CSE 211 (Theory of Computation)

Regular Languages

Dr. Muhammad Masroor Ali

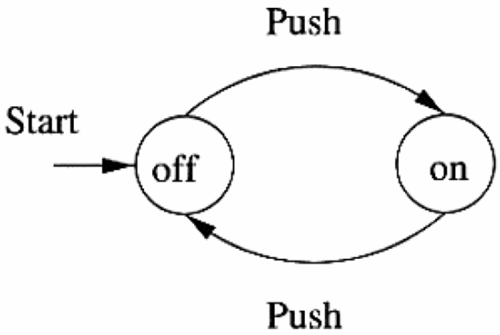
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

January 2024

Version: 1.1, Last modified: September 2, 2024

Example

Hopcroft, Motwani, and Ullman, Figure 1.1, p-3



A finite automaton modeling an on/off switch

Finite Automata

Sipser, Figure 1.4, p-34

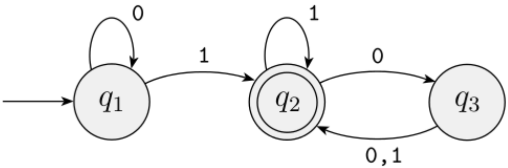


FIGURE 1.4

A finite automaton called M_1 that has three states

Finite Automata

Sipser, 1.1, p-34

- state diagram
- states
- start state
- accept state
- transitions

Finite Automata

Hopcroft, Motwani, and Ullman, 2.2, p-45

- deterministic finite automaton
- deterministic
- nondeterministic
- DFA

Formal Definition of a Finite Automaton

Sipser, Definition 1.5, p-35

DEFINITION 1.5

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

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

Formal Definition of a Finite Automaton

Sipser, 1.1, p-35

- A is the set of all strings that machine M accepts.
- We say that A is the language of machine M.
- Write $L(M) = A$.
- We say that M recognizes A or that M accepts A.

Example

Sipser, Example 1.7, p-37

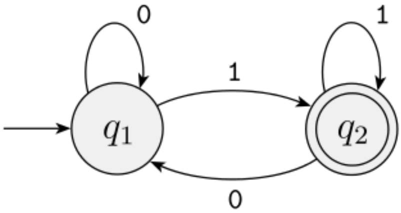


FIGURE 1.8
State diagram of the two-state finite automaton M_2

Example

Sipser, Example 1.9, p-38

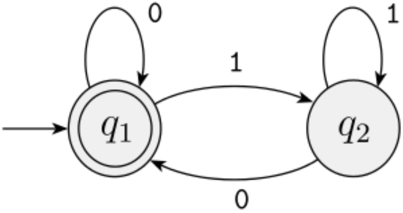


FIGURE 1.10
State diagram of the two-state finite automaton M_3

Example

Sipser, Example 1.11, p-38

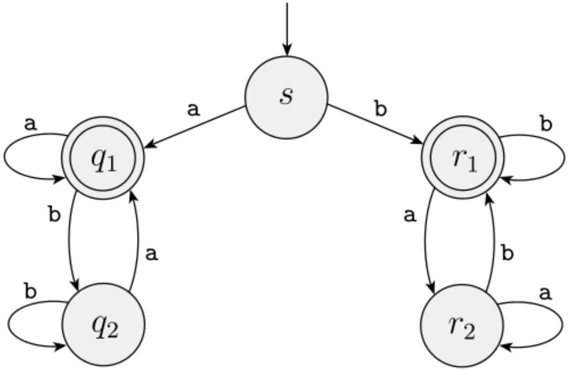


FIGURE 1.12
Finite automaton M_4

Example

Sipser, Example 1.13, p-39

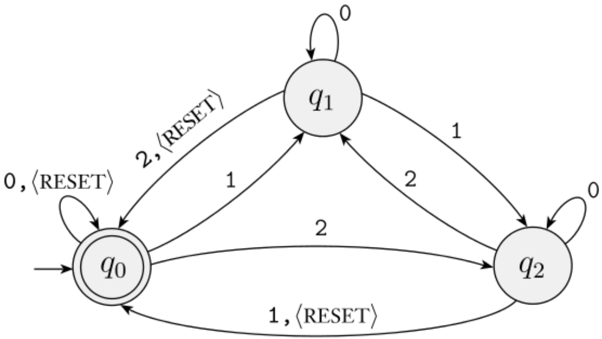


FIGURE 1.14
Finite automaton M_5

Example

Sipser, Example 1.15, p-40

- A generalization of Example 1.13.
- Same four-symbol alphabet Σ.
- For each $i \geq 1$ 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 $<\mathrm{RESET}>$ appears.
- For each A_i we give a finite automaton B_i, recognizing A_i.

Example — *continued*

Sipser, Example 1.15, p-40

- We describe the machine B_i formally as follows.
- $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, where 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.

Example — *continued*

Sipser, Example 1.15, p-40

- For each q_j let,

$$\begin{aligned}
\delta_i(q_j, 0) &= q_j, \\
\delta_i(q_j, 1) &= q_k, \text{ where } k = j + 1 \text{ modulo } i, \\
\delta_i(q_j, 2) &= q_k, \text{ where } k = j + 2 \text{ modulo } i, \text{ and} \\
\delta_i(q_j, \texttt{<RESET>}) &= q_0
\end{aligned}$$

Formal Definition of Computation

Sipser, 1.1, p-40

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton.
- Let $w_1, w_2, \ldots, w_n$ be a string where each w_i is a member of the alphabet Σ.

Formal Definition of Computation — *continued*

Sipser, 1.1, p-40

- Then M accepts w if a sequence of states $r_0, r_1, r_2, \ldots, r_n$ in Q exists with three conditions:
  1. $r_0 = q_0$,
  2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
  3. $r_n \in F$.

- Then M accepts w if a sequence of states $r_0, r_1, r_2, \ldots, r_n$ in Q exists with three conditions:
  1. $r_0 = q_0$,
  2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
  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 $A = \{w \mid M \text{ accepts } w\}$.

Formal Definition of Computation

Sipser, Definition 1.16, p-40

DEFINITION 1.16

A language is called a ***regular language*** if some finite automaton recognizes it.

Designing Finite Automata

Sipser, 1.1, p-41

- You have to figure out what you need to remember about the string as you are reading it.

Designing Finite Automata

Sipser, 1.1, p-41

- Suppose that the alphabet is $\{0, 1\}$ and that the language consists of all strings with an odd number of 1s.
- You want to construct a finite automaton E_1 to recognize this language.

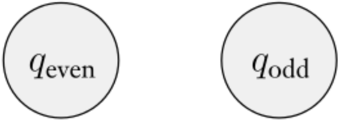


FIGURE 1.18
The two states q_{even} and q_{odd}

Designing Finite Automata

Sipser, Figure 1.19, p-42

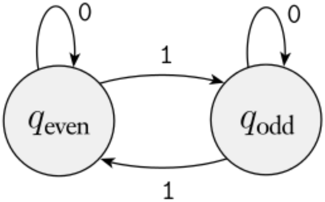


FIGURE 1.19
Transitions telling how the possibilities rearrange

Designing Finite Automata

Sipser, Figure 1.20, p-43

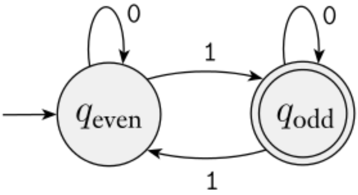


FIGURE 1.20
Adding the start and accept states

- 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, and 11111110011111 are all in the language, but 11 and 0000 are not.

Example — *continued*

Sipser, Example 1.21, p-44

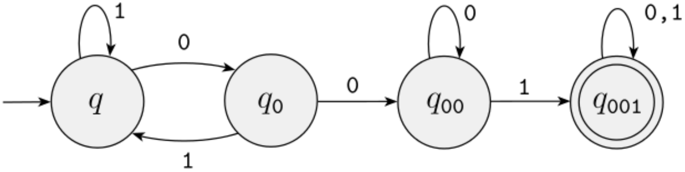


FIGURE 1.22
Accepts strings containing 001

- Let us formally specify a DFA that accepts all and only the strings of 0's and 1's that have the sequence 01 somewhere in the string.

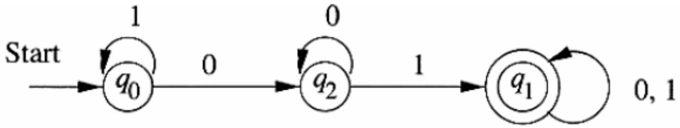
Example — *continued*

Hopcroft, Motwani, and Ullman, Example 2.1, p-46

- We can write this language L as:
  $\{w \,|\, w$ is of the form $x01y$ for some strings x and y consisting of 0's and 1's only.$\}$
- Another equivalent description, using parameters x and y to the left of the vertical bar, is:
  $\{x01y \,|\, x$ and y are any strings of 0's and 1's$\}$

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 2.1, p-46



The transition diagram for the DFA accepting all strings with a substring 01

Transition Tables

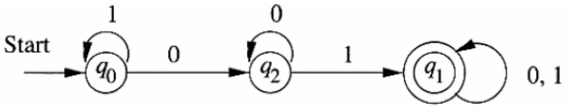
Hopcroft, Motwani, and Ullman, 2.2.3, p-48

- A transition table is a conventional, tabular representation of a function like δ that takes two arguments and returns a value.
- The rows of the table correspond to the states.
- The columns correspond to the inputs.
- The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$.

Example

Hopcroft, Motwani, and Ullman, Example 2.2.3, p-48

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1



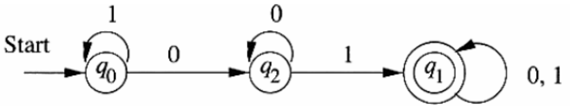
The transition diagram for the DFA accepting all strings with a substring 01

- The start state is marked with an arrow.
- The accepting states are marked with a star.

Example

Hopcroft, Motwani, and Ullman, Example 2.2.3, p-48

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1



The transition diagram for the DFA accepting all strings with a substring 01

- We can deduce the sets of states and input symbols by looking at the row and column heads.
- We can now read from the transition table all the information we need to specify the finite automaton uniquely.

Example

Hopcroft, Motwani, and Ullman, Example 2.4, p-51

- Design a DFA to accept the language $L = \{w \mid w \text{ has both an even number of 0's and an even number of 1's}\}$

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 2.4, p-51

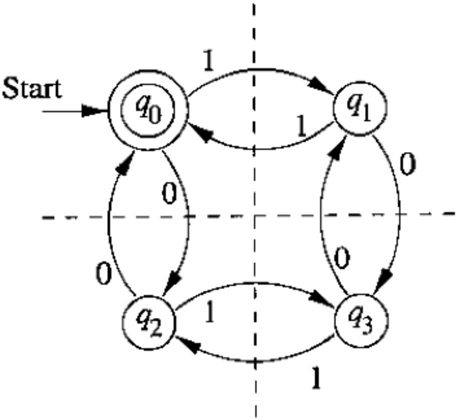
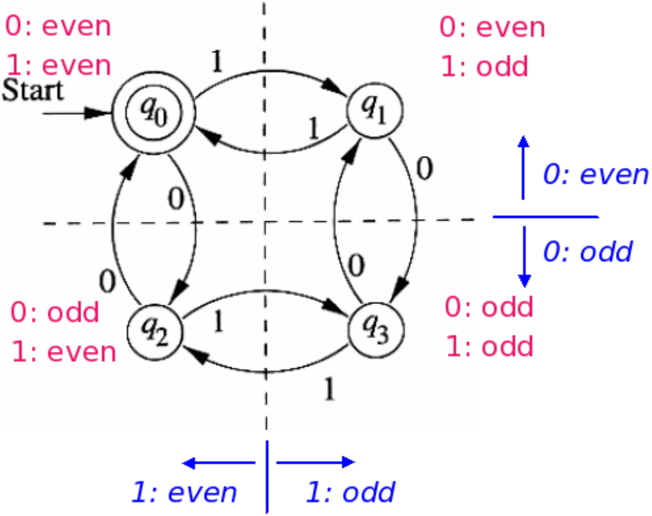


Figure 2.6: Transition diagram for the DFA of Example 2.4

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 2.4, p-51



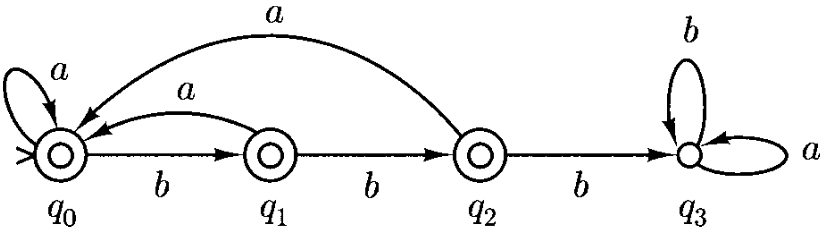
Example

Lewis and Papadimitriou, Example 2.1.2, p-59

- Design a deterministic finite automaton M that accepts the language
  $L(M) = \{w \in \{a, b\}^* :$
  w does not contain three consecutive b's$\}$.

Example — *continued*

Lewis and Papadimitriou, Example 2.1.2, p-59



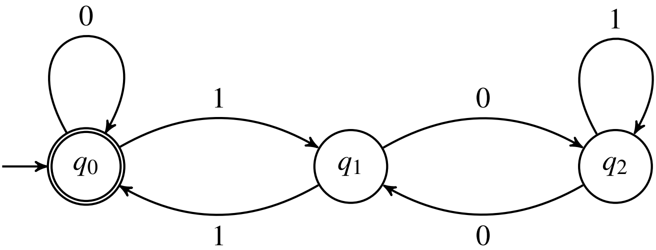
Example

http://math.stackexchange.com/questions/140283/why-does-this-fsm-accept-binary-numbers-divisible-by-three

- Design a DFA that accepts binary numbers that are divisible by three.

