3. REGULAR LANGUAGES AND FINITE AUTOMATA

3.1 Regular Languages and Regular Expressions

- A *regular* language over an alphabet Σ is one that can be obtained from certain basic languages using the operations of union, concatenation, and Kleene *.

- The basic languages are the ones of the form $\{a\}$, plus the empty language $\varnothing$ and the language $\{\Lambda\}$.

- A *regular expression* is a formula that describes a regular language.

  In a regular expression, braces are left out or replaced by parentheses.
  Union is represented by the + symbol.

- Examples of regular languages over the alphabet {0, 1} and corresponding regular expressions:

Language	***Corresponding Regular Expression***
1. $\{\Lambda\}$	Λ
2. {0}	0
3. {001} (i.e., {0}{0}{1})	001
4. {0, 1} (i.e., {0} $\cup$ {1})	0 + 1
5. {0, 10} (i.e., {0} $\cup$ {10})	0 + 10
6. {1, Λ}{001}	(1 + Λ)001
7. {110}*{0, l}	(110)*(0 + 1)
8. {1}*{10}	1*10
9. {10, 111, 11010}*	(10 + 111 + 11010)*
10. {0, 10}*({11}* $\cup$ {001, Λ})	(0 + 10)* ((11)* + 001 + Λ)

A Formal Definition of Regular Languages and Regular Expressions

- The set of regular languages, along with the regular expressions that represent them, can be defined formally using recursion.

- **Definition 3.1:** The set R of regular languages over Σ, and the corresponding regular expressions, are defined as follows.

  1. $\varnothing$ is an element of R, and the corresponding regular expression is $\varnothing$.
  2. $\{\Lambda\}$ is an element of R, and the corresponding regular expression is Λ.
  3. For each $a \in \Sigma$, $\{a\}$ is an element of R, and the corresponding regular expression is a.
  4. If L_1 and L_2 are any elements of R, and r_1 and r_2 are the corresponding regular expressions,
     (a) $L_1 \cup L_2$ is an element of R, and the corresponding regular expression is $(r_1 + r_2)$;
     (b) L_1L_2 is an element of R, and the corresponding regular expression is (r_1r_2);
     (c) L_1^* is an element of R, and the corresponding regular expression is (r_1^*).

  Only those languages that can be obtained by using statements 1–4 are regular languages over Σ.

Simplifying Regular Expressions

- Shortcuts are often used to reduce the size of regular expressions. Some common abbreviations:

  (r^2) stands for (rr)
  (r^+) stands for $((r^*)r)$

- To reduce the number of parentheses in regular expressions, operators will have the following precedence:

  Highest: Kleene *
  Concatenation
  Lowest: +

  For example, $a + b^*c$ means $(a + ((b^*)c))$.

- Two regular expressions are considered to be equal if they represent the same language and not equal otherwise.

  $a + b^*c = (a + ((b^*)c))$
  $(a + b)^* \neq a + b^*$

- Some regular expression identities:

  $1^*(1 + \Lambda) = 1^*$
  $1^*1^* = 1^*$
  $0^* + 1^* = 1^* + 0^*$
  $(0^*1^*)^* = (0 + 1)^*$
  $(0 + 1)^*01(0 + 1)^* + 1^*0^* = (0 + 1)^*$

  All five are actually special cases of more general rules. For example, for any two regular expressions r and s, $(r^*s^*)^* = (r + s)^*$.

Examples of Regular Expressions

- **Example 3.1:** Strings of Even Length

  Let $L \subseteq \{0, 1\}^*$ be the language of all strings of even length. (Since 0 is even, $\Lambda \in L$.) Regular expressions that correspond to L:

  $(00 + 01 + 10 + 11)^*$
  $((0 + 1)(0 + 1))^*$

- **Example 3.2:** Strings with an Odd Number of l's

  Let L be the language of all strings of 0's and 1's containing an odd number of 1's. Regular expressions that correspond to L:

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

  There is not necessarily a simplest or most natural regular expression for this language (or any other).

- **Example 3.3:** Strings of Length 6 or Less

  Let L be the set of all strings over $\{0, 1\}$ of length 6 or less. A simple but inelegant regular expression corresponding to L:

  $\Lambda + 0 + 1 + 00 + 01 + 10 + 11 + 000 + \ldots + 111110 + 111111$

  A more concise regular expression corresponding to L:

  $(0 + 1 + \Lambda)^6$

Examples of Regular Expressions (Continued)

- **Example 3.4:** Strings Ending in 1 and Not Containing 00

  Let L be the language

  $L = \{x \in \{0, 1\}^* \mid x$ ends with 1 and does not contain the substring 00$\}$

  Saying that a string does not contain the substring 00 is the same as saying that every 0 comes at the end or is followed immediately by 1. Adding the requirement that every string ends with 1 leads to the expression $(1 + 01)^* (1 + 01)$, or $(1 + 01)^+$.

- **Example 3.5:** The Language of C Identifiers

  Let l (for "letter") denote the regular expression

  a + b + …+ z + A + B + …+ Z

  and let d (for "digit") stand for

  0 + 1 + 2 + … + 9

  An identifier in the C programming language is any string of length 1 or more that contains only letters, digits, and underscores (_) and begins with a letter or an underscore. Therefore, a regular expression for the language of all C identifiers is $(l + \_)(l + d + \_)^*$.

- **Example 3.6:** Real Literals in Pascal

  The following regular expression represents real (i.e., floating-point) literals in Pascal:

  $sd^+(pd^+ + pd^+\mathrm{E}sd^+ + \mathrm{E}sd^+)$

  s (for "sign") represents the regular expression $\Lambda + a + m$, where a is "plus" and m is "minus." p stands for "point," and E is a symbol in the alphabet.

