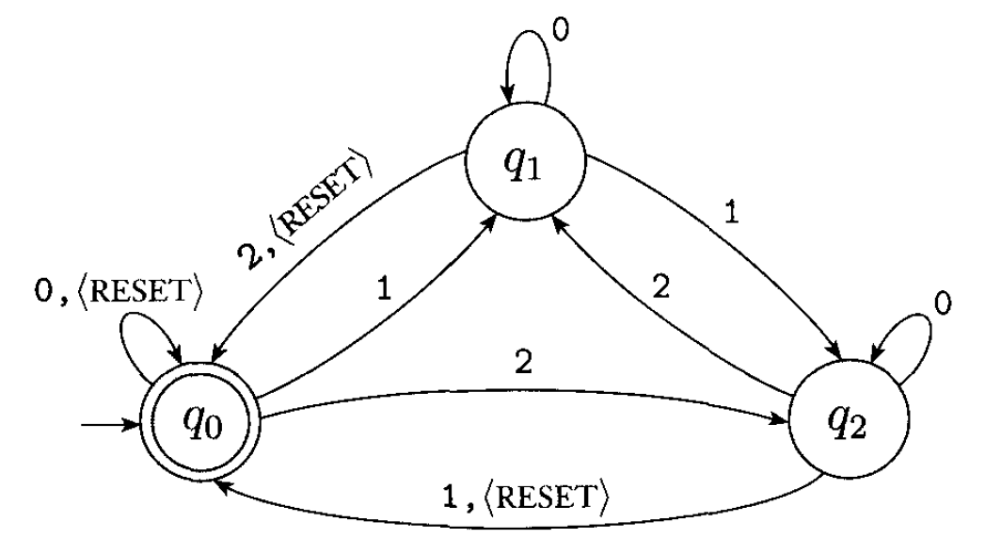


M_4 has two accept states, q_1 and r_1, and operates over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.
Some experimentation shows that it accepts strings $\mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{bb}$ and $\mathtt{bab}$, but not strings $\mathtt{ab}, \mathtt{ba}$, or $\mathtt{bbba}$.

- This machine begins in state s, and after it reads the first symbol, it goes either left into the q states or right into the r states.
  - If the first symbol in the input string is a, then it goes left and accepts when the string ends with an a.
  - Similarly, if the first symbol is a b, the machine goes right and accepts when the string ends in b.
- So M_4 accepts all strings that start and end with a or that start and end with b. In other words, M4 accepts strings that start and end with the same symbol.

Machine M_5

- Machine M_5, shown below, has a four-symbol input alphabet, $\Sigma = \{\langle \text{RESET} \rangle, 0, 1, 2\}$.



- Machine M_5 keeps a running count of the sum of the numerical input symbols it reads, modulo 3.
- Every time it receives the $\langle \text{RESET} \rangle$ symbol it resets the count to 0.
- It accepts if the sum is 0 modulo 3, i.e., if the sum is a multiple of 3.

Formal Description of Finite Automata

- If describing a finite automaton by state diagram is not possible, either because of size or because the description depends on some unspecified parameter, we resort to a formal description:
- Example: Consider a generalization of M_5, using the same four-symbol alphabet Σ. For each $i > 1$ let A_i be the language of all strings where the sum of the numbers is a multiple of i, except that the sum is reset to 0 whenever the symbol $\langle \texttt{RESET} \rangle$ appears. For each A_i we give a finite automaton B_i, recognizing A_i. B_i is described formally as $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$:
  - Q_i is the set of i states $\{q_0, q_1, q_2, \ldots, q_{i-1}\}$.
  - We design the transition function δ_i so that for each j, if B_i is in q_j, the running sum is j, modulo i: For each q_j let

$$\begin{aligned} \delta_i(q_j, 0) &= q_j, \\ \delta_i(q_j, 1) &= q_k, \text{ where } k = j + 1 \pmod{i}, \\ \delta_i(q_j, 2) &= q_k, \text{ where } k = j + 2 \pmod{i}, \\ \delta_i(q_j, \langle \texttt{RESET} \rangle) &= q_0. \end{aligned}$$

Regular Languages

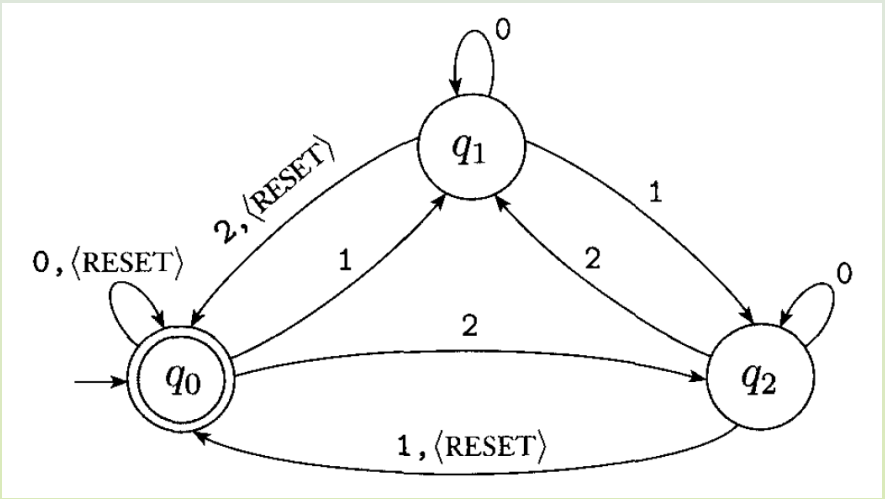
- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ.
  Then M **accepts** w if there exists a sequence of states $r_0, r_1, \ldots, r_n$ in Q satisfying:
  1. $r_0 = q_0$,
  2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$,
  3. $r_n \in F$.
- Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state.
- We say that M **recognizes language** A if $A = \{w : M \text{ accepts } w\}$.

Definition (Regular Language)

A language is called a **regular language** if it is the language recognized by some finite automaton.

An Example

- Consider again machine M_5. Let w be $10\langle\text{RESET}\rangle 22\langle\text{RESET}\rangle 012$.



- Then M_5 accepts w according to the formal definition of computation because the sequence of states it enters when computing on w is $q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0$, which satisfies the three conditions.
- The language of M_5 is

$$L(M_5) = \{w : \text{the sum of the symbols in } w \text{ is 0 modulo 3 except that } \langle\text{RESET}\rangle \text{ resets the count to 0}\}.$$

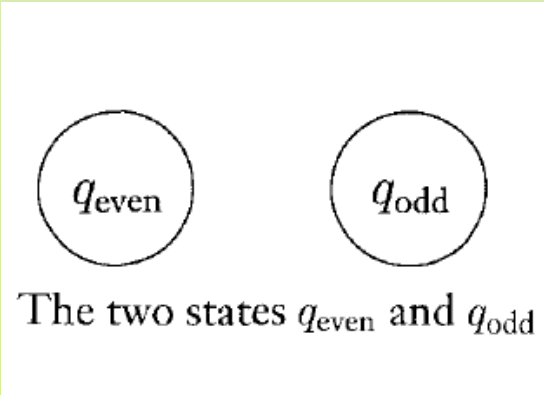
- Since M_5 recognizes this language, it is a regular language.

Tips for Designing an Automaton

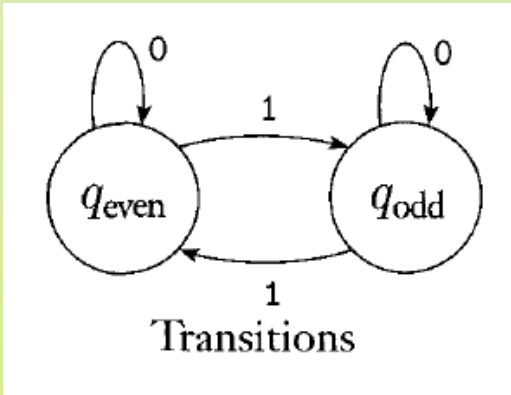
- Designing of an automaton is a creative process.
- Imagine how a machine would go about performing the relevant task.
  - It receives an input string and must determine whether it is a member of the language the automaton is supposed to recognize.
  - It gets to see the symbols in the string one by one.
  - After each symbol, it must decide whether the string seen so far is in the language.
  - Since it does not know when the end of the string is coming, it must always be ready with the answer.
- In order to make these decisions, it has to remember some characteristics of the string as it is reading it.
- For many languages, it does not need to remember the entire input, but only certain crucial information.

Designing an Automaton: A Simple Example

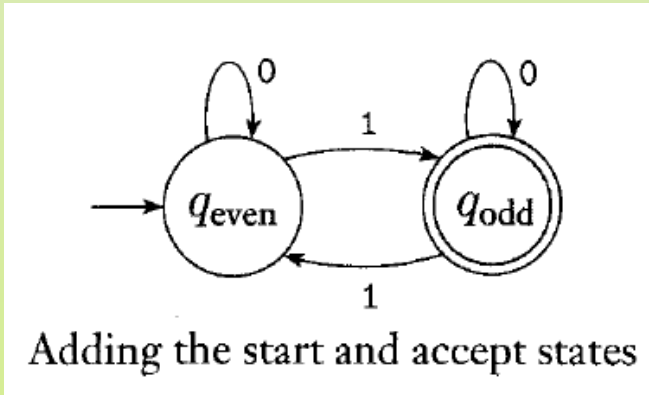
- Suppose that the alphabet is $\{0, 1\}$ and that the language consists of all strings with an odd number of 1s.
- Start by getting an input string of 0s and 1s symbol by symbol.
- We need to remember whether the number of 1s seen so far is even or odd and keep track of this information as we read new symbols.
- If we read a 1, we flip the answer, but if we read a 0, we leave it as is.
- The start state corresponds to the possibility associated with having seen 0 symbols so far, i.e., even number of 1's.
- The accept states are those corresponding to possibilities where we want to accept the input string, i.e., odd number of 1s.



The two states q_{even} and q_{odd}



Transitions



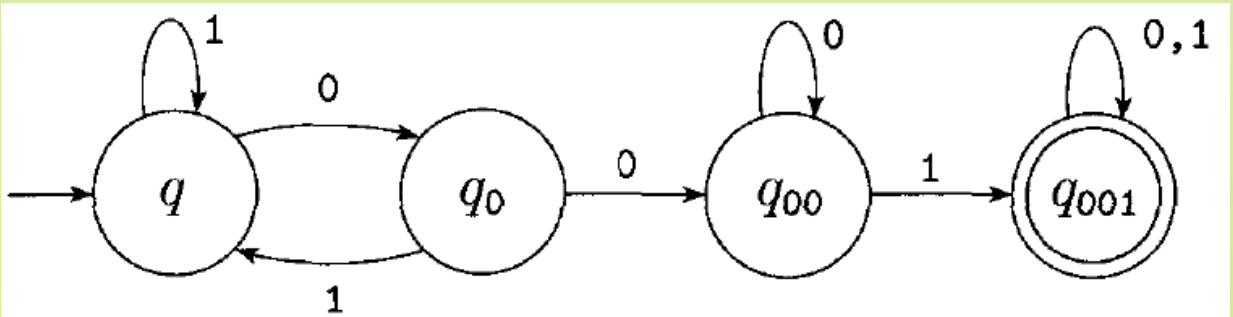
Adding the start and accept states

Designing an Automaton: Another Example

- We would like to design a finite automaton E_2 to recognize the regular language of all strings that contain the string 001 as a substring.
- For example, 0010, 1001, 001 are in the language, but 11 and 0000 are not.
- As symbols come in, we skip over all 1s.
- If we come to a 0, then we note that we may have just seen the first of the three symbols in the pattern 001 we are seeking.
- If at this point we see a 1, there were too few 0s, so we go back to skipping over 1s.
- But if we see a 0 at that point, we remember that we have just seen two symbols of the pattern.
- Now we simply need to continue scanning until seeing a 1.
- If we find it, we remember the success in finding the pattern and continue reading the input string until we get to the end.

Designing an Automaton: Another Example (Cont'd)

- Summarizing, there are four possibilities:
  1. Have not seen any symbols of the pattern.
  2. Have just seen a 0.
  3. Have just seen 00.
  4. Have seen the entire pattern 001.
- Assign the states q, q_0, q_{00} and q_{001} to these possibilities.
- Assign the transitions by observing that
  - in q reading a 1 we stay in q, but reading a 0 we move to q_0;
  - in q_0 reading a 1 we return to q, but reading a 0 we move to q_{00};
  - in q_{00}, reading a 1 we move to q_{001}, but reading a 0 leaves us in q_{00};
  - in q_{001} reading a 0 or a 1 leaves us in q_{001}.
- The start state is q, and the only accept state is q_{001}.



Regular Operations

- We develop a toolbox of techniques to use for
  - designing automata;
  - proving that certain languages are nonregular.
- In arithmetic, the basic objects are numbers and the tools are operations, such as $+$, $\times$ etc.
- In the theory of computation the objects are languages and the tools include operations for manipulating them.
- We define three operations on languages, called the **regular operations**, and use them to study regular languages.

Definition (Regular Operations)

Let A and B be languages. We define the regular operations **union**, **concatenation** and **star** as follows:

- **Union**: $A \cup B = \{x : x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{xy : x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \dots x_k : k \geq 0 \text{ and each } x_i \in A\}$.

Explanations and Examples

- The union operation takes all the strings in both A and B and collects them together into one language.
- The concatenation operation attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.
- The star operation applies to a single language rather than to two different languages. It works by attaching any number of strings in A together to get a string in the new language.
  - Because "any number" includes 0 as a possibility, the empty string ε is always a member of A^*, no matter what A is.
- Example: Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. If $A = \{\mathtt{good}, \mathtt{bad}\}$ and $B = \{\mathtt{boy}, \mathtt{girl}\}$, then
  - $A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\}$,
  - $A \circ B = \{\mathtt{goodboy}, \mathtt{goodgirl}, \mathtt{badboy}, \mathtt{badgirl}\}$,
  - $A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$.