Example — *continued*

http://math.stackexchange.com/questions/140283/why-does-this-fsm-accept-binary-numbers-divisible-by-three



Example

John Martin, Example 2.1, p-47

- A finite automaton accepting the language of strings ending with aa.

Example — *continued*

John Martin, Example 2.1, p-47

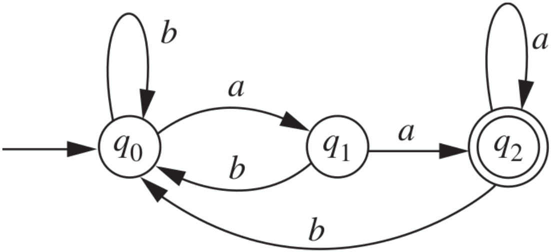


Figure 2.2 |
An FA accepting the strings ending with *aa*.

Example

Peter Linz, Example 2.2

- A finite automaton accepting the language:

$$L = \{a^n b \mid n \geq 0\}.$$

Example — *continued*

Peter Linz, Example 2.2

FIGURE 2.2

The Regular Operations

Sipser, 1.1, p-44

DEFINITION 1.23

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**: $A^* = \{x_1 x_2 \ldots x_k |\ k \geq 0 \text{ and each } x_i \in A\}$.

Example

Sipser, Example 1.24, p-45

- Alphabet Σ be the standard 26 letters $\{a, b, ..., z\}$.
- $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{boy}, \text{girl}\}$.

Example

Sipser, Example 1.24, p-45

- Alphabet Σ be the standard 26 letters $\{a, b, ..., z\}$.
- $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{boy}, \text{girl}\}$.
- $A \cup B = \{\text{good}, \text{bad}, \text{boy}, \text{girl}\}$

Example

Sipser, Example 1.24, p-45

- Alphabet Σ be the standard 26 letters $\{a, b, ..., z\}$.
- $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{boy}, \text{girl}\}$.
- $A \cup B = \{\text{good}, \text{bad}, \text{boy}, \text{girl}\}$
- $A \circ B = \{\text{goodboy}, \text{goodgirl}, \text{badboy}, \text{badgirl}\}$

Example

Sipser, Example 1.24, p-45

- Alphabet Σ be the standard 26 letters $\{a, b, ..., z\}$.
- $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{boy}, \text{girl}\}$.
- $A \cup B = \{\text{good}, \text{bad}, \text{boy}, \text{girl}\}$
- $A \circ B = \{\text{goodboy}, \text{goodgirl}, \text{badboy}, \text{badgirl}\}$
-

$$A^* = \{\epsilon, \text{good}, \text{bad}, \text{goodgood}, \text{goodbad}, \text{badgood}, \text{badbad}, \text{goodgoodgood}, \text{goodgoodbad}, \text{goodbadgood}, \text{goodbadbad}, \ldots\}$$

The Regular Operations

Sipser, 1.1, p-45

- $\mathcal{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers.
- We say that $\mathcal{N}$ is closed under multiplication.
- We mean that for any x and y in $\mathcal{N}$, the product $x \times y$ also is in $\mathcal{N}$.
- In contrast, $\mathcal{N}$ is *not* closed under division.
- 1 and 2 are in $\mathcal{N}$ but $1/2$ is not.

The Regular Operations — *continued*

Sipser, 1.1, p-45

- Generally speaking, 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.
- We show that the collection of regular languages is closed under all three of the regular operations.

The Regular Operations

Sipser, 1.1, p-45

THEOREM 1.25

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

In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

Example

Formulated

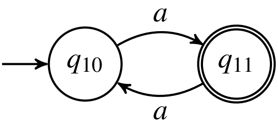
- $\Sigma = \{a\}$
- $L_1 = \{$contains an odd number of a's$\}$

  $L_2 = \{aa\}$
- Design automata M_1 and M_2 for L_1 and L_2 and then construct M which recognizes $L_1 \cup L_2$

- $L_1 = \{$contains an odd number of a's$\}$
  $L_2 = \{aa\}$

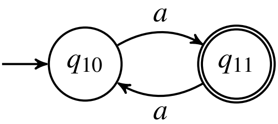
- $L_1 = \{$contains an odd number of a's$\}$
  $L_2 = \{aa\}$

M_1

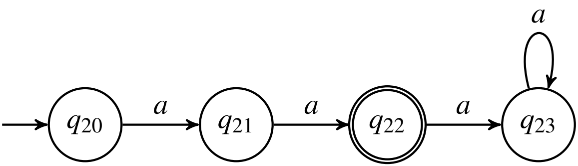


- $L_1 = \{\text{contains an odd number of } a\text{'s}\}$
  $L_2 = \{aa\}$

M_1



M_2



PROOF IDEA

- We have regular languages A_1 and A_2 and want to show that $A_1 \cup A_2$ also is regular.
- Because 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 demonstrate a finite automaton, call it M, that recognizes $A_1 \cup A_2$.

- This is a proof by construction.
- We construct M from M_1 and M_2.
- Machine M must accept its input exactly when either M_1 or M_2 would accept it in order to recognize the union language.

The Regular Operations — *continued*

Sipser, 1.1, p-45

- It works by simulating both M_1 and M_2 and accepting if either of the simulations accept.
- How can we make machine M simulate M_1 and M_2?
- Perhaps it first simulates M_1 on the input and then simulates M_2 on the input.

- But we must be careful here!
- Once the symbols of the input have been read and used to simulate M_1, we can't "rewind the input tape" to try the simulation on M_2.
- We need another approach.

The Regular Operations — *continued*

Sipser, 1.1, p-45

- Pretend that you are M.
- As the input symbols arrive one by one, you simulate both M_1 and M_2 simultaneously.
- That way, only one pass through the input is necessary.

The Regular Operations — *continued*

Sipser, 1.1, p-45

- But can you keep track of both simulations with finite memory?
- All you need to remember is the state that each machine would be in if it had read up to this point in the input.
- Therefore, you need to remember a pair of states.

The Regular Operations — *continued*

Sipser, 1.1, p-45

- How many possible pairs are there?
- If M_1 has k_1 states and M_2 has k_2 states, the number of pairs of states, one from M_1 and the other from M_2, is the product $k_1 \times k_2$.
- This product will be the number of states in M, one for each pair.

The Regular Operations — *continued*

Sipser, 1.1, p-45

- 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.

PROOF

- M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$.
- 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. $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.

- This set is the Cartesian product of sets Q_1 and Q_2 and is written $Q_1 \times Q_2$.
- It is the set of all pairs of states, the first from Q_1 and the second from Q_2.

The Regular Operations — *continued*

Sipser, 1.1, p-45

2. Σ, the alphabet, is the same as in M_1 and M_2.

- In this theorem and in all subsequent similar theorems, we assume for simplicity that both M_1 and M_2 have the same input alphabet Σ.
- The theorem remains true if they have different alphabets, Σ_1 and Σ_2.
- We would then modify the proof to let $\Sigma = \Sigma_1 \cup \Sigma_2$.

The Regular Operations — *continued*

Sipser, 1.1, p-45

3. δ, the transition function, is defined as follows.

- For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let

$$\delta\left((r_1, r_2), a\right) = \left(\delta_1(r_1, a), \delta_2(r_2, a)\right).$$

- Hence δ gets a state of M (which actually is a pair of states from M_1 and M_2), together with an input symbol, and returns M's next state.

4. q_0 is the pair (q_1, q_2).

The Regular Operations — *continued*

Sipser, 1.1, p-45

5. F is the set of pairs in which either member is an accept state of M_1 or M_2.

- We can write it as $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$.
- This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

5. F is the set of pairs in which either member is an accept state of M_1 or M_2.

- We can write it as $F = \{(r_1, r_2) \mid r_1 \in F_1$ or $r_2 \in F_2\}$.
- This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.
- Note that it is not the same as $F = F_1 \times F_2$.

THEOREM 1.26

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

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

The Regular Operations — *continued*

Sipser, 1.1, p-47

- To prove this theorem, let's try something along the lines of the proof of the union case.
- As before, we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2.

The Regular Operations — *continued*

Sipser, 1.1, p-47

- But now, instead of constructing automaton M to accept its input if either M_1 or M_2 accept, it must accept 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 doesn't know where to break its input (i.e., where the first part ends and the second begins).

Nondeterministic Finite Automata

Lewis and Papadimitriou, 2.2, p-63

- Nondeterminism is an *inessential feature* of finite automata.
- Every nondeterministic finite automaton is equivalent to a deterministic finite automaton.
- Thus we shall profit from the powerful notation of nondeterministic finite automata.
- But we always know that, if we must, we can always go back and redo everything in terms of the lower-level language of ordinary, down-to-earth deterministic automata.

- $L = (ab \cup aba)^*$

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba)\ldots(ab \cup aba)$

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$ $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba)\ldots(ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$ *belongs*

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$ *belongs*
- $abab$

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$ *belongs*
- $abab$ *belongs*

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$ *belongs*
- $abab$ *belongs*
- ϵ

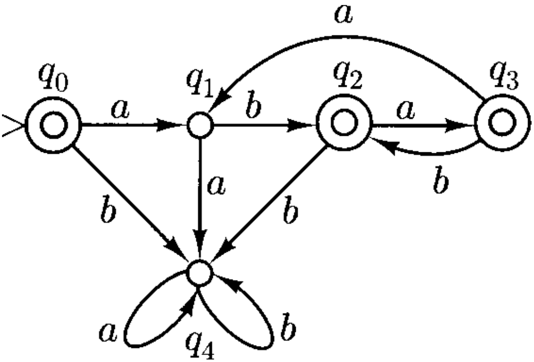
- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$ *belongs*
- $abab$ *belongs*
- ϵ *belongs*

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$ *belongs*
- $abab$ *belongs*
- ϵ *belongs*
- $abababba$

- $L = (ab \cup aba)^*$
- As many as $(ab \cup aba)$'s you like.
- $(ab \cup aba)^* =$
  $(ab \cup aba)(ab \cup aba)(ab \cup aba)(ab \cup aba) \ldots (ab \cup aba)$
- ab *belongs*
- aba *belongs*
- $ababa$ *belongs*
- $abaab$ *belongs*
- $abab$ *belongs*
- ϵ *belongs*
- $abababba$ *does not belong*

Nondeterministic Finite Automata — *continued*

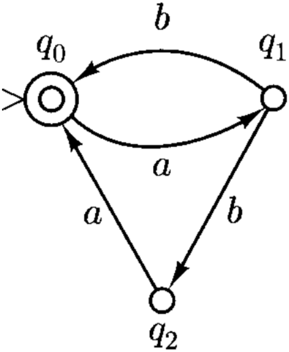
Lewis and Papadimitriou, Figure 2.4, p-64



Nondeterministic Finite Automata — *continued*

Lewis and Papadimitriou, Figure 2.5, p-65

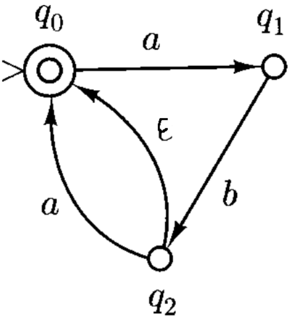
$L = (ab \cup aba)^*$



Nondeterministic Finite Automata — *continued*

Lewis and Papadimitriou, Figure 2.6, p-65

$L = (ab \cup aba)^*$



Nondeterministic Finite Automata — *continued*

Sipser, Figure 1.27, p-48

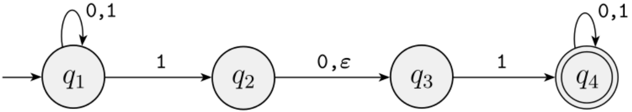


FIGURE 1.27
The nondeterministic finite automaton N_1

Nondeterministic Finite Automata — *continued*

Sipser, Figure 1.28, p-49

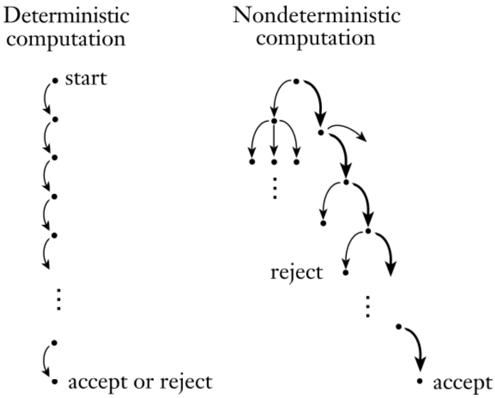


FIGURE 1.28
Deterministic and nondeterministic computations with an accepting branch

Nondeterministic Finite Automata — *continued*

Sipser, Figure 1.29, p-49

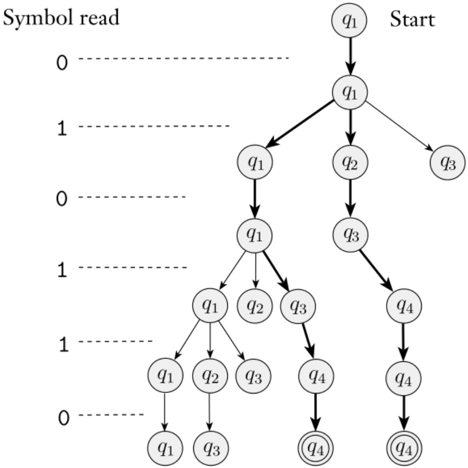


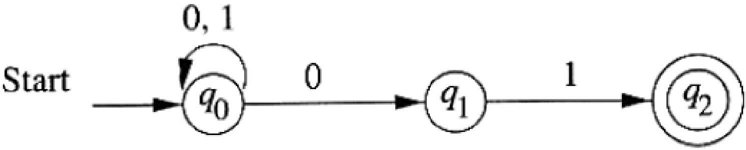
FIGURE 1.29

The computation of N_1 on input 010110

Example

Hopcroft, Motwani, and Ullman, Example 2.6, p-56

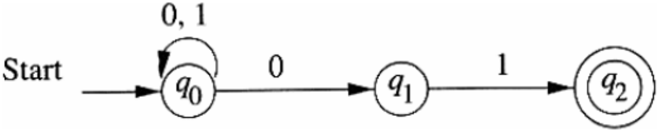
- Job of this automaton is to accept all and only the strings of 0's and 1's that end in 01.