3.2 The Memory Required to Recognize a Language

- Consider the problem of *recognizing* a language (deciding whether an arbitrary input string is in the language) subject to the following restrictions:

  Only one pass through the input (from left to right) is allowed.
  After each input symbol is read, a decision is made concerning the string seen so far.

- The question to be answered for a particular language is how much information must be remembered at each step. Two extremes:

  All input symbols seen so far must be remembered.
  Nothing about the input symbols seen so far must be remembered (as in the case of recognizing Σ^*).

The Memory Required to Recognize a Language (Continued)

- **Example 3.7:** Strings Ending with 0

  Let $L = \{0, 1\}^*\{0\}$. Recognizing strings in L requires nothing more than remembering the last symbol seen (0 or 1).

- **Example 3.8:** Strings with Next-to-Last Symbol 0

  Let L be the language of all strings in $\{0, 1\}^*$ with next-to-last symbol 0. To recognize L, it is necessary to remember the last two input symbols read. (It is not enough to remember just the next-to-last input symbol.) There are four possible combinations of two input symbols: 11, 10, 00, and 01.

  The strings Λ, 0, and 1 must be considered separately. As it turns out, Λ and 1 can be treated in the same way as 11, and 0 can be treated in the same way as 10, leading to four cases:

  a. The string is Λ or l or ends with 11.
  b. The string is 0 or ends with 10.
  c. The string ends with 00.
  d. The string ends with 01.

- **Example 3.9:** Strings Ending with 11

  Let $L = \{0, 1\}^*\{11\}$. It is not necessary to remember the last two input symbols. Instead, it is enough to remember one of three cases:

  a. The string does not end in 1. (Either it is Λ or it ends in 0.)
  b. The string is 1 or ends in 01.
  c. The string ends in 11.

The Memory Required to Recognize a Language (Continued)

- **Example 3.10:** Strings with an Even Number of 0's and an Odd Number of 1's

  Consider the language of strings x in $\{0, 1\}^*$ for which $n_0(x)$ is even and $n_1(x)$ is odd. Remembering the number of 0's and 1's read so far would require an unlimited amount of memory. Fortunately, all that is necessary is to remember remember the parity (i.e., even or odd) of both the number of 0's and the number of 1's.

- **Example 3.11:** A Recognition Algorithm for the Language in Example 3.4

  Let L be the language

  $L = \{x \in \{0, 1\}^* \mid x \text{ ends with 1 and does not contain the substring } 00\}$

  There are four cases. One of them occurs when the input seen so far includes 00 as a substring (call this case N). The remaining cases occur when 00 has not yet been seen:

  The last symbol seen was 0 (case 0).
  The last symbol seen was 1 (case 1).
  No symbols have been seen yet (case Λ).

Using Diagrams to Recognize Languages

- If a language can be recognized by remembering one of a finite set of cases, the algorithm for recognizing the language can be represented by a diagram. The diagram is a sort of flowchart, indicating how to move from case to case as each input symbol is read.

- A diagram of this type consists of circles (representing cases) connected by arrows (representing transitions from one case to another based on a particular input symbol).

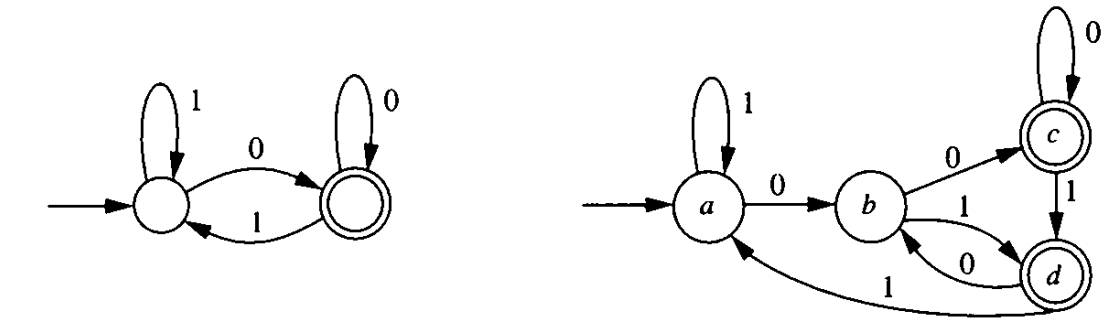
- One of the circles has arrow that does not come from another circle. This circle is the starting point of the algorithm.

- Double circles indicate cases in which the string seen so far is an element of the language.

- Each circle has a label. The labels are arbitrary, but they are often chosen to signify the meaning of a particular case.

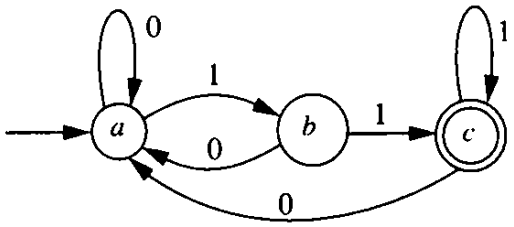
Examples of Diagrams for Recognizing Languages

- Diagrams that represent algorithms for recognizing the languages in Examples 3.7–3.11:

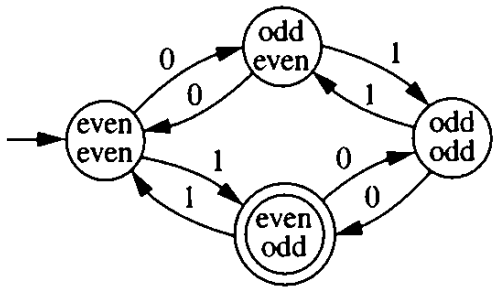


(*a*)

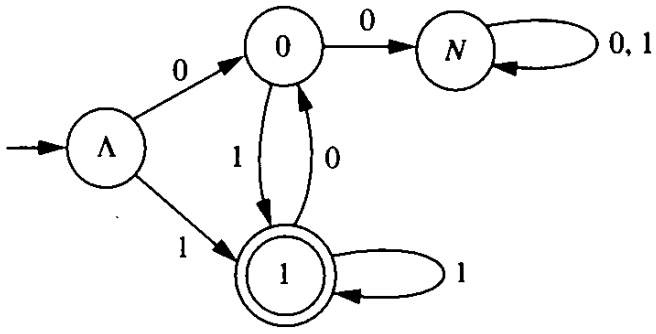
(*b*)



(*c*)



(*d*)



(*e*)

(a) Strings ending in 0
(b) Strings with next-to-last symbol 0
(c) Strings ending with 11
(d) Strings with n_0 even and n_1 odd
(e) Strings that end with 1 and do not contain the substring 00

Abstract Machines

- The diagram for recognizing a particular language L can be thought of as specifying an *abstract machine*.