Closure Under Operations

- Example: Let $\mathbb{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers.
  - When we say that $\mathbb{N}$ is closed under multiplication we mean that, for any x and y in $\mathbb{N}$, the product $x \times y$ is also in $\mathbb{N}$.
  - $\mathbb{N}$ is not closed under division, as 1 and 2 are in $\mathbb{N}$ but $1/2$ is not.
- A collection of objects is **closed under some operation** if applying that operation to members of the collection returns an object still in the collection.
- Our goal is to show that the collection of regular languages is closed under all three regular operations.

Closure Under Union

Theorem (Closure Under Union)

The class of regular languages is closed under the union operation, i.e., if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

- Since A_1 and A_2 are regular, we know that some finite automaton M_1 recognizes A_1 and some finite automaton M_2 recognizes A_2. To prove that $A_1 \cup A_2$ is regular, we construct a finite automaton M that recognizes $A_1 \cup A_2$. M works by simulating simultaneously both M_1 and M_2 and accepting if either of the simulations accept. We need to remember the state that each machine would be in if it had read up to this point in the input.
  - Since, we need to remember a pair of states, if M_1 has k_1 states and M_2 has k_2 states, M has $k_1 \times k_2$ states.
  - The transitions of M go from pair to pair, updating the current state for both M_1 and M_2.
  - The accept states of M are those pairs wherein either M_1 or M_2 is in an accept state.

Formal Proof of Closure Under Union

- Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$:
  1. $Q = \{(r_1, r_2) : r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$. This set is the Cartesian product of sets Q_1 and Q_2, written $Q_1 \times Q_2$.
  2. The alphabet Σ is the same as in M_1 and M_2. For simplicity, we assume that both M_1 and M_2 have the same input alphabet Σ. If they have different alphabets, Σ_1 and Σ_2, we let $\Sigma = \Sigma_1 \cup \Sigma_2$.
  3. The transition function δ is defined as follows: For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
  4. q_0 is the pair (q_1, q_2).
  5. F is the set of pairs in which either member is an accept state of M_1 or M_2, i.e., $F = \{(r_1, r_2) : r_1 \in F_1 \text{ or } r_2 \in F_2\} = (F_l \times Q_2) \cup (Q_1 \times F_2)$.

  The construction is fairly simple, and, thus, its correctness is evident from the strategy yielding the construction.
- More complicated constructions often require additional discussion to prove correctness.

Closure Under Concatenation and Nondeterminism

- Since the union of two regular languages is regular, the class of regular languages is closed under union.

Theorem

The class of regular languages is closed under the concatenation operation, i.e., if A_1 and A_2 are regular languages, then so is $A_1 \circ A_2$.

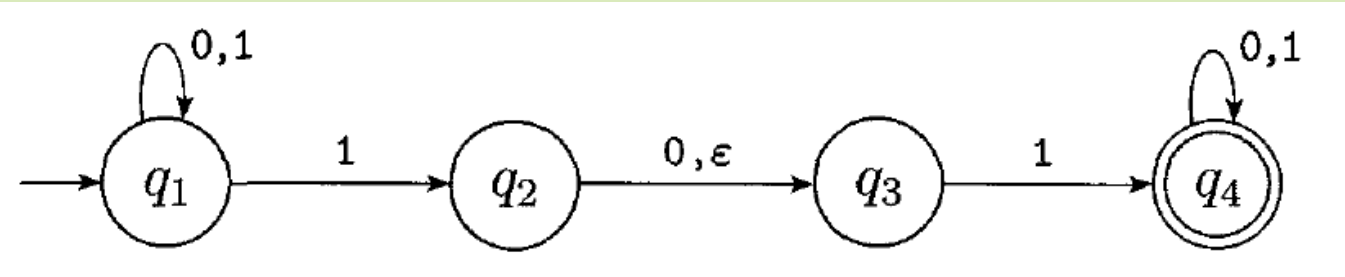
- To prove this theorem we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2.
- We need to construct an automaton M that accepts if its input can be broken into two pieces, where M_1 accepts the first piece and M_2 accepts the second piece.
- The problem is that M does not know where to break its input (i.e., where the first part ends and the second begins).
- To solve this problem we introduce a new technique called nondeterminism.

Subsection 2

Nondeterminism

Nondeterminism

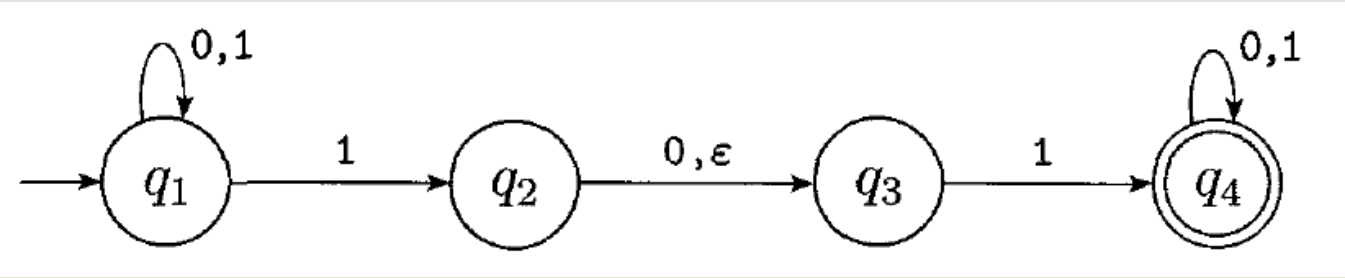
- In finite automata, every step of a computation follows in a unique way from the preceding step.
- When the machine is in a given state and reads the next input symbol, we know what the next state will be - it is determined.
- We call this **deterministic computation**.
- In a **nondeterministic machine**, several choices may exist for the next state at any point.



  - In a Nondeterministic Finite Automaton (NFA), a state may have zero, one, or many exiting arrows for each alphabet symbol.
  - An NFA may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.

Operation of a Nondeterministic Finite Automaton

- Consider again the NFA N_1:

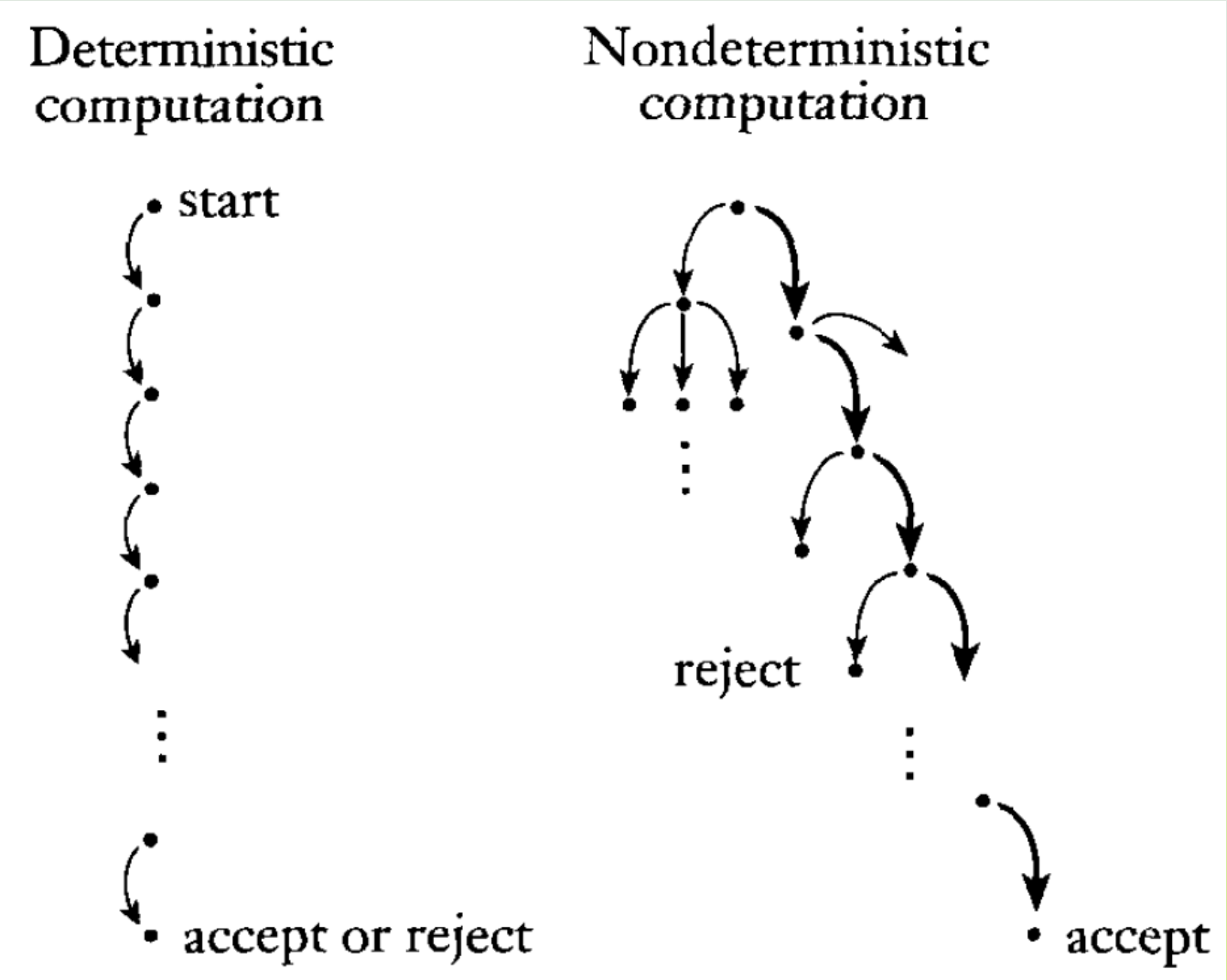


- Say that we are in state q_1 and the next input symbol is a 1.
- The machine splits into multiple copies of itself and follows all the possibilities in parallel.
- If there are subsequent choices, the machine splits again.
- If the next input symbol does not appear on any of the arrows exiting the state occupied by a copy of the machine, that copy dies.
- Finally, if any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.
- In a state with an ε on an exiting arrow, without reading any input, the machine splits following an ε-arrow or staying at current state.

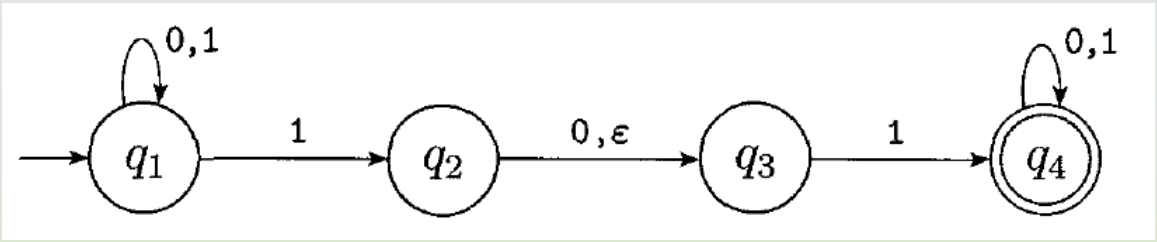
Nondeterminism and Parallelism

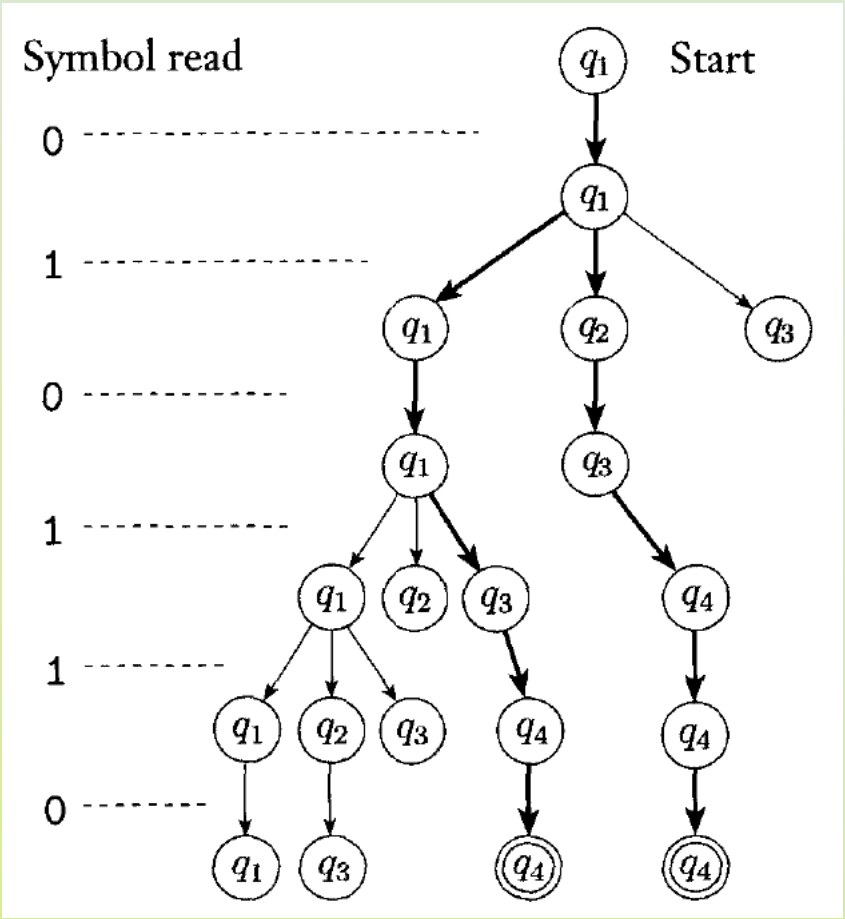
- Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent "processes" or "threads" can be running concurrently.
  - The NFA splitting to follow several choices corresponds to a process "forking" into several children, each proceeding separately.
  - If at least one of these processes accepts, then the entire computation accepts.
- Another way to think of a nondeterministic computation is as a tree of possibilities:
  - The root of the tree corresponds to the start of the computation.
  - Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices.
  - The machine accepts if at least one of the computation branches ends in an accept state.