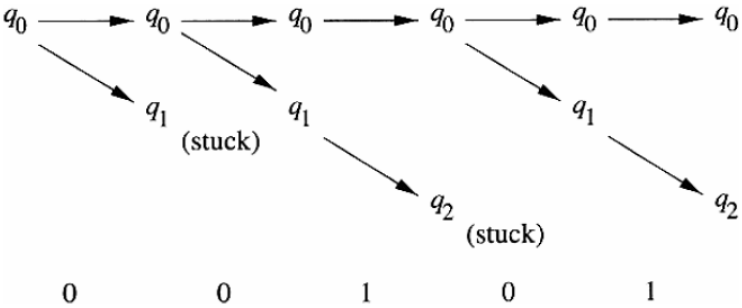


An NFA accepting all strings that end in 01

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 2.6, p-56





Example

Sipser, Example 1.30, p-51

- Let A be the language consisting of all strings over $\{0, 1\}$ containing a 1 in the third position from the end.
- 000100 is in A but 0011 is not.

Example — *continued*

Sipser, Example 1.30, p-51

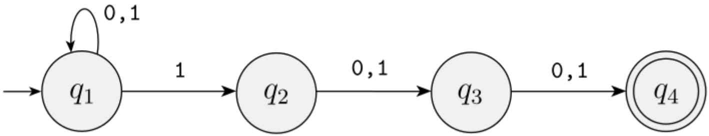


FIGURE 1.31
The NFA N_2 recognizing A

Nondeterministic Finite Automata — *continued*

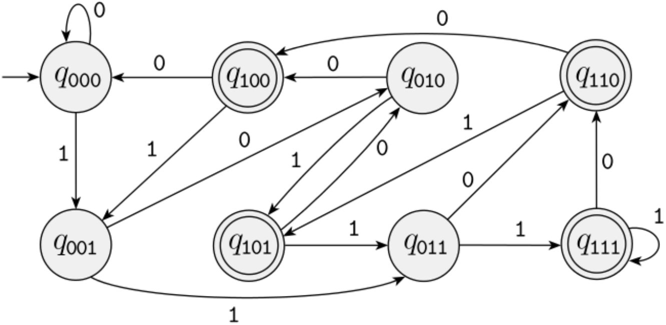


FIGURE 1.32
A DFA recognizing A

- Accepts all strings of the form 0^k where k is a multiple of 2 or 3.
- N_3 accepts the strings ϵ, 00, 000, 0000, and 000000, but not 0 or 00000.

Nondeterministic Finite Automata — *continued*

Sipser, Example 1.33, p-52

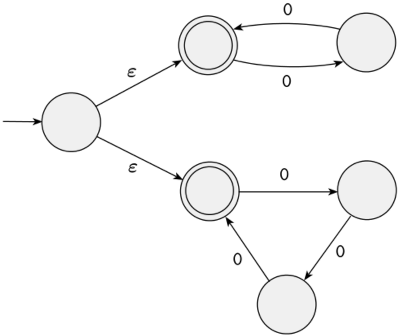


FIGURE 1.34
The NFA N_3

- Has an input alphabet $\{0\}$ consisting of a single symbol.
- An alphabet containing only one symbol is called a unary alphabet.

Sipser, Example 1.35, p-52

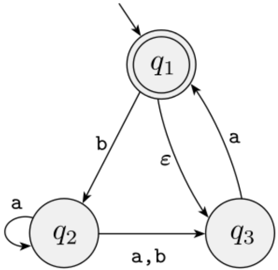


FIGURE 1.36
The NFA N_4

- It accepts the strings ϵ, `a`, `baba`, and `baa`.
- But that it doesn't accept the strings `b`, `bb`, and `babba`.

Formal Definition of a Nondeterministic Finite Automaton

Sipser, 1.2, p-53

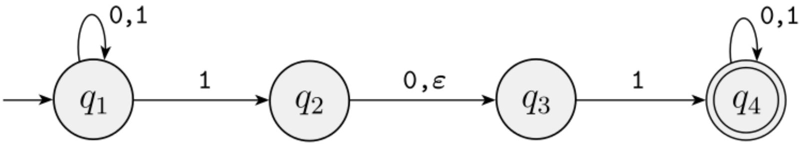
DEFINITION 1.37

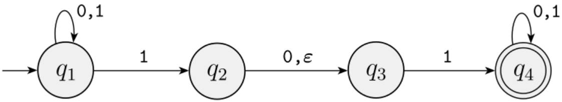
A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

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

Formal Definition of. . . — *continued*

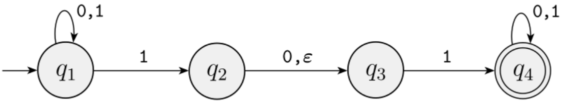
Sipser, Example 1.38, p-54





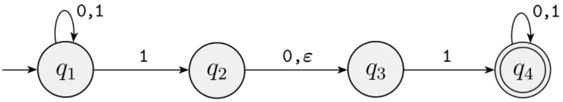
The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where,

1. $Q = \{q_1, q_2, q_3, q_4\}$



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where,

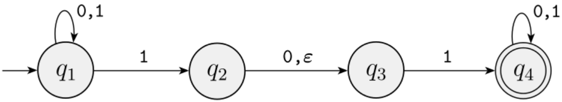
2. $\Sigma = \{0, 1\}$



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where,

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	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

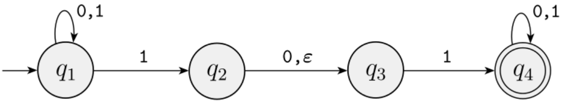


The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where,

4. q_1 is the start state.

Formal Definition of. . . — *continued*

Sipser, Example 1.38, p-54



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where,

5. $F = \{q_4\}$.

Equivalence of NFAs AND DFAs

Sipser, 1.2, p-54

- Deterministic and nondeterministic finite automata recognize the same class of languages.
- Such equivalence is both surprising and useful.
- It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages.
- It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.
- Say that two machines are equivalent if they recognize the same language.

Equivalence of NFAs AND DFAs

Sipser, 1.2, p-55

THEOREM 1.39

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

PROOF IDEA

- If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes it.
- The idea is to convert the NFA into an equivalent DFA that simulates the NFA.
- Recall the “reader as automaton” strategy for designing finite automata.

- How would you simulate the NFA if you were pretending to be a DFA?
- What do you need to keep track of as the input string is processed?
- In the examples of NFA's, you kept track of the various branches of the computation by placing a finger on each state that could be active at given points in the input.
- You updated the simulation by moving, adding, and removing fingers according to the way the NFA operates.
- All you needed to keep track of was the set of states having fingers on them.

- If k is the number of states of the NFA, it has 2^k subsets of states.
- Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^k states.
- Now we need to figure out which will be the start state and accept states of the DFA.
- What will be its transition function.
- We can discuss this more easily after setting up some formal notation.

Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

PROOF

- 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.

Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

- Before doing the full construction, let's first consider the easier case wherein N has no ϵ arrows.
- Later we take the ϵ arrows into account.

Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

1. $Q' = P(Q)$.

- Every state of M is a set of states of N.
- Recall that $P(Q)$ is the set of subsets of Q.

2. For $R \in Q'$ and $a \in \Sigma$, let
   $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a)$ 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.
- Another way to write this expression is,

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$

Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

3. ${q_0}' = \{q_0\}$.

- M starts in the state corresponding to the collection containing just the start state of N.

Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$.

- The machine M accepts if one of the possible states that N could be in at this point is an accept state.

Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

- Now we need to consider the ϵ arrows.
- To do so, we set up an extra bit of notation.
- For any state R of M, we define $E(R)$ to be the collection of states that can be reached from members of R by going only along ϵ arrows, including the members of R themselves.

Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

- Formally, for $R \subseteq Q$ let

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

- Then 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 $\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a))$ for some $r \in R\}$.
- Additionally, we need to 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 $E\left(\{q_0\}\right)$ achieves this effect.

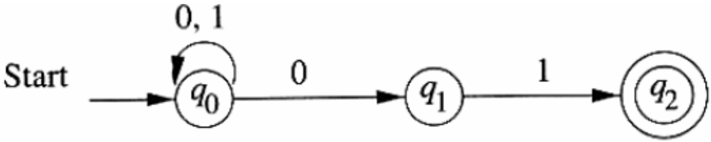
Equivalence of NFAs AND DFAs — *continued*

Sipser, 1.2, p-55

- We have now completed the construction of the DFA M that simulates the NFA N.
- The construction of M obviously works correctly.
- At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point.
- Thus our proof is complete.

Example

Hopcroft, Motwani, and Ullman, Example 2.10, p-61



An NFA accepting all strings that end in 01

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 2.10, p-61

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

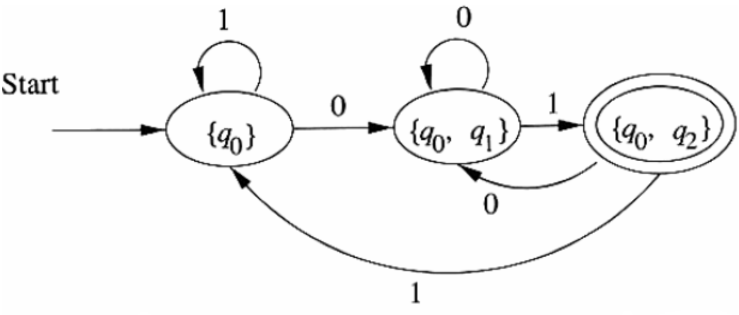
Figure 2.12: The complete subset construction from Fig. 2.9

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Renaming the states

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 2.10, p-61



The DFA constructed from the NFA

Example

Sipser, Example 1.41, p-56

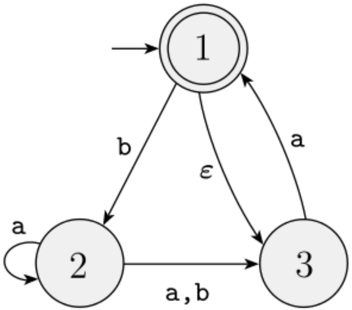


FIGURE 1.42
The NFA N_4

Example — *continued*

Sipser, Example 1.41, p-56

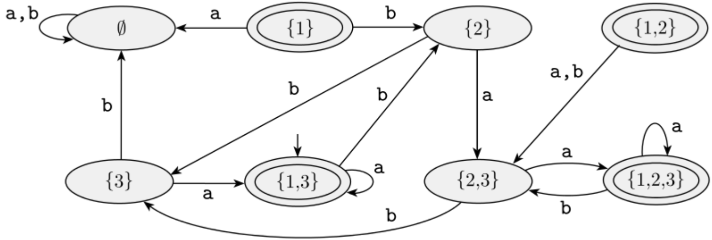


FIGURE 1.43
A DFA D that is equivalent to the NFA N_4

Example — *continued*

Sipser, Example 1.41, p-56

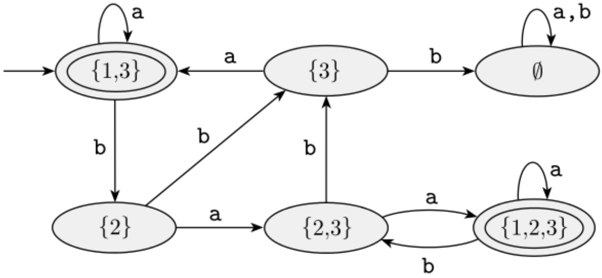
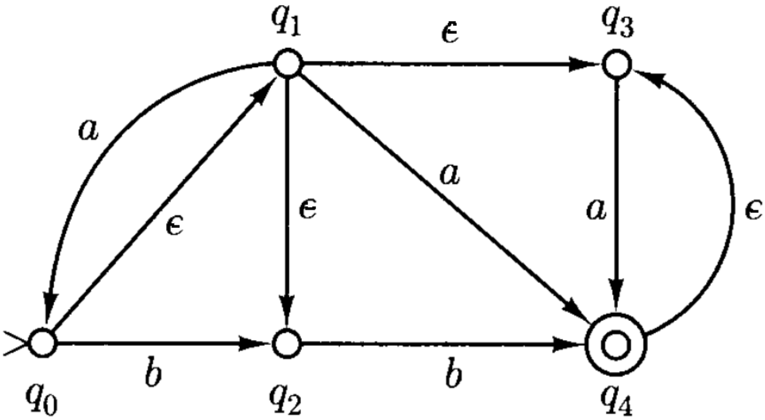


FIGURE 1.44
DFA D after removing unnecessary states

Example

Lewis and Papadimitriou, Example 2.2.3, p-70

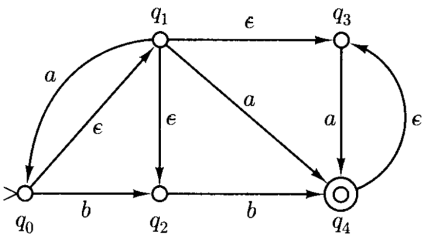
We find the DFA equivalent to the nondeterministic automaton.



Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$

- Q' is the power set of Q.

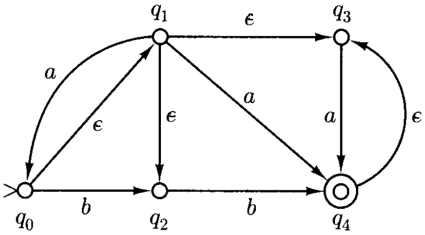


- Since N has 5 states, D will have $2^5 = 32$ states.
- However, only a few of these states will be relevant to the operation of D.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad\qquad D = (Q', \Sigma, \delta', {q_0}', F')$$

- Q' is the power set of Q.