  Such a machine is at any time in one of a finite number of possible *states.*
  The machine receives successive *inputs*, and as a result of being in a certain state and receiving a certain input, it moves to the state specified by the corresponding arrow.
  Certain states are *accepting* states. A string is in L if and only if the state the machine is in as a result of processing that string is an accepting state.

- An abstract machine could be implemented in hardware or software if desired.

- The heart of an abstract machine is the set of states and the function that specifies, for each combination of state and input symbol, the machine's next state.

- The crucial property is the finiteness of the set of states. Being able to distinguish between these states is the only form of memory the machine has.

- The more states a machine of this type has, the more complicated a language it will be able to recognize.

- Many languages cannot be recognized by this type of abstract machine. Chapter 4 shows that the languages that can be recognized in this way are precisely the regular languages.

3.3 Finite Automata

- The simple type of language-recognizing machine described in Section 3.2 is known as a *finite automaton*.

- **Definition 3.2:** A *finite automaton,* or *finite-state machine* (abbreviated FA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where

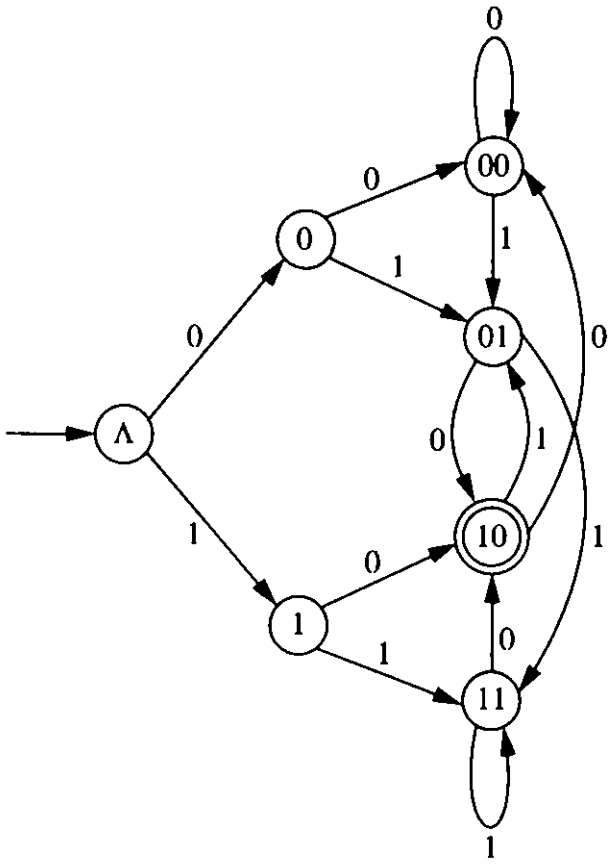
  Q is a finite set of *states*;
  Σ is a finite alphabet of *input symbols*;
  $q_0 \in Q$ (the *initial* state);
  $A \subseteq Q$ (the set of *accepting* states);
  δ is a function from $Q \times \Sigma$ to Q (the *transition* function).

  For any $q \in Q$ and any $a \in \Sigma$, $\delta(q, a)$ is the state to which the FA moves if it is in state q and receives the input a.

An Example of a Finite Automaton

- **Example 3.12:** Strings Ending with 10

  The transition diagram for an FA that recognizes $L = \{0, 1\}^*\{10\}$, the language of all strings in $\{0, 1\}^*$ ending in 10.



  Until it gets more than two inputs, the FA remembers exactly what it has received. After that, it cycles back and forth among four states, "remembering" the last two symbols it has received.

An Example of a Finite Automaton (Continued)

A transition table for the FA that recognizes $L = \{0, 1\}^*\{10\}$:

		input	
		0	**1**
state	Λ	0	1
	0	00	01
	1	10	11
	00	00	01
	01	10	11
	10	00	01
	11	10	11

The rows for the three states 1, 01, and 11 in the transition table are exactly the same, suggesting that these states can be merged into a single state named *B*.

The rows for the three states 0, 00, and 10 are also identical. The two nonaccepting states (0 and 00) can be merged into a new state named *A*.

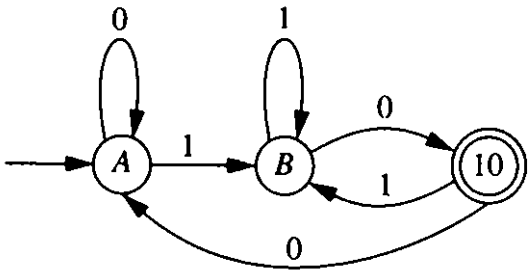
At this point, the number of states has been reduced to 4:

		input	
		0	**1**
state	Λ	*A*	*B*
	A	*A*	*B*
	B	10	*B*
	10	*A*	*B*

In this new FA, the rows for states Λ and *A* are the same, and neither Λ nor *A* is an accepting state. As a result, these states can be merged.

An Example of a Finite Automaton (Continued)

The final transition diagram and corresponding transition table:



		input	
		0	**1**
state	*A*	*A*	*B*
	B	10	*B*
	10	*A*	*B*

- The analysis that led to the simplification of the FA in this example can be turned into a systematic procedure for minimizing the number of states in an FA.