Deterministic versus Nondeterministic Computation

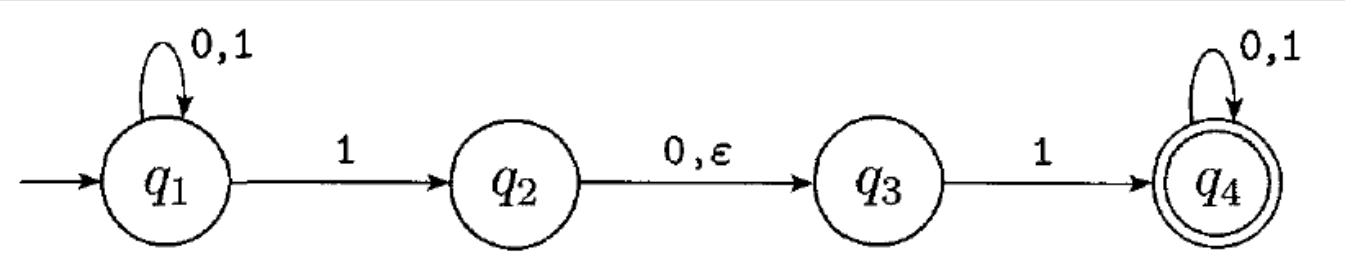


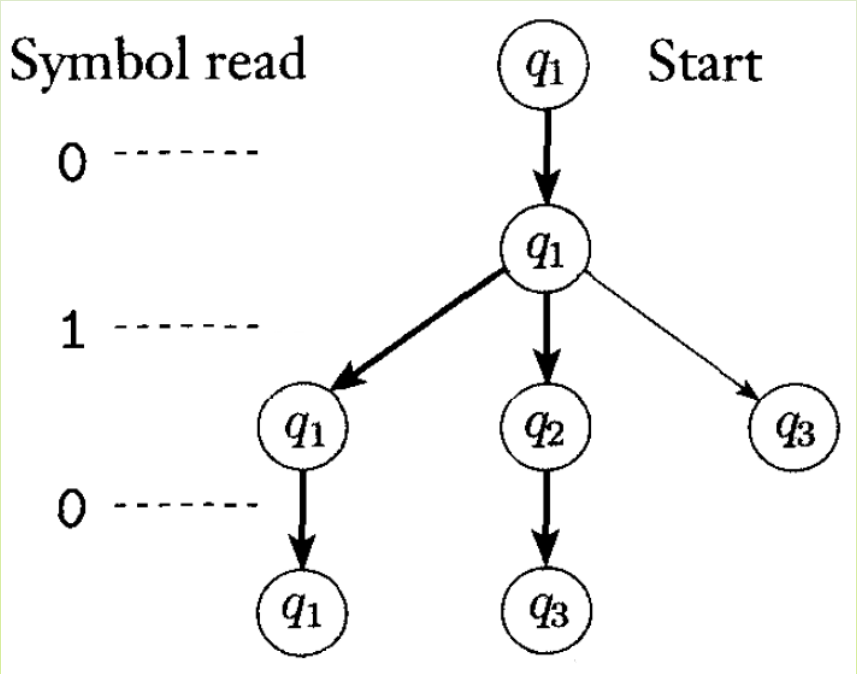
Sample Run of N_1 on Input 010110





Sample Run of N_1 on Input 010

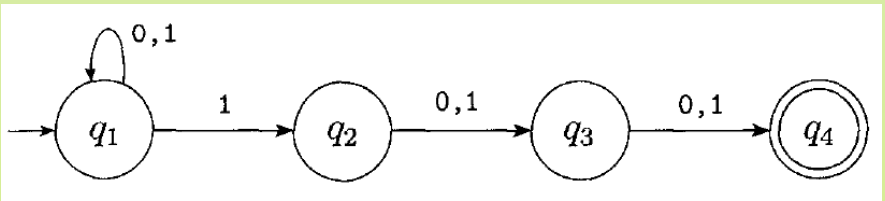




In fact N_1 accepts all strings that contain either 101 or 11 as a substring.

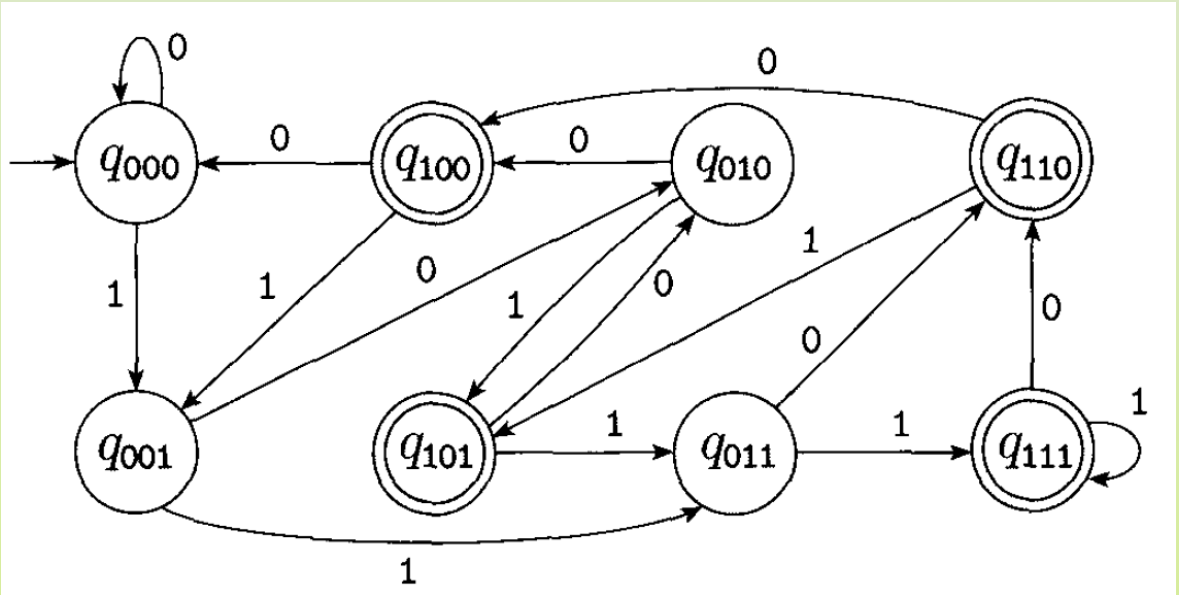
Nondeterminism: Advantages and Example

- Nondeterministic finite automata are useful in several respects.
  - Every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs.
  - An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand.
  - Nondeterminism in finite automata is a good introduction to nondeterminism in more powerful computational models because finite automata are especially easy to understand.
- Example: Let A be the language consisting of all strings over $\{0, 1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A: N_2 stays in the start state q_1, until it "guesses" that it is three places from the end. At that point, if the input symbol is a 1, it branches to state q_2 and uses q_3 and q_4 to "check" whether the guess was correct.



A DFA Recognizing A

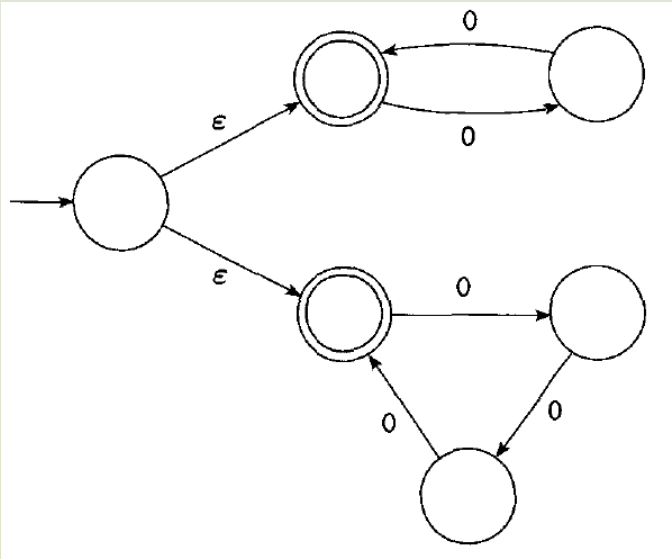
- Every NFA can be converted into an equivalent DFA, but sometimes that DFA may have many more states.
- The smallest DFA for A, the language consisting of all strings over $\{0, 1\}$ containing a 1 in the third position from the end, contains eight states.



- Understanding the functioning of the NFA is much easier.

An NFA with a Unary Alphabet

- Consider the following NFA N_3 that has an input alphabet $\{0\}$ consisting of a single symbol.

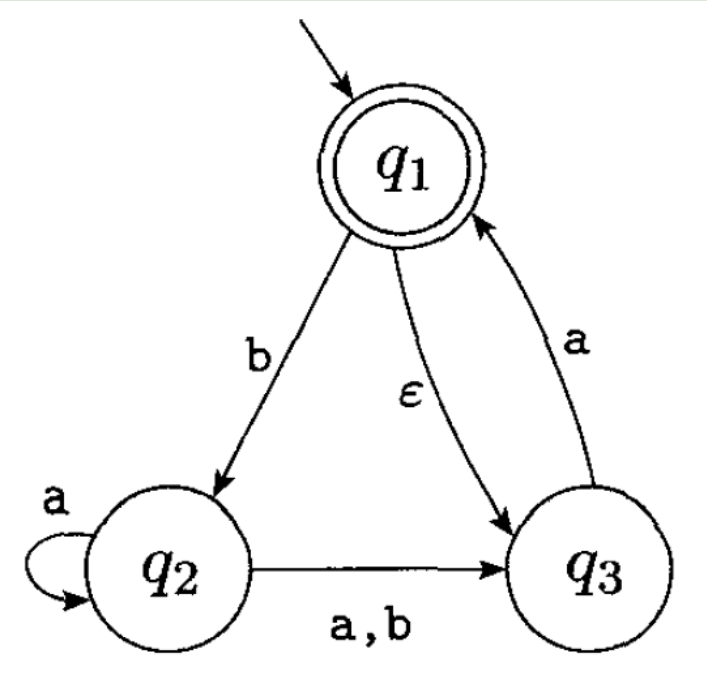


An alphabet containing only one symbol is called a **unary alphabet**. This machine demonstrates the convenience of having ε arrows.

- It accepts all strings of the form 0^k where k is a multiple of 2 or 3. E.g., N_3 accepts $\varepsilon, 00, 000, 0000$ and 000000, but not 0 or 00000.
- The machine operates by
  - initially guessing whether to test for a multiple of 2 or a multiple of 3 by branching into either the top loop or the bottom loop;
  - then checking whether its guess was correct.

The NFA N_4

- NFA N_4 is given in the following figure:



- It accepts the strings ε, a, baba and baa.
- It does not accept the strings b, bb and babba.

Features of Nondeterministic Finite Automata

- Like deterministic finite automata, nondeterministic ones have states, an input alphabet, a transition function, a start state, and a collection of accept states.
- They differ from deterministic ones in the type of transition function.
  - In a DFA the transition function takes a state and an input symbol and produces the next state.
  - In an NFA the transition function takes a state and an input symbol or the empty string and produces the set of possible next states.
- For any set Q we write $\mathcal{P}(Q)$ to be the collection of all subsets of Q, called the **power set** of Q.
- For any alphabet Σ we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$.
- The formal description of the type of the transition function in an NFA is written $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$.

Nondeterministic Finite Automata

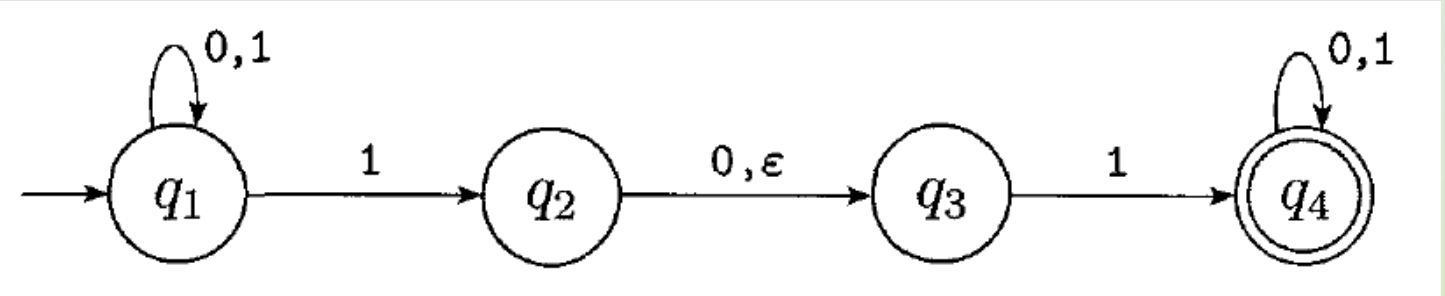
Definition (Nondeterministic Finite Automaton)

A **nondeterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set of **states**,
2. Σ is a finite **alphabet**,
3. $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the **transition function**,
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the set of **accept states**.

Example of a Nondeterministic Finite Automaton

- Example: Recall the NFA N_1:



1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

4. q_1 is the start state, and
5. $F = \{q_4\}$.