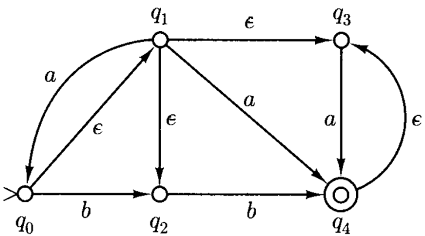


- Namely, those states that can be reached from state ${q_0}'$ by reading some input string.
- Obviously, any state in D that is not reachable from ${q_0}'$ is irrelevant to the operation of D.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad\qquad D = (Q', \Sigma, \delta', {q_0}', F')$$

- Q' is the power set of Q.

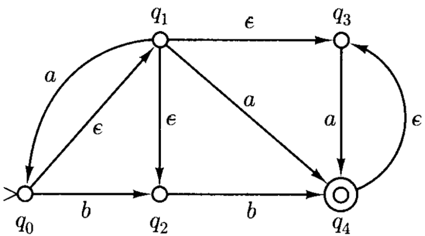


- We shall build this by *lazy evaluation* on the subsets.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$

- ${q_0}' = E(q_0)$.

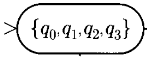


- ${q_0}' = E(q_0) = \{q_0, q_1, q_2, q_3\}$.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', q_0', F')$$

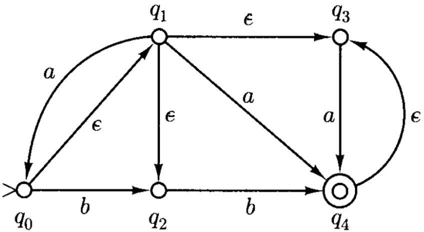
- $q_0' = E(q_0)$.



- $q_0' = E(q_0) = \{q_0, q_1, q_2, q_3\}$.

Example — *continued*

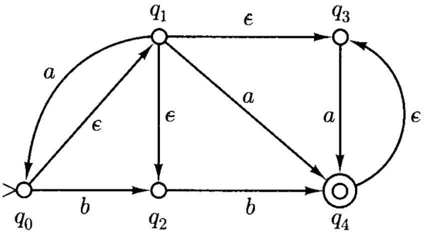
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- $\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a) \cup \delta(q_3, a) = \emptyset \cup \{q_0, q_4\} \cup \emptyset \cup \{q_4\} = \{q_0, q_4\}$

Example — *continued*

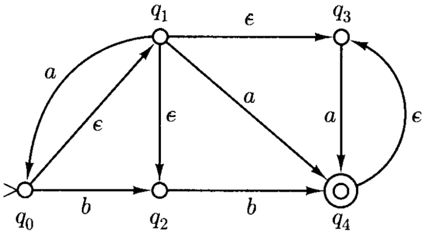
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- $E(q_0) = \{q_0, q_1, q_2, q_3\}$, and $E(q_4) = \{q_3, q_4\}$.
- $\delta'({q_0}', a) = \{q_0, q_1, q_2, q_3\} \cup \{q_3, q_4\} = \{q_0, q_1, q_2, q_3, q_4\}$.

Example — *continued*

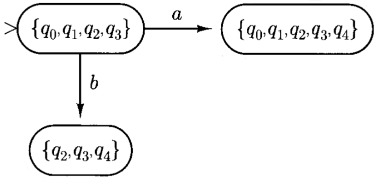
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- Similarly, $\delta'({q_0}', b) = \{q_2, q_3, q_4\}$.

Example — *continued*

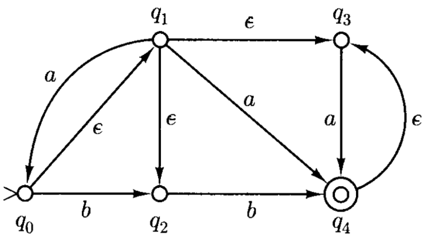
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- Similarly, $\delta'({q_0}', b) = \{q_2, q_3, q_4\}$.

Example — *continued*

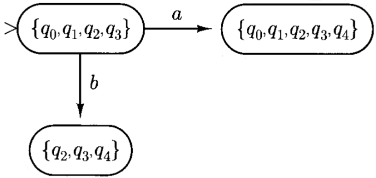
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- We repeat the calculation for the newly introduced states.
- $\delta'(\{q_0, q_1, q_2, q_3, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$, and
- $\delta'(\{q_0, q_1, q_2, q_3, q_4\}, b) = \{q_2, q_3, q_4\}$.

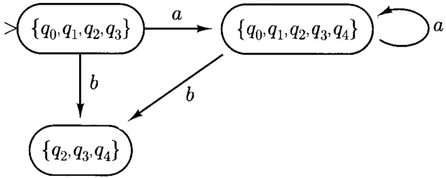
Example — *continued*

$N = (Q, \Sigma, \delta, q_0, F)$ $\qquad$ $D = (Q', \Sigma, \delta', q_0{}', F')$



- We repeat the calculation for the newly introduced states.
- $\delta'(\{q_0, q_1, q_2, q_3, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$, and
- $\delta'(\{q_0, q_1, q_2, q_3, q_4\}, b) = \{q_2, q_3, q_4\}$.

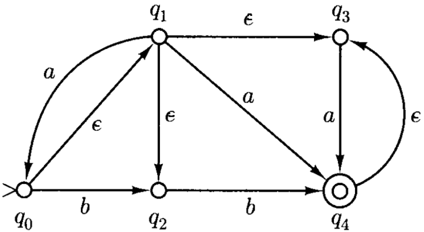
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- We repeat the calculation for the newly introduced states.
- $\delta'(\{q_0, q_1, q_2, q_3, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$, and
- $\delta'(\{q_0, q_1, q_2, q_3, q_4\}, b) = \{q_2, q_3, q_4\}$.

Example — *continued*

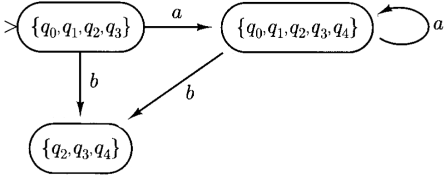
$N = (Q, \Sigma, \delta, q_0, F)$ $D = (Q', \Sigma, \delta', {q_0}', F')$



- Also we get.
- $\delta'(\{q_2, q_3, q_4\}, a) = \{q_3, q_4\}$, and
- $\delta'(\{q_2, q_3, q_4\}, b) = \{q_3, q_4\}$.

Example — *continued*

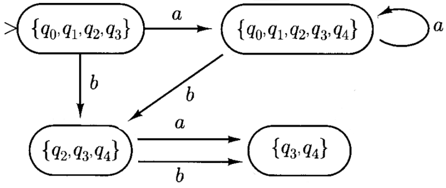
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', q_0{}', F')$$



- Also we get.
- $\delta'(\{q_2, q_3, q_4\}, a) = \{q_3, q_4\}$, and
- $\delta'(\{q_2, q_3, q_4\}, b) = \{q_3, q_4\}$.

Example — *continued*

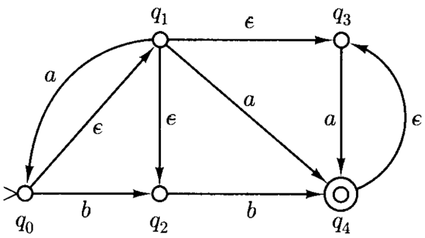
$N = (Q, \Sigma, \delta, q_0, F)$ $\quad$ $D = (Q', \Sigma, \delta', {q_0}', F')$



- Also we get.
- $\delta'(\{q_2, q_3, q_4\}, a) = \{q_3, q_4\}$, and
- $\delta'(\{q_2, q_3, q_4\}, b) = \{q_3, q_4\}$.

Example — *continued*

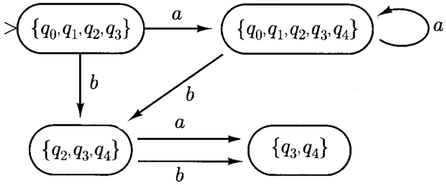
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- Next we get.
- $\delta'(\{q_3, q_4\}, a) = \{q_3, q_4\}$, and
- $\delta'(\{q_3, q_4\}, b) = \emptyset$.

Example — *continued*

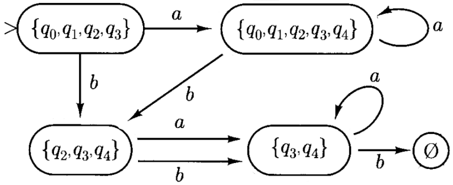
$N = (Q, \Sigma, \delta, q_0, F)$ $D = (Q', \Sigma, \delta', {q_0}', F')$



- Next we get.
- $\delta'(\{q_3, q_4\}, a) = \{q_3, q_4\}$, and
- $\delta'(\{q_3, q_4\}, b) = \emptyset$.

Example — *continued*

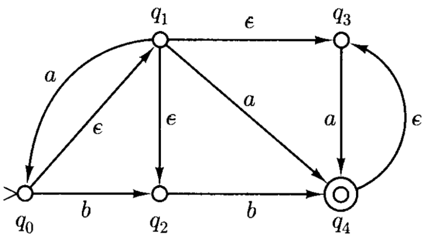
$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- Next we get.
- $\delta'(\{q_3, q_4\}, a) = \{q_3, q_4\}$, and
- $\delta'(\{q_3, q_4\}, b) = \emptyset$.

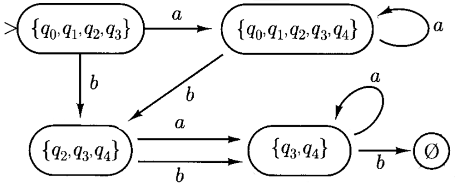
Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$



- Finally, we get.
- $\delta'(\emptyset, a) = \delta'(\emptyset, b) = \emptyset$.

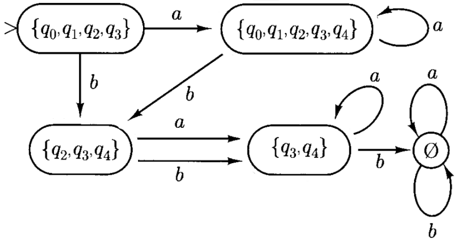
$N = (Q, \Sigma, \delta, q_0, F)$ $D = (Q', \Sigma, \delta', q_0', F')$



- Finally, we get.
- $\delta'(\emptyset, a) = \delta'(\emptyset, b) = \emptyset$.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', q_0{}', F')$$

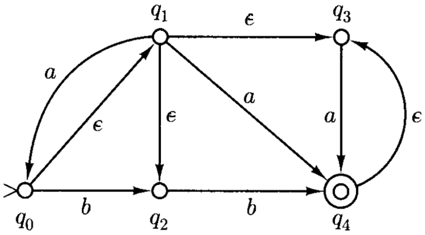


- Finally, we get.
- $\delta'(\emptyset, a) = \delta'(\emptyset, b) = \emptyset$.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', {q_0}', F')$$

- F' is those sets of states that contain at least one accepting state of N.

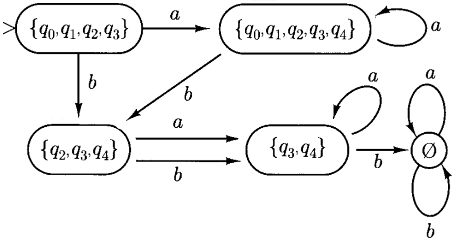


- q_4 is the sole member of F.
- The set of final states, contains each set of states of which q_4 is a member.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', q_0', F')$$

- F' is those sets of states that contain at least one accepting state of N.

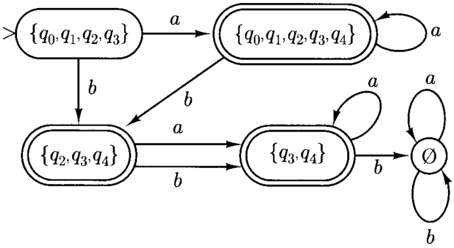


- q_4 is the sole member of F.
- The set of final states, contains each set of states of which q_4 is a member.

Example — *continued*

$$N = (Q, \Sigma, \delta, q_0, F) \qquad D = (Q', \Sigma, \delta', q_0', F')$$

- F' is those sets of states that contain at least one accepting state of N.



- The three states $\{q_0, q_1, q_2, q_3, q_4\}$, $\{q_2, q_3, q_4\}$, and $\{q_3, q_4\}$ are final.

Equivalence of NFAs AND DFAs

Sipser, 1.2, p-56

COROLLARY 1.40

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

THEOREM 1.45

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

Closure under the Regular Operations — *continued*

Sipser, 1.2, p-59

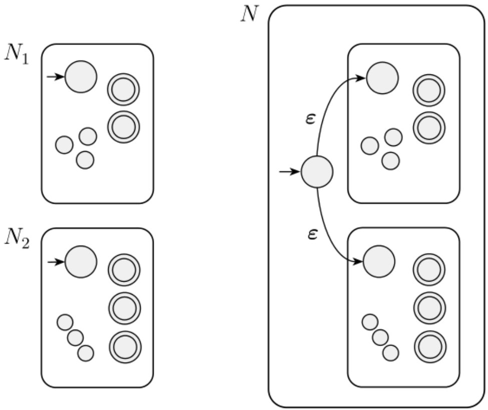
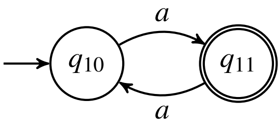


FIGURE 1.46
Construction of an NFA N to recognize $A_1 \cup A_2$

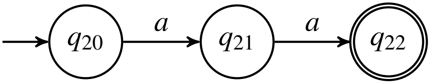
- $L_1 = \{$contains an odd number of a's$\}$

  $L_2 = \{aa\}$

N_1



N_2



PROOF

- 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 set of accept states $F = F_1 \cup F_2$.

