1

REGULAR LANGUAGES

The theory of computation begins with a question: What is a computer? It is perhaps a silly question, as everyone knows that this thing I type on is a computer. 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. What can a computer do with such a small 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. Often found at supermarket entrances and exits, automatic doors swing open when the controller senses that a person is approaching. 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. This configuration is shown in the following figure.

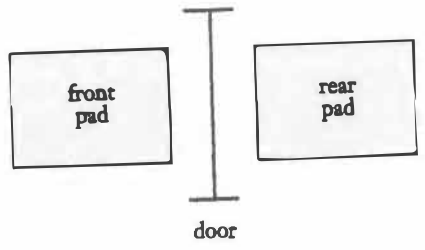

FIGURE 1.1
Top view of an automatic door

The controller is in either of two states: "OPEN" or "CLOSED," representing the corresponding condition of the door. As shown in the following figures, there are four possible input conditions: "FRONT" (meaning that a person is standing on the pad in front of the doorway), "REAR" (meaning that a person is standing on the pad to the rear of the doorway), "BOTH" (meaning that people are standing on both pads), and "NEITHER" (meaning that no one is standing on either pad).

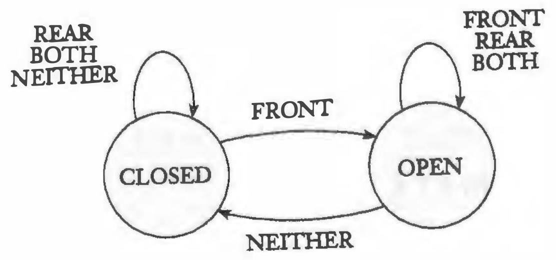

FIGURE 1.2
State diagram for an automatic door controller

		input signal			
		NEITHER	FRONT	REAR	BOTH
state	CLOSED	CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

FIGURE 1.3
State transition table for an automatic door controller

The controller moves from state to state, depending on the input it receives. When in the CLOSED state and receiving input NEITHER or REAR, it remains in the CLOSED state. In addition, if the input BOTH is received, it stays CLOSED because opening the door risks knocking someone over on the rear pad. But if the input FRONT arrives, it moves to the OPEN state. In the OPEN state, if input FRONT, REAR, or BOTH is received, it remains in OPEN. If input NEITHER arrives, it returns to CLOSED.

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 and 1.3. 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. In an elevator controller, a state may represent the floor the elevator is on and the inputs might be the signals received from the buttons. This computer might need several bits to keep track of this information. Controllers for various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators, are 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. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets.

We will now take a closer look at finite automata from a mathematical perspective. We will develop a precise definition of a finite automaton, terminology for describing and manipulating finite automata, and theoretical results that describe their power and limitations. Besides giving you a clearer understanding of what finite automata are and what they can and cannot do, this theoretical development will allow you to practice and become more comfortable with mathematical definitions, theorems, and proofs in a relatively simple setting.

In beginning to describe the mathematical theory of finite automata, we do so in the abstract, without reference to any particular application. The following figure depicts a finite automaton called M_1.

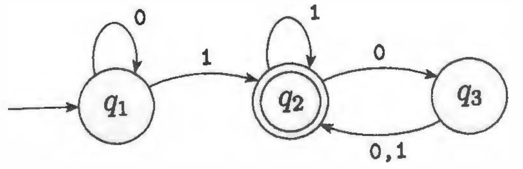

FIGURE 1.4
A finite automaton called M_1 that has three states

Figure 1.4 is called the ***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. The output is either *accept* or *reject*. We will consider only this yes/no type of output for now to keep things simple. The processing begins in M_1's start state. The automaton receives the symbols from the input string one by one from left to right. After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label. When it reads the last symbol, M_1 produces its output. The output is *accept* if M_1 is now in an accept state and *reject* if it is not.

For example, when we feed the input string 1101 into the machine M_1 in Figure 1.4, the processing proceeds as follows:

1. Start in state q_1.
2. Read 1, follow transition from q_1 to q_2.
3. Read 1, follow transition from q_2 to q_2.
4. Read 0, follow transition from q_2 to q_3.
5. Read 1, follow transition from q_3 to q_2.
6. *Accept* because M_1 is in an accept state q_2 at the end of the input.