Computation of an NFA

- The formal definition of computation for an NFA is similar to that for a DFA.
- Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. Then we say that N **accepts** w if w can be written in the form $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ε, and a sequence of states $r_0, r_1, \ldots, r_m$ exists in Q, satisfying the three conditions:
  1. $r_0 = q_0$;
  2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m-1$, and
  3. $r_m \in F$.
- Condition 1 says that the machine starts out in the start state.
  Condition 2 says that state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1}.
  Condition 3 says that the machine accepts its input if the last state is an accept state.

Equivalence of NFAs and DFAs: Proof Idea

- Deterministic and nondeterministic finite automata recognize the same class of languages.
  - This is surprising because NFAs appear to have more power than DFAs.
  - It is also useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.
- We call two machines **equivalent** if they recognize the same language.

Theorem (Equivalence of DFAs and NFAs)

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

- Idea of Proof: The idea is to convert the NFA into an equivalent DFA that simulates the NFA. For the simulation, we need to keep track of the set of states in the parallel computation paths of the NFA. If the NFA has k states, it has 2^k subsets of states. Since the DFA must remember the current set of states, it will have 2^k states. We also need to find the start and accept states and the transition function.

Proof of the Equivalence I

- Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A. We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A. Before doing the full construction, we first consider the easier case wherein N has no ε arrows. Later we take the ε arrows into account.
  1. $Q' = \mathcal{P}(Q)$. Every state of M is a set of states of N.
  2. For $R \in Q'$ and $a \in \Sigma$ let $\delta'(R, a) = \{q \in Q : q \in \delta(r, a), \text{ for some } r \in R\}$. If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets. i.e., $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$.
  3. $q_0' = \{q_0\}$. M starts in the state corresponding to the collection containing just the start state of N.
  4. $F' = \{R \in Q' : R \text{ contains an accept state of } N\}$. The machine M accepts if one of the possible states that N could be in at this point is an accept state.

Proof of the Equivalence II

- Now we need to consider the ε arrows.
- For any state R of M we define $E(R)$ to be the collection of states that can be reached from R by going only along ε arrows, including the members of R themselves. Formally, for $R \subseteq Q$, let

$$E(R) = \{q : q \text{ can be reached from } R \text{ by traveling along 0 or more } \varepsilon \text{ arrows}\}.$$

- We modify the transition function of M to place additional fingers on all states that can be reached by going along ε arrows after every step. Replacing $\delta(r, a)$ by $E(\delta(r, a))$ achieves this effect. Thus,

$$\delta'(R, a) = \{q \in Q : q \in E(\delta(r, a)), \text{ for some } r \in R\}.$$

- We also modify the start state of M to move the fingers initially to all possible states that can be reached from the start state of N along the ε arrows. Changing q_0' to be $E(\{q_0\})$ achieves this effect. This completes the construction of the DFA M that simulates N.

Characterization of Regular Languages Using NFAs

- The preceding theorem states that every NFA can be converted into an equivalent DFA. Thus, nondeterministic finite automata give an alternative way of characterizing the regular languages:

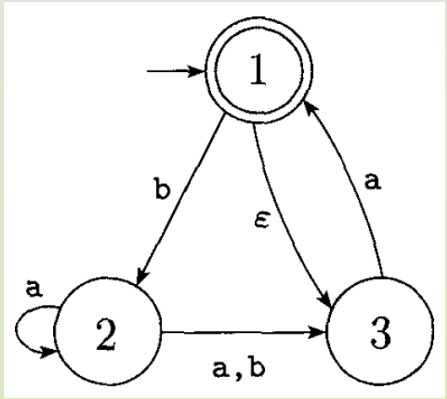
Corollary (Characterization of Regular Languages Using NFAs)

A language is regular if and only if some nondeterministic finite automaton recognizes it.

- One direction of the "if and only if" condition states that a language is regular if some NFA recognizes it. The theorem shows that any NFA can be converted into an equivalent DFA. Consequently, if an NFA recognizes some language, so does some DFA, and hence the language is regular.
- The other direction of the "if and only if" condition states that if a language is regular, some NFA must be recognizing it. Obviously, this condition is true because a regular language has a DFA recognizing it and any DFA is also an NFA.

Illustration of the Proof

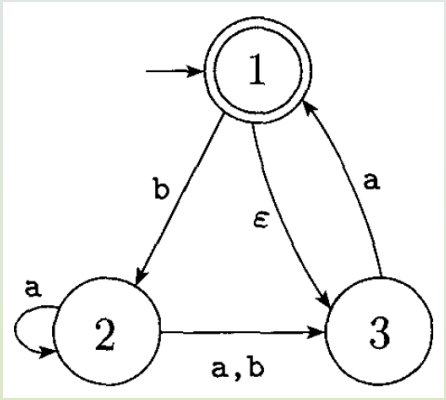
- Consider the machine N_4:



The formal description is $N_4 = (Q, \{\mathtt{a}, \mathtt{b}\}, \delta, 1, \{1\})$. We construct a DFA D that is equivalent to N_4:

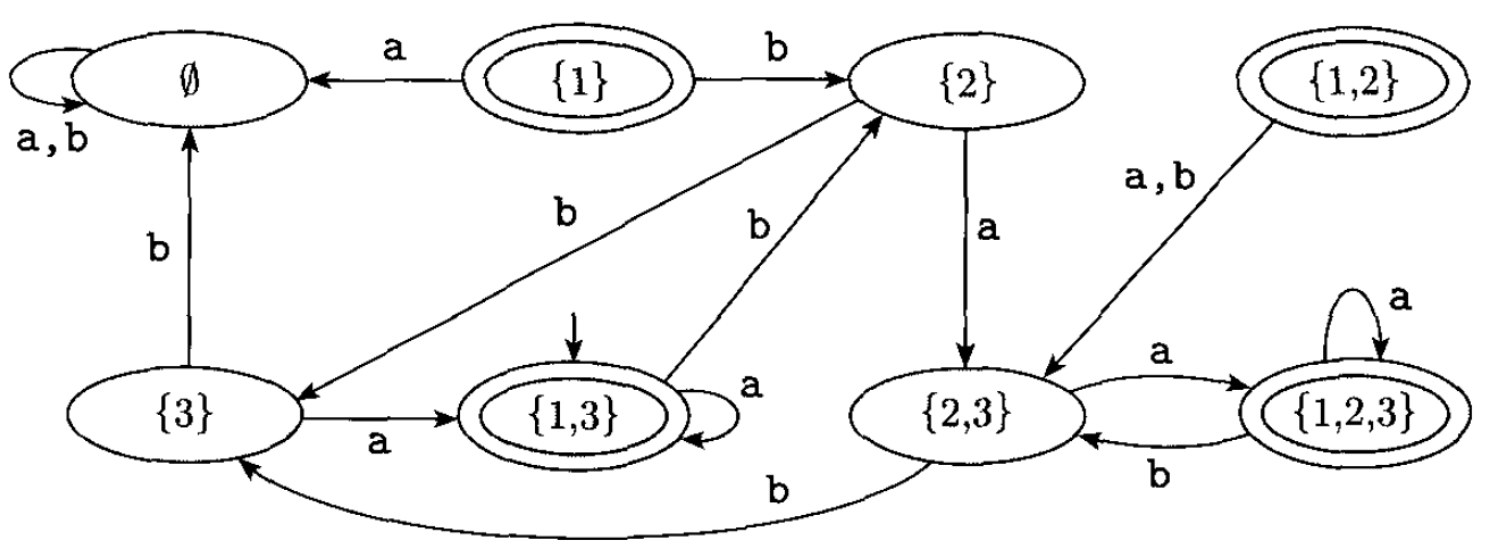
- We first determine D's states. Since N_4 has three states, D has eight states $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.
- The start state of D is $E(\{1\})$, the set of states that are reachable from 1 by traveling along ε arrows, plus 1 itself. An ε arrow goes from 1 to 3, so $E(\{1\}) = \{1,3\}$.
- The new accept states are those containing N_4's accept state, i.e., $\{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$.

Illustration of the Proof (Transitions)

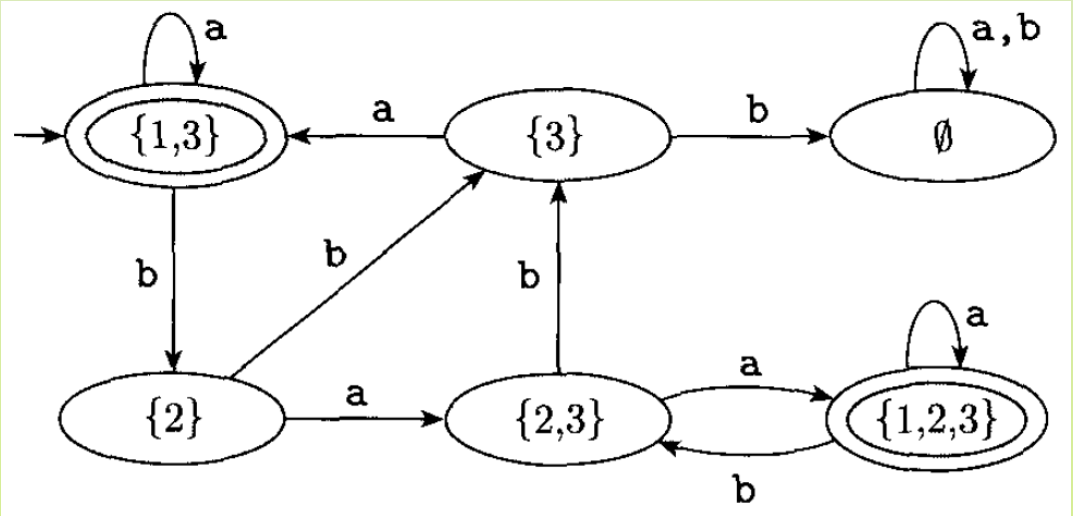


- Finally, we determine D's transition function. Each of D's states goes to one place on input a and one place on input b.
- E.g., in D, state $\{2\}$ goes to $\{2,3\}$ on input a, because in N_4, state 2 goes to both 2 and 3 on input a and we cannot go farther from 2 or 3 along ε arrows.
- State $\{2\}$ goes to state $\{3\}$ on input b, because in N_4, state 2 goes only to state 3 on input b and we cannot go farther along ε arrows.
- State $\{1\}$ goes to $\emptyset$ on a. It goes to $\{2\}$ on b.
- State $\{3\}$ goes to $\{1,3\}$ on a, because in N_4, state 3 goes to 1 on a and 1 in turn goes to 3 with an ε arrow. State $\{3\}$ on b goes to $\emptyset$.
- State $\{1,2\}$ on a goes to $\{2,3\}$ because 1 points at no states with a arrows and 2 points at both 2 and 3 with a arrows and neither points anywhere with ε arrows. State $\{1,2\}$ on b goes to $\{2,3\}$.

Illustration of the Proof (Final Diagrams)



We can simplify the machine since no arrows point at $\{1\}$ and $\{1, 2\}$:



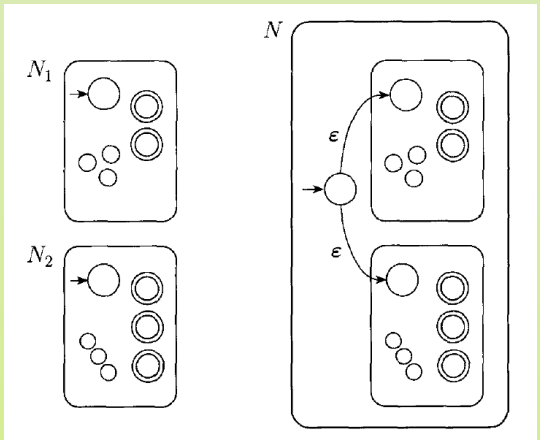
Idea of Proof of Closure Under Union Using NFAs

- We proved closure under union by simulating deterministically both machines simultaneously via a Cartesian product construction.
- We give a new proof using the technique of nondeterminism.

Theorem

The class of regular languages is closed under the union operation.

- We have regular languages A_1 and A_2 and want to prove that $A_1 \cup A_2$ is regular. We take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into one new NFA N. Machine N must accept its input if either N_1 or N_2 accepts this input.



The new machine has a new start state that branches to the start states of the old machines with ε arrows. Thus, the new machine nondeterministically guesses which of the two machines accepts the input. If one of them accepts the input, N also accepts.

Proof of Closure Under Union Using NFAs

- Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2. Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$:
  1. $Q = \{q_0\} \cup Q_1 \cup Q_2$. The states of N are all the states of N_1 and N_2, with the addition of a new start state q_0.
  2. The state q_0 is the start state of N.
  3. The accept states $F = F_1 \cup F_2$. The accept states of N are all the accept states of N_1 and N_2. So N accepts if either N_1 accepts or N_2 accepts.
  4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

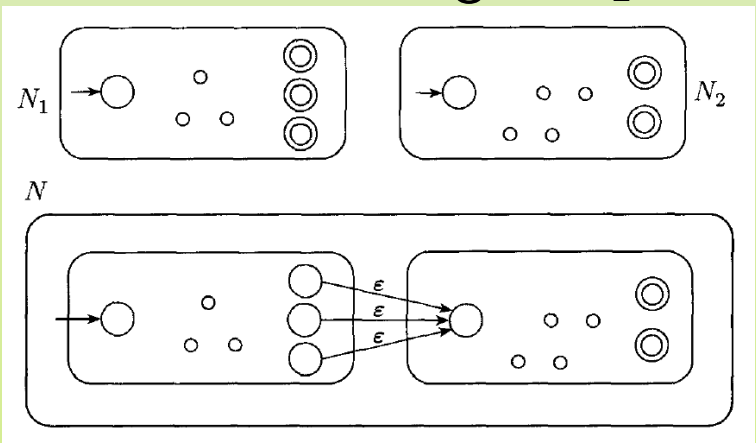
$$\delta(q,a) = \begin{cases} \delta_1(q,a), & \text{if } q \in Q_1 \\ \delta_2(q,a), & \text{if } q \in Q_2 \\ \{q_1, q_2\}, & \text{if } q = q_0 \text{ and } a = \varepsilon \\ \emptyset, & \text{if } q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

Idea of Proof of Closure Under Concatenation

Theorem (Closure Under Concatenation)

The class of regular languages is closed under the concatenation operation.