- The accept states of N are all the accept states of N_1 and N_2.
- That way, N accepts if either N_1 accepts or N_2 accepts.

4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$,

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

THEOREM 1.47

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

Closure under the Regular Operations — *continued*

Sipser, 1.2, p-61

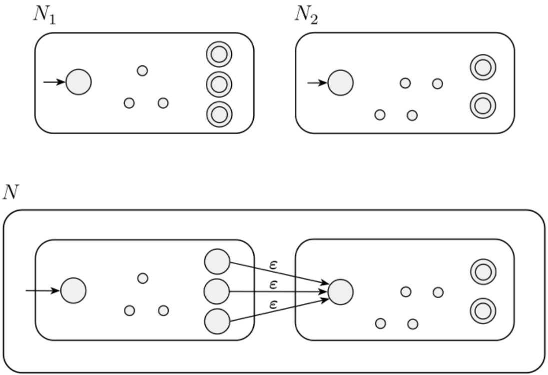
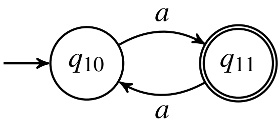


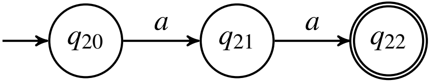
FIGURE 1.48
Construction of N to recognize $A_1 \circ A_2$

- $L_1 = \{$contains an odd number of a's$\}$
  $L_2 = \{aa\}$

N_1



N_2



Closure under the Regular Operations — *continued*

Sipser, 1.2, p-61

PROOF

- 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 $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.

- The states of N are all the states of N_1 and N_2.

Closure under the Regular Operations — *continued*

Sipser, 1.2, p-61

2. The state q_1 is the start state of N.

3. The set of accept states $F = F_2$.

- The accept states F are the same as the accept states of N_2.

4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$,

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

Closure under the Regular Operations

Sipser, 1.2, p-62

THEOREM 1.49

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

Closure under the Regular Operations — *continued*

Sipser, 1.2, p-62

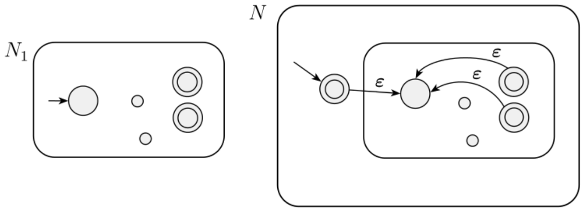
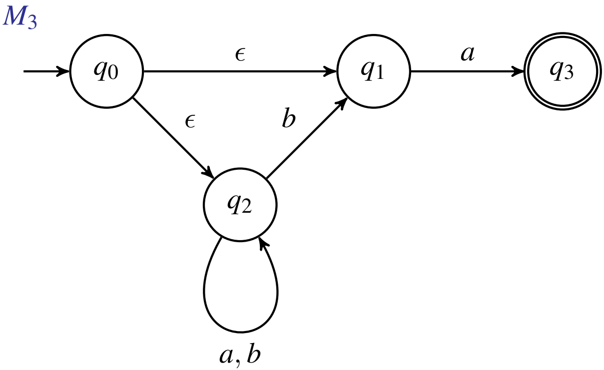


FIGURE 1.50
Construction of N to recognize A^*

- $\sum = \{a, b\}$, $L_3 = \{$ends in exactly one a at the end$\}$

- $\sum = \{a, b\}$, $L_3 = \{$ends in exactly one a at the end$\}$



PROOF

- Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.
- Construct $N = (Q, \Sigma, \delta, q_0, F)$ 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.

Closure under the Regular Operations — *continued*

Sipser, 1.2, p-62

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 $a \in \Sigma_\epsilon$,

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

Regular Expressions

Sipser, 1.3, p-63

- In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as $(5+3) \times 4$.
- Similarly, we can use the regular operations to build up expressions describing languages.
- These are called regular expressions.
- An example is:

$$(0 \cup 1)0^*$$

Regular Expressions

Sipser, 1.3, p-64

DEFINITION 1.52

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, or
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.

https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

- Stephen Cole Kleene (January 5, 1909 – January 25, 1994) was an American mathematician.

Kleene Star

Stephen Cole Kleene

`https://en.wikipedia.org/wiki/Stephen_Cole_Kleene` — *continued*

- He was one of the students of Alonzo Church.
- Kleene, along with Rózsa Péter, Alan Turing, Emil Post, and others is known as a founder of the branch of mathematical logic known as recursion theory.
- This subsequently helped to provide the foundations of theoretical computer science.
- Kleene's work grounds the study of which functions are computable.

- A number of mathematical concepts are named after him:
  - Kleene hierarchy,
  - Kleene algebra,
  - the Kleene star (Kleene closure),
  - Kleene's recursion theorem and
  - the Kleene fixpoint theorem.

- He also invented regular expressions, and made significant contributions to the foundations of mathematical intuitionism.

Kleene Star

https://en.wikipedia.org/wiki/Kleene_star

- In mathematical logic and computer science, the Kleene star (or Kleene operator or Kleene closure) is a unary operation, either on sets of strings or on sets of symbols or characters.
- The application of the Kleene star to a set V is written as V^*.
- It is widely used for regular expressions, which is the context in which it was introduced by Stephen Kleene to characterise certain automata, where it means "zero or more".

Kleene Star — *continued*

https://en.wikipedia.org/wiki/Kleene_star

- If V is a set of strings, then V^* is defined as the smallest superset of V that contains the empty string ϵ and is closed under the string concatenation operation.
- If V is a set of symbols or characters, then V^* is the set of all strings over symbols in V, including the empty string ϵ.

Kleene Star — *continued*

https://en.wikipedia.org/wiki/Kleene_star

- The set V^* can also be described as the set of finite-length strings that can be generated by concatenating arbitrary elements of V, allowing the use of the same element multiple times.

Kleene Star — *continued*

https://en.wikipedia.org/wiki/Kleene_star

- If V is either the empty set $\emptyset$ or the singleton set $\{\epsilon\}$, then $V^* = \{\epsilon\}$.
- If V is any other finite set, then V^* is a countably infinite set.

- Given a set V define,

  $V_0 = \{\epsilon\}$ (the language consisting only of the empty string),
  $V_1 = V$.

- And define recursively the set,
  $V_{i+1} = \{wv : w \in V_i \text{ and } v \in V\}$ for each $i > 0$.

Kleene Star — *continued*

https://en.wikipedia.org/wiki/Kleene_star

- If V is a formal language, then V_i, the i-th power of the set V, is a shorthand for the concatenation of set V with itself i times.
- That is, V_i can be understood to be the set of all strings that can be represented as the concatenation of i strings in V.
- The definition of Kleene star on V is

$$V^* = \bigcup_{i \in \mathcal{N}} V_i = \{\epsilon\} \cup V \cup V_2 \cup V_3 \cup V_4 \cup \ldots.$$

- In some formal language studies, a variation on the Kleene star operation called the Kleene plus is used.
- The Kleene plus omits the V_0 term in the union.
- In other words, the Kleene plus on V is,

$$V^+ = \bigcup_{i \in \mathcal{N} \setminus \{0\}} V_i = V_1 \cup V_2 \cup V_3 \cup \ldots.$$

Kleene Star — *continued*

Example https://en.wikipedia.org/wiki/Kleene_star

- Example of Kleene star applied to set of strings:

  $\{\texttt{ab},\ \texttt{c}\}^* = \{\epsilon,\ \texttt{ab},\ \texttt{c},\ \texttt{abab},\ \texttt{abc},\ \texttt{cab},\ \texttt{cc},\ \texttt{ababab},\ \texttt{ababc},\ \texttt{abcab},\ \texttt{abcc},\ \texttt{cabab},\ \texttt{cabc},\ \texttt{ccab},\ \texttt{ccc},\ \ldots\}$.

Kleene Star — *continued*

Example https://en.wikipedia.org/wiki/Kleene_star

- Example of Kleene star applied to set of characters:

  $\{\texttt{a, b, c}\}^* = \{\epsilon\texttt{, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, ...}\}$.

- Example of Kleene star applied to the empty set:

  $\emptyset^* =$

Kleene Star — *continued*

Example https://en.wikipedia.org/wiki/Kleene_star

- Example of Kleene star applied to the empty set:

$$\emptyset^* = \{\epsilon\}.$$

- Example of Kleene star applied to the empty set:

$$\emptyset^* = \{\epsilon\}.$$

- Example of Kleene plus applied to the empty set:

$$\emptyset^+ =$$

Kleene Star — *continued*

Example https://en.wikipedia.org/wiki/Kleene_star

- Example of Kleene star applied to the empty set:

$$\emptyset^* = \{\epsilon\}.$$

- Example of Kleene plus applied to the empty set:

$$\emptyset^+ = \emptyset\,\emptyset^* = \{\} = \emptyset.$$

Why is the Kleene star of a null set an empty string?

https://math.stackexchange.com/q/908192/62254

- How come a null set when taken zero times can give you an empty string?
- To begin with, a null set has only zero strings.
- But a set with an empty string has one string with zero length.

Why is the Kleene star of a null... — *continued*

https://math.stackexchange.com/q/908192/62254

- By definition, the strings in X^* (for any language X, whether $X = \emptyset$ or not) are those constructed by taking some finite number (possibly 0) of strings from X and concatenating them.
- If you take 0 strings and concatenate them, you get ϵ.

Why is the Kleene star of a null... — *continued*

https://math.stackexchange.com/q/908192/62254

- Note that this has nothing to do with whether $X = \emptyset$ or not.
- The empty string ϵ is *always* in X^* regardless of what X is.

Why is the Kleene star of a null... — *continued*

https://math.stackexchange.com/q/908192/62254

- When $X = \emptyset$, there are no other strings in X^*, because you cannot take more than 0 strings from $\emptyset$.
- So the only string in $\emptyset^*$ is ϵ.
- Thus $\emptyset^* = \{\epsilon\}$.

Why is the Kleene star of a null... — *continued*

https://math.stackexchange.com/q/908192/62254

- Now let X be some nonempty language, say $X = \{a\}$.
- Then $X^* = \{\epsilon, \texttt{ a, aa, aaa, aaaa,} \ldots\}$.
- Notice that ϵ is still in X^*.
- But now there are other strings because I can concatenate one or more a's together.
- The ϵ is what I get by concatenating zero a's.

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

1. 0^*10^*

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

1. 0^*10^*

   $\{w \mid w$ contains a single $1\}$.

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

1. 0^*10^*

   $\{w \mid w \text{ contains a single } 1\}$.
2. $\Sigma^*1\Sigma^*$

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

1. 0^*10^*

   $\{w \mid w$ contains a single 1$\}$.
2. $\Sigma^*1\Sigma^*$

   $\{w \mid w$ has at least one 1$\}$.

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

1. 0^*10^*

   $\{w \mid w$ contains a single 1$\}$.
2. $\Sigma^*1\Sigma^*$

   $\{w \mid w$ has at least one 1$\}$.
3. $\Sigma^*001\Sigma^*$

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

1. 0^*10^*

   $\{w \mid w$ contains a single 1$\}$.
2. $\Sigma^*1\Sigma^*$

   $\{w \mid w$ has at least one 1$\}$.
3. $\Sigma^*001\Sigma^*$

   $\{w \mid w$ contains the string 001 as a substring$\}$.

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

1. 0^*10^*

   $\{w \mid w$ contains a single 1$\}$.
2. $\Sigma^*1\Sigma^*$

   $\{w \mid w$ has at least one 1$\}$.
3. $\Sigma^*001\Sigma^*$

   $\{w \mid w$ contains the string 001 as a substring$\}$.
4. $1^*(01^+)^*$

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

1. 0^*10^*

   $\{w \mid w$ contains a single 1$\}$.
2. $\Sigma^*1\Sigma^*$

   $\{w \mid w$ has at least one 1$\}$.
3. $\Sigma^*001\Sigma^*$

   $\{w \mid w$ contains the string 001 as a substring$\}$.
4. $1^*(01^+)^*$

   $\{w \mid$ every 0 in w is followed by at least one 1$\}$.

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

1. 0^*10^*

   $\{w \mid w$ contains a single 1$\}$.
2. $\Sigma^*1\Sigma^*$

   $\{w \mid w$ has at least one 1$\}$.
3. $\Sigma^*001\Sigma^*$

   $\{w \mid w$ contains the string 001 as a substring$\}$.
4. $1^*(01^+)^*$

   $\{w \mid$ every 0 in w is followed by at least one 1$\}$.
5. $(\Sigma\Sigma)^*$

Example

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

1. 0^*10^*

   $\{w \mid w$ contains a single 1$\}$.
2. $\Sigma^*1\Sigma^*$

   $\{w \mid w$ has at least one 1$\}$.
3. $\Sigma^*001\Sigma^*$

   $\{w \mid w$ contains the string 001 as a substring$\}$.
4. $1^*(01^+)^*$

   $\{w \mid$ every 0 in w is followed by at least one 1$\}$.
5. $(\Sigma\Sigma)^*$

   $\{w \mid w$ is a string of even length$\}$.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

6. $(\Sigma\Sigma\Sigma)^*$

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

6. $(\Sigma\Sigma\Sigma)^*$

   $\{w \mid$ the length of w is a multiple of 3$\}$.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

6. $(\Sigma\Sigma\Sigma)^*$

   $\{w \mid$ the length of w is a multiple of 3$\}$.

7. $01 \cup 10$

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

6. $(\Sigma\Sigma\Sigma)^*$

   $\{w \mid$ the length of w is a multiple of 3$\}$.