Extending the Transition Function

- Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. The δ function can be extended to describe the behavior of M when given more than one symbol of input.

- The value of $\delta^*(q, x)$ will indicate the state in which M ends up if started in state q and given input string x.

- If x is the string $a_1a_2\ldots a_n$, the value of $\delta^*(q, x)$ is obtained by first going to the state q_1 to which M goes from state q on input a_1; then going to the state q_2 to which M goes from q_1 on input a_2; …; and finally, going to the state q_n to which M goes from q_{n-1} on input a_n.

- To make the definition of δ^* formal, a recursive definition is needed.

- **Definition 3.3:** Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. The function

  $\delta^* : Q \times \Sigma^* \rightarrow Q$

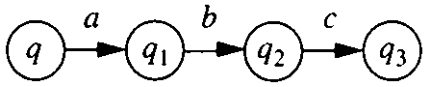
  is defined as follows:

  1. For any $q \in Q$, $\delta^*(q, \Lambda) = q$
  2. For any $q \in Q$, $y \in \Sigma^*$, and $a \in \Sigma$,

  $\delta^*(q, ya) = \delta(\delta^*(q, y), a)$

Extending the Transition Function (Continued)

- Suppose that M contains the following transitions:



- Definition 3.3 can be used to calculate the value of $\delta^*(q, abc)$:

$$\begin{aligned}
\delta^*(q, abc) &= \delta(\delta^*(q, ab), c) \\
&= \delta(\delta(\delta^*(q, a), b), c) \\
&= \delta(\delta(\delta^*(q, \Lambda a), b), c) \\
&= \delta(\delta(\delta(\delta^*(q, \Lambda), a), b), c) \\
&= \delta(\delta(\delta(q, a), b), c) \\
&= \delta(\delta(q_1, b), c) \\
&= \delta(q_2, c) \\
&= q_3
\end{aligned}$$

- For strings of length 1 (i.e., elements of Σ), δ and δ^* can be used interchangeably.

- Other properties of δ^* can be derived from its definition. For example, it is possible to prove by mathematical induction that $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$ for any $q \in Q$ and any strings x and y.

Acceptance by a Finite Automaton

- Using the extended transition function, it is possible to give a precise definition of what it means for an FA to accept a string and what it means for an FA to accept a language.

- **Definition 3.4:** Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. A string $x \in \Sigma^*$ is *accepted* by M if $\delta^*(q_0, x) \in A$. If a string is not accepted, it is *rejected* by M. The *language* accepted by M, or the language recognized by M, is the set

  $L(M) = \{x \in \Sigma^* \mid x \text{ is accepted by } M\}$

  If L is any language over Σ, L is accepted, or recognized, by M if and only if $L = L(M)$.

- The last statement in the definition does *not* say that L is accepted by M if every string in L is accepted by M. If it did, the following FA could be used to accept any language:

  The power of a machine does not lie in the number of strings it accepts, but in its ability to accept some and reject others.

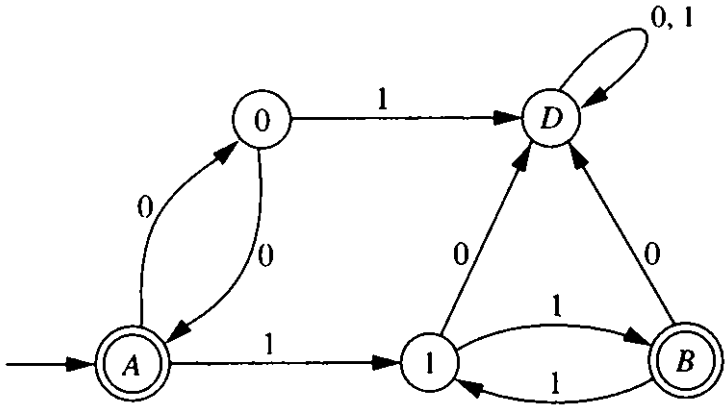
- **Theorem 3.1.** A language L over the alphabet Σ is regular if and only if there is an FA with input alphabet Σ that accepts L.

- Chapter 4 will prove this theorem by showing how to construct a regular expression from an FA and vice-versa. In simple cases, however, these constructions can be done informally.

Finding a Regular Expression Corresponding to an FA

- **Example 3.13:** Finding a Regular Expression Corresponding to an FA

  Consider the problem of finding a regular expression for the language *L* accepted by the following FA:



  To be accepted, a string must cause the FA to go from state *A* to state *A* or from state *A* to state *B*.

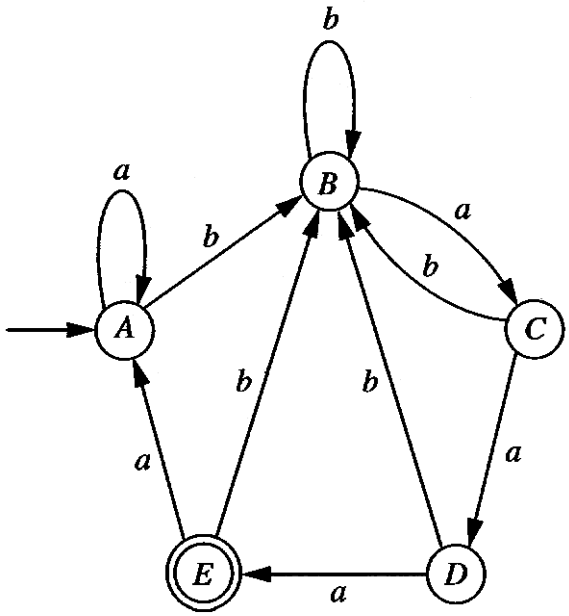
  The strings that cause the FA to go from state *A* to state *A* correspond to the regular expression (00)*.
  The strings that cause the FA to go from state *A* to state *B* correspond to the regular expression (00)*11(11)*.