- Given regular languages A_1 and A_2, and want to prove that $A_1 \circ A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into a new NFA N. Assign N's start state to be the start state of N_1. The accept states of N_1 have additional ε arrows that nondeterministically allow branching to N_2 whenever N_1 is in an accept state, i.e., when it has found an initial piece of the input that constitutes a string in A_1.



The accept states of N are the accept states of N_2 only. Therefore it accepts when the input can be split into two parts, the first accepted by N_1 and the second by N_2.

Proof of Closure Under Concatenation

- Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2. Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$:
  1. $Q = Q_1 \cup Q_2$. The states of N are all the states of N_1 and N_2.
  2. The state q_1, is the same as the start state of N_1.
  3. The accept states F_2 are the same as the accept states of N_2.
  4. Define δ so that for any $q \in Q$ and any $a \in \Sigma$,

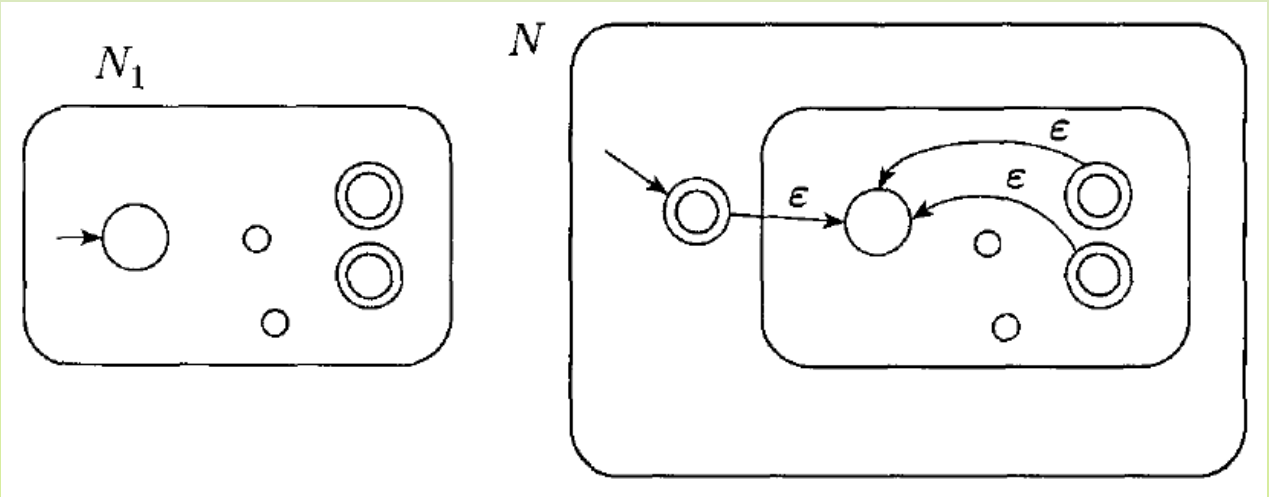
$$\delta(q,a) = \begin{cases} \delta_1(q,a), & \text{if } q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a), & \text{if } q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\}, & \text{if } q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q,a), & \text{if } q \in Q_2 \end{cases}.$$

Idea of Proof of Closure Under Star

Theorem (Closure Under Star)

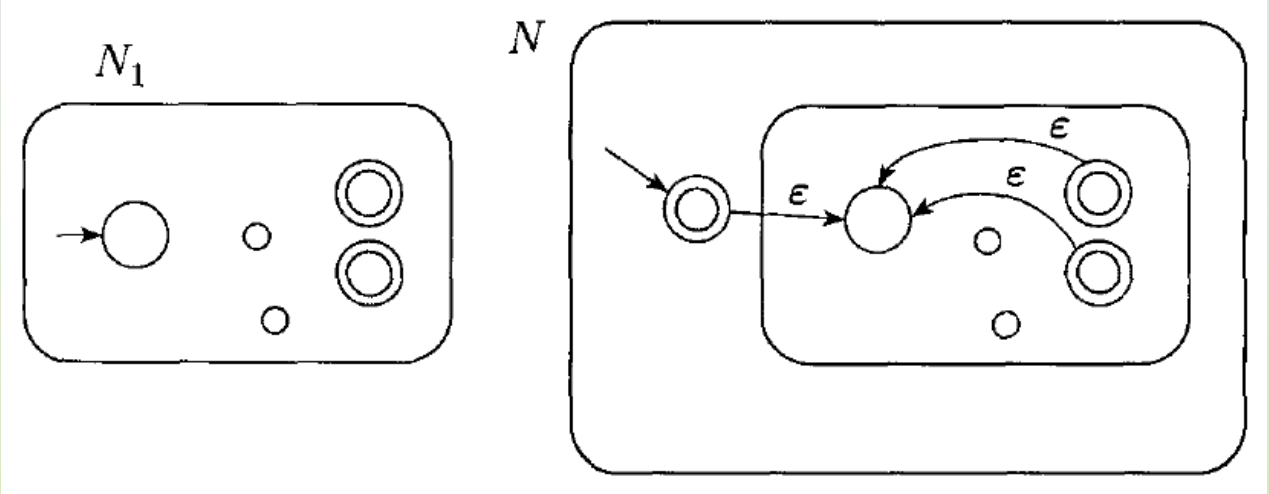
The class of regular languages is closed under the star operation.

- We have a regular language A_1 and want to prove that A_1^* also is regular. We take an NFA N_1 for A_1 and modify it to recognize A_1^*. The resulting NFA N will accept its input whenever it can be broken into several pieces and N_1 accepts each piece.



We can construct N like N_1 with additional ε arrows returning to the start state from the accept states.

Idea of Proof of Closure Under Star



- This way, when processing gets to the end of a piece that N_1 accepts, the machine N has the option of jumping back to the start state to try to read in another piece that N_1 accepts.
- In addition we must modify N so that it accepts ε, which always is a member of A_1^*. One (slightly bad) idea is simply to add the start state to the set of accept states. This approach certainly adds ε to the recognized language, but it may also add other, undesired strings. To fix this, we add a new start state, which also is an accept state, and which has an ε arrow to the old start state.

Proof of Closure Under Star

- Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1. Construct $N = (Q, A, \delta, q_0, F)$ to recognize A_1^*.
  1. $Q = \{q_0\} \cup Q_1$. The states of N are the states of N_1 plus a new start state.
  2. The state q_0 is the new start state.
  3. $F = \{q_0\} \cup F_1$. The accept states are the old accept states plus the new start state.
  4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & \text{if } q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a), & \text{if } q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\}, & \text{if } q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\}, & \text{if } q = q_0 \text{ and } a = \varepsilon \\ \emptyset, & \text{if } q = q_0 \text{ and } a \neq \varepsilon \end{cases}.$$

Subsection 3

Regular Expressions

Introducing Regular Expressions

- In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as $(5 + 3) \times 4$.
- In "language calculus" we can use the regular operations to build up expressions describing languages, called **regular expressions**.
- Example: $(0 \cup 1)0^*$.
- The value of a regular expression is a language.
- In the example the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s:
  - The symbols 0 and 1 are shorthand for the sets $\{0\}$ and $\{1\}$. So $(0 \cup 1)$ means $(\{0\} \cup \{1\})$. The value of this part is the language $\{0, 1\}$.
  - The part 0^* means $\{0\}^*$. Its value is the language consisting of all strings containing any number of 0s.
  - The regular expressions $(0 \cup 1)0^*$ actually is shorthand for $(0 \cup 1) \circ 0^*$. The concatenation attaches the strings from the two parts to obtain the value of the entire expression.
- Regular expressions have an important role in computer science applications, especially for describing text patterns.

Examples and Order of Regular Operations

- Example: Another regular expression is $(0 \cup 1)^*$.
  It starts with the language $(0 \cup 1)$ and applies the $*$ operation. The value of this expression is the language consisting of all possible strings of 0s and 1s. If $\Sigma = \{0, 1\}$, we can write Σ as shorthand for the regular expression $(0 \cup 1)$ to get $(0 \cup 1)^* = \Sigma^*$.
- More generally, if Σ is any alphabet:
  - The regular expression Σ describes the language consisting of all strings of length 1 over this alphabet.
  - Σ^* describes the language consisting of all strings over that alphabet.
  - Σ^*1 is the language that contains all strings that end in a 1.
  - The language $(0\Sigma^*) \cup (\Sigma^*1)$ consists of all strings that either start with a 0 or end with a 1.
- In arithmetic, we say that $\times$ has precedence over $+$ to mean that, when there is a choice, we do the $\times$ operation first.
- In regular expressions the order is $*, \circ, \cup$, unless parentheses are used to change the usual order.

Regular Expressions

Definition (Regular Expression)

We say that R is a **regular expression** if R is:

1. a, for some a in the alphabet Σ;
2. ε;
3. $\emptyset$;
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions;
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions;
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

- Be careful to distinguish the languages represented by ε and $\emptyset$.

Remarks and Notation

- In defining a notion in terms of itself, one commits the fallacy of a **circular definition**.
- Note that in the definition of regular expression, R_1 and R_2 are always smaller than R. Thus, regular expressions are defined in terms of smaller regular expressions avoiding circularity.
- A definition of this type is called an **inductive definition**.
- Parentheses in an expression may be omitted, in which case evaluation is done in the precedence order: star, then concatenation, then union.
- For convenience, we let R^+ be shorthand for RR^*. So the language R^+ has all strings that are 1 or more concatenations of strings from R.
- So $R^+ \cup \varepsilon = R^*$.
- R^k is shorthand for the concatenation of k R's with each other.
- When we want to distinguish between a regular expression R and the language that it describes, we write $L(R)$ to be the language of R.

Examples of Regular Expressions and Their Languages

- Consider the alphabet $\Sigma = \{0, 1\}$.
  - $0^*10^* = \{w : w \text{ contains a single } 1\}$.
  - $\Sigma^*1\Sigma^* = \{w : w \text{ has at least one } 1\}$.
  - $\Sigma^*001\Sigma^* = \{w : w \text{ contains the string } 001 \text{ as a substring}\}$.
  - $1^*(01^+)^* = \{w : \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$.
  - $(\Sigma\Sigma)^* = \{w : w \text{ is a string of even length}\}$.
  - $(\Sigma\Sigma\Sigma)^* = \{w : \text{the length of } w \text{ is a multiple of three}\}$.
  - $01 \cup 10 = \{01, 10\}$.
  - $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w : w \text{ starts and ends with the same symbol}\}$.
  - $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
    The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
  - $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
  - $1^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
  - $\emptyset^* = \{\varepsilon\}$.
    If the language is empty, the star operation can only put together 0 strings, giving only the empty string.

Simple Identities Involving Regular Expressions

- If we let R be any regular expression, we have the following identities:
  - $R \cup \emptyset = R$.
    Adding the empty language to any other language will not change it.
  - $R \circ \varepsilon = R$.
    Joining the empty string to any string will not change it.
    However, exchanging $\emptyset$ and ε in the preceding identities may cause the equalities to fail.
  - $R \cup \varepsilon$ may not equal R.
    For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \cup \varepsilon) = \{0, \varepsilon\}$.
  - $R \circ \emptyset$ may not equal R.
    For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \circ \emptyset) = \emptyset$.

Equivalence of Regular Languages and Regular Expressions

- Even though finite automata and regular expressions appear to be different, they are equivalent in their descriptive power.

  Any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa.
- Recall that a regular language is one that is recognized by some finite automaton.

Theorem (Equivalence of Regular Languages and Regular Expressions)

A language is regular if and only if some regular expression describes it.

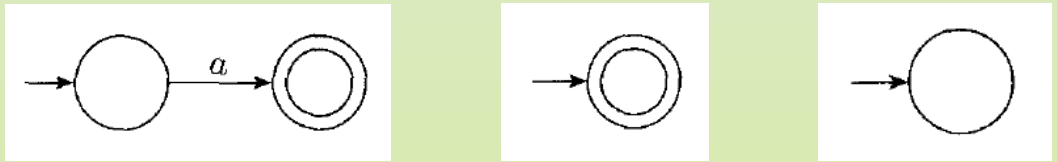
- This theorem has two directions which are proven as separate lemmas.

Regular Expressions to Automata

Lemma

If a language is described by a regular expression, then it is regular.