Experimenting with this machine on a variety of input strings reveals that it accepts the strings 1, 01, 11, and 0101010101. In fact, M_1 accepts any string that ends with a 1, as it goes to its accept state q_2 whenever it reads the symbol 1. 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. Can you describe the language consisting of all strings that M_1 accepts? We will do so shortly.

FORMAL DEFINITION OF A FINITE AUTOMATON

In the preceding section, we used state diagrams to introduce finite automata. Now we define finite automata formally. Although state diagrams are easier to grasp intuitively, we need the formal definition, too, for two specific reasons.

First, a formal definition is precise. It resolves any uncertainties about what is allowed in a finite automaton. If you were uncertain about whether finite automata were allowed to have 0 accept states or whether they must have exactly one transition exiting every state for each possible input symbol, you could consult the formal definition and verify that the answer is yes in both cases. Second, a formal definition provides notation. Good notation helps you think and express your thoughts clearly.

The language of a formal definition is somewhat arcane, having some similarity to the language of a legal document. Both need to be precise, and every detail must be spelled out.

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a start state and a set of accept states. The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language, a list of five elements is often called a 5-tuple. Hence we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a ***transition function***, frequently denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol 1, that means that if the automaton is in state x when it reads a 1, it then moves to state y. We can indicate the same thing with the transition function by saying that $\delta(x, 1) = y$. This notation is a kind of mathematical shorthand. Putting it all together, we arrive at the formal definition of finite automata.

DEFINITION 1.5

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

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

[1]Refer back to page 7 if you are uncertain about the meaning of $\delta: Q \times \Sigma \longrightarrow Q$.
[2]Accept states sometimes are called ***final states***.

The formal definition precisely describes what we mean by a finite automaton. For example, returning to the earlier question of whether 0 accept states is allowable, you can see that setting F to be the empty set $\emptyset$ yields 0 accept states, which is allowable. Furthermore, the transition function δ specifies exactly one next state for each possible combination of a state and an input symbol. That answers our other question affirmatively, showing that exactly one transition arrow exits every state for each possible input symbol.

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. For example, let's return to the finite automaton M_1 we discussed earlier, redrawn here for convenience.

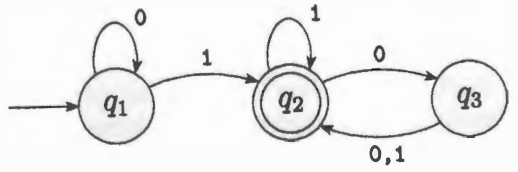

FIGURE 1.6
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

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

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

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

If A is the set of all strings that machine M accepts, we say that A is the ***language of machine*** M and write $L(M) = A$. We say that M ***recognizes*** A or that M ***accepts*** A. Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term *recognize* for languages in order to avoid confusion.

A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language—namely, the empty language $\emptyset$.

In our example, let

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

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

EXAMPLES OF FINITE AUTOMATA

EXAMPLE 1.7

Here is the state diagram of finite automaton M_2.

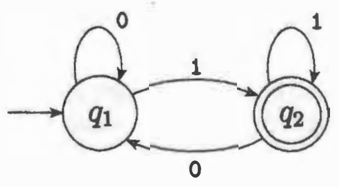

FIGURE 1.8
State diagram of the two-state finite automaton M_2

In the formal description, M_2 is $(\{q_1, q_2\}, \{0,1\}, \delta, q_1, \{q_2\})$. The transition function δ is

	0	1
q_1	q_1	q_2
q_2	q_1	q_2.

Remember that the state diagram of M_2 and the formal description of M_2 contain the same information, only in different forms. You can always go from one to the other if necessary.

A good way to begin understanding any machine is to try it on some sample input strings. When you do these "experiments" to see how the machine is working, its method of functioning often becomes apparent. On the sample string 1101, the machine M_2 starts in its start state q_1 and proceeds first to state q_2 after reading the first 1, and then to states q_2, q_1, and q_2 after reading 1, 0, and 1. The string is accepted because q_2 is an accept state. But string 110 leaves M_2 in state q_1, so it is rejected. After trying a few more examples, you would see that M_2 accepts all strings that end in a 1. Thus $L(M_2) = \{w \mid w \text{ ends in a } 1\}$.

EXAMPLE 1.9

Consider the finite automaton M_3.

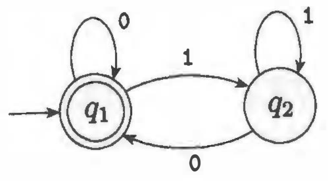

FIGURE 1.10
State diagram of the two-state finite automaton M_3

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

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

EXAMPLE 1.11

The following figure shows a five-state machine M_4.

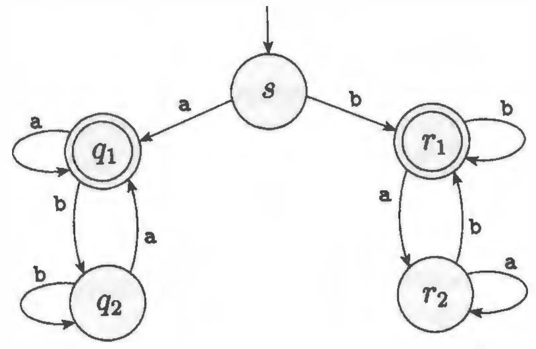

FIGURE 1.12
Finite automaton M_4

Machine M_4 has two accept states, q_1 and r_1, and operates over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$. Some experimentation shows that it accepts strings a, b, aa, bb, and bab, but not strings ab, ba, or bbba. This machine begins in state s, and after it reads the first symbol in the input, it goes either left into the q states or right into the r states. In both cases, it can never return to the start state (in contrast to the previous examples), as it has no way to get from any other state back to s. If the first symbol in the input string is a, then it goes left and accepts when the string ends with an a. Similarly, if the first symbol is a b, the machine goes right and accepts when the string ends in b. So M_4 accepts all strings that start and end with a or that start and end with b. In other words, M_4 accepts strings that start and end with the same symbol.

EXAMPLE 1.13

Figure 1.14 shows the three-state machine M_5, which has a four-symbol input alphabet, $\Sigma = \{\langle\text{RESET}\rangle, 0, 1, 2\}$. We treat $\langle\text{RESET}\rangle$ as a single symbol.

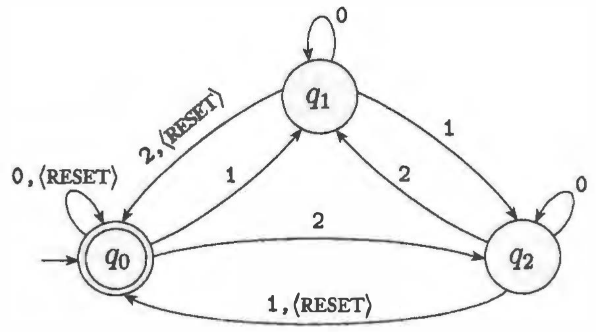

FIGURE 1.14
Finite automaton M_5

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.

Describing a finite automaton by state diagram is not possible in some cases. That may occur when the diagram would be too big to draw or if, as in the next example, the description depends on some unspecified parameter. In these cases, we resort to a formal description to specify the machine.

EXAMPLE 1.15

Consider a generalization of Example 1.13, using the same four-symbol alphabet Σ. For each $i \geq 1$ let A_i be the language of all strings where the sum of the numbers is a multiple of i, except that the sum is reset to 0 whenever the symbol $\langle\text{RESET}\rangle$ appears. For each A_i we give a finite automaton B_i, recognizing A_i. We describe the machine B_i formally as follows: $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, where Q_i is the set of i states $\{q_0, q_1, q_2, \ldots, q_{i-1}\}$, and we design the transition function δ_i so that for each j, if B_i is in q_j, the running sum is j, modulo i. For each q_j let

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

FORMAL DEFINITION OF COMPUTATION

So far we have described finite automata informally, using state diagrams, and with a formal definition, as a 5-tuple. The informal description is easier to grasp at first, but the formal definition is useful for making the notion precise, resolving any ambiguities that may have occurred in the informal description. 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. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w \mid M \text{ accepts } w\}$.

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

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

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. As such, it cannot be reduced to a simple recipe or formula. However, you might find a particular approach helpful when designing various types of automata. That is, 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. Pretending that you are the machine is a psychological trick that helps engage your whole mind in the design process.

Let's design a finite automaton using the "reader as automaton" method just described. 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 the automaton is supposed to recognize. 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. The reason is that you, like the machine, don't know when the end of the string is coming, so you must always be ready with the answer.

First, in order to make these decisions, you have to figure out what you need to remember about the string as you are reading it. Why not simply remember all you have seen? Bear in mind that you are pretending to be a finite automaton and that this type of machine has only a finite number of states, which means a finite memory. Imagine that the input is extremely long—say, from here to the moon—so that you could not possibly remember the entire thing. You have a finite memory—say, a single sheet of paper—which has a limited storage capacity. Fortunately, for many languages you don't need to remember the entire input. You need to remember only certain crucial information. Exactly which information is crucial depends on the particular language considered.

For example, suppose that the alphabet is $\{0,1\}$ and that the language consists of all strings with an odd number of 1s. You want to construct a finite automaton E_1 to recognize this language. Pretending to be the automaton, 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? Of course not. 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. If you read a 1, flip the answer; but if you read a 0, leave the answer as is.

But how does this help you design E_1? Once you have determined the necessary information to remember about the string as it is being read, 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.

Then you assign a state to each of the possibilities. These are the states of E_1, as shown here.

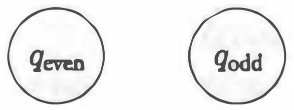

FIGURE 1.18
The two states q_{even} and q_{odd}

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, as shown here.

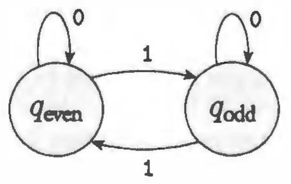

FIGURE 1.19
Transitions telling how the possibilities rearrange

Next, you set the start state to be the state corresponding to the possibility associated with having seen 0 symbols so far (the empty string ε). In this case, the start state corresponds to state q_{even} because 0 is an even number. Last, set the accept states to be those corresponding to possibilities where you want to accept the input string. Set q_{odd} to be an accept state because you want to accept

when you have seen an odd number of 1s. These additions are shown in the following figure.

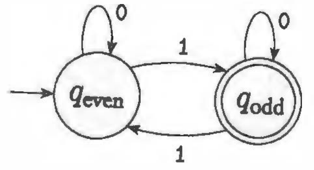

FIGURE 1.20
Adding the start and accept states

EXAMPLE 1.21

This example shows how to design a finite automaton E_2 to recognize the regular language of all strings that contain the string 001 as a substring. For example, 0010, 1001, 001, and 11111110011111 are all in the language, but 11 and 0000 are not. How would you recognize this language if you were pretending to be E_2? 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.

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}, as shown in Figure 1.22.

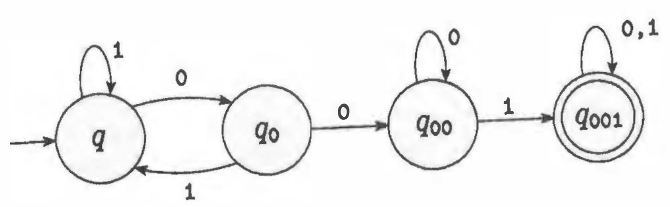

FIGURE 1.22
Accepts strings containing 001

THE REGULAR OPERATIONS

In the preceding two sections, 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 for designing automata to recognize particular languages. The toolbox also will include ways of proving that certain other languages are nonregular (i.e., beyond the capability of finite automata).

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as + and ×. In the theory of computation, the objectstare 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:** $A \cup B = \{x \mid x \in A \text{ or } x \in$t$B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star:** A^*t$= \{x_1x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in$t$A\}$.

You are already familiar with the union operation. It simply takes all the strings in both A and B and lumps them together into one language.

The concatenation operation is a little trickier. It attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

The star operation is a bit different from the other two because it applies to a single language rather than to two different languages. That is, the star operation is a ***unary operation*** instead of a ***binary operation***. It works by attaching any number of strings in A together to get a string in the new language. Because

"any number" includes 0 as a possibility, the empty string ε is always a member of A^*, no matter what A is.

EXAMPLE 1.24

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

$A \cup B = \{\text{good}, \text{bad}, \text{boy}, \text{girl}\}$,

$A \circ B = \{\text{goodboy}, \text{goodgirl}, \text{badboy}, \text{badgirl}\}$, and

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

Let $\mathcal{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers. When we say that $\mathcal{N}$ is ***closed under multiplication***, we mean that for any x and y in $\mathcal{N}$, the product $x \times y$ also is in $\mathcal{N}$. In contrast, $\mathcal{N}$ is not closed under division, as 1 and 2 are in $\mathcal{N}$ but 1/2 is not. Generally speaking, a collection of objects is ***closed*** under some operation if applying that operation to members of the collection returns an object still in the collection. We show that the collection of regular languages is closed under all three of the regular operations. In Section 1.3, we show that these are useful tools for manipulating regular languages and understanding the power of finite automata. We begin with the 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 We have regular languages A_1 and A_2 and want to show that $A_1 \cup A_2$ also is regular. Because A_1 and A_2 are regular, we know that some finite automaton M_1 recognizes A_1 and some finite automaton M_2 recognizes A_2. To prove that $A_1 \cup A_2$ is regular, we demonstrate a finite automaton, call it M, that recognizes $A_1 \cup A_2$.

This is a proof by construction. We construct M from M_1 and M_2. Machine M must accept its input exactly when either M_1 or M_2 would accept it in order to recognize the union language. It works by ***simulating*** both M_1 and M_2 and accepting if either of the simulations accept.

How can we make machine M simulate M_1 and M_2? Perhaps it first simulates M_1 on the input and then simulates M_2 on the input. But we must be careful here! Once the symbols of the input have been read and used to simulate M_1, we can't "rewind the input tape" to try the simulation on M_2. We need another approach.

Pretend that you are M. As the input symbols arrive one by one, you simulate both M_1 and M_2 simultaneously. That way, only one pass through the input is necessary. But can you keep track of both simulations with finite memory? All you need to remember is the state that each machine would be in if it had read up to this point in the input. Therefore, you need to remember a pair of states. How many possible pairs are there? If M_1 has k_1 states and M_2 has k_2 states, the number of pairs of states, one from M_1 and the other from M_2, is the product $k_1 \times k_2$. This product will be the number of states in M, one for each pair. The 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

Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and
M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) | \, r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
   This set is the ***Cartesian product*** of sets Q_1 and Q_2 and is written $Q_1 \times Q_2$. It is the set of all pairs of states, the first from Q_1 and the second from Q_2.
2. Σ, the alphabet, is the same as in M_1 and M_2. In this theorem and in all subsequent similar theorems, we assume for simplicity that both M_1 and M_2 have the same input alphabet Σ. The theorem remains true if they have different alphabets, Σ_1 and Σ_2. We would then modify the proof to let $\Sigma = \Sigma_1 \cup \Sigma_2$.
3. δ, the transition function, is defined as follows. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let
   $$\delta\big((r_1, r_2), a\big) = \big(\delta_1(r_1, a), \delta_2(r_2, a)\big).$$
   Hence δ gets a state of M (which actually is a pair of states from M_1 and M_2), together with an input symbol, and returns M's next state.
4. q_0 is the pair (q_1, q_2).
5. F is the set of pairs in which either member is an accept state of M_1 or M_2. We can write it as
   $$F = \{(r_1, r_2) | \, r_1 \in F_1 \text{ or } r_2 \in F_2\}.$$
   This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Note that it is *not* the same as $F = F_1 \times F_2$. What would that give us instead?[3])

[3] This expression would define M's accept states to be those for which *both* members of the pair are accept states. In this case, M would accept a string only if both M_1 *and* M_2 accept it, so the resulting language would be the *intersection* and not the union. In fact, this result proves that the class of regular languages is closed under intersection.