Part One: Automata and Languages
Chapter 1. Regular Languages

Wonhong Nam

Konkuk University

September 13, 2017

Part One: Automata and Languages

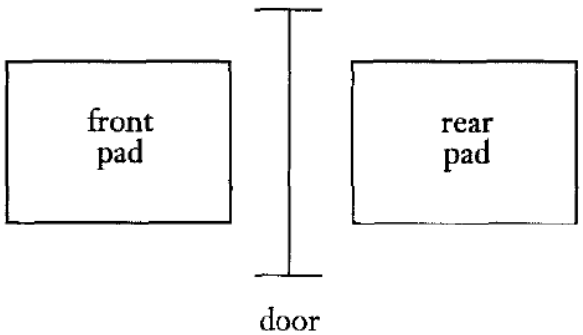
What is a computer?

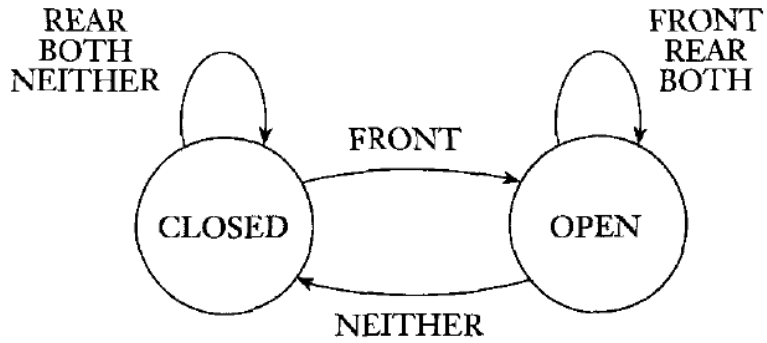
- It is perhaps a silly question, but these real computers are quite complicated—too much so to allow us to set up a manageable mathematical theory of them directly.
- Instead we use an idealized computer called a **computational model**.
- As with any model in science, a computational model may be accurate in some ways but perhaps not in others.
- Thus we will use several different computational models, depending on the features we want to focus on.
- We begin with the simplest model, called the **finite state machine** or **finite automaton**.

1.1 Finite Automata

Finite automata are good models for computers with an extremely limited amount of memory.

- Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.
- The controller for an automatic door is one example of such a device.
  - An automatic door has a pad in front to detect the presence of a person about to walk through the doorway.
  - Another pad is located to the rear of the doorway so that the controller can hold the door open long enough for the person to pass all the way through and also so that the door does not strike someone standing behind it as it opens.
  - The controller is in either of two states: "OPEN" or "CLOSED."



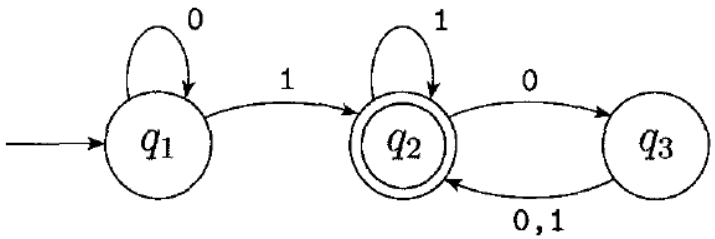


Automatic Door Controller

- For example, a controller might start in state CLOSED and receive the series of input signals FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, REAR, and NEITHER.
- It then would go through the series of states CLOSED (starting), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED, CLOSED, and CLOSED.
- Thinking of an automatic door controller as a finite automaton is useful because that suggests standard ways of representation as in Figures 1.2.
- This controller is a computer that has just a single bit of memory, capable of recording which of the two states the controller is in.
- Other common devices have controllers with somewhat larger memories, e.g., an elevator controller.
- Controllers for various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators, are additional examples of computers with limited memories.
- The design of such devices requires keeping the methodology and terminology of finite automata in mind.
- Finite automata and their probabilistic counterpart **Markov chains** are useful tools when we are attempting to recognize patterns in data.

Finite Automaton

- Figure 1.4 is called the **finite automaton** (**state diagram**) of M_1.
- It has three **states**, labeled q_1, q_2, and q_3.
- The **start state**, q_1, is indicated by the arrow pointing at it from nowhere.
- The **accept state**, q_2, is the one with a double circle.
- The arrows going from one state to another are called **transitions**.
- When this automaton receives an input string such as 1101, it processes that string and produces an output which is either **accept** or **reject**.
- *Accept* because M_1 is in an accept state q_2 at the end of the input.
- Experimenting with this machine on a variety of input strings reveals that it accepts the strings 1, 01, 11, and 0101010101.
- In fact, M_1 accepts any string that ends with a 1.
- In addition, it accepts strings 100, 0100, 110000, and 0101000000, and any string that ends with an even number of 0s following the last 1.
- It rejects other strings, such as 0, 10, 101000.



Formal Definition of a Finite Automaton

Definition 1.5

A finite automation 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. $\delta : Q \times \Sigma \to Q$ is the **transition function**,
4. $q_0 \in Q$ is the **start state (initial state)**, and
5. $F \subseteq Q$ is the set of **accept states (goal states)**.

- The transition function δ specifies exactly one next state for each possible combination of a state and an input symbol.
- 0 accept states is allowable.
- We can use the notation of the formal definition to describe individual finite automata by specifying each of the five parts listed in Definition 1.5.

Formal Definition of a Finite Automaton

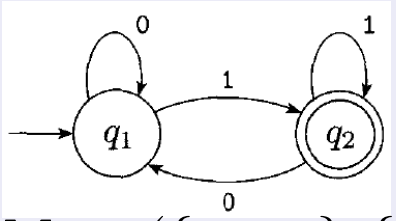
- For example, let's return to the finite automaton M_1 we discussed earlier.
- $M_1 = (Q, \Sigma, \delta, q_1, F)$, where $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, δ below, q_1 is the start state, and $F = \{q_2\}$.

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- If A is the set of all strings that machine M accepts, we say that A is the **language of machine** M and write $L(M) = A$.
- We say that M **recognizes** A or that M **accepts** A.
- Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term recognize for languages in order to avoid confusion.
- In our example, let $A = \{w \mid w$ contains at least one 1 and an even number of 0s following the last 1$\}$.
- Then $L(M_1) = A$, or equivalently, M_1 recognizes A.

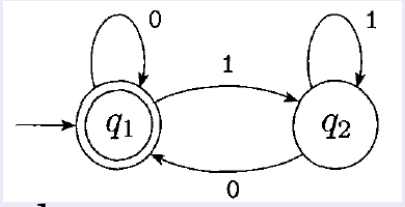
Examples of Finite Automata

Example 1.7: Here is the state diagram of finite automaton M_2.



- In the formal description $M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$.
- A good way to begin understanding any machine is to try it on some sample input strings. E.g., 0, 1, 00, 01, 10, 11, $\cdots$.
- $L(M_2) = \{w \mid w \text{ ends in a } 1\}$.

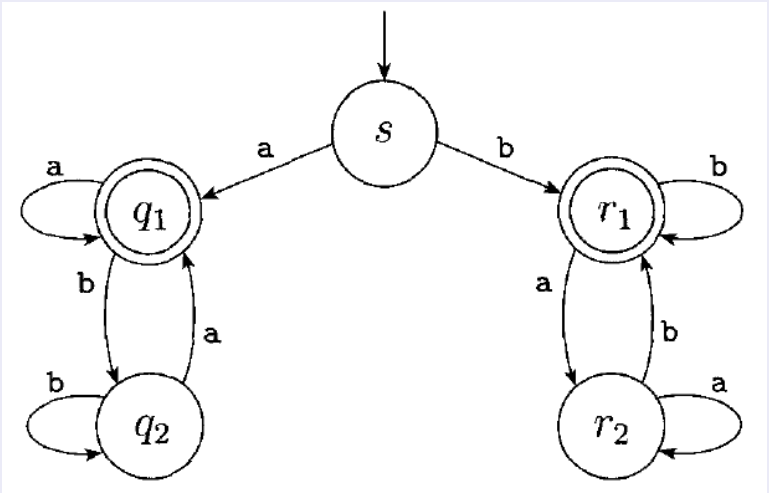
Example 1.9: Consider the finite automaton M_3.



- Because the start state is also an accept state, M_3 accepts the empty string ε.
- $L(M_3) = \{w \mid w \text{ is the empty string } \varepsilon \text{ or ends in a } 0\}$.