  Combining the two regular expressions yields (00)* + (00)*11(11)*, which can be simplified to (00)*(11)*.

Finding a Regular Expression Corresponding to an FA (Continued)

- **Example 3.14:** Another Example of a Regular Expression Corresponding to an FA

  Let *M* be the following FA:



  Every string ending in *b* causes *M* to be in state *B*, which means that any string ending in *baaa* is accepted by *M*. It is also true that any string accepted by *M* must end in *baaa*, so *L*(*M*) is the set of all strings ending in *baaa*.

  A regular expression corresponding to *L*(*M*) is (*a* + *b*)**baaa*.

  Developing a systematic approach to finding a regular expression for *L*(*M*) is difficult because of the many loops in *M*.

Finding an FA Corresponding to a Regular Expression

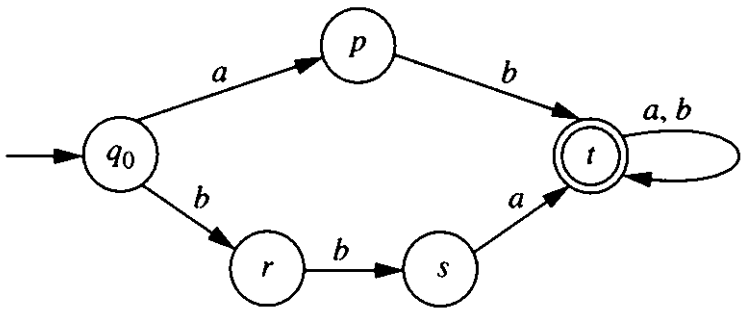
- **Example 3.15:** Strings Containing Either *ab* or *bba*

  Let L be the language of all strings in $\{a, b\}^*$ that contain *ab* or *bba* as a substring. L corresponds to the regular expression $(a + b)^*(ab + bba)(a + b)^*$.

  A finite automaton accepting L can be constructed based on the following observations:

  The FA should accept the strings *ab* and *bba*.
  If *x* is accepted, then any other string obtained by adding symbols to the end of *x* should also be accepted.

  A first attempt at an FA:

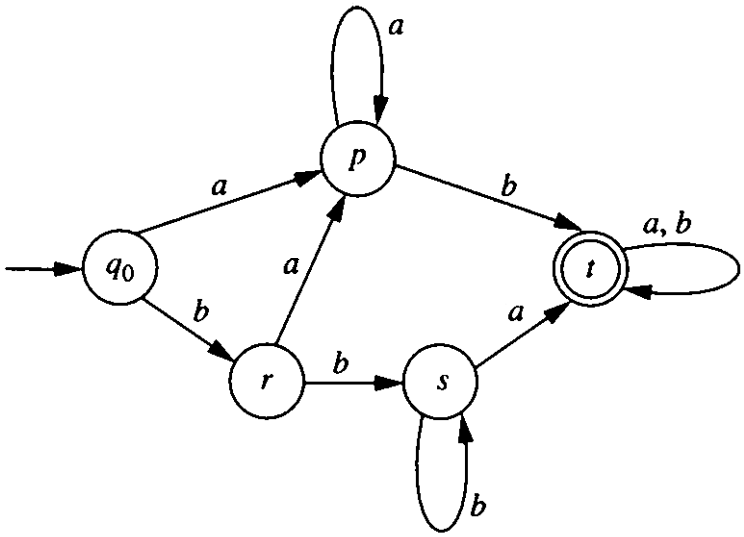


  States *p*, *r*, and *s* are missing some of their transitions. Before filling in these transitions, it is helpful to determine what each of these states represents:

  p is the state reached if the last input symbol was *a* and the FA has not yet seen either *ab* or *bba*.
  r is the state reached if the last input symbol was *b*, and it was not preceded by a *b*, and the FA has not yet arrived in the accepting state.
  s is the state reached if the FA has just received two consecutive *b*'s but has not yet reached the accepting state.

Finding an FA Corresponding to a Regular Expression (Continued)

- The transition diagram for the completed FA:



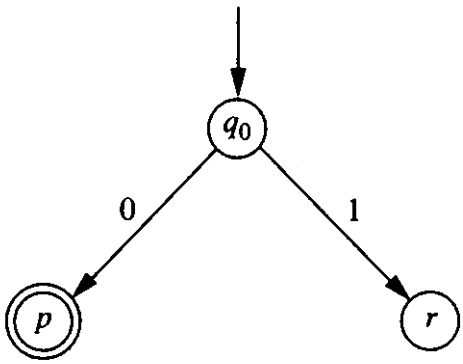
Finding an FA Corresponding to a Regular Expression (Continued)

- **Example 3.16:** Another Example of an FA Corresponding to a Regular Expression

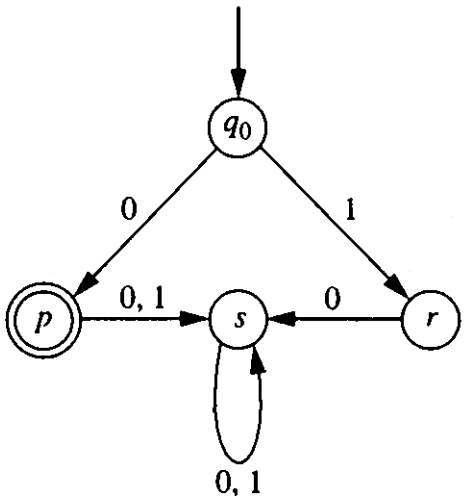
  Constructing an FA from a regular expression is not always a straightforward process. Consider the problem of constructing an FA from the regular expression (11 + 110)*0.

  When the structure of the FA is not clear, it is best just to proceed one symbol at a time, checking to see if each new transition goes to an existing state or requires a new state.