7. $01 \cup 10$

   $\{01, 10\}$.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

6. $(\Sigma\Sigma\Sigma)^*$

   $\{w \mid$ the length of w is a multiple of 3$\}$.
7. $01 \cup 10$

   $\{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

6. $(\Sigma\Sigma\Sigma)^*$

   $\{w \mid$ the length of w is a multiple of 3$\}$.
7. $01 \cup 10$

   $\{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$

   $\{w \mid w$ starts and ends with the same symbol$\}$.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

6. $(\Sigma\Sigma\Sigma)^*$

   $\{w \mid$ the length of w is a multiple of 3$\}$.
7. $01 \cup 10$

   $\{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$

   $\{w \mid w$ starts and ends with the same symbol$\}$.
9. $(0 \cup \epsilon)1^* = 01^* \cup 1^*$

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

6. $(\Sigma\Sigma\Sigma)^*$

   $\{w \mid$ the length of w is a multiple of 3$\}$.
7. $01 \cup 10$

   $\{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$

   $\{w \mid w$ starts and ends with the same symbol$\}$.
9. $(0 \cup \epsilon)1^* = 01^* \cup 1^*$

   The expression $0 \cup \epsilon$ describes the language $\{0, \epsilon\}$, so the concatenation operation adds either 0 or ϵ before every string in 1^*.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

10. $(0 \cup \epsilon)(1 \cup \epsilon)$

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

10. $(0 \cup \epsilon)(1 \cup \epsilon)$

    $\{\epsilon, 0, 1, 01\}$.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

10. $(0 \cup \epsilon)(1 \cup \epsilon)$

    $\{\epsilon, 0, 1, 01\}$.

11. $1^*\emptyset = \emptyset$

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

10. $(0 \cup \epsilon)(1 \cup \epsilon)$

    $\{\epsilon, 0, 1, 01\}$.

11. $1^*\emptyset = \emptyset$

    Concatenating the empty set to any set yields the empty set.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

10. $(0 \cup \epsilon)(1 \cup \epsilon)$

    $\{\epsilon, 0, 1, 01\}$.

11. $1^*\emptyset = \emptyset$

    Concatenating the empty set to any set yields the empty set.

12. $\emptyset^* = \{\epsilon\}$

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

10. $(0 \cup \epsilon)(1 \cup \epsilon)$

    $\{\epsilon, 0, 1, 01\}$.

11. $1^*\emptyset = \emptyset$

    Concatenating the empty set to any set yields the empty set.

12. $\emptyset^* = \{\epsilon\}$

    The star operation puts together any number of strings from the language to get a string in the result.

Example — *continued*

Sipser, Example 1.53, p-65

- In the following instances, we assume that the alphabet Σ is $\{0, 1\}$.

---

10. $(0 \cup \epsilon)(1 \cup \epsilon)$

    $\{\epsilon, 0, 1, 01\}$.

11. $1^*\emptyset = \emptyset$

    Concatenating the empty set to any set yields the empty set.

12. $\emptyset^* = \{\epsilon\}$

    The star operation puts together any number of strings from the language to get a string in the result.

    If the language is empty, the star operation can put together 0 strings, giving only the empty string.

Example

Hopcroft, Motwani, and Ullman, Example 3.2, p-87

- Let us write a regular expression for the set of strings that consist of alternating 0's and 1's.
- First, let us develop a regular expression for the language consisting of the single string 01.
- We can then use the star operator to get an expression for all strings of the form $0101\ldots01$.

Example

Hopcroft, Motwani, and Ullman, Example 3.2, p-87

- Let us write a regular expression for the set of strings that consist of alternating 0's and 1's.
- First, let us develop a regular expression for the language consisting of the single string 01.
- We can then use the star operator to get an expression for all strings of the form $0101\ldots01$.

Example — *continued*

alternating 0's and 1's

- The basis rule for regular expressions tells us that 0 and 1 are expressions denoting the languages $\{0\}$ and $\{1\}$, respectively.
- If we concatenate the two expressions, we get a regular expression for the language $\{01\}$.
- This expression is 01.

Example — *continued*

alternating 0's and 1's

- As a general rule, if we want a regular expression for the language consisting of only the string w, we use w itself as the regular expression.

alternating 0's and 1's

- To get all strings consisting of zero or more occurrences of 01; we use the regular expression $(01)^*$.
- We first put parentheses around 01, to avoid confusing with the expression 01^*.
- Language of 01^* is all strings consisting of a 0 and any number of 1's.
- The star takes precedence over dot.
- Therefore the argument of the star is selected before performing any concatenations.

alternating 0's and 1's

- However, $L((01)^*)$ is not exactly the language that we want.
- It includes only those strings of alternating 0's and 1's that begin with 0 and end with 1.
- We also need to consider the possibility that there is a 1 at the beginning and/or a 0 at the end.

Example — *continued*

alternating 0's and 1's

- One approach is to construct three more regular expressions that handle the other three possibilities.

alternating 0's and 1's

- One approach is to construct three more regular expressions that handle the other three possibilities.
- $(10)^*$ represents those alternating strings that begin with 1 and end with 0.

Example — *continued*

alternating 0's and 1's

- One approach is to construct three more regular expressions that handle the other three possibilities.
- $(10)^*$ represents those alternating strings that begin with 1 and end with 0.
- $0(10)^*$ can be used for strings that both begin and end with 0.

Example — *continued*

alternating 0's and 1's

- One approach is to construct three more regular expressions that handle the other three possibilities.
- $(10)^*$ represents those alternating strings that begin with 1 and end with 0.
- $0(10)^*$ can be used for strings that both begin and end with 0.
- $1(01)^*$ serves for strings that begin and end with 1.

Example — *continued*

alternating 0's and 1's

- One approach is to construct three more regular expressions that handle the other three possibilities.
- $(10)^*$ represents those alternating strings that begin with 1 and end with 0.
- $0(10)^*$ can be used for strings that both begin and end with 0.
- $1(01)^*$ serves for strings that begin and end with 1.
- The entire regular expression is

$$(01)^* + (10)^* + 0(10)^* + 1(01)^*$$

Example — *continued*

alternating 0's and 1's

$$(01)^* + (10)^* + 0(10)^* + 1(01)^*$$

- Notice that we use the $+$ operator to take the union of the four languages that together give us all the strings with alternating 0's and 1's.

alternating 0's and 1's

- However, there is another approach that yields a regular expression that looks rather different and is also somewhat more succinct.
- Start again with the expression $(01)^*$.
- We can add an optional 1 at the beginning if we concatenate on the left with the expression $\epsilon + 1$.
- Likewise, we add an optional 0 at the end with the expression $\epsilon + 0$.
- For instance, using the definition of the $+$ operator:

$$L(\epsilon + 1) = L(\epsilon) \cup L(1) = \{\epsilon\} \cup \{1\} = \{\epsilon, 1\}$$

Example — *continued*

alternating 0's and 1's

$L(\epsilon + 1) = L(\epsilon) \cup L(1) = \{\epsilon\} \cup \{1\} = \{\epsilon, 1\}$

- If we concatenate this language with any other language L, the ϵ choice gives us all the strings in L.
- Choice 1 gives us $1w$ for every string w in L.
- Thus, another expression for the set of strings that alternate 0's and 1's is:

$$(\epsilon + 1)(01)^*(\epsilon + 0)$$

- Note that we need parentheses around each of the added expressions, to make sure the operators group properly.

Precedence of Regular-Expression Operators

Hopcroft, Motwani, and Ullman, 3.1.3, p-88

- Like other algebras, the regular-expression operators have an assumed order of "precedence,".
- Operators are associated with their operands in a particular order.
- We are familiar with the notion of precedence from ordinary arithmetic expressions.

- For instance, we know that $xy + z$ groups the product xy before the sum.
- It is equivalent to the parenthesized expression $(xy) + z$ and not to the expression $x(y + z)$.
- Similarly, we group two of the same operators from the left in arithmetic.
- So $x - y - z$ is equivalent to $(x - y) - z$, and not to $x - (y - z)$.

Precedence of . . . Operators — *continued*

Hopcroft, Motwani, and Ullman, 3.1.3, p-88

For regular expressions, the following is the order of precedence for the operators:

1. The star operator is of highest precedence.
   - That is, it applies only to the smallest sequence of symbols to its left that is a well-formed regular expression.

Precedence of . . . Operators — *continued*

Hopcroft, Motwani, and Ullman, 3.1.3, p-88

For regular expressions, the following is the order of precedence for the operators:

2. Next in precedence comes the concatenation or “dot” operator.
   - After grouping all stars to their operands, we group concatenation operators to their operands.
   - That is, all expressions that are juxtaposed (adjacent, with no intervening operator) are grouped together.

Precedence of . . . Operators — *continued*

Hopcroft, Motwani, and Ullman, 3.1.3, p-88

For regular expressions, the following is the order of precedence for the operators:

3. Finally, all unions ($+$ operators) are grouped with their operands.
   - Since union is also associative, it again matters little in which order consecutive unions are grouped.

- Sometimes we do not want the grouping in a regular expression to be as required by the precedence of the operators.
- If so, we are free to use parentheses to group operands exactly as we choose.
- In addition, there is never anything wrong with putting parentheses around operands that you want to group.
- Even if the desired grouping is implied by the rules of precedence.

- If we let R be any regular expression, we have the following identities.

Regular Expressions — *continued*

Sipser, 1.3, p-63

- $R \cup \emptyset = R$.
- Adding the empty language to any other language will not change it.

- $R \circ \epsilon = R$.
- 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.

- $R \cup \epsilon$ may not equal R.
- For example, if $R = 0$, then $L(R) = \{0\}$.
- But $L(R \cup \epsilon) = \{0, \epsilon\}$.

- $R \circ \emptyset$ may not equal R.
- For example, if $R = 0$, then $L(R) = \{0\}$.
- But $L(R \circ \emptyset) = \emptyset$.

Equivalence with Finite Automata

Sipser, 1.3, p-66

- Regular expressions and finite automata are equivalent in their descriptive power.
- This fact is surprising because finite automata and regular expressions superficially appear to be rather different.
- However, 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.

Equivalence with Finite Automata — *continued*

Sipser, 1.3, p-66

THEOREM 1.54

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

- This theorem has two directions.
- We state and prove each direction as a separate lemma.

LEMMA 1.55

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

PROOF IDEA

- Say that we have a regular expression R describing some language A.
- We show how to convert R into an NFA recognizing A.
- By Corollary 1.40, if an NFA recognizes A then A is regular.

LEMMA 1.55

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

PROOF

- Let's convert R into an NFA N.
- We consider the six cases in the formal definition of regular expressions.

Equivalence with Finite Automata — *continued*

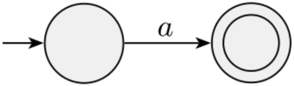
Sipser, 1.3, p-66

1. $R = a$ for some $a \in \Sigma$.

- Then $L(R) = \{a\}$.

1. $R = a$ for some $a \in \Sigma$.

- Then $L(R) = \{a\}$.

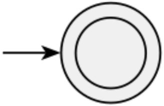


2. $R = \epsilon$.

- Then $L(R) = \{\epsilon\}$.

2. $R = \epsilon$.

- Then $L(R) = \{\epsilon\}$.



Equivalence with Finite Automata — *continued*

Sipser, 1.3, p-66

3. $R = \emptyset$.

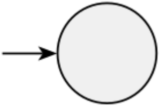
- Then $L(R) = \emptyset$.

Equivalence with Finite Automata — *continued*

Sipser, 1.3, p-66

3. $R = \emptyset$.

- Then $L(R) = \emptyset$.



Equivalence with Finite Automata — *continued*

Sipser, 1.3, p-66

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = {R_1}^*$.

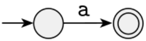
- We use the constructions given in the proofs that the class of regular languages is closed under the regular operations.
- In other words, we construct the NFA for R from the NFA's for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