Examples of Finite Automata

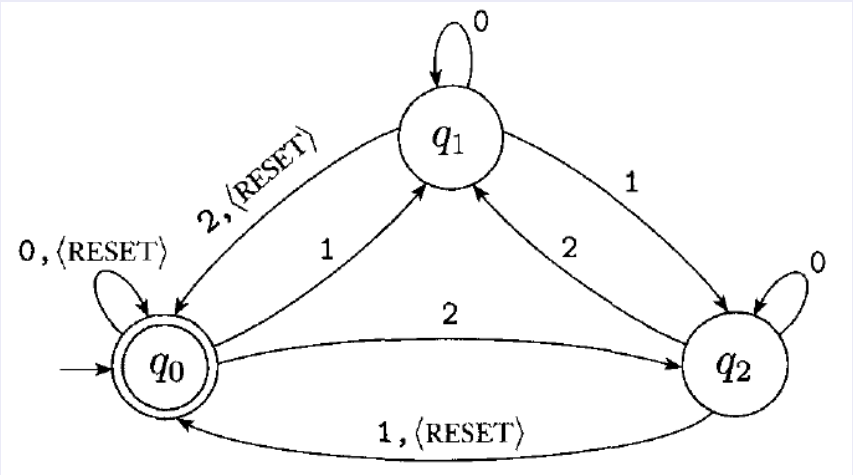
Example 1.11: The following figure shows a five-state machine M_4.



- Some experimentation shows that it accepts strings `a`, `b`, `aa`, `bb`, and `bab`, but not strings `ab`, `ba`, or `bbba`.
- If the first symbol in the input string is `a`, then it goes left and accepts when the string ends with an `a`.
- Similarly, if the first symbol is a `b`, the machine goes right and accepts when the string ends in `b`.
- So M_4 accepts all strings that start and end with `a` or that start and end with `b`.

Examples of Finite Automata

Example 1.13: Figure 1.14 shows machine M_5.



- $\Sigma = \{\langle \text{RESET} \rangle, 0, 1, 2\}$. We treat $\langle \text{RESET} \rangle$ as a single symbol.
- Machine M_5 keeps a running count of the sum of the numerical input symbols it reads, modulo 3.
- Every time it receives the $\langle \text{RESET} \rangle$ symbol it resets the count to 0.
- It accepts if the sum is 0 modulo 3, or in other words, if the sum is a multiple of 3.

Formal Definition of Computation

- So far we have described finite automata informally, using state diagrams, and with a formal definition, as a 5-tuple.
- Next we do the same for a finite automaton's computation. We already have an informal idea of the way it computes, and we now formalize it mathematically.
- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M accepts w if a sequence of states $r_0, r_1, \ldots, r_n$ 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. $r_n \in F$.
- We say that M **recognizes language** A if $A = \{w \mid M \text{ accepts } w\}$.

Definition 1.16

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

Example 1.17

- Take machine M_5 from Example 1.13.
- Let w be the string $10\langle\text{RESET}\rangle 22\langle\text{RESET}\rangle 012$.
- Then M_5 accepts w according to the formal definition of computation because the sequence of states it enters when computing on w is
- $q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0$,
- which satisfies the three conditions.
- The language of M_5 is
- $L(M_5) = \{w \mid$ the sum of the symbols in w is 0 modulo 3, except that $\langle\text{RESET}\rangle$ resets the count to $0\}$.
- As M_5 recognizes this language, it is a regular language.

Designing Finite Automata

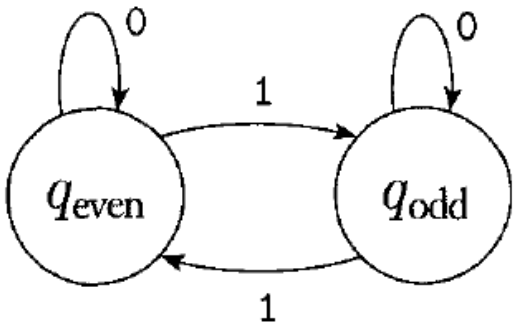
- Whether it be of automaton or artwork, design is a creative process.
- Put *yourself* in the place of the machine you are trying to design and then see how you would go about performing the machine's task.
- Suppose that you are given some language and want to design a finite automaton that recognizes it.
- Pretending to be the automaton, you receive an input string and must determine whether it is a member of the language.
- You get to see the symbols in the string one by one.
- After each symbol you must decide whether the string seen so far is in the language.
- You have to figure out what you need to remember about the string as you are reading it.
- You need to remember only certain crucial information.

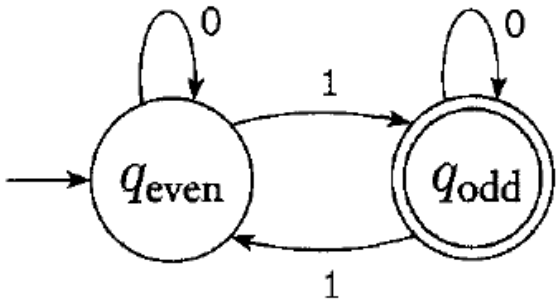
Suppose that the alphabet is $\{0, 1\}$ and the language consists of all strings with an odd number of 1s.

- You want to construct a finite automaton E_1 to recognize this language.
- You start getting an input string of 0s and 1s symbol by symbol.
- Do you need to remember the entire string seen so far in order to determine whether the number of 1s is odd?
- Simply remember whether the number of 1s seen so far is even or odd and keep track of this information as you read new symbols.
- Once you have determined the necessary information to remember about the string, you represent this information as a finite list of possibilities.
- In this instance, the possibilities would be
  1. even so far, and
  2. odd so far.
- These are the states of E_1, as shown here.

The language with all strings with an odd number of 1s.

- Next, you assign the transitions by seeing how to go from one possibility to another upon reading a symbol.
- So, if state q_{even} represents the even possibility and state q_{odd} represents the odd possibility, you would set the transitions to flip state on a 1 and stay put on a 0.
- Next, you set the start state to be the state corresponding to the possibility associated with having seen 0 symbols so far.
- In this case the start state corresponds to state q_{even} because 0 is an even number.
- Last, set the accept states.
- Set q_{odd} to be an accept state because you want to accept when you have seen an odd number of 1s.



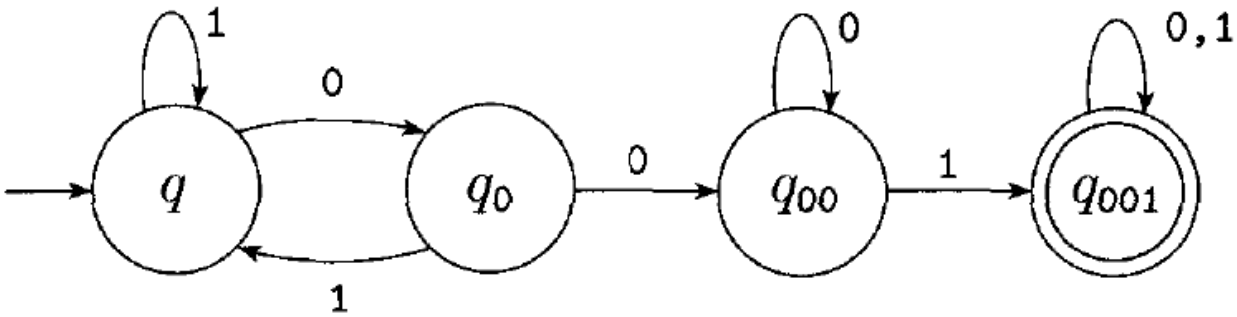


Example 1.21: E_2 to recognize the regular language of all strings that contain the string 001 as a substring

- For example, 0010, 1001, 001, and 11111110011111 are all in the language, but 11 and 0000 are not.
- As symbols come in, you would initially skip over all 1s.
- If you come to a 0, then you note that you may have just seen the first of the three symbols in the pattern 001 you are seeking.
- If at this point you see a 1, there were too few 0s, so you go back to skipping over 1s.
- But if you see a 0 at that point, you should remember that you have just seen two symbols of the pattern.
- Now you simply need to continue scanning until you see a 1.
- If you find it, remember that you succeeded in finding the pattern and continue reading the input string until you get to the end.
- So there are four possibilities: You
  1. haven't just seen any symbols of the pattern,
  2. have just seen a 0,
  3. have just seen 00, or
  4. have seen the entire pattern 001.

Example 1.21: all strings with 001 as a substring

- Assign the states q, q_0, q_{00}, and q_{001} to these possibilities.
- You can assign the transitions by observing that from q reading a 1 you stay in q, but reading a 0 you move to q_0.
- In q_0 reading a 1 you return to q, but reading a 0 you move to q_{00}.
- In q_{00}, reading a 1 you move to q_{001}, but reading a 0 leaves you in q_{00}.
- Finally, in q_{001} reading a 0 or a 1 leaves you in q_{001}.
- The start state is q, and the only accept state is q_{001}.



The Regular Operations

- So far, we introduced and defined finite automata and regular languages.
- We now begin to investigate their properties. Doing so will help develop a toolbox of techniques to use.
- In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as $+$ and $\times$.
- In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them.
- We define three operations on languages, called the **regular operations**, and use them to study properties of the regular languages.

Definition 1.23

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

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

- The empty string ε is always a member of A^*, no matter what A is.

The Regular Operations

Example 1.24

Let the alphabet Σ be the standard 26 letters $\{a, b, \ldots, z\}$. If $A = \{\text{good, bad}\}$ and $B = \{\text{boy, girl}\}$, then

- $A \cup B = \{\text{good, bad, boy, girl}\}$,
- $A \circ B = \{\text{goodboy, goodgirl, badboy, badgirl}\}$, and
- $A^* = \{\varepsilon, \text{good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, } \ldots\}$.