  At the beginning, there will be transitions from the initial state on both 0 and 1:



  At this point, it is clear that a "dead state" *s* will be needed to handle strings that cannot be accepted, such as strings that begin with 00, 01, or 10:

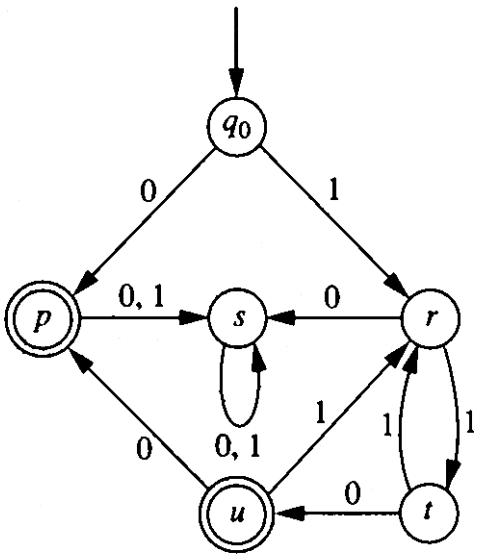


Finding an FA Corresponding to a Regular Expression (Continued)

If the FA is in state *r* and receives the input 1, a new state *t* will be needed.

If the FA is in state *t* and receives the input 0, a new accepting state *u* will be needed.

At this point, all new transitions can go to existing states, so no further states are needed:



The process of adding new states is guaranteed to stop eventually provided that the language is regular. (If the language is not regular, the process will not stop.)

Chapter 4 gives an algorithm for converting a regular expression to an FA.

3.4 Distinguishing One String From Another

- A finite automaton relies on the fact that it doesn't have to remember all the symbols it has seen so far. Instead, each state in the FA represents a group of strings that don't need to be distinguished from each other.

- The following definition specifies the circumstances under which an FA recognizing a language L must distinguish between two strings x and y.

- **Definition 3.5:** Let L be a language in Σ^*, and x any string in Σ^*. The set L/x is defined as follows:

  $L/x = \{z \in \Sigma^* \mid xz \in L\}$

  Two strings x and y are said to be *distinguishable with respect to L* if $L/x \neq L/y$. Any string z that is in one of the two sets but not the other (i.e., for which $xz \in L$ and $yz \notin L$, or vice versa) is said to distinguish x and y with respect to L. If $L/x = L/y$, x and y are indistinguishable with respect to L.

- To show that two strings x and y are distinguishable with respect to a language L, it is sufficient to find a string z so that either $xz \in L$ and $yz \notin L$, or $xz \notin L$ and $yz \in L$.

Distinguishing One String From Another (Continued)

- **Lemma 3.1.** Suppose $L \subseteq \Sigma^*$ and $M = (Q, \Sigma, q_0, A, \delta)$ is an FA recognizing L. If x and y are two strings in Σ^* that are distinguishable with respect to L, then $\delta^*(q_0, x) \neq \delta^*(q_0, y)$.

  Proof: If x and y are distinguishable with respect to L, then one of the strings xz and yz is in L and the other is not. Because M accepts L, it follows that one of the two states $\delta^*(q_0, xz)$ and $\delta^*(q_0, yz)$ is an accepting state and the other is not. In particular, therefore,

  $\delta^*(q_0, xz) \neq \delta^*(q_0, yz)$

  It is easy that show that

  $\delta^*(q_0, xz) = \delta^*(\delta^*(q_0, x), z)$
  $\delta^*(q_0, yz) = \delta^*(\delta^*(q_0, y), z)$

  Because the left sides of these two equations are unequal, the right sides must be, and therefore $\delta^*(q_0, x) \neq \delta^*(q_0, y)$.

- **Theorem 3.2.** Suppose $L \subseteq \Sigma^*$ and, for some positive integer n, there are n strings in Σ^*, any two of which are distinguishable with respect to L. Then every FA recognizing L must have at least n states.

  Proof: Suppose $x_1, x_2, \ldots, x_n$ are n strings, any two of which are distinguishable with respect to L. If $M = (Q, \Sigma, q_0, A, \delta)$ is any FA with fewer than n states, then by the pigeonhole principle, the states $\delta^*(q_0, x_1)$, $\delta^*(q_0, x_2)$, …, $\delta^*(q_0, x_n)$ are not all distinct, and so for some $i \neq j$, $\delta^*(q_0, x_i) = \delta^*(q_0, x_j)$. Since x_i and x_j are distinguishable with L, it follows from the lemma that M cannot recognize L.

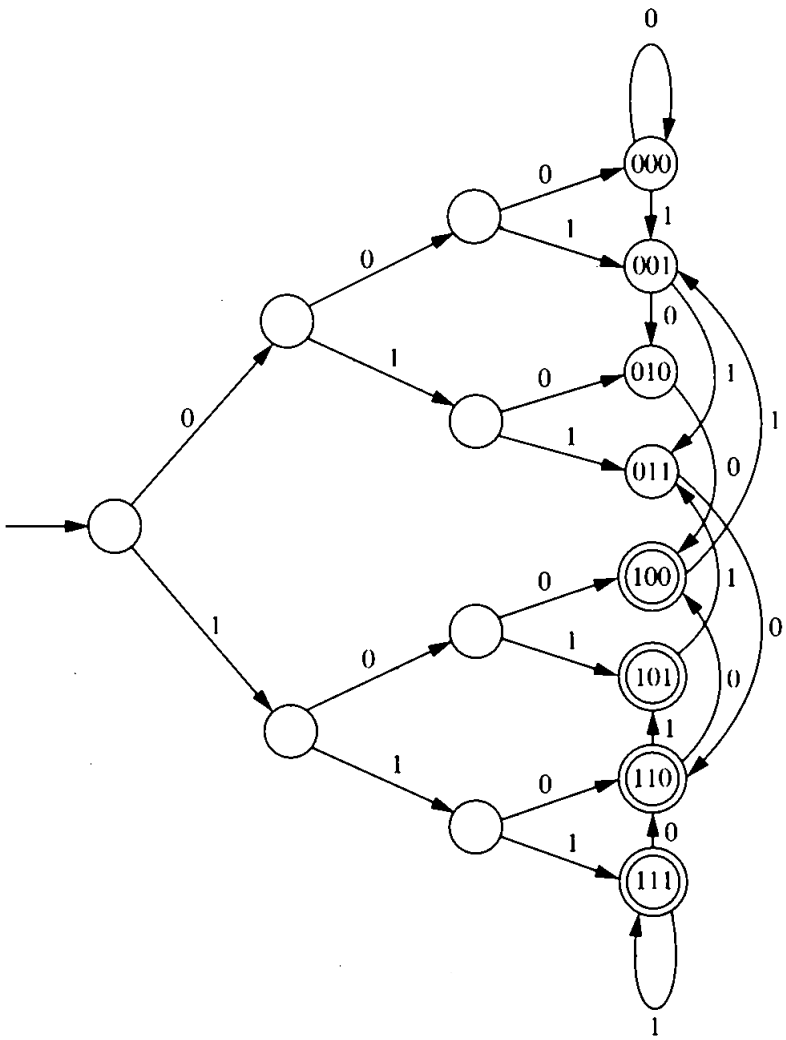
Distinguishing One String From Another (Continued)