Example

Sipser, Example 1.56, p-68

$(ab \cup a)^*$

a

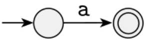


Example

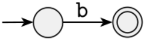
Sipser, Example 1.56, p-68

$(ab \cup a)^*$

a



b

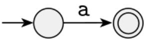


Example

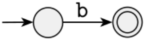
Sipser, Example 1.56, p-68

$(ab \cup a)^*$

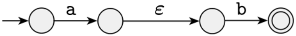
a



b



ab



Example

Sipser, Example 1.56, p-68

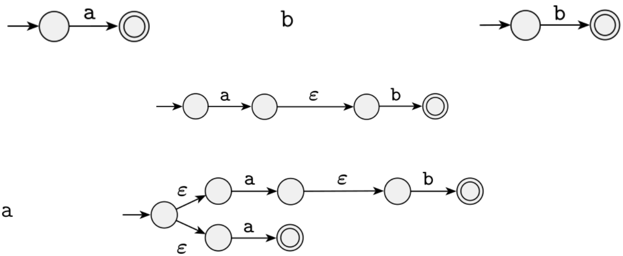
$(ab \cup a)^*$

a

b

ab

ab ∪ a

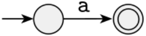


Example

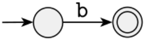
Sipser, Example 1.56, p-68

$(ab \cup a)^*$

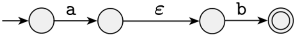
a



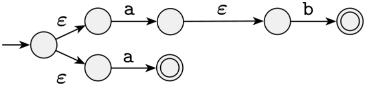
b



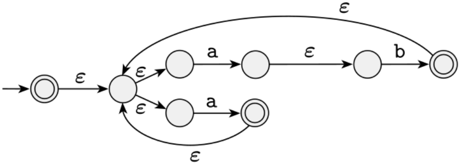
ab



ab ∪ a



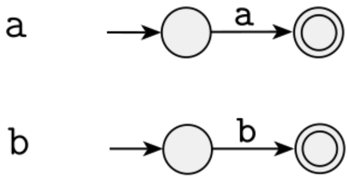
(ab ∪ a)*



Example

Sipser, Example 1.59, p-69

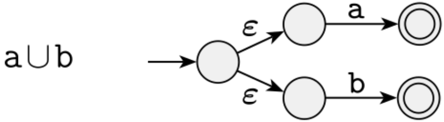
$(a \cup b)^* aba$



Example

Sipser, Example 1.59, p-69

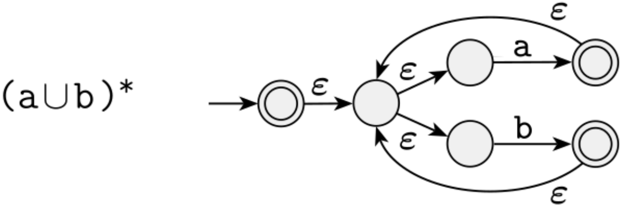
$(a \cup b)^*aba$



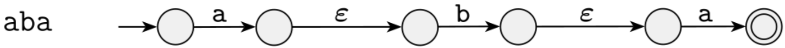
Example

Sipser, Example 1.59, p-69

$(a \cup b)^* aba$



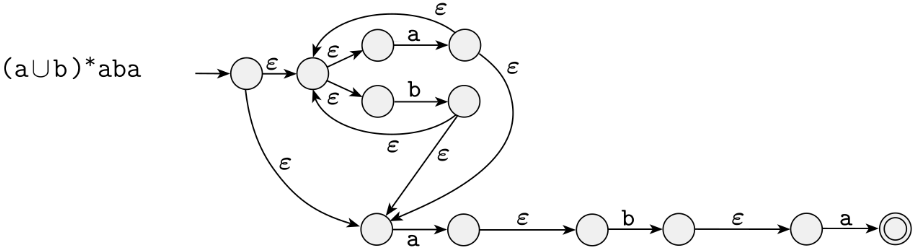
$(a \cup b)^* aba$



Example

Sipser, Example 1.59, p-69

$(a \cup b)^* aba$



LEMMA 1.60

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

Nonregular Languages

Sipser, 1.4, p-77

$B = \{0^n1^n \mid n \geq 0\}$

$C = \{w \mid w \text{ has an equal number of 0's and 1's}\}$

Nonregular Languages

Sipser, 1.4, p-77

$B = \{0^n 1^n \mid n \geq 0\}$

$C = \{w \mid w \text{ has an equal number of 0's and 1's}\}$

$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*
- ϵ

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*
- ϵ *belongs*

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*
- ϵ *belongs*
- 10

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*
- ϵ *belongs*
- 10 *does not belong*

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*
- ϵ *belongs*
- 10 *does not belong*
- 110

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 — *belongs*
- 01100 — *belongs*
- 1101110011 — *belongs*
- ϵ — *belongs*
- 10 — *does not belong*
- 110 — *does not belong*

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*
- ϵ *belongs*
- 10 *does not belong*
- 110 *does not belong*
- 1101

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \;\middle|\; \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 — *belongs*
- 01100 — *belongs*
- 1101110011 — *belongs*
- ϵ — *belongs*
- 10 — *does not belong*
- 110 — *does not belong*
- 1101 — *belongs*

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

- 0110 *belongs*
- 01100 *belongs*
- 1101110011 *belongs*
- ϵ *belongs*
- 10 *does not belong*
- 110 *does not belong*
- 1101 *belongs*

- w should *toggle* between 0 and 1 an equal number of times.

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

$$= \left\{ w \,\middle|\, \begin{array}{l} w = 1,\ w = 0,\ w = \epsilon \text{ or } w \text{ starts with a 0 and} \\ \text{ends with a 0 or } w \text{ starts with a 1 and ends} \\ \text{with a 1} \end{array} \right\}$$

Nonregular Languages

http://www.eecs.berkeley.edu/~sseshia/172/lectures/Slides3.pdf, Slide 31

$$D = \left\{ w \,\middle|\, \begin{array}{l} w \text{ has an equal number of occurrences of 01} \\ \text{and 10 as substrings} \end{array} \right\}$$

$$= \left\{ w \,\middle|\, \begin{array}{l} w = 1,\ w = 0,\ w = \epsilon \text{ or } w \text{ starts with a 0 and} \\ \text{ends with a 0 or } w \text{ starts with a 1 and ends} \\ \text{with a 1} \end{array} \right\}$$

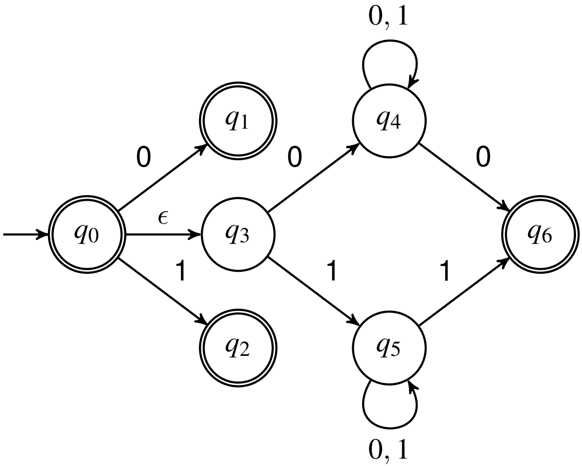
- $\epsilon \cup 0 \cup 1 \cup (0\Sigma^*0) \cup (1\Sigma^*1)$

Nonregular Languages

$\epsilon \cup 0 \cup 1 \cup (0\Sigma^*0) \cup (1\Sigma^*1)$

Nonregular Languages

$\epsilon \cup 0 \cup 1 \cup (0\Sigma^*0) \cup (1\Sigma^*1)$



The Pumping Lemma for Regular Languages

Sipser, 1.4, p-77

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where 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 $i \geq 0$, $xy^iz \in A$,
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.
- y^0 equals ϵ.

The Pumping Lemma for Regular Languages

Sipser, 1.4, p-77

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where 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 $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

- When s is divided into xyz, either x or z may be ϵ.
- But condition 2 says that $y \neq \epsilon$.
- Without condition 2 the theorem would be trivially true.

The Pumping Lemma for Regular Languages

Sipser, 1.4, p-77

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where 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 $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

- Condition 3 states that the pieces x and y together have length at most p.
- It is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular.

PROOF IDEA

- 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 our three conditions.

The Pumping Lemma for Regular Languages — *continued*

Sipser, 1.4, p-77

- What if no strings in A are of length at least p?
- Then our task is even easier because the theorem becomes vacuously true.
- Obviously the three conditions hold for all strings of length at least p if there aren't any such strings.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-77

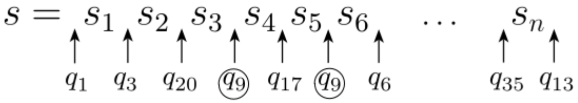


FIGURE 1.71
Example showing state q_9 repeating when M reads s

- If s in A has length at least p, consider the sequence of states that M goes through when computing with input s.
- It starts with q_1 the start state, then goes to, say, q_3, then, say, q_{20}, then q_9, and so on, until it reaches the end of s in state q_{13}.

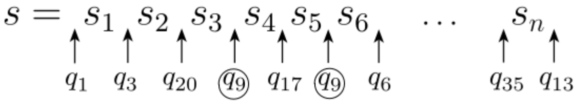


FIGURE 1.71
Example showing state q_9 repeating when M reads s

- 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 $q_1, q_3, q_{20}, q_9, \dots, q_{13}$ has length $n + 1$.

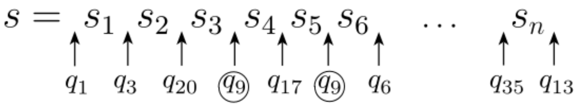


FIGURE 1.71

Example showing state q_9 repeating when M reads s

- 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.
- This result is an example of the pigeonhole principle.

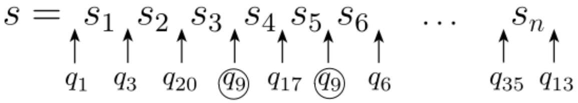


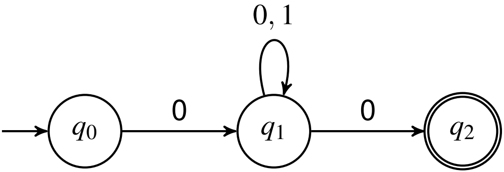
FIGURE 1.71
Example showing state q_9 repeating when M reads s

- State q_9 is the one that repeats.

The Pumping Lemma for Regular Languages

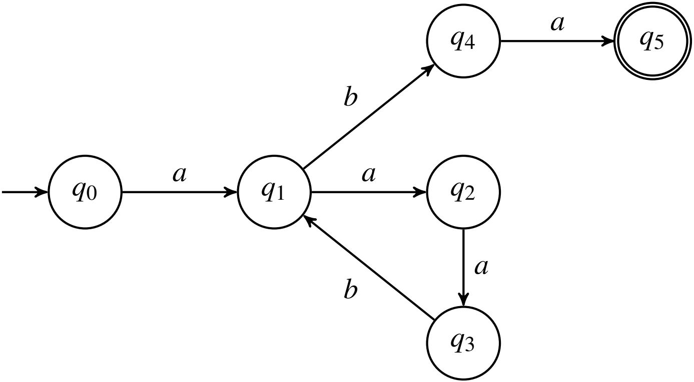
$L = \{w \mid w \text{ starts and ends with 0}, |w| \geq 2\}$

$L = 0\Sigma^*0$



The Pumping Lemma for Regular Languages

$L = a(aab)^*ba$



The Pumping Lemma... — *continued*

Sipser, 1.4, p-77

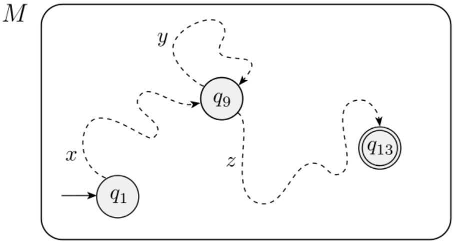


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- 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, coming after the second occurrence of q_9.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-77

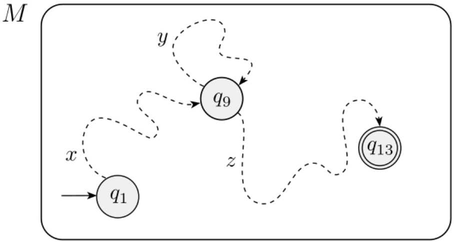


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- x takes M from the state q_1 to q_9.
- y takes M from q_9 back to q_9.
- z takes M from q_9 to the accept state q_{13}.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-77

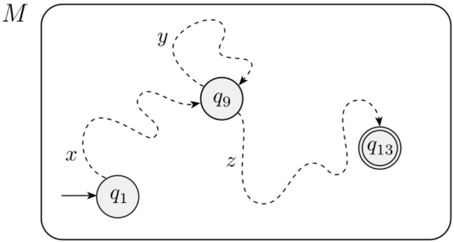


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- Suppose that we run M on input $xyyz$.
- We know that x takes M from q_1 to q_9.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-77

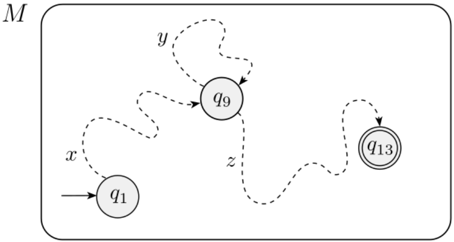


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- Then the first y takes it from q_9 back to q_9, as does the second y.
- Then z takes it to q_{13}.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-77

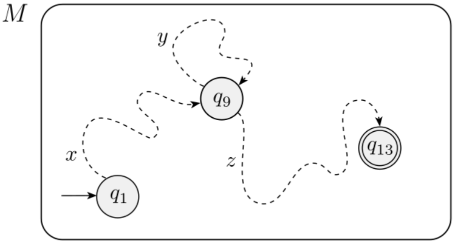


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- With q_{13} being an accept state, M accepts input $xyyz$.
- Similarly, it will accept xy^iz for any $i > 0$.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-77

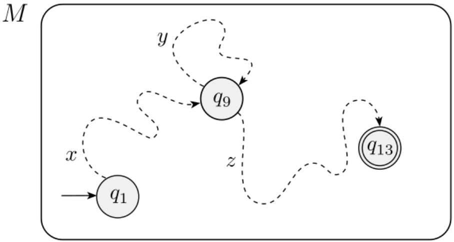


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- For the case $i = 0$, $xy^iz = xz$, which is accepted for similar reasons.
- That establishes condition 1.

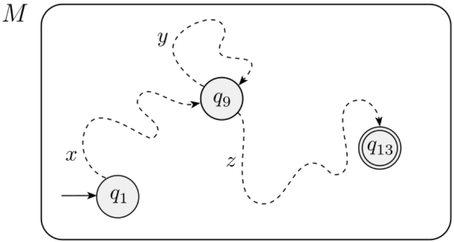


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- Checking condition 2, we see that $|y| > 0$, as it was the part of s that occurred between two different occurrences of state q_9.

The Pumping Lemma… — *continued*

Sipser, 1.4, p-77

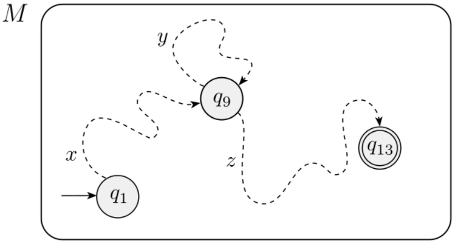


FIGURE 1.72
Example showing how the strings x, y, and z affect M

- In order to get condition 3, we make sure that q_9 is the first repetition in the sequence.
- By the pigeonhole principle, the first $p + 1$ states in the sequence must contain a repetition.
- Therefore, $|xy| \leq p$.

PROOF

- Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognizing A and p be the number of states of M.
- Let $s = s_1 s_2 \ldots s_n$ be a string in A of length n, where $n \geq p$.
- Let $r_1, r_2, \ldots, r_{n+1}$ be the sequence of states that M enters while processing s.
- So $r_{i+1} = \delta(r_i, s_i)$ 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 $\ell \le p+1$.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-77

Let,

- $x = s_1 \ldots s_{j-1}$.
- $y = s_j \ldots s_{\ell-1}$.
- $z = s_\ell \ldots s_n$.

Let,

- $x = s_1 \dots s_{j-1}$.
- $y = s_j \dots s_{\ell-1}$.
- $z = s_\ell \dots s_n$.
- x takes M from r_1 to r_j.
- y takes M from r_j to r_ℓ.
- z takes M from r_ℓ to r_{n+1}, which is an accept state, M must accept xy^iz 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 conditions of the pumping lemma.

A Pumping Lemma

Peter Linz, 4.3

- We have given the pumping lemma only for infinite languages.
- Finite languages, although always regular, cannot be pumped since pumping automatically creates an infinite set.
- The theorem does hold for finite languages, but it is vacuous.
- The p in the pumping lemma becomes larger than the longest string, so that no string can be pumped.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-80

- To use the pumping lemma to prove that a language B is not regular, first assume that B is regular in order to obtain a contradiction.
- Then 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.

- Next, find a string s in B that has length p or greater but that cannot be pumped.

- Finally, demonstrate that s cannot be pumped by considering all ways of dividing s into x, y, and z (taking condition 3 of the pumping lemma into account if convenient).
- For each such division, find a value i where $xy^iz \notin B$.

- This final 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.
- Hence B cannot be regular.

The Pumping Lemma. . . — *continued*

Sipser, 1.4, p-80

- Finding s sometimes takes a bit of creative thinking.
- You may need to hunt through several candidates for s before you discover one that works.
- Try members of B that seem to exhibit the “essence” of B’s nonregularity.

- If you are still uncomfortable on this topic, you may want to watch this presentation:

  Pumping Lemma For Regular by Didem Yalcin

Example

Sipser, Example 1.73, p-80

- Let B be the language $\{0^n 1^n \mid n \geq 0\}$.
- We use the pumping lemma to prove that B is not regular.
- The proof is by contradiction.

- Assume to the contrary that B is regular.
- Let p be the pumping length given by the pumping lemma.

Example — *continued*

Sipser, Example 1.73, p-80

- 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, $s = xyz$.
- Where 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, violating condition 1 of the pumping lemma.
- This case is a contradiction.

Example — *continued*

Sipser, Example 1.73, p-80

- We consider three cases to show that this result is impossible.

---

2. The string y consists only of 1s.

- This case also gives a contradiction.

Example — *continued*

Sipser, Example 1.73, p-80

- We consider three cases to show that this result is impossible.

---

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, which is a contradiction.

Example — *continued*

Sipser, Example 1.73, p-80

- Thus a contradiction is unavoidable if we make the assumption that B is regular.
- So B is not regular.

Example — *continued*

Sipser, Example 1.73, p-80

- Note that we can simplify this argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3.

Example

Sipser, Example 1.74, p-80

- $C = \{w \mid w$ has an equal number of 0s and 1s$\}$.
- We use the pumping lemma to prove that C is not regular.
- The proof is by contradiction.

Example — *continued*

Sipser, Example 1.74, p-80

- Assume to the contrary that C is regular.
- Let p be the pumping length given by the pumping lemma.
- 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.

Example — *continued*

Sipser, Example 1.74, p-80

- We would like to show that this outcome is impossible.

Example — *continued*

Sipser, Example 1.74, p-80

- But wait, it is possible!
- 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.

Example — *continued*

Sipser, Example 1.74, p-80

- Here condition 3 in the pumping lemma is useful.
- It stipulates that when pumping s, it must be divided so that $|xy| \leq p$.
- That restriction on the way that s may be divided makes it easier to show that the string $s = 0^p 1^p$ we selected cannot be pumped.
- If $|xy| \leq p$, then y must consist only of 0s, so $xyyz \notin C$.
- Therefore, s cannot be pumped.
- That gives us the desired contradiction.

Example

Sipser, Example 1.75, p-81

- $F = \{ww \mid w \in \{0,1\}^*\}$.
- We use the pumping lemma to prove that F is not regular.

Example — *continued*

Sipser, Example 1.75, p-81

- Assume to the contrary that F is regular.
- Let p be the pumping length given by the pumping lemma.
- 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.

Example — *continued*

Sipser, Example 1.75, p-81

- We show that this outcome is impossible.
- Condition 3 is once again 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 0's, 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.

Example

Sipser, Example 1.76, p-82

- We demonstrate a nonregular unary language.
- $D = \left\{ 1^{n^2} \mid n \geq 0 \right\}$.
- We use the pumping lemma to prove that D is not regular.
- The proof is by contradiction.

- Assume to the contrary that D is regular.
- Let p be the pumping length given by the pumping lemma.

- 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.

Example — *continued*

Sipser, Example 1.76, p-82

- We show that this outcome is impossible.
- The sequence of perfect squares:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

- Note the growing gap between successive members of this sequence.
- Large members of this sequence cannot be near each other.

Example — *continued*

Sipser, Example 1.76, p-82

- Now consider the two strings xyz and xy^2z.
- These strings differ from each other by a single repetition of y.
- Consequently their lengths differ by the length of y.
- By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$.

Example — *continued*

Sipser, Example 1.76, p-82

- We have $|xyz| = p^2$ and 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 this length cannot be a perfect square itself.
- So we arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular.

Example

Sipser, Example 1.77, p-82

- Let E be the language $\{0^i 1^j \mid i > j\}$.
- We use the pumping lemma to prove that E is not regular.
- The proof is by contradiction.

Example — *continued*

Sipser, Example 1.77, p-82

- Assume that E is regular.
- Let p be the pumping length for E given by the pumping lemma.

Example — *continued*

Sipser, Example 1.77, p-82

- 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.

Example — *continued*

Sipser, Example 1.77, p-82

- Let's examine the string $xyyz$ to see whether it can be in E.
- 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 need to try something else.
- The pumping lemma states that $xy^iz \in E$ even when $i = 0$.

Example — *continued*

Sipser, Example 1.77, p-82

- So let's 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.

Example

Hopcroft, Motwani, and Ullman, Example 4.3, p-129

- Let us show that the language L_{pr} consisting of all strings of 1's whose length is a prime is not a regular language.
- Suppose it were.
- Then there would be a constant p satisfying the conditions of the pumping Lemma.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 4.3, p-129

- Consider some prime $n \geq p+2$.
- There must be such an n, since there are an infinity of primes.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 4.3, p-129

- Let $w = 1^n$.
- By the pumping lemma, we can break $w = xyz$ such that $y \neq \epsilon$ and $|xy| \leq p$.
- Let $|y| = m$.
- Then $|xz| = n - m$.

- Now consider the string $xy^{n-m}z$.
- This must be in L_{pr} by the pumping lemma, if L_{pr} really is regular.
- However,

$$\begin{aligned} |xy^{n-m}z| &= |xz| + (n-m)|y| \\ &= n - m + (n-m)m \\ &= (m+1)(n-m) \end{aligned}$$

- It looks like $|xy^{n-m}z|$ is not a prime, since it has two factors $(m+1)$ and $(n-m)$.

- However, we must check that neither of these factors are 1.
- Since then $(m+1)(n-m)$ might be a prime after all.
- But $m+1>1$, since $y \neq \epsilon$ tells us $m \geq 1$.
- Also, $n-m \geq 1$, since $n \geq p+2$ was chosen, and $m \leq p$ since

$$m = |y| \leq |xy| \leq p$$

- Thus, $n-m \geq 2$.

Example — *continued*

Hopcroft, Motwani, and Ullman, Example 4.3, p-129

- Again we have started by assuming the language in question was regular.
- We derived a contradiction by showing that some string not in the language was required by the pumping lemma to be in the language.
- Thus, we conclude that L_{pr} is not a regular language.

Example

Peter Linz, Example 4.9

- $\Sigma = \{a, b\}$.
- The language $L = \{w \in \Sigma^* \mid n_a(w) < n_b(w)\}$ is not regular.

- We pick $w = a^p b^{p+1}$.
- Now, because $|xy|$ cannot be greater than p, y will be all a's.
- That is $y = a^k$, $1 \leq k \leq p$.

Example — *continued*

Peter Linz, Example 4.9

- We now pump up, using $i = m$
- The resulting string, $a^{p+mk}b^{p+1}$, $1 \leq m$, is not in L.
- Therefore, the pumping lemma is violated.
- L is not regular.

Example

Peter Linz, Example 4.10

- The language $L = \{(ab)^n a^k \mid n > k, k \geq 0\}$ is not regular.
- We pick as our string $w = (ab)^{p+1}a^p$, which is in L.

Example — *continued*

Peter Linz, Example 4.10

- Because of the constraint $|xy| \leq p$, both x and y must be in the part of the string made up of ab's.
- The choice of x does not affect the argument, so let us see what can be done with y.
- For $y = a$, we choose $i = 0$ and get a string $((ab)^* a^*)$ not in L.

Example — *continued*

Peter Linz, Example 4.10

- If we pick $y = ab$, we can choose $i = 0$ again.
- Now we get the string $(ab)^p a^p$, which is not in L.
- In the same way, we can deal with any possible choice of y, thereby proving our claim.

Example

Peter Linz, Example 4.13

- Show that the language $L = \{a^n b^l \mid n \neq l\}$ is not regular.

Example — *continued*

Peter Linz, Example 4.13

- Here we need a bit of ingenuity to apply the pumping lemma directly.
- Choosing a string with, $l = p + 1$, or, $l = p + 2$, will not do.
- We can always choose a decomposition that will make it impossible to pump the string out of the language.
- That is, we can not pump it so that it does not always have an unequal number of a's and b's.

- We must be more inventive.
- Let us take $n = p!$ and $l = (p+1)!$.
- We now choose y (by necessity consisting of all a's) of length $0 < k \leq p$.
- In the string xy^iz, we will have $(p! + (i-1)k)$ a's.
- We can get a contradiction of the pumping lemma if we can pick i such that $p! + (i-1)k = (p+1)!$

Example — *continued*

Peter Linz, Example 4.13

- $$\begin{aligned} p! + (i-1)k &= (p+1)!, \\ (i-1)k &= (p+1)! - p!, \\ &= (p+1)p! - p!, \\ &= p!(p+1-1), \\ ik - k &= p!p, \\ i &= 1 + \frac{p!p}{k}. \end{aligned}$$
- This is always possible since $k \leq p$.
- The right side is therefore an integer.
- We have succeeded in violating the conditions of the pumping lemma.

Example — *continued*

Peter Linz, Example 4.13, comments added by the instructor

What is wrong with starting with a string like $a^{p+1}b^p$?

- This string has a length $\geq p$.
- Also, it belongs to the given language, $L = \{a^n b^l \mid n \neq l\}$.
- Then, what is wrong with taking, $y = a$, pumping it out, and getting a string, $a^p b^p \notin L$?
- Hence, show that L is not regular.

Example — *continued*

Peter Linz, Example 4.13, comments added by the instructor

- The problem is, we have not tried *all* possible y's, as we should always do.

- If we take, $y = aa$, we will observe that, xy^iz will always belong to L, for all values of i.

Example — *continued*

Peter Linz, Example 4.13, comments added by the instructor

- So, our string was *not* chosen properly.

A Pumping Lemma

Peter Linz, 4.3 (adapted)

- The pumping lemma is difficult to understand.
- It is easy to go astray when applying it.
- Here are some common pitfalls.
- Watch out for them.

A Pumping Lemma — *continued*

Peter Linz, 4.3 (adapted)

- One mistake is to try using the pumping lemma to show that a language is regular.
- Even if you can show that all possible strings in L always obey the pumping lemma, you cannot conclude that L is regular.
- The pumping lemma can *only* be used to prove that a language is not regular.

A Pumping Lemma — *continued*

Peter Linz, 4.3 (adapted)

- Another mistake is to start (usually inadvertently) with a string not in L.
- For example, suppose we try to show that $L = \{a^n \mid \; n \text{ is a prime number}\}$ is not regular.
- An argument starts with "Given p, let $w = a^p \ldots$".
- This is incorrect since p is not necessarily prime.

A Pumping Lemma — *continued*

Peter Linz, 4.3 (adapted)

- To avoid this pitfall, we need to start with something like, “Given p, let $w = a^M$, where M is a prime number larger than p.”

A Pumping Lemma — *continued*

Peter Linz, 4.3 (adapted)

- Finally, perhaps the most common mistake is to make some assumptions about the decomposition xyz.
- The only thing we can say about the decomposition is what the pumping lemma tells us, namely,
  - that y is not empty and
  - that $|xy| \leq p$, that is, that y must be within p symbols of the left end of the string.
- Anything else makes the argument invalid.

A Pumping Lemma — *continued*

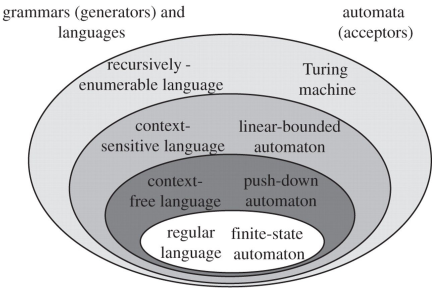
Peter Linz, 4.3 (adapted)

- But even if you master the technical difficulties of the pumping lemma, it may still be hard to see exactly how to use it.
- The pumping lemma is like a game with complicated rules.

A Pumping Lemma — *continued*

Peter Linz, 4.3 (adapted)

- Knowledge of the rules is essential, but that alone is not enough to play a good game.
- You also need a good strategy to win.



End of Slides