- 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.

Union operation

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 $A_1 \cup A_2$.

Proof idea: proof by construction

- 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 $A_1 \cup A_2$.
- 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.
- It works by *simulating* both M_1 and M_2 and accepting if either of the simulations accept.
- Perhaps it first simulates M_1 on the input and then simulates M_2 on the input. No.

Proof idea: proof by construction

- As the input symbols arrive one by one, you simulate both M_1 and M_2 simultaneously.
- 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.
- 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 is the product $k_1 \times k_2$.
- 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: proof by construction

- Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.
- Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.
- $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
- This set is the *Cartesian product* of Q_1 and Q_2 and is written $Q_1 \times Q_2$.
- Σ, the alphabet, is the same as in M_1 and M_2 .
- The theorem remains true if they have different alphabets, Σ_1 and Σ_2.
- δ, the transition function, is defined as follows.
- For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
- q_0 is the pair (q_1, q_2).
- 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 construction is fairly simple, and thus its correctness is evident from the strategy described in the proof idea.
- The class of regular languages is closed under the union operation.

Theorem 1.26

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 $A_1 \circ A_2$.

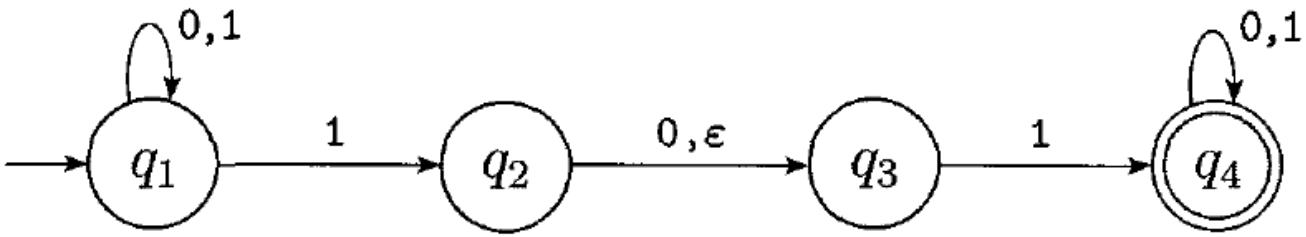
- Instead of constructing automaton M to accept its input if either M_1 or M_2 accepts, it must accept if its input can be broken into two pieces, where M_1 accepts the first piece and M_2 accepts the second piece.
- The problem is that M doesn't know where to break its input (i.e., where the first part ends and the second begins).
- To solve this problem we introduce a new technique called *nondeterminism*.

1.2 Nondeterminism

- Nondeterminism is a useful concept that has had great impact on the theory of computation.
- So far, every step of a computation follows in a unique way from the preceding step.
- When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined.
- We call this **deterministic** computation.
- In a **nondeterministic** machine, several choices may exist for the next state at any point.

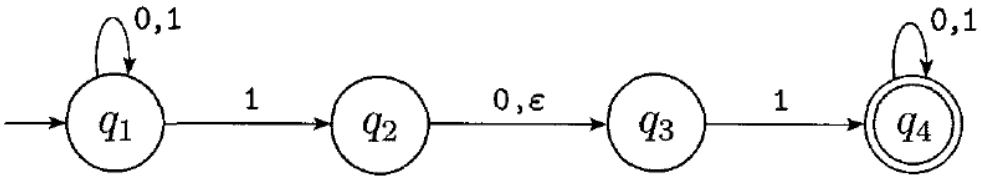
DFA vs. NFA

- The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is as follows.
- First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet.
- The nondeterministic automaton shown in Figure 1.27 violates that rule.
- State q_1 has one exiting arrow for 0, but it has two for 1; q_2 has one arrow for 0, but it has none for 1.
- In an NFA a state may have zero, one, or many exiting arrows for each alphabet symbol.
- An NFA may have arrows labeled with members of the alphabet or ε.
- Zero, one, or many arrows may exit from each state with the label ε.



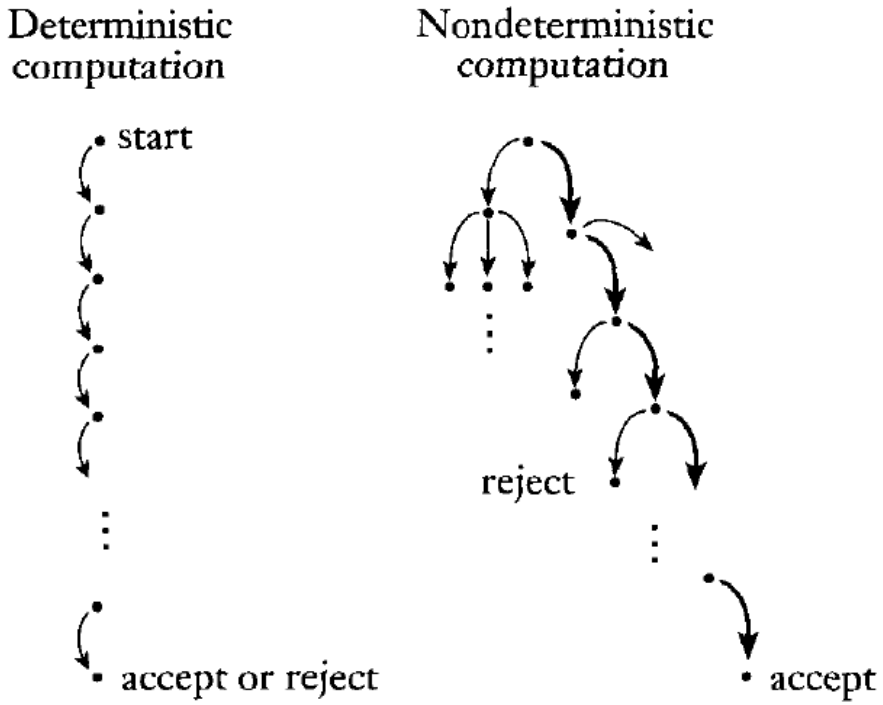
How does an NFA compute?

- Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed.
- For example, say that we are in state q_1 in NFA N_1 and that the next input symbol is a 1.
- After reading that symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel.
- If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it.
- Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.
- If a state with an ε symbol on an exiting arrow is encountered, something similar happens.
- Without reading any input, the machine splits into multiple copies, one following each of the exiting ε-labeled arrows and one staying at the current state.

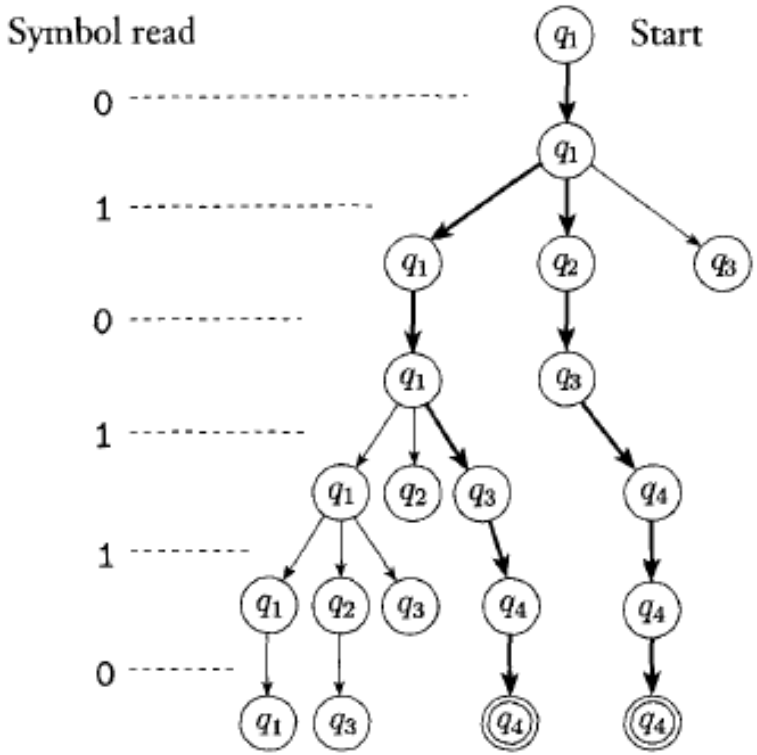


Deterministic and nondeterministic computations

- Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent "processes" or "threads" can be running concurrently.
- Another way to think of a nondeterministic computation is as a tree of possibilities.
- The machine accepts if at least one of the computation branches ends in an accept state, as shown in Figure 1.28.