- **Example 3.17:** The Language L_n

  Suppose $n \geq 1$, and let

  $L_n = \{x \in \{0, 1\}^* \mid |x| \geq n$ and the nth symbol from the right in x is $1\}$

  One way to construct an FA for L_n is to have a distinct state for every possible substring of length n or less. If $n = 3$, the FA will have the following appearance:



  Some transitions from the rightmost eight states are not shown. The rule for these transitions is $\delta(abc, d) = bcd$.

Distinguishing One String From Another (Continued)

- In general, the construction of Example 3.17 will produce an FA for L_n with $2^{n+1} - 1$ states.

- The number of states can be reduced somewhat, using the technique of Example 3.12. However, Theorem 3.2 can be used to show that any FA that recognizes L_n must have at least 2^n states.

- Using Theorem 3.2 requires showing that any two strings of length n (of which there are 2^n) are distinguishable with respect to L_n.

  If x and y are two distinct strings of length n, they must differ in the ith symbol (from the left), for some i with $1 \le i \le n$.
  For the string z that we will use to distinguish these two strings with respect to L, we can choose any string of length $i - 1$.
  Then xz and yz still differ in the ith symbol, and now the ith position is precisely the nth from the right.
  One of the strings xz and yz is in L and the other is not, so $L/x \neq L/y$. Therefore, x and y are distinguishable with respect to L_n.

Proving That a Language Is Not Regular

- Theorem 3.2 can be used to show that a language L is not regular. The technique is to find an infinite set S of strings whose elements are pairwise distinguishable with respect to L. Since S is infinite, there can be no FA that recognizes L.

- In Theorem 3.3, the set S is taken to be Σ^*, the set of all strings.

- **Theorem 3.3.** The language *pal* of palindromes over the alphabet {0, 1} cannot be accepted by any finite automaton, and it is therefore not regular.

  Proof: The goal is to show that for any two distinct strings x and y in $\{0, 1\}^*$, x and y are distinguishable with respect to *pal*.

  First consider the case when $|x| = |y|$, and let $z = x^r$. Then $xz = xx^r$, which is in *pal*, and yz is not.

  If $|x| \neq |y|$, assume without loss of generality that $|x| < |y|$, and let $y = y_1y_2$, where $|y_1| = |x|$. Let w be any string different from y_2 but of the same length, and let $z = ww^rx^r$.

  Now, $xz = xww^rx^r$, so $xz \in pal$.

  In order for the string $yz = y_1y_2z = y_1y_2ww^rx^r$ to be in *pal*, w^r must be the reverse of y_2, which is impossible, because $w \neq y_2$.

  In either case, x and y are distinguishable with respect to *pal*.

- Chapter 5 introduces other methods for demonstrating that a language is nonregular.

3.5 Unions, Intersections, and Complements

- If L_1 and L_2 are regular languages accepted by FAs M_1 and M_2, it should be possible to construct FAs that accept $L_1 \cup L_2$, L_1L_2, and L_1*, because these languages are also regular.

- Constructing an FA that accepts $L_1 \cup L_2$ is fairly simple. The same construction—with minor changes—can be used to construct an FA that accepts $L_1 \cap L_2$ and $L_1 - L_2$.

- Chapter 4 shows how to construct FAs for L_1L_2 and L_1*.

- Suppose that $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are FAs that accept L_1 and L_2, respectively. An FA that accepts $L_1 \cup L_2$ will need to test an input string x to see if either M_1 or M_2 accepts it. This can be done by having M simulate the behavior of both M_1 and M_2 when given x as their input.

- The states of M will have the form (p, q), where $p \in Q_1$ and $q \in Q_2$. Each move of M will update p based on the move M_1 would have made and q based on the move M_2 would have made.

- For M to accept $L_1 \cup L_2$, the accepting states of M will have the form (p, q), where p is an accepting state of M_1 and/or q is an accepting state of M_2.

- For M to accept $L_1 \cap L_2$ or $L_1 - L_2$, only the definition of the accepting states will need to be changed.

Unions, Intersections, and Complements (Continued)

- **Theorem 3.4.** Suppose $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ accept languages L_1 and L_2, respectively. Let M be an FA defined by $M = (Q, \Sigma, q_0, A, \delta)$, where

  $Q = Q_1 \times Q_2$
  $q_0 = (q_1, q_2)$

  and the transition function δ is defined by the formula

  $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$

  (for any $p \in Q_1$, $q \in Q_2$, and $a \in \Sigma$). Then

  1. If $A = \{(p, q) \mid p \in A_1 \text{ or } q \in A_2\}$, M accepts the language $L_1 \cup L_2$;
  2. If $A = \{(p, q) \mid p \in A_1 \text{ and } q \in A_2\}$, M accepts the language $L_1 \cap L_2$;
  3. If $A = \{(p, q) \mid p \in A_1 \text{ and } q \notin A_2\}$, M accepts the language $L_1 - L_2$.