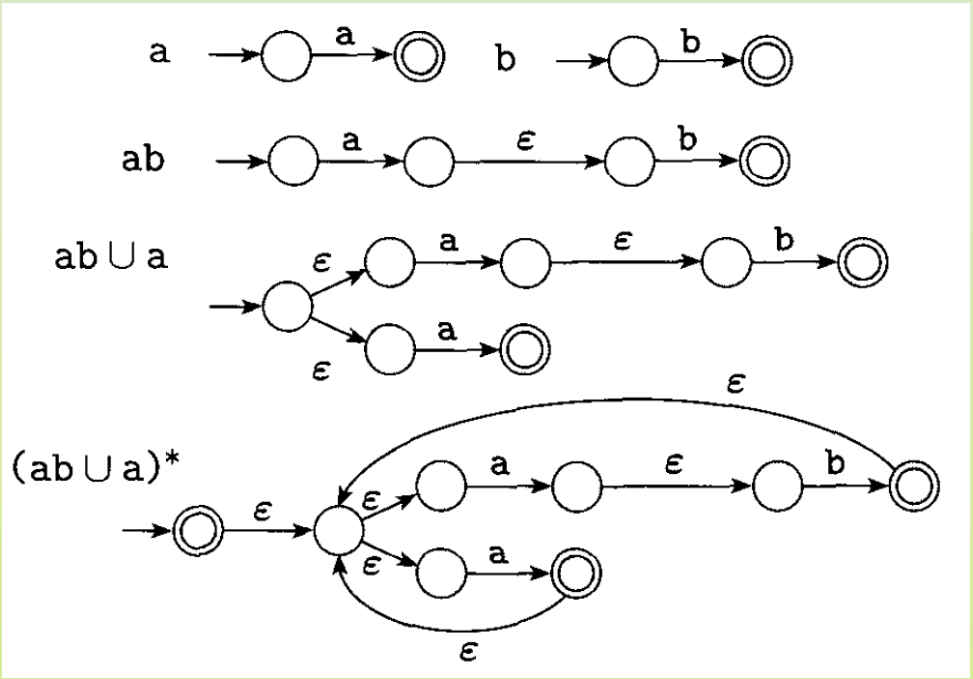
- Let R be a regular expression describing some language A. We show how to convert R into an NFA recognizing A. We consider the six cases in the formal definition of regular expressions.
  1. $R = a$, for some $a \in \Sigma$. Then $L(R) = \{a\}$, recognized by the NFA:



     Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and $\delta(r, b) = \emptyset$, for $r \neq q_1$ or $b \neq a$.
  2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$. The second NFA recognizes $L(R)$. Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$, for any r and b.
  3. $R = \emptyset$. Then $L(R) = \emptyset$. The last NFA recognizes $L(R)$. Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$, for any r and b.

Regular Expressions to Automata (Cont'd)

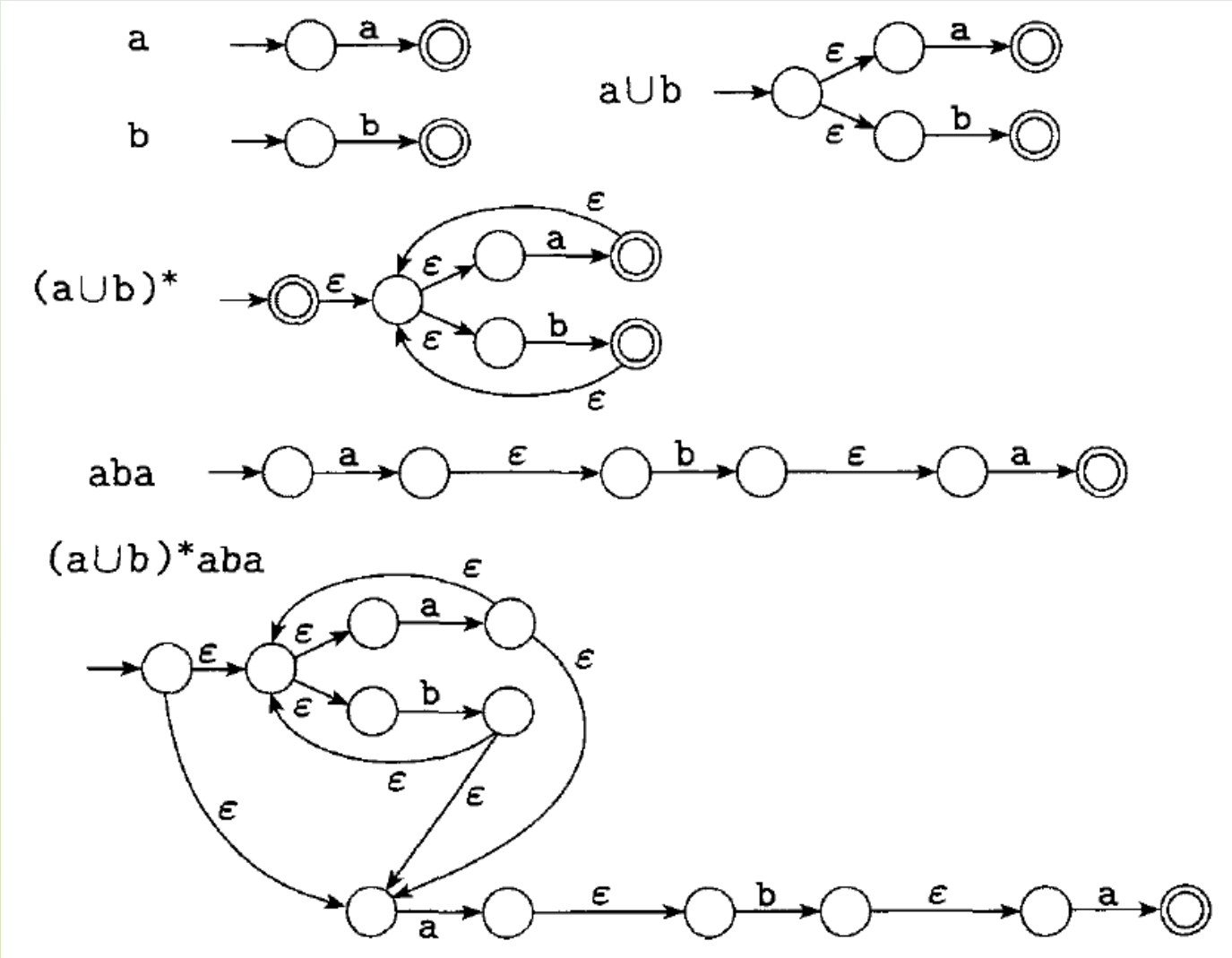
- The last three cases in the formal definition of regular expressions:
  4. $R = R_1 \cup R_2$.
  5. $R = R_1 \circ R_2$.
  6. $R = R_1^*$. For 4,5 and 6 we use the constructions showing the closure of the class of regular languages under these operations.
- Example: Convert the regular expression $(ab \cup a)^*$ to an NFA:



We do not generally get the NFA with the fewest states.

Another Example

- Convert the regular expression $(a \cup b)^* aba$ to an NFA.



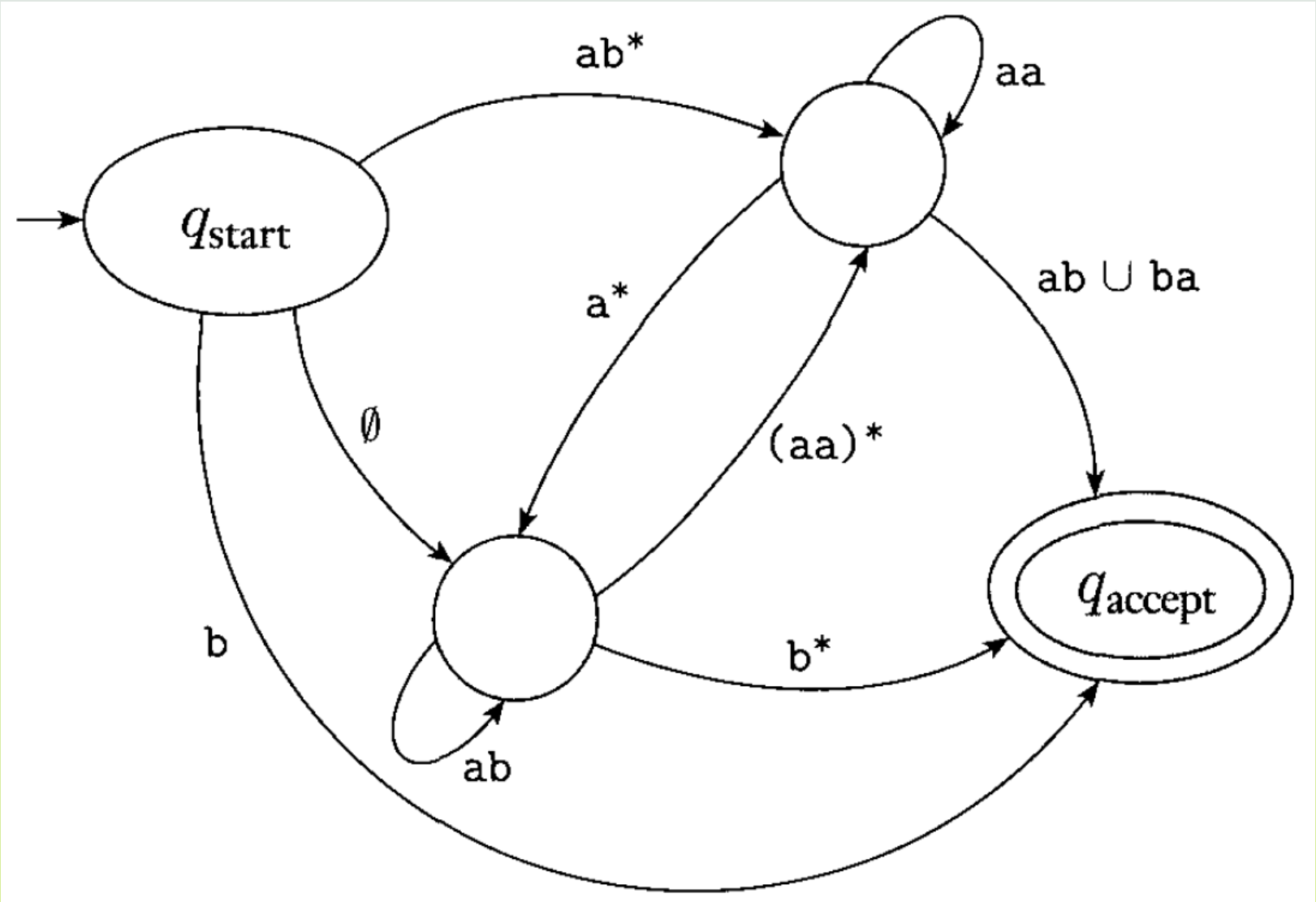
DFAs to Regular Expressions through GNFAs

Lemma

If a language is regular, then it is described by a regular expression.

- We describe a procedure for converting DFAs into equivalent regular expressions. We use a new type of finite automaton called a **generalized nondeterministic finite automaton**, **GNFA**.
  - First we show how to convert DFAs into GNFAs.
  - Then convert GNFAs into regular expressions.
- GNFAs are NFAs wherein the transition arrows may have any regular expressions as labels, instead of only members of Σ or ε.
  - The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time. It moves along transitions by reading blocks of symbols as described by the regular expressions.
  - A GNFA is nondeterministic and so may have several different ways to process the same input string.
  - It accepts if it can be in an accept state at the end of the input.

A Generalized Nondeterministic Finite Automaton

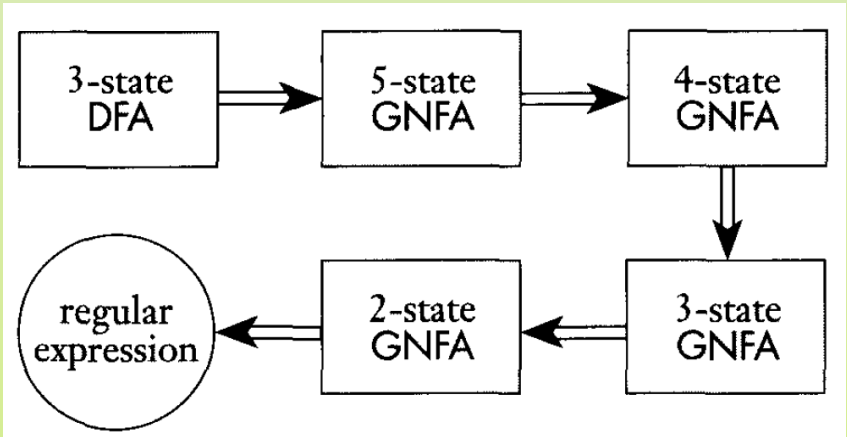


Special Form of GNFAS

- For convenience we require that GNFAs always have a special form:
  - The start state has transition arrows going to every other state but no arrows coming in from any other state.
  - There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
  - Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.
- We can easily convert a DFA into a GNFA in the special form:
  - We simply add a new start state with an ε arrow to the old start state and a new accept state with ε arrows from the old accept states.
  - If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels.
  - Finally, we add arrows labeled $\emptyset$ between states that had no arrows. This does not change the language recognized since a transition labeled with $\emptyset$ can never be used.

Reducing the GNFA's Number of States

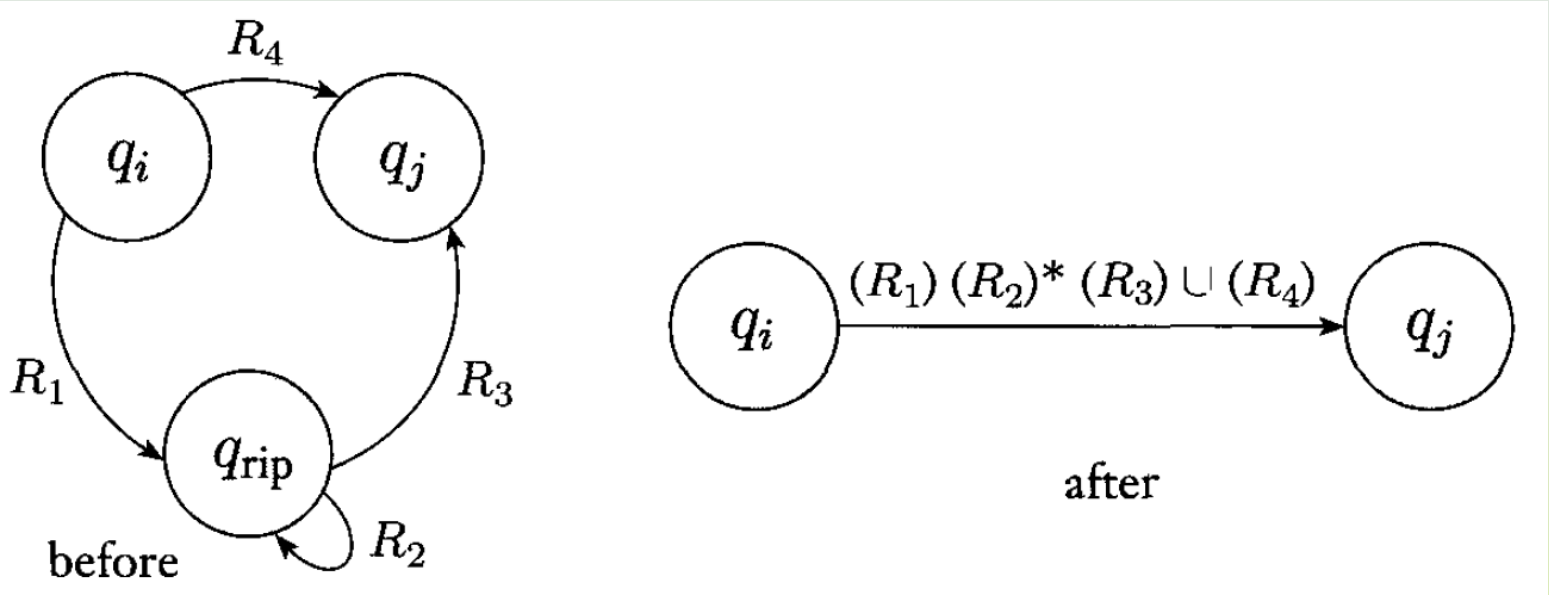
- Now we show how to convert a GNFA into a regular expression.
  - Say that the GNFA has k states.
  - Because a GNFA must have a start and an accept state and they must be different, we know that $k \geq 2$.
  - If $k > 2$, we construct an equivalent GNFA with $k - 1$ states.
    - This step can be repeated until the GNFA is reduced to two states.
  - If $k = 2$, the GNFA has a single arrow going from the start to the accept state.
    - The label of this arrow is the equivalent regular expression.
- Example: The stages in converting a DFA with three states to an equivalent regular expression are:



Choosing a State to Purge

- To construct an equivalent GNFA with one fewer state when $k > 2$, we select a state q_{rip}, remove it and repair the remainder so that the same language is still recognized.
- Any state will do, provided that it is not the start or accept state.
- After removing q_{rip} we repair the machine by altering the regular expressions that label each of the remaining arrows.
- The new labels compensate for the absence of q_{rip} by adding back the lost computations.
- The new label going from a state q_i to a state q_j is a regular expression that describes all strings that would take the machine from q_i to q_j either directly or via q_{rip}.

Purging the Chosen State



- If in the old machine q_i goes to q_{rip} with an arrow R_1, q_{rip} goes to itself with an arrow R_2, q_{rip} goes to q_j with an arrow labeled R_3, and q_i goes to q_j with an arrow R_4, then in the new machine the arrow from q_i to q_j gets the label $(R_1)(R_2)^*(R_3) \cup (R_4)$.
- We make this change for each arrow going from any state q_i to any state q_j, including the case where $q_i = q_j$.

Generalized Nondeterministic Finite Automata

- The transition function is $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to \mathcal{R}$, where $\mathcal{R}$ is the collection of all regular expressions over the alphabet Σ, and q_{start} and q_{accept} are the start and accept states.
- If $\delta(q_i, q_j) = R$, the arrow from state q_i to state q_j has the regular expression R as its label.

Definition (GNFA)

A **generalized nondeterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where:

1. Q is the finite set of **states**;
2. Σ is the **input alphabet**;
3. $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to \mathcal{R}$ is the **transition function**;
4. q_{start} is the **start state**;
5. q_{accept} is the **accept state**.

Acceptance Condition of a GNFA

- A GNFA $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ **accepts** a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots, q_k$ exists, such that
  1. $q_0 = q_{\text{start}}$ is the start state,
  2. $q_k = q_{\text{accept}}$ is the accept state, and
  3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Converting a GNFA to a Regular Expression

- Returning to the proof, we let M be the DFA for language A. Then we convert M to a GNFA G by adding a new start state and a new accept state and additional transition arrows as necessary.
- We use the procedure $\textsc{Convert}(G)$, which takes a GNFA and returns an equivalent regular expression.

$\textsc{Convert}(G)$

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.
3. If $k > 2$, select any $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, where $Q' = Q - \{q_{\text{rip}}\}$, and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$ let
$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$
for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute $\textsc{Convert}(G')$ and return this value.

Correctness of the Conversion

Claim (Correctness of CONVERT(G))

For any GNFA G, CONVERT(G) is equivalent to G.

- By induction on k, the number of states of the GNFA.
  - Basis: If G has only two states, it can have only a single arrow, which goes from the start state to the accept state. The regular expression label on this arrow describes all the strings that allow G to get to the accept state. Hence this expression is equivalent to G.
  - Induction step: Assume that the claim is true for $k-1$ states. We show that G and G' recognize the same language.
    - Suppose that G accepts an input w. Let q_{start}, q_1, q_2, ..., q_{accept} be an accepting computation of G. If no state is q_{rip}, clearly G' also accepts w. If q_{rip} does appear, removing each run of consecutive q_{rip} states forms an accepting computation for G'. The states q_i and q_j bracketing a run have a new regular expression on the arrow between them that describes all strings taking q_i to q_j via q_{rip} on G.

Correctness of the Conversion (Cont'd)

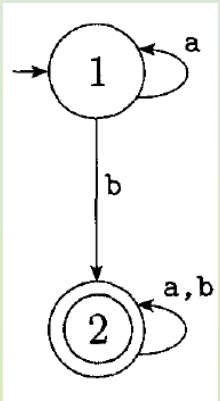
- We finish the Induction step:
  - Conversely, suppose that G' accepts an input w. Each arrow between any two states q_i and q_j in G' describes the collection of strings taking q_i to q_j in G either directly or via q_{rip}. Thus, G must also accept w.

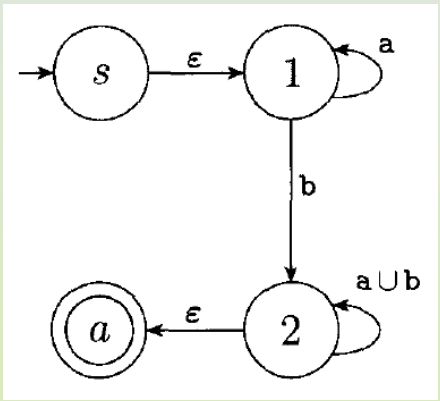
  Thus G and G' are equivalent.

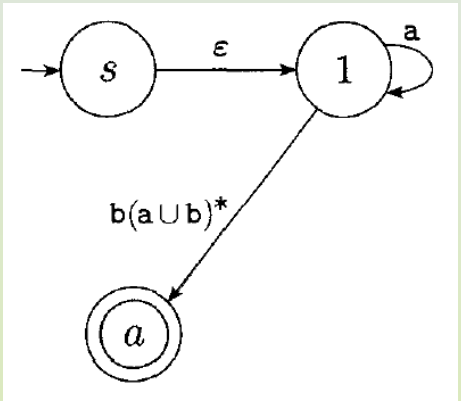
  The induction hypothesis states that when the algorithm calls itself recursively on input G', the result is a regular expression that is equivalent to G' because G' has $k-1$ states. Hence this regular expression is also equivalent to G, and the algorithm is correct.

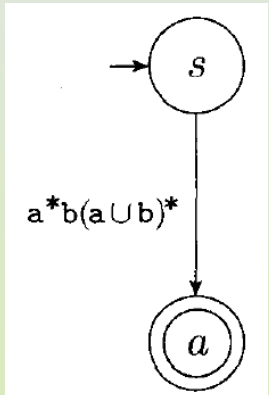
Illustrating the Conversion

- Consider the two-state DFA:



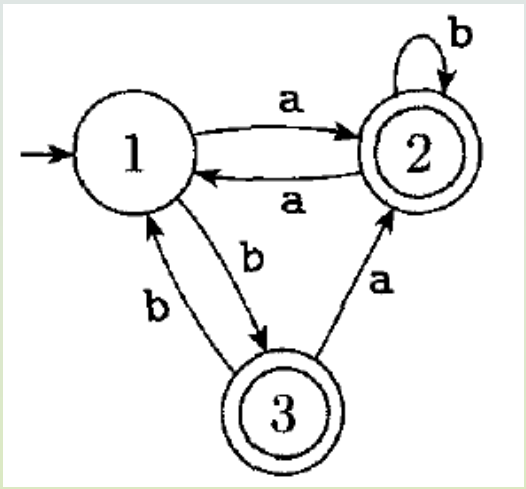




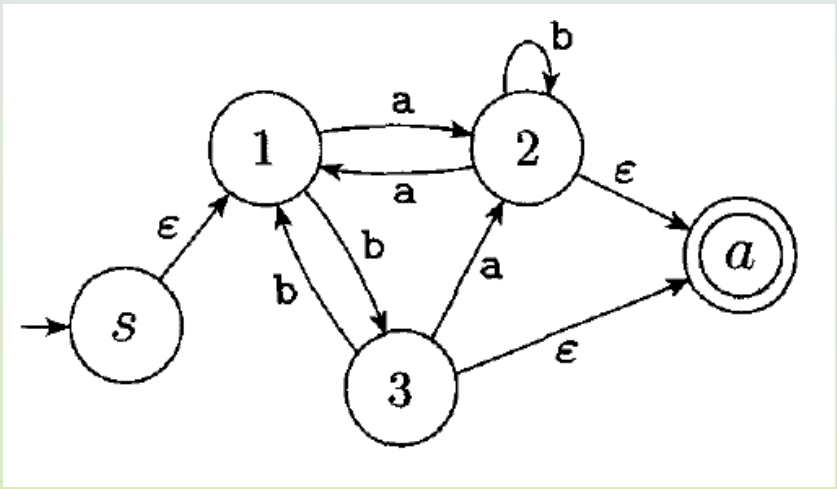


- We make a GNFA by adding a new start state and a new accept state. We did not draw the arrows labeled $\emptyset$, even though they are present.
- We remove State 2, and update the remaining arrow labels. The only label that changes is the one from 1 to a. It was $\emptyset$, but became $b(a \cup b)^*$ according to Step 3 of the CONVERT. CONVERT gives $(b)(a \cup b)^*(\varepsilon) \cup \emptyset$, which can be simplified to $b(a \cup b)^*$.
- Finally, remove State 1 and follow the same procedure.

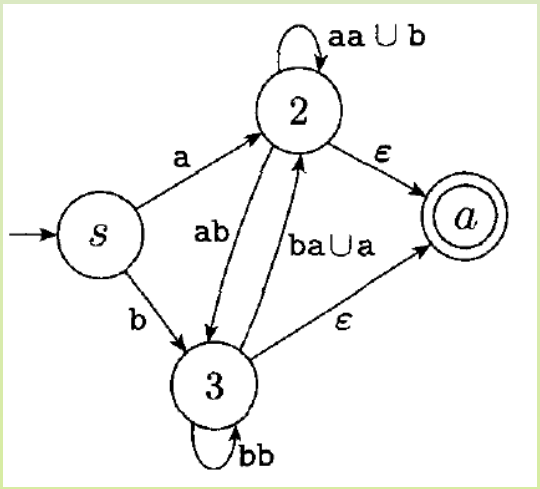
Another Example



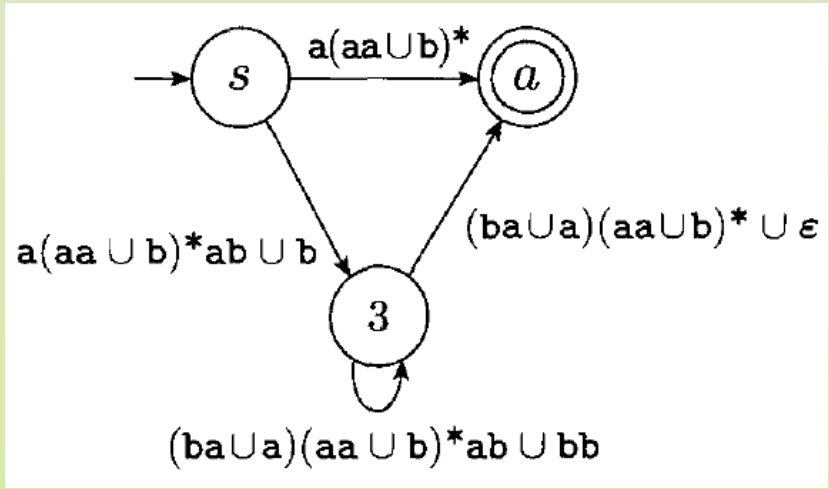
add s and a



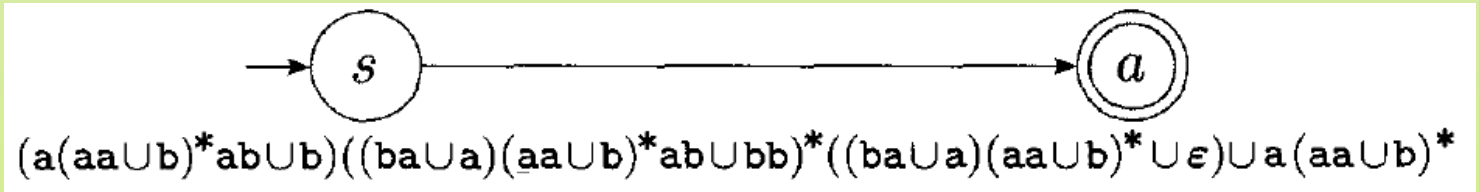
remove 1



remove 2



remove 3



Subsection 4

Nonregular Languages

Limitations of Finite Automata

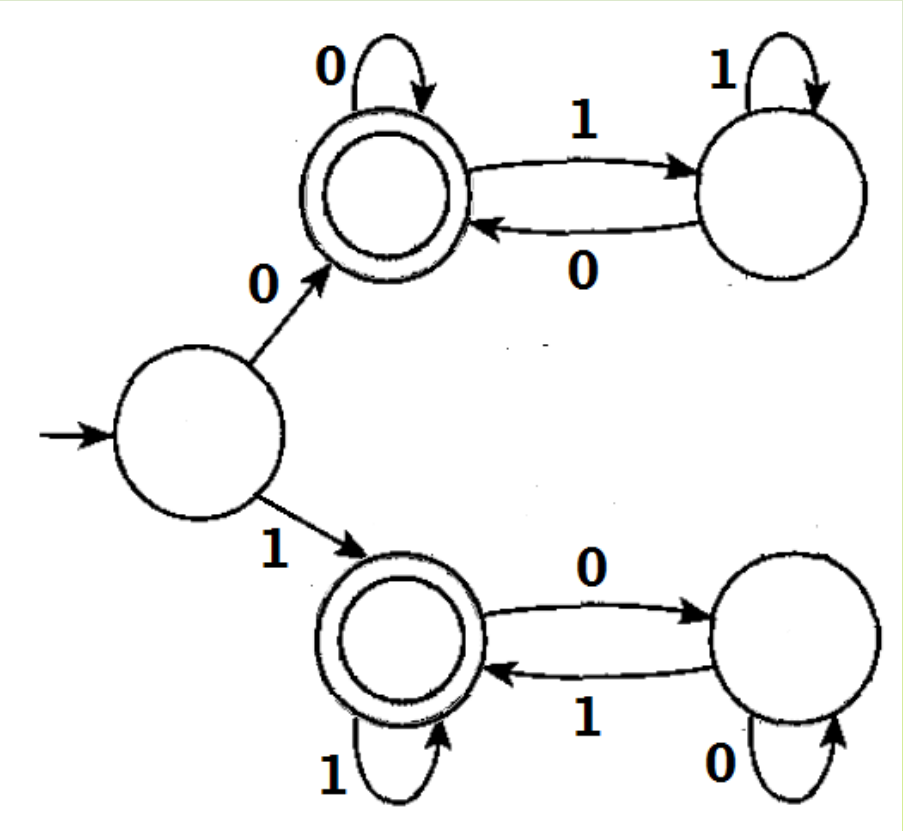
- Finite automata have limitations: we show how to prove that certain languages cannot be recognized by any finite automaton.
- Example: Consider the language $B = \{0^n 1^n : n \geq 0\}$. An attempt to find a DFA that recognizes B indicates that the machine needs to remember how many 0s have been seen so far as it reads the input. Because the number of 0s is not limited, the machine will have to keep track of an unlimited number of possibilities. This cannot be accomplished with a finite number of states.
- We present a method for proving that languages are not regular.
- A formal method is needed: just because the language appears to require unbounded memory does not mean that it actually does.
- Example: Consider two languages over the alphabet $\Sigma = \{0, 1\}$: $C = \{w : w \text{ has an equal number of 0s and 1s}\}$, and $D = \{w : w \text{ has an equal number of occurrences of 01 and 10 as substrings}\}$. Neither language appears to be regular. C is not, but D is!

A Finite Automaton Recognizing D

- Consider the language

$$D = \{w : w \text{ has an equal number of 01 and 10 as substrings}\}.$$

It is regular since it is recognized by the following NFA:



Introducing the Pumping Lemma

- The technique for proving nonregularity relies on a theorem about regular languages, called the **pumping lemma**.
- It states that all regular languages have a special property. If we can show that a language does not have this property, then the language is not regular.
- The property states that all strings in the language can be "pumped" if they are at least as long as a certain special value, called the **pumping length**.
- This means that each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

The Pumping Lemma

Theorem (The Pumping Lemma)

If A is a regular language, then there is a number p (the **pumping length**), such that, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i > 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

- $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.
- When s is divided into xyz, either x or z may be ε, but Condition 2 says that $y \neq \varepsilon$. This ensures the theorem is not trivially true.
- Condition 3 states that the pieces x and y together have length at most p. This is a useful technical condition in applications.

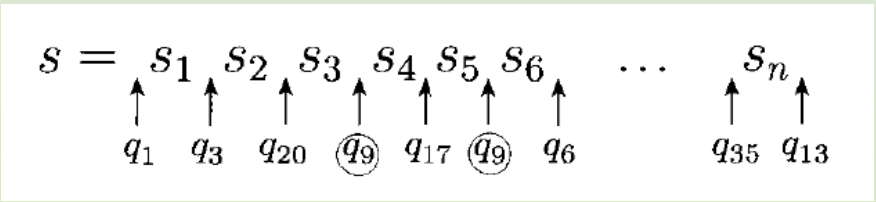
Idea Behind the Proof of the Pumping Lemma

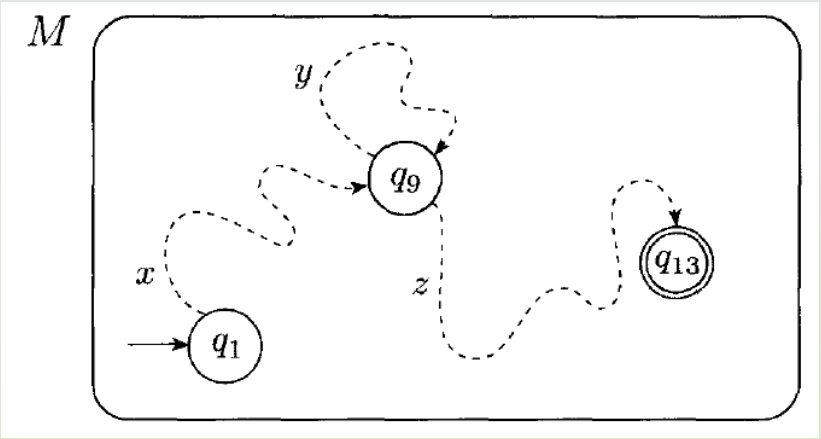
- Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes A. We assign the pumping length p to be the number of states of M. We show that any string s in A of length at least p may be broken into the three pieces xyz satisfying the three conditions.
  - If no strings in A are of length at least p, the task is easier because the theorem becomes vacuously true.
  - If s in A has length at least p, consider the sequence of states that M goes through when computing with input s:

$$\overbrace{q_1}^{\text{start state}} \to q_3 \to q_{20} \to q_9 \to \cdots \to \overbrace{q_{13}}^{\text{end of input}}$$

With s in A, we know that M accepts s, so q_{13} is an accept state. If we let n be the length of s, the sequence of states $q_1, q_3, q_{20}, q_9, \ldots, q_{13}$ has length $n + 1$. Because n is at least p, we know that $n + 1$ is greater than p, the number of states of M. Therefore, the sequence must contain a repeated state.

Idea Behind the Proof of the Pumping Lemma (Cont'd)





- Suppose state q_9 is the one that repeats. We divide s into the three pieces x, y and z:
  - Piece x is the part of s appearing before q_9;
  - Piece y is the part between the two appearances of q_9;
  - Piece z is the remaining part of s.

  So x takes M from the state q, to q_9, y takes M from q_9 back to q_9 and z takes M from q_9 to the accept state q_{13}:
  - If we run M on input xy^iz, M accepts it.
  - $|y| > 0$, as it was the part of s that occurred between two different occurrences of state q_9.
  - By taking q_9 to be the first repetition, we ensure $|xy| < p$.

Formal Proof of the Pumping Lemma

- Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognizing A and p be the number of states of M. Let $s = s_1 s_2 \cdots s_n$ be a string in A of length n, where $n \geq p$. Let $r_1, , \ldots, r_{n+1}$ be the sequence of states that M enters while processing s, i.e., $r_{i+1} = \delta(r_i, s_i)$, for $1 \leq i \leq n$. This sequence has length $n + 1$, which is at least $p + 1$. Among the first $p + 1$ elements in the sequence, two must be the same state, by the pigeonhole principle. We call the first of these r_j and the second r_ℓ. Because r_ℓ occurs among the first $p + 1$ places in a sequence starting at r_1, we have $\ell \leq p + 1$. Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{\ell-1}$ and $z = s_\ell \cdots s_n$.
  - As x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accept state, M must accept $ry^i z$, for $i \geq 0$.
  - We know that $j \neq \ell$, so $|y| > 0$;
  - $\ell \leq p + 1$, so $|xy| \leq p$.

  Thus we have satisfied all three conditions of the pumping lemma.

How to Apply the Pumping Lemma to Prove Nonregularity

- Using the pumping lemma to prove that a language B is not regular:
  - Assume that B is regular in order to obtain a contradiction.
  - Use the pumping lemma to guarantee the existence of a pumping length p such that all strings of length p or greater in B can be pumped.
  - Find a string s in B of length p or greater that cannot be pumped.
  - Demonstrate that s cannot be pumped:
    - Consider all ways of dividing s into x, y and z (taking Condition 3 of the pumping lemma into account, if convenient) and, for each such division, finding a value i where $xy^iz \notin B$.
    - This step often involves grouping the various ways of dividing s into several cases and analyzing them individually.
  - The existence of s contradicts the pumping lemma if B were regular, so B cannot be regular.
- Finding s is not always straightforward: We try members of B that seem to exhibit the "essence" of B's nonregularity.

The Language $\{0^n1^n : n \geq 0\}$

- Let B be the language $\{0^n1^n : n \geq 0\}$. We use the pumping lemma to prove that B is not regular.
  Assume that B is regular. Let p be the pumping length. Choose s to be the string 0^p1^p. Because s is a member of B and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, such that, for any $i \geq 0$, the string xy^iz is in B. We consider three cases to show that this result is impossible:
  1. The string y consists only of 0s: In this case the string $xyyz$ has more 0s than 1s and so is not a member of B, a contradiction.
  2. The string y consists only of 1s: This case also gives a contradiction.
  3. The string y consists of both 0s and 1s: In this case the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B, a contradiction.

  Thus, assuming B regular leads to a contradiction, so B is not regular.
- In this example, finding the string s was easy, because any string in B of length p or more would work.
- We next look at an example where some choices for s do not work.

Equal Number of 0s and 1s

- Let $C = \{w : w \text{ has an equal number of 0s and 1s}\}$. We use the pumping lemma to prove that C is not regular.

  Assume that C is regular. Let p be the pumping length. Let s be the string 0^p1^p. With s being a member of C and having length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where, for any $i \geq 0$, the string xy^iz is in C. Even though we wish to show that this is impossible, it is not the case! If we let x and z be the empty string and y be the string 0^p1^p, then xy^iz always has an equal number of 0s and 1s and hence is in C. So it seems that s can be pumped.

  Condition 3 in the pumping lemma, however, stipulates that when pumping s, it must be divided so that $|xy| \leq p$. If $|xy| \leq p$, then y must consist only of 0s, so $xyyz \notin C$. Therefore s cannot be pumped, a contradiction.

The Language $\{ww : w \in \{0,1\}^*\}$

- Let $F = \{ww : w \in \{0,1\}^*\}$. We show that F is nonregular, using the pumping lemma.

  Assume that F is regular. Let p be the pumping length. Let s be the string 0^p10^p1. Because s is a member of F and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

  Condition 3 is crucial, because without it we could pump s if we let x and z be the empty string. With Condition 3, the proof follows because y must consist only of 0s, so $xyyz \notin F$.

- Observe that we chose $s = 0^p10^p1$ to be a string that exhibits the "essence" of the nonregularity of F, as opposed to, say, the string 0^p0^p. Even though 0^p0^p is a member of F, it fails to demonstrate a contradiction because it can be pumped.

Perfect Squares

- Let $D = \{1^{n^2} : n \geq 0\}$. In other words, D contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that D is not regular.
  Assume that D is regular. Let p be the pumping length. Let s be the string 1^{p^2}. Because s is a member of D and s has length at least p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where, for any $i \geq 0$, the string xy^iz is in D. As in the preceding examples, we show that this outcome is impossible.
  Consider the two strings xyz and xy^2z. Their lengths differ by the length of y. By Condition 3 of the pumping lemma, $|xy| \leq p$, whence $|y| \leq p$. We have $|xyz| = p^2$. so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Moreover, condition 2 implies that y is not the empty string and so $|xy^2z| > p^2$. Therefore, the length of xy^2z lies strictly between the consecutive perfect squares p^2 and $(p+1)^2$. Hence, it cannot be a perfect square itself, i.e., $xy^2z \notin D$, a contradiction. Thus, D is not regular.

"Pumping Down"

- We use the pumping lemma to show that $E = \{0^i 1^j : i > j\}$ is not regular.

  Assume that E is regular. Let p be the pumping length. Let $s = 0^{p+1}1^p$. Then s can be split into xyz, satisfying the conditions of the pumping lemma. By Condition 3, y consists only of 0s. Consider the string $xyyz$. Adding an extra copy of y increases the number of 0s. But, E contains all strings in 0^*1^* that have more 0s than 1s, so increasing the number of 0s will still give a string in E. No contradiction occurs.

  We try something else: The pumping lemma states that $xy^iz \in E$ even when $i = 0$. Consider the string $xy^0z = xz$. Removing string y decreases the number of 0s in s. Recall that s has just one more 0 than 1. Therefore, xz cannot have more 0s than 1s, so it cannot be a member of E. Thus, we obtain a contradiction.