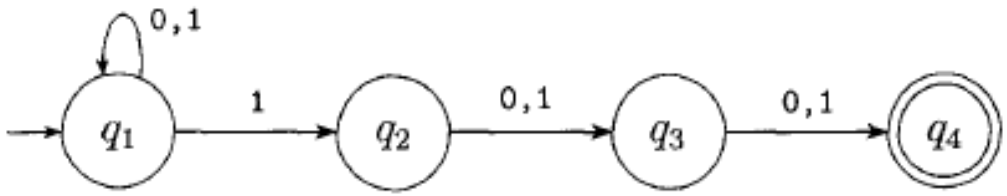


Sample runs of the NFA N_1 shown in Figure 1.27

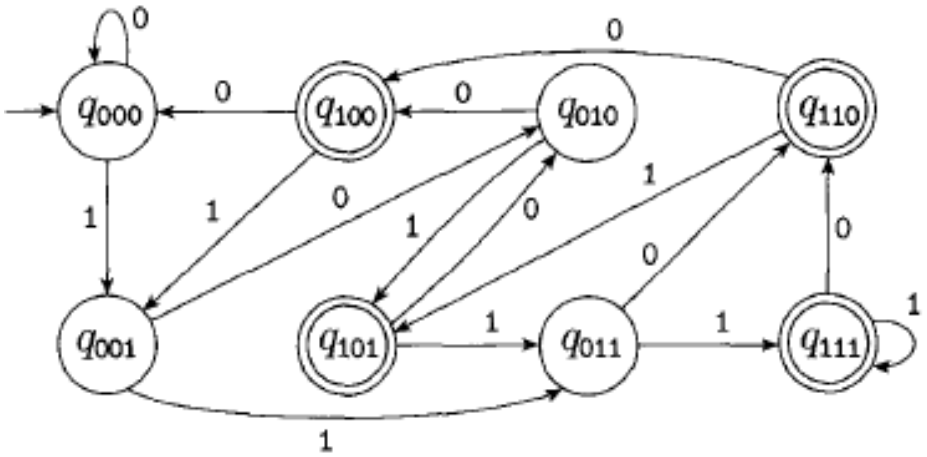


- The computation of N_1 on input 010110 is depicted in the following figure.
- Note that an ε arrow exits state q_2.
- As q_4 is an accept state, N_1 accepts this string.
- What does N_1 do on input 010?
- By continuing to experiment in this way, you will see that N_1 accepts all strings that contain either 101 or 11 as a substring.

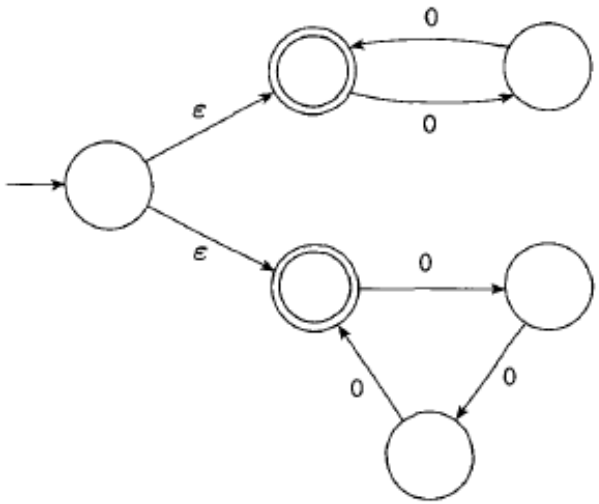
Example 1.30



- Let A be the language consisting of all strings over 0,1 containing a 1 in the third position from the end (e.g., 01100 is in A but 0011 is not).
- The above four-state NFA N_2 recognizes A.
- Every NFA can be converted into an equivalent DFA, but sometimes that DFA may have many more states.
- The smallest DFA for A contains eight states.



Example 1.33



- Consider the following NFA N_3 that has an input alphabet $\{0\}$ consisting of a single symbol.
- This machine demonstrates the convenience of having ε arrows.
- It accepts all strings of the form 0^k where k is a multiple of 2 or 3.
- N_3 accepts the strings ε, 00, 000, 0000, and 000000, but not 0 or 00000.
- Think of the machine operating by initially guessing whether to test for a multiple of 2 or a multiple of 3 and then checking whether its guess was correct.
- Of course, we could replace this machine by one that doesn't have ε arrows or even any nondeterminism at all, but the machine shown is the easiest one to understand for this language.

Formal Definition of a NFA

- The formal definition of a nondeterministic finite automaton is similar to that of a deterministic finite automaton.
- Both have states, an input alphabet, a transition function, a start state, and a collection of accept states.
- However, they differ in one essential way: in the type of transition function.
- In a DFA the transition function takes a state and an input symbol and produces the next state.
- In an NFA the transition function takes a state and an input symbol *or the empty string* and produces *the set of possible next states.*
- For any set Q we write $\mathcal{P}(Q)$ to be the collection of all subsets of Q. Here $\mathcal{P}(Q)$ is called the **power set** of Q.
- For any alphabet Σ we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$.
- Now we can write the formal description of the type of the transition function in an NFA as $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$.

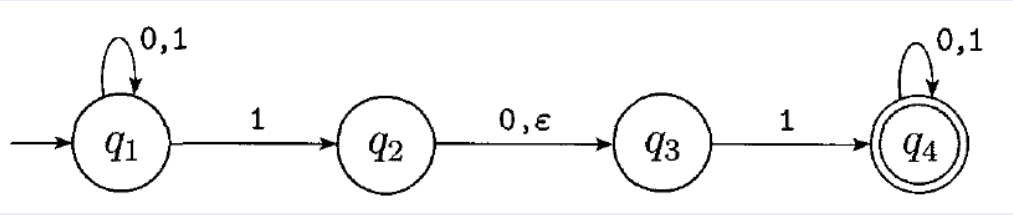
Formal Definition of a NFA

Definition 1.37

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

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

Example 1.38



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

The formal definition of computation for an NFA

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

Equivalence of NFAs and DFAs

- Deterministic and nondeterministic finite automata recognize the same class of languages.
- 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.

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.
- How would you simulate the NFA if you were pretending to be a DFA?
- In the examples of NFAs 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.
- 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, and what will be its transition function.
- We can discuss this more easily after setting up some formal notation.

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.
- 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.
  1. $Q' = \mathcal{P}(Q)$.
     Every state of M is a set of states of N. Recall that $\mathcal{P}(Q)$ is the set of subsets of Q.
  2. For $R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$. Another way for this expression is $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$.
     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.
  3. $q_0' = \{q_0\}$.
     M starts in the state corresponding to the collection containing just the start state of N.
  4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$.
     The machine M accepts if one of the possible states that N could be in at this point is an accept state.

Proof: 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 R by going only along ε arrows, including the members of R themselves.
- Formally, for $R \subseteq Q$ let $E(R) = \{q \mid q$ can be reached from R by traveling along 0 or more ε 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 $E(\{q_0\})$ achieves this effect.
- 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.

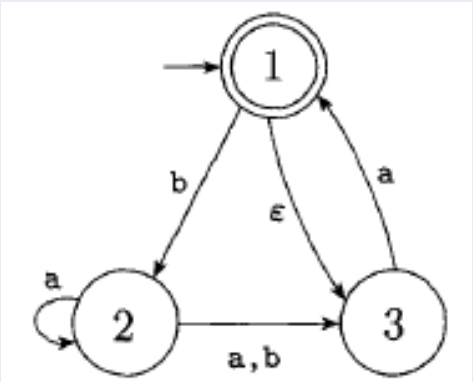
Every NFA can be converted into an equivalent DFA.

Corollary 1.40

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

- $\rightarrow$: Because a regular language has a DFA recognizing it and any DFA is also an NFA.
- $\leftarrow$: If an NFA recognizes some language, so does some DFA, and hence the language is regular.

Example 1.41: converting an NFA N_4 to a DFA

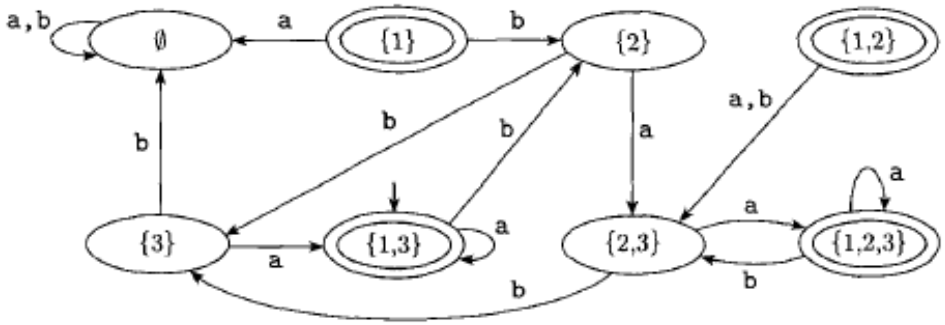


Example 1.41: converting an NFA N_4 to a DFA

- To construct a DFA D that is equivalent to N_4, we first determine D's states.
- N_4 has three states, $\{1, 2, 3\}$, so we construct D with eight states, one for each subset of N_4's states.
- Thus D's state set is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.
- Next, we determine the start and accept states of D.
- The start state is $E(\{1\})$, the set of states that are reachable from 1 by traveling along ε arrows, plus 1 itself.
- An ε arrow goes from 1 to 3, so $E(\{1\}) = \{1, 3\}$.
- The new accept states are those containing N_4's accept state; thus $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$.
- Finally, we determine D's transition function.
- Each of D's states goes to one place on input a and one place on input b.

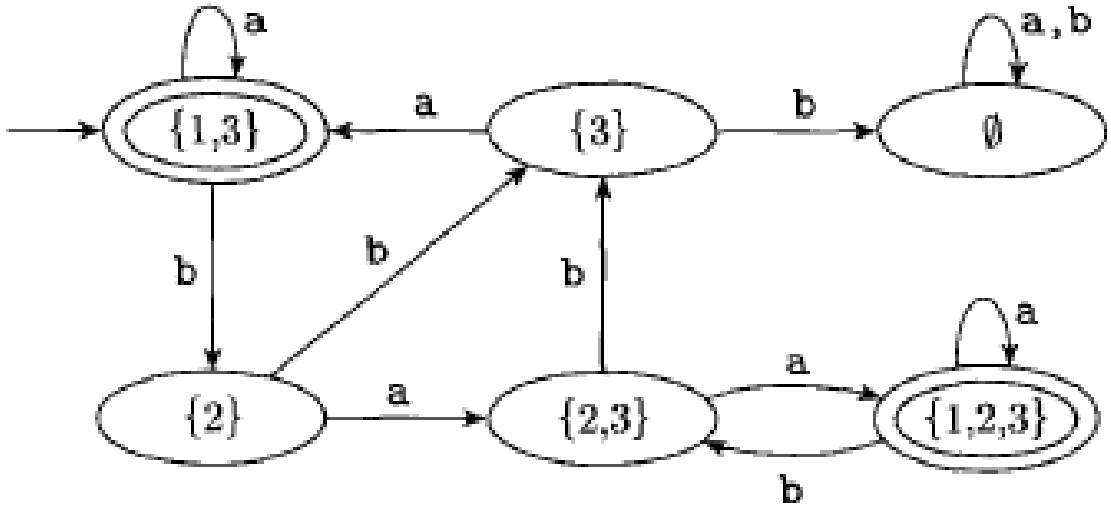
Example 1.41: converting an NFA N_4 to a DFA

- In D, state $\{2\}$ goes to $\{2, 3\}$ on input a, because in N_4, state 2 goes to both 2 and 3 on input a.
- State $\{2\}$ goes to state $\{3\}$ on input b, because in N_4, state 2 goes only to state 3 on input b.
- State $\{1\}$ goes to $\emptyset$ on a, because no a arrows exit it. It goes to $\{2\}$ on b.
- Note that the procedure in Theorem 1.39 specifies that we follow the ε arrows *after* each input symbol is read.
- State $\{3\}$ goes to $\{1, 3\}$ on a, because in N_4, state 3 goes to 1 on a and 1 in turn goes to 3 with an ε arrow.
- State $\{3\}$ on b goes to $\emptyset$.
- State $\{1, 2\}$ on a goes to $\{2, 3\}$ because 1 points at no states with a arrows and 2 points at both 2 and 3 with a arrows.
- State $\{1, 2\}$ on b goes to $\{2, 3\}$. $\cdots$



Example 1.41: converting an NFA N_4 to a DFA

- We may simplify this machine by observing that no arrows point at states $\{1\}$ and $\{1, 2\}$, so they may be removed without affecting the performance of the machine.
- Doing so yields the following figure.



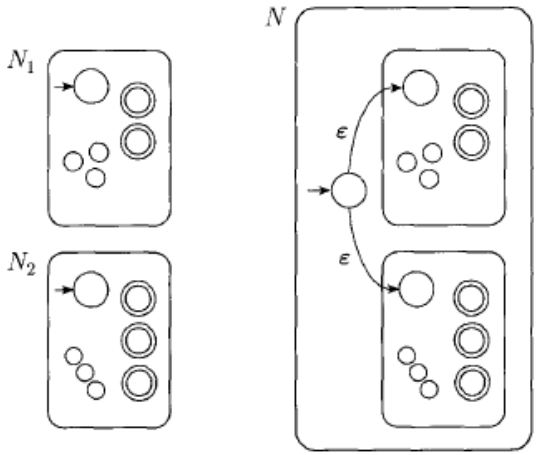
Closure under the Regular Operations

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

Theorem 1.45

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

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



Proof: Closure under Union

- Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.
- Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$.
  1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
     The states of N are all the states of N_1 and N_2, with the addition of a new start state q_0.
  2. The state q_0 is the start state of N.
  3. The accept states $F = F_1 \cup F_2$.
     The accept states of N are all the accept states of N_1 and N_2. 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 $a \in \Sigma_\varepsilon$,

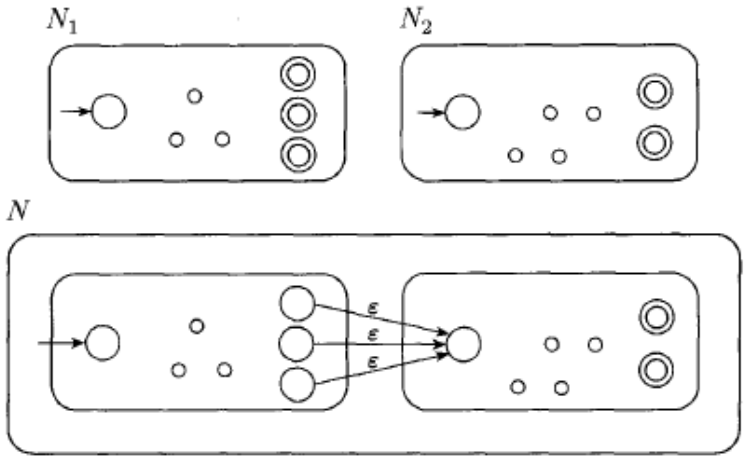
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

Closure under Concatenation

Theorem 1.47

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

- We have regular languages A_1 and A_2 and want to prove that $A_1 \circ A_2$ is regular.
- The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into one new NFA, N.
- Assign N's start state to be the start state of N_1.
- The accept states of N_1 have additional ε arrows that allow nondeterministically branching to N_2 whenever N_1 is in an accept state.
- The accept states of N are the accept states of N_2 only.



Proof: Closure under Concatenation

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

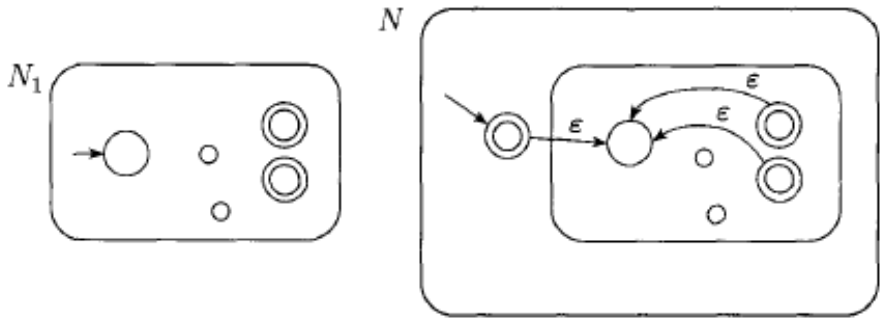
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

Closure under Star Operation

Theorem 1.49

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

- We have regular languages A_1 and want to prove that A_1^* also is regular.
- We take an NFA N_1 for A_1 and modify it to recognize A_1^*.
- The resulting NFA N will accept its input whenever it can be broken into several pieces and N_1 accepts each piece.
- We can construct N like N_1 with additional ε arrows returning to the start state from the accept states.
- In addition we must modify N so that it accepts ε.
- We add a new start state, which also is an accept state, and which has an ε arrow to the old start state.



Proof: Closure under Star Operation

- 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.
  2. The state q_0 is the new start state.
  3. $F = \{q_0\} \cup F_1$. The accept states are the old accept states plus the new start state.
  4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

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

1.3 Regular Expressions

- 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, which are called *regular expressions*.
- An example is: $(0 \cup 1)0^*$.
- The value of the arithmetic expression is the number 32.
- The value of a regular expression is a language.
- In this case, the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s.
- The symbols 0 and 1 are shorthand for the sets $\{0\}$ and $\{1\}$. So $(0 \cup 1)$ means $(\{0\} \cup \{1\})$.
- The value of this part is the language $\{0, 1\}$.
- The part 0^* means $\{0\}^*$, and its value is the language consisting of all strings containing any number of 0s.
- The concatenation symbol $\circ$ often is implicit in regular expressions.
- Thus $(0 \cup 1)0^*$ actually is shorthand for $(0 \cup 1) \circ 0^*$.

1.3 Regular Expressions

- Regular expressions have an important role in computer science applications.
- In applications involving text, users may want to search for strings that satisfy certain patterns.
- Text editors all provide mechanisms for the description of patterns by using regular expressions.

Example 1.51

Another example of a regular expression is $(0 \cup 1)^*$.

- It starts with the language $(0 \cup 1)$ and applies the * operation.
- The value of this expression is the language consisting of all possible strings of 0s and 1s.
- If $\Sigma = \{0, 1\}$, we can write Σ as shorthand for the regular expression $(0 \cup 1)$.
- More generally, if Σ is any alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet, and Σ^* describes the language consisting of all strings over that alphabet.

Formal Definition of a Regular Expression

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.

Formal Definition of a Regular Expression

- Don't confuse the regular expressions ε and $\emptyset$.
- The expression ε represents the language containing a single string—namely, the empty string—whereas $\emptyset$ represents the language that doesn't contain any strings.
- Parentheses in an expression may be omitted.
- If they are, evaluation is done in the precedence order: star, then concatenation, then union.
- We let R^+ be shorthand for RR^*. So $R^+ \cup \varepsilon = R^*$.
- When we want to distinguish between a regular expression R and the language that it describes, we write $L(R)$ to be the language of R.

Example 1.53

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. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}$.
4. $1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of three}\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$. Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
    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.

Regular Expressions

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

Regular Expressions

- Regular expressions are useful tools in the design of compilers for programming languages.
- Elemental objects in a programming language, called *tokens*, such as the variable names and constants, may be described with regular expressions.
- For example, a numerical constant that may include a fractional part and/or a sign may be described as a member of the language
- $(+ \cup - \cup \varepsilon)(D^+ \cup D^+.D^* \cup D^*.D^+)$
- where $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is the alphabet of decimal digits.
- Examples of generated strings are: 72, 3.14159, +7., and −.01.
- Once the syntax of the tokens of the programming language have been described with regular expressions, automatic systems can generate the *lexical analyzer*, the part of a compiler that initially processes the input program.

Equivalence with Finite Automata

- 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.

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.

Equivalence with Finite Automata

Lemma 1.55

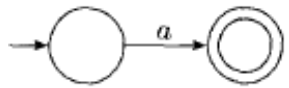
If a language is described by a regular expression, then it is regular.

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.

Proof

- Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.
  - $R = a$ for some a in Σ. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.

    Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here but an NFA is all we need for now, and it is easier to describe.
    Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$, or $b \neq a$.
  - $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.
    Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any r and b.

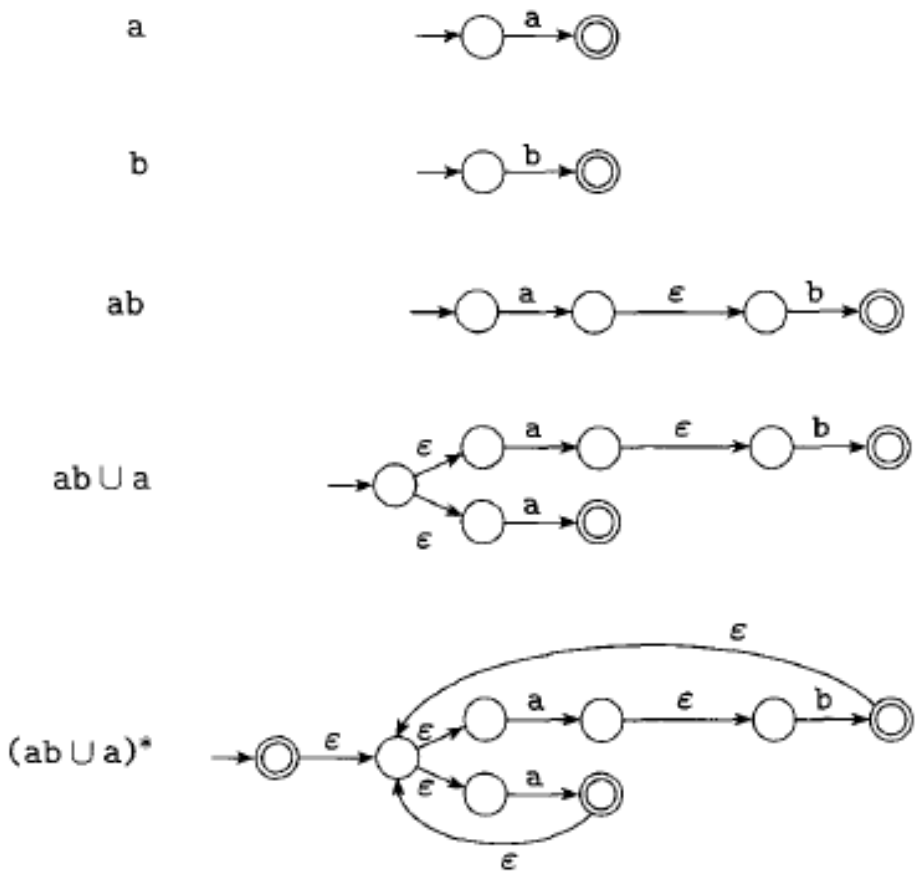
Proof

- Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.
  - $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$. Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b.

  - $R = R_1 \cup R_2$.
  - $R = R_1 \circ R_2$.
  - $R = R_1^*$.
  - For the last three cases 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 NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

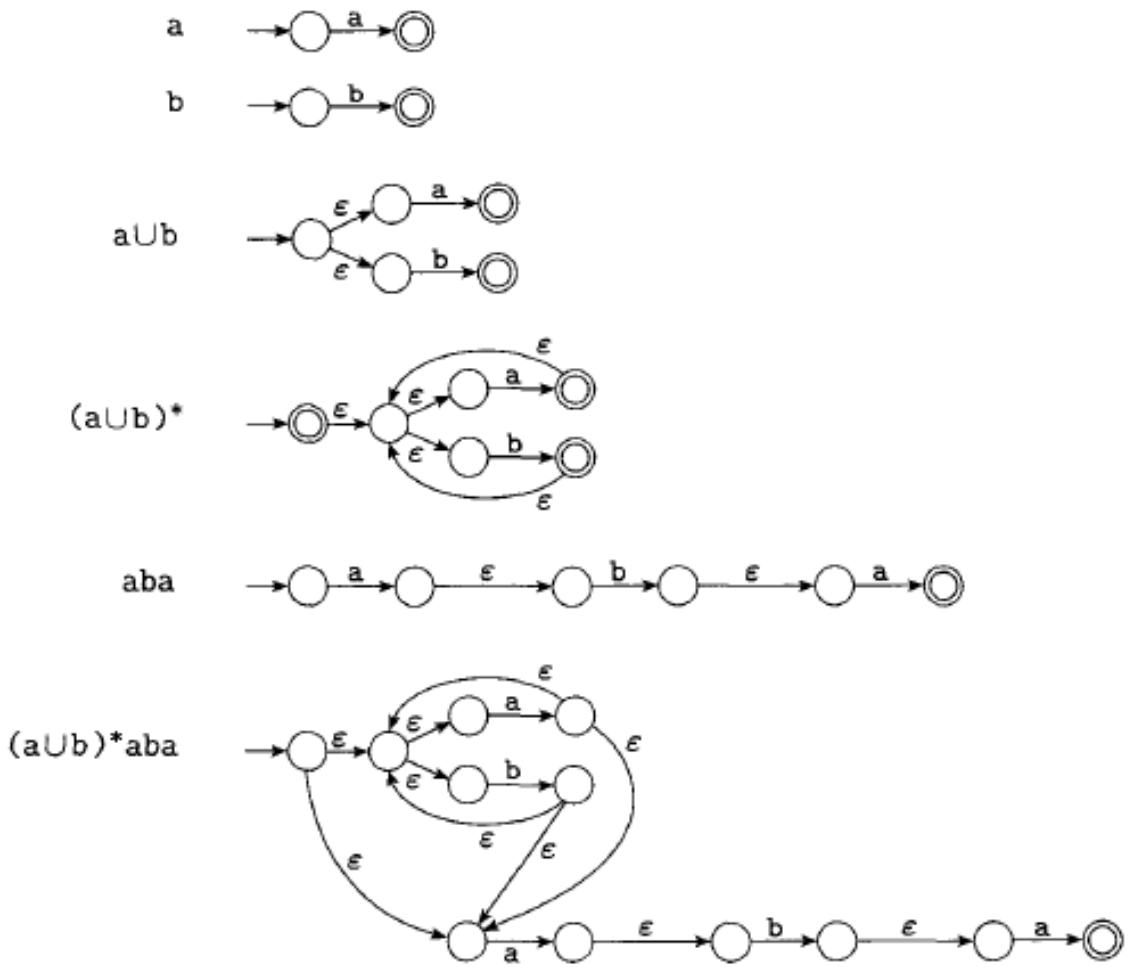
Example 1.56

- We convert the regular expression $(ab \cup a)^*$ to an NFA in a sequence of stages.
- We build up from the smallest subexpressions to larger subexpressions.



Example 1.58

- In Figure 1.59, we convert the regular expression $(a \cup b)^* aba$ to an NFA. A few of the minor steps are not shown.



Lemma 1.60

Lemma 1.60

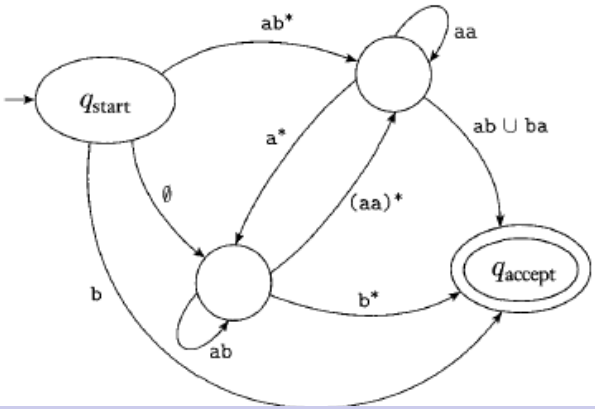
If a language is regular, then it is described by a regular expression.

Proof Idea

- We need to show that, if a language A is regular, a regular expression describes it.
- Because A is regular, it is accepted by a DFA.
- We describe a procedure for converting DFAs into equivalent regular expressions.
- We break this procedure into two parts, using a new type of finite automaton called a **generalized nondeterministic finite automaton**, GNFA.
- First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Proof Idea (continued)

- Generalized nondeterministic finite automata (GNFA) are simply NFAs wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ε.
- GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA.
- The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow.
- A GNFA is nondeterministic and so may have several different ways to process the same input string.
- It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input.

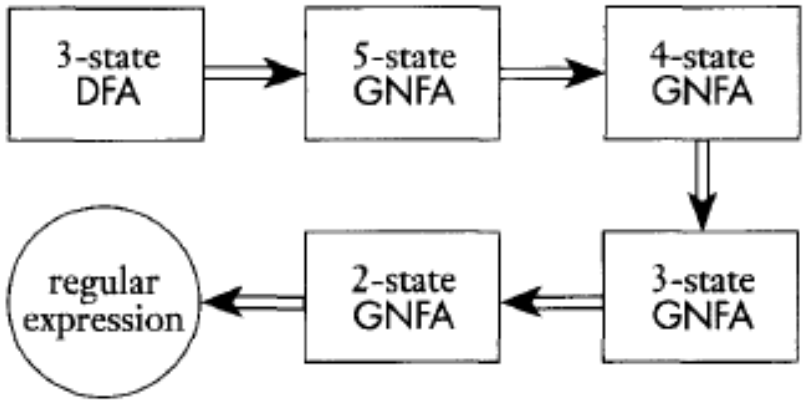


Proof Idea (continued)

- For convenience we require that GNFAs always have a special form that meets the following conditions.
  - The start state has transition arrows going to every other state but no arrows coming in from any other state.
  - There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
  - Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.
- We can easily convert a DFA into a GNFA in the special form.
- We simply add a new start state with an ε arrow to the old start state and a new accept state with ε arrows from the old accept states.
- If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels.
- Finally, we add arrows labeled $\emptyset$ between states that had no arrows.

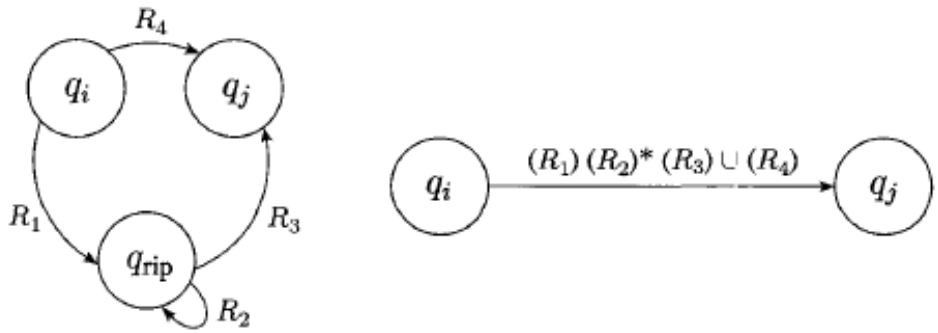
Proof Idea (continued)

- Now we show how to convert a GNFA into a regular expression.
- Say that the GNFA has k states.
- Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that $k > 2$.
- If $k > 2$, we construct an equivalent GNFA with $k - 1$ states.
- This step can be repeated on the new GNFA until it is reduced to two states.
- If $k = 2$, the GNFA has a single arrow that goes from the start state to the accept state.
- The label of this arrow is the equivalent regular expression.



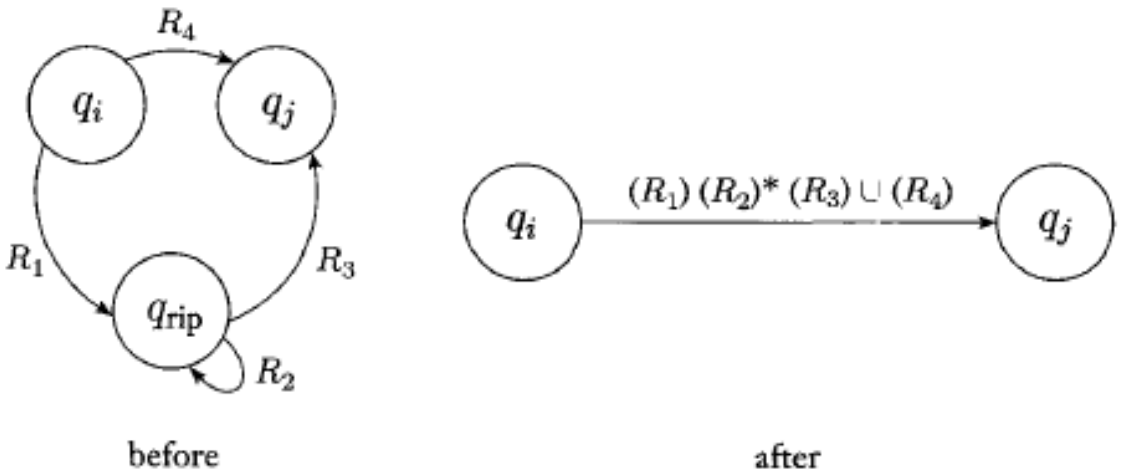
Proof Idea (continued)

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



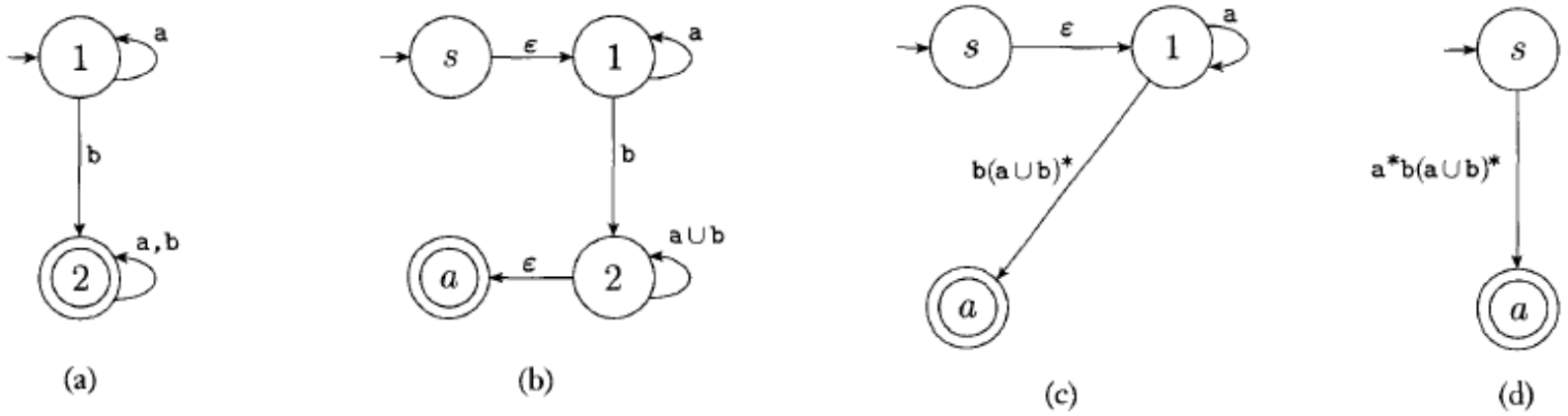
Proof Idea (continued)

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



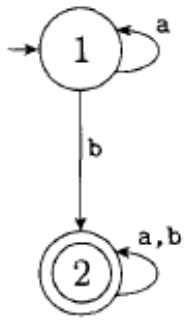
Example 1.66

- In this example we use the preceding algorithm to convert a DFA into a regular expression.
- We begin with the two-state DFA in Figure 1.67(a).
- In Figure 1.67(b) we make a four-state GNFA by adding a new start state and a new accept state, called s and a.
- To avoid cluttering up the figure, we do not draw the arrows labeled $\emptyset$, even though they are present.
- Note that we replace the label a, b on the self-loop at state 2 on the DFA with the label $a \cup b$ at the corresponding point on the GNFA.

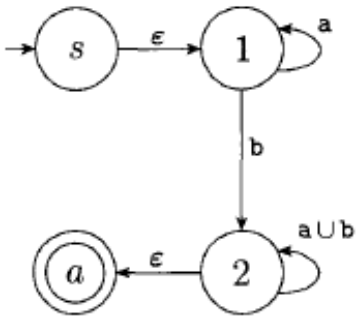


Example 1.66

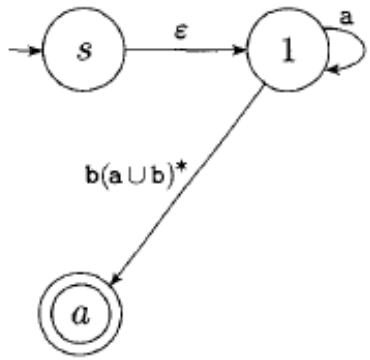
- In Figure 1.67(c) we remove state 2, and update the remaining arrow labels.
- In this case the only label that changes is the one from 1 to a.
- In part (b) it was $\emptyset$, but in part (c) it is $b(a \cup b)^*$.
- In Figure 1.67(d) we remove state 1 from part (c) and follow the same procedure.
- Because only the start and accept states remain, the label on the arrow joining them is the regular expression that is equivalent to the original DFA.
- What about removing state 1 first?
- Study Example 1.68 by yourself.



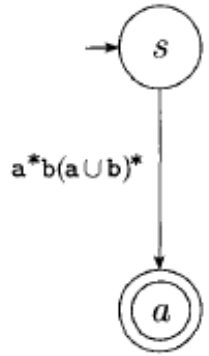
(a)



(b)



(c)



(d)

1.4 Nonregular Languages

- To understand the power of finite automata you must also understand their limitations.
- In this section we show how to prove that certain languages cannot be recognized by any finite automaton.
- Let's take the language $B = \{0^n 1^n \mid n \geq 0\}$.
- If we attempt to find a DFA that recognizes B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input.
- Because the number of 0s isn't limited, the machine will have to keep track of an unlimited number of possibilities.
- But it cannot do so with any finite number of states.
- $C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$.
- Next, we present a method for proving that languages such as B are not regular.

The Pumping Lemma for Regular Languages

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

The Pumping Lemma for Regular Languages

Theorem 1.70: **Pumping lemma**

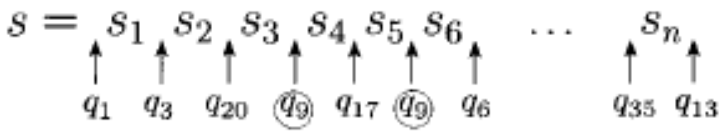
If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

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

- Recall the notation where $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.
- When s is divided into xyz, either x or z may be ε, but condition 2 says that $y \neq \varepsilon$.
- Condition 3 is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular. See Example 1.74 for an application of condition 3.

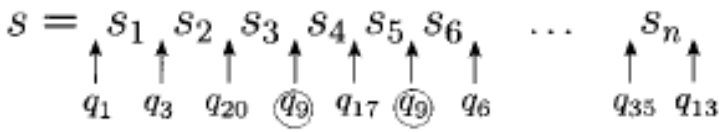
Proof Idea

- 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.
- 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.
- 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}.
- With s in A, we know that M accepts s, so q_{13} is an accept state.

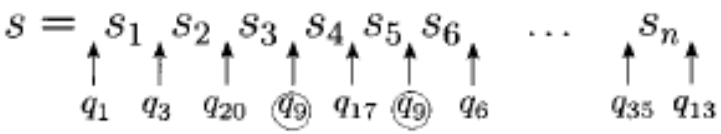


Proof Idea

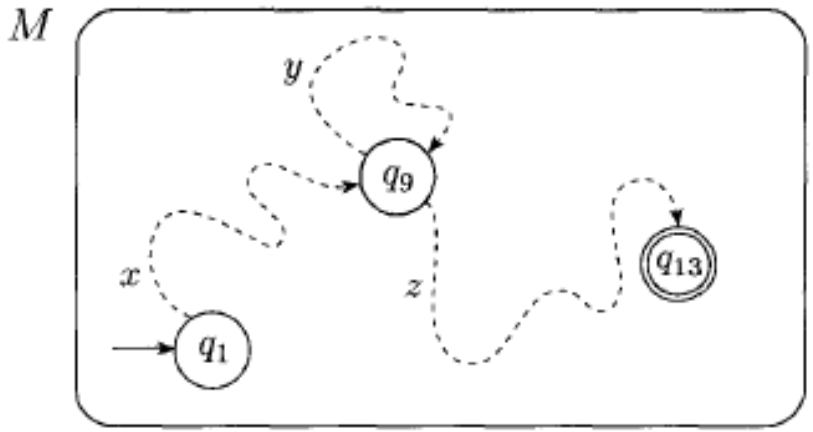
- If we let n be the length of s, the sequence of states $q_1, q_3, q_{20}, q_9, \ldots, q_{13}$ has length $n + 1$.
- Because n is at least p, we know that $n + 1$ is greater than p, the number of states of M.
- Therefore the sequence must contain a repeated state.
- This result is an example of the **pigeonhole principle**, a fancy name for the rather obvious fact that if p pigeons are placed into fewer than p holes, some hole has to have more than one pigeon in it.
- State q_9 is the one that repeats.



Proof Idea



- We now divide s into the three pieces x, y, and z.
- Piece x is the part of s appearing before q_9, piece y is the part between the two appearances of q_9, and piece z is the remaining part of s, coming after the second occurrence of q_9.
- So x takes M from the state q_1, to q_9, y takes M from q_9 back to q_9 and z takes M from q_9 to the accept state q_{13}.



Proof Idea

- Let's see why this division of s satisfies the three conditions.
- Suppose that we run M on input $xyyz$.
- We know that x takes M from q_1, to q_9, and then the first y takes it from q_9 back to q_9, as does the second y, and then z takes it to q_{13}.
- With q_{13} being an accept state, M accepts input $xyyz$.
- Similarly, it will accept xy^iz for any $i \geq 0$.
- For the case $i = 0$, $xy^iz = xz$, which is accepted for similar reasons.
- That establishes condition 1.
- 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.
- 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 $|xy| \leq p$.
- For the formal proof, you read the textbook.

The Pumping Lemma for Regular Languages

- 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) and, for each such division, finding a value i where $xy^iz \notin B$.
- The existence of s contradicts the assumption that B were regular.
- Hence B cannot be regular.

Example 1.73: Let B be the language $\{0^n1^n \mid n \geq 0\}$

- 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.
- 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.
  2. The string y consists only of 1s. This case also gives a contradiction.
  3. The string y consists of both 0s and 1s. In this case the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B, which is a contradiction.
- Note that we can simplify this argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3.

Example 1.74

- Let $C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$.
- We use the pumping lemma to prove that C is not regular.
- Assume that C is regular.
- Let p be the pumping length given by the pumping lemma.
- As in Example 1.73, 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.
- We would like to show that this outcome is impossible. 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.
- Here condition 3 in the pumping lemma is useful.
- It stipulates that when pumping s it must be divided so that $|xy| \leq p$.
- If $|xy| \leq p$, then y must consist only of 0s, so $xyyz \notin C$.
- Therefore s cannot be pumped. That gives us the desired contradiction.

Example 1.74

- Selecting the string s in this example required more care than in Example 1.73.
- If we had chosen $s = (01)^p$ instead, we would have run into trouble because we need a string that *cannot* be pumped and that string *can* be pumped, even taking condition 3 into account.
- One way to do so sets $x = \varepsilon, y = 01$, and $z = (01)^{p-1}$.
- Then $xy^iz \in C$ for every value of i.

Example 1.75

- Let $F = \{ww \mid w \in \{0,1\}^*\}$.
- We show that F is nonregular, using the pumping lemma.
- 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.
- We show that this outcome is impossible.
- With condition 3, the proof follows because y must consist only of 0s, so $xyyz \notin F$.
- Condition 3 is once again crucial, because without it we could pump s if we let x and z be the empty string.
- Even though 0^p0^p is a member of F, it fails to demonstrate a contradiction because it can be pumped.

Example 1.76: Let $D = \{1^{n^2} \mid n \geq 0\}$.

- We use the pumping lemma to prove that D is not regular.
- 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.
- Now consider the two strings xyz and xy^2z.
- These strings differ from each other by a single repetition of y, and consequently their lengths differ by the length of y.
- By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$.
- We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$.
- $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. So, $|xy^2z| < (p+1)^2$.
- 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 $(p+1)^2$; that is, $p^2 < |xy^2z| < (p+1)^2$.
- We arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular.

Example 1.77: Let $E = \{0^i 1^j \mid i > j\}$.

- Sometimes "pumping down" is useful when we apply the pumping lemma.
- Assume that E is regular.
- Let p be the pumping length for E given by the pumping lemma.
- Let $s = 0^{p+1}1^p$.
- s can be split into xyz, satisfying the conditions of the pumping lemma.
- By condition 3, y consists only of 0s.
- 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 $i = 0$, 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.