  Proof: Since acceptance by M_1 and M_2 is defined in terms of the functions δ_1^* and δ_2^*, respectively, and acceptance by M in terms of δ^*, we need the formula

  $\delta^*((p, q), x) = (\delta_1^*(p, x), \delta_2^*(q, x))$

  which holds for any $x \in \Sigma^*$ and any $(p, q) \in Q$, and can be verified easily by using mathematical induction. A string x is accepted by M if and only if $\delta^*((q_1, q_2), x) \in A$. By our formula this is true if and only if

  $(\delta_1^*(q_1, x), \delta_2^*(q_2, x)) \in A$

  If the set A is defined as in case 1, this is the same as saying that $\delta_1^*(q_1, x) \in A_1$ or $\delta_2^*(q_2, x) \in A_2$, or in other words, that $x \in L_1 \cup L_2$. Cases 2 and 3 are similar.

- An FA accepting L_2' could be constructed by letting M_1 be an FA with one state that accepts Σ^* and then applying the $L_1 - L_2$ construction. However, there is an easier way to construct an FA that accepts L_2':

  $M_2' = (Q_2, \Sigma, q_2, Q_2 - A_2, \delta_2)$

Unions, Intersections, and Complements (Continued)

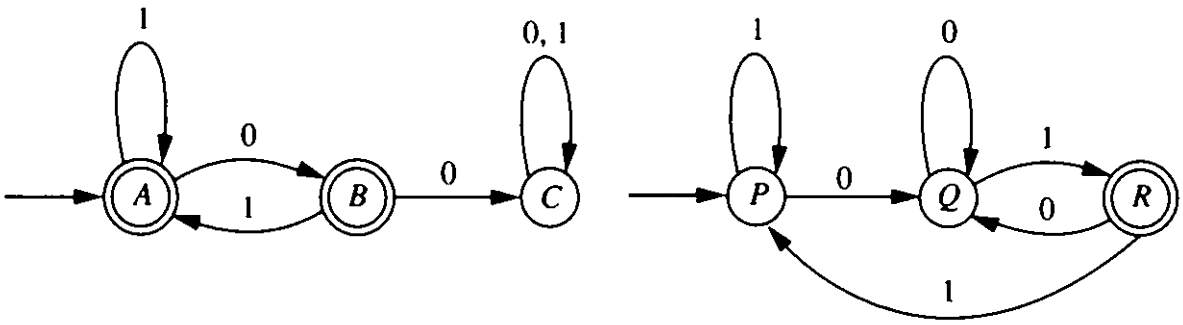
- An FA created by the construction in Theorem 3.4 can often be simplified.

- **Example 3.18:** An FA Accepting $L_1 - L_2$

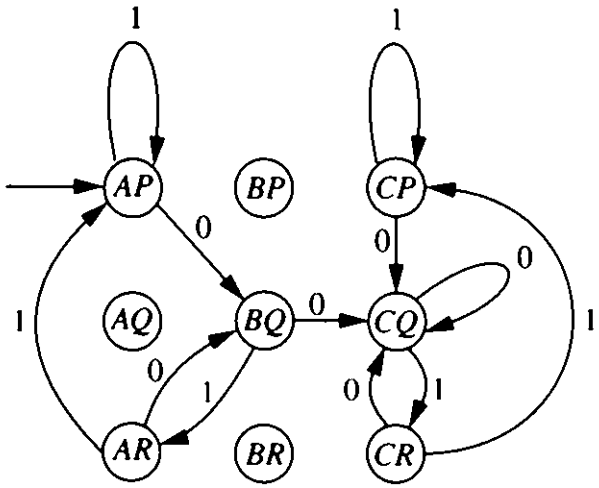
  Let L_1 and L_2 be the following languages:

  $L_1 = \{x \in \{0, 1\}^* \mid 00 \text{ is not a substring of } x\}$
  $L_2 = \{x \in \{0, 1\}^* \mid x \text{ ends with } 01\}$

  L_1 and L_2 are recognized by the following FAs:



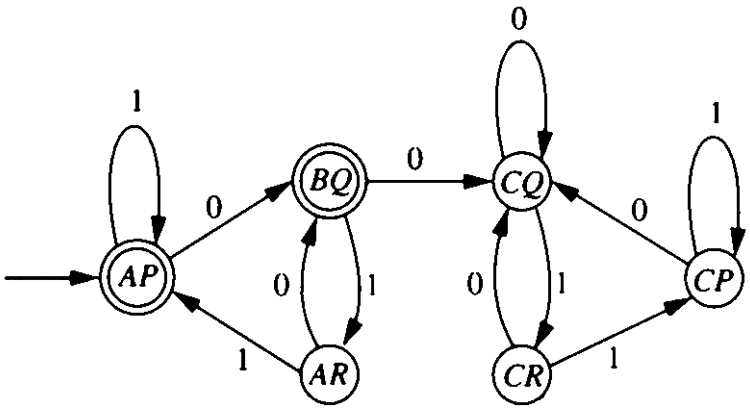
  The construction in Theorem 3.4 produces an FA with nine states:



  Three states can be removed because they are not reachable from the initial state.

Unions, Intersections, and Complements (Continued)

The FA constructed so far can be made to recognize the language $L_1 - L_2$ by designating as the accepting states those states (X, Y) for which X is either A or B and Y is not R:



This FA can be further simplified by observing that once the machine enters any of the states (C, P), (C, Q), or (C, R), it remains in one of them. Therefore, these states can be replaced by a single state: