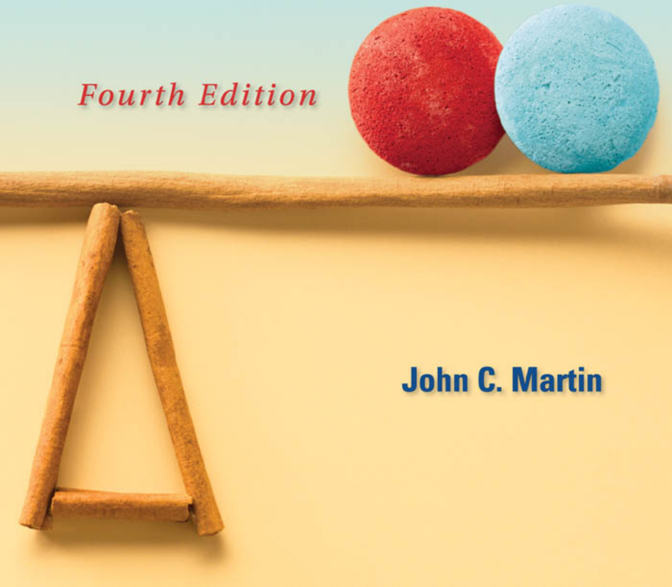
Introduction to Languages *and the* Theory of Computation

Fourth Edition



John C. Martin

Introduction to Languages and The Theory of Computation

Fourth Edition

John C. Martin
North Dakota State University

Mc Graw Hill
Connect Learn Succeed™

INTRODUCTION

Computers play such an important part in our lives that formulating a "theory of computation" threatens to be a huge project. To narrow it down, we adopt an approach that seems a little old-fashioned in its simplicity but still allows us to think systematically about what computers do. Here is the way we will think about a computer: It receives some input, in the form of a string of characters; it performs some sort of "computation"; and it gives us some output.

In the first part of this book, it's even simpler than that, because the questions we will be asking the computer can all be answered either yes or no. For example, we might submit an input string and ask, "Is it a legal algebraic expression?" At this point the computer is playing the role of a *language acceptor*. The language accepted is the set of strings to which the computer answers yes—in our example, the language of legal algebraic expressions. Accepting a language is approximately the same as solving a *decision problem*, by receiving a string that represents an *instance* of the problem and answering either yes or no. Many interesting computational problems can be formulated as decision problems, and we will continue to study them even after we get to models of computation that are capable of producing answers more complicated than yes or no.

If we restrict ourselves for the time being, then, to computations that are supposed to solve decision problems, or to accept languages, then we can adjust the level of complexity of our model in one of two ways. The first is to vary the problems we try to solve or the languages we try to accept, and to formulate a model appropriate to the level of the problem. Accepting the language of legal algebraic expressions turns out to be moderately difficult; it can't be done using the first model of computation we discuss, but we will get to it relatively early in the book. The second approach is to look at the computations themselves: to say at the outset how sophisticated the steps carried out by the computer are allowed to be, and to see what sorts of languages can be accepted as a result. Our first model, a *finite automaton*, is characterized by its lack of any auxiliary memory, and a language accepted by such a device can't require the acceptor to remember very much information during its computation.

A finite automaton proceeds by moving among a finite number of distinct states in response to input symbols. Whenever it reaches an *accepting* state, we think of it as giving a "yes" answer for the string of input symbols it has received so far. Languages that can be accepted by finite automata are regular languages; they can be described by either *regular expressions* or *regular grammars*, and generated by combining one-element languages using certain simple operations. One step up from a finite automaton is a *pushdown automaton*, and the languages these devices accept can be generated by more general grammars called *context-free* grammars. Context-free grammars can describe much of the syntax of high-level programming languages, as well as related languages like legal algebraic expressions and balanced strings of parentheses. The most general model of computation we will study is the Turing machine, which can in principle carry out any algorithmic procedure. It is as powerful as any computer. Turing machines accept recursively enumerable languages, and one way of generating these is to use *unrestricted* grammars.

Turing machines do not represent the only general model of computation, and in Chapter 10 we consider Kleene's alternative approach to computability. The class of computable functions, which turn out to be the same as the Turing-computable ones, can be described by specifying a set of "initial" functions and a set of operations that can be applied to functions to produce new ones. In this way the computable functions can be characterized in terms of the operations that can actually be carried out algorithmically.

As powerful as the Turing machine model is potentially, it is not especially user-friendly, and a Turing machine leaves something to be desired as an actual computer. However, it can be used as a yardstick for comparing the inherent complexity of one solvable problem to that of another. A simple criterion involving the number of steps a Turing machine needs to solve a problem allows us to distinguish between problems that can be solved in a reasonable time and those that can't. At least, it allows us to distinguish between these two categories in principle; in practice it can be very difficult to determine which category a particular problem is in. In the last chapter, we discuss a famous open question in this area, and look at some of the ways the question has been approached.

The fact that these elements (abstract computing devices, languages, and various types of grammars) fit together so nicely into a theory is reason enough to study them—for people who enjoy theory. If you're not one of those people, or have not been up to now, here are several other reasons.

The algorithms that finite automata can execute, although simple by definition, are ideally suited for some computational problems—they might be the algorithms of choice, even if we have computers with lots of horsepower. We will see examples of these algorithms and the problems they can solve, and some of them are directly useful in computer science. Context-free grammars and push-down automata are used in software form in compiler design and other eminently practical areas.

A model of computation that is inherently simple, such as a finite automaton, is one we can understand thoroughly and describe precisely, using appropriate mathematical notation. Having a firm grasp of the principles governing these devices makes it easier to understand the notation, which we can then apply to more complicated models of computation.

A Turing machine is simpler than any actual computer, because it is abstract. We can study it, and follow its computation, without becoming bogged down by hardware details or memory restrictions. A Turing machine is an implementation of an algorithm. Studying one in detail is equivalent to studying an algorithm, and studying them in general is a way of studying the algorithmic method. Having a precise model makes it possible to identify certain types of computations that Turing machines cannot carry out. We said earlier that Turing machines accept recursively enumerable languages. These are not *all* languages, and Turing machines can't solve every problem. When we find a problem a finite automaton can't solve, we can look for a more powerful type of computer, but when we find a problem that can't be solved by a Turing machine (and we will discuss several examples of such "undecidable" problems), we have found a limitation of the algorithmic method.

CHAPTER 1

Mathematical Tools and Techniques

When we discuss formal languages and models of computation, the definitions will rely mostly on familiar mathematical objects (logical propositions and operators, sets, functions, and equivalence relations) and the discussion will use common mathematical techniques (elementary methods of proof, recursive definitions, and two or three versions of mathematical induction). This chapter lays out the tools we will be using, introduces notation and terminology, and presents examples that suggest directions we will follow later.

The topics in this chapter are all included in a typical beginning course in discrete mathematics, but you may be more familiar with some than with others. Even if you have had a discrete math course, you will probably find it helpful to review the first three sections. You may want to pay a little closer attention to the last three, in which many of the approaches that characterize the subjects in this course first start to show up.

1.1 | LOGIC AND PROOFS

In this first section, we consider some of the ingredients used to construct logical arguments. Logic involves *propositions*, which have *truth values*, either the value *true* or the value *false*. The propositions "$0 = 1$" and "peanut butter is a source of protein" have truth values *false* and *true*, respectively. When a simple proposition, which has no variables and is not constructed from other simpler propositions, is used in a logical argument, its truth value is the only information that is relevant.

A proposition involving a variable (a *free* variable, terminology we will explain shortly) may be true or false, depending on the value of the variable. If the domain, or set of possible values, is taken to be $\mathcal{N}$, the set of nonnegative integers, the proposition "$x - 1$ is prime" is true for the value $x = 8$ and false when $x = 10$.

Compound propositions are constructed from simpler ones using *logical connectives*. We will use five connectives, which are shown in the table below. In each case, p and q are assumed to be propositions.

Connective	Symbol	Typical Use	English Translation
conjunction	$\wedge$	$p \wedge q$	p and q
disjunction	$\vee$	$p \vee q$	p or q
negation	$\neg$	$\neg p$	not p
conditional	$\rightarrow$	$p \rightarrow q$	if p then q
			p only if q
biconditional	$\leftrightarrow$	$p \leftrightarrow q$	p if and only if q

Each of these connectives is defined by saying, for each possible combination of truth values of the propositions to which it is applied, what the truth value of the result is. The truth value of $\neg p$ is the opposite of the truth value of p. For the other four, the easiest way to present this information is to draw a *truth table* showing the four possible combinations of truth values for p and q.

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Many of these entries don't require much discussion. The proposition $p \wedge q$ ("p and q") is true when both p and q are true and false in every other case. "p or q" is true if either or both of the two propositions p and q are true, and false only when they are both false.

The conditional proposition $p \rightarrow q$, "if p then q", is defined to be false when p is true and q is false; one way to understand why it is defined to be true in the other cases is to consider a proposition like

$$x < 1 \rightarrow x < 2$$

where the domain associated with the variable x is the set of natural numbers. It sounds reasonable to say that this proposition ought to be true, no matter what value is substituted for x, and you can see that there is no value of x that makes $x < 1$ true and $x < 2$ false. When $x = 0$, both $x < 1$ and $x < 2$ are true; when $x = 1$, $x < 1$ is false and $x < 2$ is true; and when $x = 2$, both $x < 1$ and $x < 2$ are false; therefore, the truth table we have drawn is the only possible one if we want this compound proposition to be true in every case.

In English, the word order in a conditional statement can be changed without changing the meaning. The proposition $p \rightarrow q$ can be read either "if p then q" or "q if p". In both cases, the "if" comes right before p. The other way to read $p \rightarrow q$, "p only if q", may seem confusing until you realize that "only if" and "if" mean different things. The English translation of the biconditional statement

$p \leftrightarrow q$ is a combination of "p if q" and "p only if q". The statement is true when the truth values of p and q are the same and false when they are different.

Once we have the truth tables for the five connectives, finding the truth values for an arbitrary compound proposition constructed using the five is a straightforward operation. We illustrate the process for the proposition

$$(p \vee q) \wedge \neg(p \to q)$$

We begin filling in the table below by entering the values for p and q in the two leftmost columns; if we wished, we could copy one of these columns for each occurrence of p or q in the expression. The order in which the remaining columns are filled in (shown at the top of the table) corresponds to the order in which the operations are carried out, which is determined to some extent by the way the expression is parenthesized.

		1	**4**	**3**	**2**
$\boldsymbol{p}$	$\boldsymbol{q}$	$\boldsymbol{(p \vee q)}$	$\boldsymbol{\wedge}$	$\boldsymbol{\neg}$	$\boldsymbol{(p \to q)}$
T	T	T	F	F	T
T	F	T	T	T	F
F	T	T	F	F	T
F	F	F	F	F	T

The first two columns to be computed are those corresponding to the subexpressions $p \vee q$ and $p \to q$. Column 3 is obtained by negating column 2, and the final result in column 4 is obtained by combining columns 1 and 3 using the $\wedge$ operation.

A *tautology* is a compound proposition that is true for every possible combination of truth values of its constituent propositions—in other words, true in every case. A *contradiction* is the opposite, a proposition that is false in every case. The proposition $p \vee \neg p$ is a tautology, and $p \wedge \neg p$ is a contradiction. The propositions p and $\neg p$ by themselves, of course, are neither.

According to the definition of the biconditional connective, $p \leftrightarrow q$ is true precisely when p and q have the same truth values. One type of tautology, therefore, is a proposition of the form $P \leftrightarrow Q$, where P and Q are compound propositions that are *logically equivalent*—i.e., have the same truth value in every possible case. Every proposition appearing in a formula can be replaced by any other logically equivalent proposition, because the truth value of the entire formula remains unchanged. We write $P \Leftrightarrow Q$ to mean that the compound propositions P and Q are logically equivalent. A related idea is *logical implication*. We write $P \Rightarrow Q$ to mean that in every case where P is true, Q is also true, and we describe this situation by saying that P logically implies Q.

The proposition $P \to Q$ and the assertion $P \Rightarrow Q$ look similar but are different kinds of things. $P \to Q$ is a proposition, just like P and Q, and has a truth value in each case. $P \Rightarrow Q$ is a "meta-statement", an assertion about the relationship between the two propositions P and Q. Because of the way we have defined the conditional, the similarity between them can be accounted for by observing that $P \Rightarrow Q$ means $P \rightarrow Q$ is a tautology. In the same way, as we have already observed, $P \Leftrightarrow Q$ means that $P \leftrightarrow Q$ is a tautology.

There is a long list of logical identities that can be used to simplify compound propositions. We list just a few that are particularly useful; each can be verified by observing that the truth tables for the two equivalent statements are the same.

The commutative laws:	$p \vee q \Leftrightarrow q \vee p$
	$p \wedge q \Leftrightarrow q \wedge p$
The associative laws:	$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$
	$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
The distributive laws:	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
The De Morgan laws:	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$

Here are three more involving the conditional and biconditional.

$$(p \rightarrow q) \Leftrightarrow (\neg p \vee q)$$
$$(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$$
$$(p \leftrightarrow q) \Leftrightarrow ((p \rightarrow q) \wedge (q \rightarrow p))$$

The first and third provide ways of expressing $\rightarrow$ and $\leftrightarrow$ in terms of the three simpler connectives $\vee$, $\wedge$, and $\neg$. The second asserts that the conditional proposition $p \rightarrow q$ is equivalent to its *contrapositive*. The *converse* of $p \rightarrow q$ is $q \rightarrow p$, and these two propositions are not equivalent, as we suggested earlier in discussing *if* and *only if*.

We interpret a proposition such as "$x - 1$ is prime", which we considered earlier, as a statement about x, which may be true or false depending on the value of x. There are two ways of attaching a *logical quantifier* to the beginning of the proposition; we can use the universal quantifier "for every", or the existential quantifier "for some". We will write the resulting quantified statements as

$$\forall x(x - 1 \text{ is prime})$$
$$\exists x(x - 1 \text{ is prime})$$

In both cases, what we have is no longer a statement about x, which still appears but could be given another name without changing the meaning, and it no longer makes sense to substitute an arbitrary value for x. We say that x is no longer a free variable, but is bound to the quantifier. In effect, the statement has become a statement about the domain from which possible values may be chosen for x. If as before we take the domain to be the set $\mathcal{N}$ of nonnegative integers, the first statement is false, because "$x - 1$ is prime" is not true for every x in the domain (it is false when $x = 10$). The second statement, which is often read "there exists x such that $x - 1$ is prime", is true; for example, $8 - 1$ is prime.

An easy way to remember the notation for the two quantifiers is to think of $\forall$ as an upside-down A, for "all", and to think of $\exists$ as a backward E, for "exists". Notation for quantified statements sometimes varies; we use parentheses in order to specify clearly the *scope* of the quantifier, which in our example is the statement "$x - 1$ is prime". If the quantified statement appears within a larger formula, then an appearance of x outside the scope of this quantifier means something different.

We assume, unless explicitly stated otherwise, that in statements containing two or more quantifiers, the same domain is associated with all of them. Being able to understand statements of this sort requires paying particular attention to the scope of each quantifier. For example, the two statements

$$\forall x(\exists y((x < y))$$
$$\exists y(\forall x((x < y))$$

are superficially similar (the same variables are bound to the same quantifiers, and the inequalities are the same), but the statements do not express the same idea. The first says that for every x, there is a y that is larger. This is true if the domain in both cases is $\mathcal{N}$, for example. The second, on the other hand, says that there is a single y such that no matter what x is, x is smaller than y. This statement is false, for the domain $\mathcal{N}$ and every other domain of numbers, because if it were true, one of the values of x that would have to be smaller than y is y itself. The best way to explain the difference is to observe that in the first case the statement $\exists y(x < y)$ is within the scope of $\forall x$, so that the correct interpretation is "there exists y, which may depend on x".

Manipulating quantified statements often requires negating them. If it is not the case that for every x, $P(x)$, then there must be some value of x for which $P(x)$ is not true. Similarly, if there does not exist an x such that $P(x)$, then $P(x)$ must fail for every x. The general procedure for negating a quantifed statement is to reverse the quantifier (change $\forall$ to $\exists$, and vice versa) and move the negation inside the quantifier. $\neg(\forall x(P(x)))$ is the same as $\exists x(\neg P(x))$, and $\neg(\exists x(P(x)))$ is the same as $\forall x(\neg P(x))$. In order to negate a statement with several nested quantifiers, such as

$$\forall x(\exists y(\forall z(P(x, y, z))))$$

apply the general rule three times, moving from the outside in, so that the final result is

$$\exists x(\forall y(\exists z(\neg P(x, y, z))))$$

We have used "$\exists x(x - 1$ is prime)" as an example of a quantified statement. To conclude our discussion of quantifiers, we consider how to express the statement "x is prime" itself using quantifiers, where again the domain is the set $\mathcal{N}$. A prime is an integer greater than 1 whose only divisors are 1 and itself; the statement "x is prime" can be formulated as "$x > 1$, and for every k, if k is a divisor of x, then either k is 1 or k is x". Finally, the statement "k is a divisor of x" means that there is an integer m with $x = m * k$. Therefore, the statement we are looking for can be written

$$(x > 1) \wedge \forall k((\exists m(x = m * k)) \rightarrow (k = 1 \vee k = x))$$

A typical step in a *proof* is to derive a statement from initial assumptions and hypotheses, or from statements that have been derived previously, or from other generally accepted facts, using principles of logical reasoning. The more formal the proof, the stricter the criteria regarding what facts are "generally accepted", what principles of reasoning are allowed, and how carefully they are elaborated.

You will not learn how to write proofs just by reading this section, because it takes a lot of practice and experience, but we will illustrate a few basic proof techniques in the simple proofs that follow.

We will usually be trying to prove a statement, perhaps with a quantifier, involving a conditional proposition $p \to q$. The first example is a *direct* proof, in which we assume that p is true and derive q. We begin with the definitions of odd integers, which appear in this example, and even integers, which will appear in Example 1.3.

An integer n is odd if there exists an integer k so that $n = 2k + 1$.

An integer n is even if there exists an integer k so that $n = 2k$.

In Example 1.3, we will need the fact that every integer is either even or odd and no integer can be both (see Exercise 1.51).

EXAMPLE 1.1 The Product of Two Odd Integers Is Odd

To Prove: For every two integers a and b, if a and b are odd, then ab is odd.

■ Proof

The conditional statement can be restated as follows: If there exist integers i and j so that $a = 2i + 1$ and $b = 2j + 1$, then there exists an integer k so that $ab = 2k + 1$. Our proof will be *constructive*—not only will we show that there exists such an integer k, but we will demonstrate how to construct it. Assuming that $a = 2i + 1$ and $b = 2j + 1$, we have

$$\begin{aligned} ab &= (2i + 1)(2j + 1) \\ &= 4ij + 2i + 2j + 1 \\ &= 2(2ij + i + j) + 1 \end{aligned}$$

Therefore, if we let $k = 2ij + i + j$, we have the result we want, $ab = 2k + 1$.

An important point about this proof, or any proof of a statement that begins "for every", is that a "proof by example" is not sufficient. An example can constitute a proof of a statement that begins "there exists", and an example can disprove a statement beginning "for every", by serving as a counterexample, but the proof above makes no assumptions about a and b except that each is an odd integer.

Next we present examples illustrating two types of *indirect proofs*, proof by contrapositive and proof by contradiction.

Proof by Contrapositive — EXAMPLE 1.2

To Prove: For every three positive integers i, j, and n, if $ij = n$, then $i \le \sqrt{n}$ or $j \le \sqrt{n}$.

■ **Proof**

The conditional statement $p \to q$ inside the quantifier is logically equivalent to its contrapositive, and so we start by assuming that there exist values of i, j, and n such that

$$\text{not } (i \le \sqrt{n} \text{ or } j \le \sqrt{n})$$

According to the De Morgan law, this implies

$$\text{not } (i \le \sqrt{n}) \text{ and not } (j \le \sqrt{n})$$

which in turn implies $i > \sqrt{n}$ and $j > \sqrt{n}$. Therefore,

$$ij > \sqrt{n}\sqrt{n} = n$$

which implies that $ij \ne n$. We have constructed a direct proof of the contrapositive statement, which means that we have effectively proved the original statement.

For every proposition p, p is equivalent to the conditional proposition *true* $\to p$, whose contrapositive is $\neg p \to$ *false*. A proof of p by contradiction means assuming that p is false and deriving a contradiction (i.e., deriving the statement *false*). The example we use to illustrate proof by contradiction is more than two thousand years old and was known to members of the Pythagorean school in Greece. It involves positive rational numbers: numbers of the form m/n, where m and n are positive integers.

Proof by Contradiction: The Square Root of 2 Is Irrational — EXAMPLE 1.3

To Prove: There are no positive integers m and n satisfying $m/n = \sqrt{2}$.

■ **Proof**

Suppose for the sake of contradiction that there are positive integers m and n with $m/n = \sqrt{2}$. Then by dividing both m and n by all the factors common to both, we obtain $p/q = \sqrt{2}$, for some positive integers p and q with no common factors. If $p/q = \sqrt{2}$, then $p = q\sqrt{2}$, and therefore $p^2 = 2q^2$. According to Example 1.1, since p^2 is even, p must be even; therefore, $p = 2r$ for some positive integer r, and $p^2 = 4r^2$. This implies that $2r^2 = q^2$, and the same argument we have just used for p also implies that q is even. Therefore, 2 is a common factor of p and q, and we have a contradiction of our previous statement that p and q have no common factors.

It is often necessary to use more than one proof technique within a single proof. Although the proof in the next example is not a proof by contradiction, that technique is used twice within it. The statement to be proved involves the factorial of a positive integer n, which is denoted by $n!$ and is the product of all the positive integers less than or equal to n.

EXAMPLE 1.4 There Must Be a Prime Between n and $n!$

To Prove: For every integer $n > 2$, there is a prime p satisfying $n < p < n!$.

■ **Proof**

Because $n > 2$, the distinct integers n and 2 are two of the factors of $n!$. Therefore,

$$n! - 1 \geq 2n - 1 = n + n - 1 > n + 1 - 1 = n$$

The number $n! - 1$ has a prime factor p, which must satisfy $p \leq n! - 1 < n!$. Therefore, $p < n!$, which is one of the inequalities we need. To show the other one, suppose for the sake of contradiction that $p \leq n$. Then by the definition of factorial, p must be one of the factors of $n!$. However, p cannot be a factor of both $n!$ and $n! - 1$; if it were, it would be a factor of 1, their difference, and this is impossible because a prime must be bigger than 1. Therefore, the assumption that $p \leq n$ leads to a contradiction, and we may conclude that $n < p < n!$.

EXAMPLE 1.5 Proof by Cases

The last proof technique we will mention in this section is proof by cases. If P is a proposition we want to prove, and P_1 and P_2 are propositions, at least one of which must be true, then we can prove P by proving that P_1 implies P and P_2 implies P. This is sufficient because of the logical identities

$$\begin{aligned}(P_1 \to P) \wedge (P_2 \to P) &\Leftrightarrow (P_1 \vee P_2) \to P \\ &\Leftrightarrow \mathit{true} \to P \\ &\Leftrightarrow P\end{aligned}$$

which can be verified easily (saying that P_1 or P_2 must be true is the same as saying that $P_1 \vee P_2$ is equivalent to *true*).

The principle is the same if there are more than two cases. If we want to show the first distributive law

$$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$$

for example, then we must show that the truth values of the propositions on the left and right are the same, and there are eight cases, corresponding to the eight combinations of truth values for p, q, and r. An appropriate choice for P_1 is "p, q, and r are all true".

1.2 | SETS

A finite set can be described, at least in principle, by listing its elements. The formula

$$A = \{1, 2, 4, 8\}$$

says that A is the set whose elements are 1, 2, 4, and 8.

For infinite sets, and even for finite sets if they have more than just a few elements, ellipses (...) are sometimes used to describe how the elements might be listed:

$$B = \{0, 3, 6, 9, \dots\}$$
$$C = \{13, 14, 15, \dots, 71\}$$

A more reliable and often more informative way to describe sets like these is to give the property that characterizes their elements. The sets B and C could be described this way:

$$B = \{x \mid x \text{ is a nonnegative integer multiple of } 3\}$$
$$C = \{x \mid x \text{ is an integer and } 13 \le x \le 71\}$$

We would read the first formula "B is the set of all x such that x is a nonnegative integer multiple of 3". The expression before the vertical bar represents an arbitrary element of the set, and the statement after the vertical bar contains the conditions, or restrictions, that the expression must satisfy in order for it to represent a legal element of the set.

In these two examples, the "expression" is simply a variable, which we have arbitrarily named x. We often choose to include a little more information in the expression; for example,

$$B = \{3y \mid y \text{ is a nonnegative integer}\}$$

which we might read "B is the set of elements of the form $3y$, where y is a nonnegative integer". Two more examples of this approach are

$$D = \{\{x\} \mid x \text{ is an integer such that } x \ge 4\}$$
$$E = \{3i + 5j \mid i \text{ and } j \text{ are nonnegative integers}\}$$

Here D is a set of sets; three of its elements are $\{4\}$, $\{5\}$, and $\{6\}$. We could describe E using the formula

$$E = \{0, 3, 5, 6, 8, 9, 10, \dots\}$$

but the first description of E is more informative, even if the other seems at first to be more straightforward.

For any set A, the statement that x is an element of A is written $x \in A$, and $x \notin A$ means x is not an element of A. We write $A \subseteq B$ to mean A is a *subset* of B, or that every element of A is an element of B; $A \not\subseteq B$ means that A is not a subset of B (there is at least one element of A that is not an element of B). Finally, the *empty set*, the set with no elements, is denoted by $\emptyset$.

A set is determined by its elements. For example, the sets $\{0, 1\}$ and $\{1, 0\}$ are the same, because both contain the elements 0 and 1 and no others; the set $\{0, 0, 1, 1, 1, 2\}$ is the same as $\{0, 1, 2\}$, because they both contain 0, 1, and 2 and no other elements (no matter how many times each element is written, it's the same element); and there is only one empty set, because once you've said that a set contains no elements, you've described it completely. To show that two sets A and B are the same, we must show that A and B have exactly the same elements—i.e., that $A \subseteq B$ and $B \subseteq A$.

A few sets will come up frequently. We have used $\mathcal{N}$ in Section 1.1 to denote the set of *natural numbers*, or nonnegative integers; $\mathcal{Z}$ is the set of all integers, $\mathcal{R}$ the set of all real numbers, and $\mathcal{R}^+$ the set of nonnegative real numbers. The sets B and E above can be written more concisely as

$$B = \{3y \mid y \in \mathcal{N}\} \qquad E = \{3i + 5j \mid i, j \in \mathcal{N}\}$$

We sometimes relax the { expression | conditions } format slightly when we are describing a subset of another set, as in

$$C = \{x \in \mathcal{N} \mid 13 \le x \le 71\}$$

which we would read "C is the set of all x in $\mathcal{N}$ such that ..."

For two sets A and B, we can define their *union* $A \cup B$, their *intersection* $A \cap B$, and their *difference* $A - B$, as follows:

$$\begin{aligned} A \cup B &= \{x \mid x \in A \text{ or } x \in B\} \\ A \cap B &= \{x \mid x \in A \text{ and } x \in B\} \\ A - B &= \{x \mid x \in A \text{ and } x \notin B\} \end{aligned}$$

For example,

$$\begin{aligned} \{1, 2, 3, 5\} \cup \{2, 4, 6\} &= \{1, 2, 3, 4, 5, 6\} \\ \{1, 2, 3, 5\} \cap \{2, 4, 6\} &= \{2\} \\ \{1, 2, 3, 5\} - \{2, 4, 6\} &= \{1, 3, 5\} \end{aligned}$$

If we assume that A and B are both subsets of some "universal" set U, then we can consider the special case $U - A$, which is written A' and referred to as the *complement* of A.

$$A' = U - A = \{x \in U \mid x \notin A\}$$

We think of A' as "the set of everything that's not in A", but to be meaningful this requires context. The complement of $\{1, 2\}$ varies considerably, depending on whether the universal set is chosen to be $\mathcal{N}$, $\mathcal{Z}$, $\mathcal{R}$, or some other set.

If the intersection of two sets is the empty set, which means that the two sets have no elements in common, they are called *disjoint* sets. The sets in a collection of sets are *pairwise disjoint* if, for every two distinct ones A and B ("distinct" means not identical), A and B are disjoint. A *partition* of a set S is a collection of pairwise disjoint subsets of S whose union is S; we can think of a partition of S as a way of dividing S into non-overlapping subsets.

There are a number of useful "set identities", but they are closely analogous to the logical identities we discussed in Section 1.1, and as the following example demonstrates, they can be derived the same way.

The First De Morgan Law **EXAMPLE 1.6**

There are two De Morgan laws for sets, just as there are for propositions; the first asserts that for every two sets A and B,

$$(A \cup B)' = A' \cap B'$$

We begin by noticing the resemblance between this formula and the logical identity

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$$

The resemblance is not just superficial. We defined the logical connectives such as $\wedge$ and $\vee$ by drawing truth tables, and we could define the set operations $\cap$ and $\cup$ by drawing *membership* tables, where T denotes membership and F nonmembership:

A	B	$A \cap B$	$A \cup B$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

As you can see, the truth values in the two tables are identical to the truth values in the tables for $\wedge$ and $\vee$. We can therefore test a proposed set identity the same way we can test a proposed logical identity, by constructing tables for the two expressions being compared. When we do this for the expressions $(A \cup B)'$ and $A' \cap B'$, or for the propositions $\neg(p \vee q)$ and $\neg p \wedge \neg q$, by considering the four cases, we obtain identical values in each case. We may conclude that no matter what case x represents, $x \in (A \cup B)'$ if and only if $x \in A' \cap B'$, and the two sets are equal.

The associative law for unions, corresponding to the one for $\vee$, says that for arbitrary sets A, B, and C,

$$A \cup (B \cup C) = (A \cup B) \cup C$$

so that we can write $A \cup B \cup C$ without worrying about how to group the terms. It is easy to see from the definition of union that

$$A \cup B \cup C = \{x \mid x \text{ is an element of at least one of the sets } A, B, \text{ and } C\}$$

For the same reasons, we can consider unions of any number of sets and adopt notation to describe such unions. For example, if $A_0, A_1, A_2, \ldots$ are sets,

$$\bigcup\{A_i \mid 0 \le i \le n\} = \{x \mid x \in A_i \text{ for at least one } i \text{ with } 0 \le i \le n\}$$

$$\bigcup\{A_i \mid i \ge 0\} = \{x \mid x \in A_i \text{ for at least one } i \text{ with } i \ge 0\}$$

In Chapter 3 we will encounter the set

$$\bigcup \{\delta(p, \sigma) \mid p \in \delta^*(q, x)\}$$

In all three of these formulas, we have a set S of sets, and we are describing the union of all the sets in S. We do not need to know what the sets $\delta^*(q, x)$ and $\delta(p, \sigma)$ are to understand that

$$\bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\} = \{x \mid x \in \delta(p, \sigma) \text{ for at least one element } p \text{ of } \delta^*(q, x)\}$$

If $\delta^*(q, x)$ were $\{r, s, t\}$, for example, we would have

$$\bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\} = \delta(r, \sigma) \cup \delta(s, \sigma) \cup \delta(t, \sigma)$$

Sometimes the notation varies slightly. The two sets

$$\bigcup\{A_i \mid i \geq 0\} \quad \text{and} \quad \bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\}$$

for example, might be written

$$\bigcup_{i=0}^{\infty} A_i \quad \text{and} \quad \bigcup_{p \in \delta^*(q,x)} \delta(p, \sigma)$$

respectively.

Because there is also an associative law for intersections, exactly the same notation can be used with $\cap$ instead of $\cup$.

For a set A, the set of all subsets of A is called the *power set* of A and written 2^A. The reason for the terminology and the notation is that if A is a finite set with n elements, then 2^A has exactly 2^n elements (see Example 1.23). For example,

$$2^{\{a,b,c\}} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

This example illustrates the fact that the empty set is a subset of every set, and every set is a subset of itself.

One more set that can be constructed from two sets A and B is $A \times B$, their *Cartesian product*:

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

For example,

$$\{0, 1\} \times \{1, 2, 3\} = \{(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3)\}$$

The elements of $A \times B$ are called *ordered* pairs, because $(a, b) = (c, d)$ if and only if $a = c$ and $b = d$; in particular, (a, b) and (b, a) are different unless a and b happen to be equal. More generally, $A_1 \times A_2 \times \cdots \times A_k$ is the set of all "ordered k-tuples" $(a_1, a_2, \ldots, a_k)$, where a_i is an element of A_i for each i.

1.3 | FUNCTIONS AND EQUIVALENCE RELATIONS

If A and B are two sets (possibly equal), a *function* f from A to B is a rule that assigns to each element x of A an element $f(x)$ of B. (Later in this section we will mention a more precise definition, but for our purposes the informal "rule" definition will be sufficient.) We write $f : A \to B$ to mean that f is a function from A to B.

Here are four examples:

1. The function $f : \mathcal{N} \to \mathcal{R}$ defined by the formula $f(x) = \sqrt{x}$. (In other words, for every $x \in \mathcal{N}$, $f(x) = \sqrt{x}$.)
2. The function $g : 2^{\mathcal{N}} \to 2^{\mathcal{N}}$ defined by the formula $g(A) = A \cup \{0\}$.
3. The function $u : 2^{\mathcal{N}} \times 2^{\mathcal{N}} \to 2^{\mathcal{N}}$ defined by the formula $u(S, T) = S \cup T$.
4. The function $i : \mathcal{N} \to \mathcal{Z}$ defined by

$$i(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ (-n-1)/2 & \text{if } n \text{ is odd} \end{cases}$$

For a function f from A to B, we call A the *domain* of f and B the *codomain* of f. The domain of a function f is the set of values x for which $f(x)$ is defined. We will say that two functions f and g are the same if and only if they have the same domain, they have the same codomain, and $f(x) = g(x)$ for every x in the domain.

In some later chapters it will be convenient to refer to a *partial* function f from A to B, one whose domain is a subset of A, so that f may be undefined at some elements of A. We will still write $f : A \to B$, but we will be careful to distinguish the set A from the domain of f, which may be a smaller set. When we speak of a *function* from A to B, without any qualification, we mean one with domain A, and we might emphasize this by calling it a *total* function.

If f is a function from A to B, a third set involved in the description of f is its *range*, which is the set

$$\{f(x) \mid x \in A\}$$

(a subset of the codomain B). The range of f is the set of elements of the codomain that are actually assigned by f to elements of the domain.

Definition 1.7 One-to-One and Onto Functions

A function $f : A \to B$ is *one-to-one* if f never assigns the same value to two different elements of its domain. It is *onto* if its range is the entire set B. A function from A to B that is both one-to-one and onto is called a *bijection* from A to B.

Another way to say that a function $f : A \to B$ is one-to-one is to say that for every $y \in B$, $y = f(x)$ for *at most* one $x \in A$, and another way to say that f is onto is to say that for every $y \in B$, $y = f(x)$ for *at least* one $x \in A$. Therefore, saying that f is a bijection from A to B means that every element y of the codomain B is $f(x)$ for *exactly* one $x \in A$. This allows us to define another function f^{-1} from B to A, by saying that for every $y \in B$, $f^{-1}(y)$ is the element $x \in A$ for which

$f(x) = y$. It is easy to check that this "inverse function" is also a bijection and satisfies these two properties: For every $x \in A$, and every $y \in B$,

$$f^{-1}(f(x)) = x \qquad f(f^{-1}(y)) = y$$

Of the four functions defined above, the function f from $\mathcal{N}$ to $\mathcal{R}$ is one-to-one but not onto, because a real number is the square root of at most one natural number and might not be the square root of any. The function g is not one-to-one, because for every subset A of $\mathcal{N}$ that doesn't contain 0, A and $A \cup \{0\}$ are distinct and $g(A) = g(A \cup \{0\})$. It is also not onto, because every element of the range of g is a set containing 0 and not every subset of $\mathcal{N}$ does. The function u is onto, because $u(A, A) = A$ for every $A \in 2^{\mathcal{N}}$, but not one-to-one, because for every $A \in 2^{\mathcal{N}}$, $u(A, \emptyset)$ is also A.

The formula for i seems more complicated, but looking at this partial tabulation of its values

x	0	1	2	3	4	5	6	...
$i(x)$	0	−1	1	−2	2	−3	3	...

makes it easy to see that i is both one-to-one and onto. No integer appears more than once in the list of values of i, and every integer appears once.

In the first part of this book, we will usually not be concerned with whether the functions we discuss are one-to-one or onto. The idea of a bijection between two sets, such as our function i, will be important in Chapter 8, when we discuss infinite sets with different sizes.

An *operation on a set A* is a function that assigns to elements of A, or perhaps to combinations of elements of A, other elements of A. We will be interested particularly in *binary* operations (functions from $A \times A$ to A) and *unary* operations (functions from A to A). The function u described above is an example of a binary operation on the set $2^{\mathcal{N}}$, and for every set S, both union and intersection are binary operations on 2^S. Familar binary operations on $\mathcal{N}$, or on $\mathcal{Z}$, include addition and multiplication, and subtraction is a binary operation on $\mathcal{Z}$. The complement operation is a unary operation on 2^S, for every set S, and negation is a unary operation on the set $\mathcal{Z}$. The notation adopted for some of these operations is different from the usual functional notation; we write $U \cup V$ rather than $\cup(U, V)$, and $a - b$ rather than $-(a, b)$.

For a unary operation or a binary operation on a set A, we say that a subset A_1 of A is *closed under the operation* if the result of applying the operation to elements of A_1 is an element of A_1. For example, if $A = 2^{\mathcal{N}}$, and A_1 is the set of all *nonempty* subsets of $\mathcal{N}$, then A_1 is closed under union (the union of two nonempty subsets of $\mathcal{N}$ is a nonempty subset of $\mathcal{N}$) but not under intersection. The set of all subsets of $\mathcal{N}$ with fewer than 100 elements is closed under intersection but not under union. If $A = \mathcal{N}$, and A_1 is the set of even natural numbers, then A_1 is closed under both addition and multiplication; the set of odd natural numbers is closed under multiplication but not under addition. We will return to this idea later in this chapter, when we discuss recursive definitions of sets.

We can think of a function f from a set A to a set B as establishing a relationship between elements of A and elements of B; every element $x \in A$ is "related" to exactly one element $y \in B$, namely, $y = f(x)$. A *relation* R from A to B may be more general, in that an element $x \in A$ may be related to no elements of B, to one element, or to more than one. We will use the notation aRb to mean that a is related to b with respect to the relation R. For example, if A is the set of people and B is the set of cities, we might consider the "has-lived-in" relation R from A to B: If $x \in A$ and $y \in B$, xRy means that x has lived in y. Some people have never lived in a city, some have lived in one city all their lives, and some have lived in several cities.

We've said that a function is a "rule"; exactly what is a relation?

Definition 1.8 A Relation from A to B, and a Relation on A

For two sets A and B, a relation from A to B is a subset of $A \times B$. A relation on the set A is a relation from A to A, or a subset of $A \times A$.

The statement "a is related to b with respect to R" can be expressed by either of the formulas aRb and $(a, b) \in R$. As we have already pointed out, a function f from A to B is simply a relation having the property that for every $x \in A$, there is exactly one $y \in B$ with $(x, y) \in f$. Of course, in this special case, a third way to write "x is related to y with respect to f" is the most common: $y = f(x)$.

In the has-lived-in example above, the statement "Sally has lived in Atlanta" seems easier to understand than the statement "(Sally, Atlanta)$\in R$", but this is just a question of notation. If we understand what R is, the two statements say the same thing. In this book, we will be interested primarily in relations on a set, especially ones that satisfy the three properties in the next definition.

Definition 1.9 Equivalence Relations

A relation R on a set A is an *equivalence relation* if it satisfies these three properties.

1. R is *reflexive*: for every $x \in A$, xRx.
2. R is *symmetric*: for every x and every y in A, if xRy, then yRx.
3. R is *transitive*: for every x, every y, and every z in A, if xRy and yRz, then xRz.

If R is an equivalence relation on A, we often say "x is equivalent to y" instead of "x is related to y". Examples of relations that do not satisfy all three properties can be found in the exercises. Here we present three simple examples of equivalence relations.

EXAMPLE 1.10 The Equality Relation

We can consider the relation of equality on every set A, and the formula $x = y$ expresses the fact that (x, y) is an element of the relation. The properties of reflexivity, symmetry, and transitivity are familiar properties of equality: Every element of A is equal to itself; for every x and y in A, if $x = y$, then $y = x$; and for every x, y, and z, if $x = y$ and $y = z$, then $x = z$. This relation is the prototypical equivalence relation, and the three properties are no more than what we would expect of any relation we described as one of *equivalence*.

EXAMPLE 1.11 The Relation on A Containing All Ordered Pairs

On every set A, we can also consider the relation $R = A \times A$. Every possible ordered pair of elements of A is in the relation—every element of A is related to every other element, including itself. This relation is also clearly an equivalence relation; no statement of the form "(under certain conditions) xRy" can possibly fail if xRy for every x and every y.

EXAMPLE 1.12 The Relation of Congruence Mod n on $\mathcal{N}$

We consider the set $\mathcal{N}$ of natural numbers, and, for some positive integer n, the relation R on $\mathcal{N}$ defined as follows: for every x and y in $\mathcal{N}$,

$$xRy \text{ if there is an integer } k \text{ so that } x - y = kn$$

In this case we write $x \equiv_n y$ to mean xRy. Checking that the three properties are satisfied requires a little more work this time, but not much. The relation is reflexive, because for every $x \in \mathcal{N}$, $x - x = 0 * n$. It is symmetric, because for every x and every y in $\mathcal{N}$, if $x - y = kn$, then $y - x = (-k)n$. Finally, it is transitive, because if $x - y = kn$ and $y - z = jn$, then

$$x - z = (x - y) + (y - z) = kn + jn = (k + j)n$$

One way to understand an equivalence relation R on a set A is to consider, for each $x \in A$, the subset $[x]_R$ of A containing all the elements equivalent to x. Because an equivalence relation is reflexive, one of these elements is x itself, and we can refer to the set $[x]_R$ as *the equivalence class containing* x.

Definition 1.13 The Equivalence Class Containing x

For an equivalence relation R on a set A, and an element $x \in A$, the equivalence class containing x is

$$[x]_R = \{y \in A \mid yRx\}$$

If there is no doubt about which equivalence relation we are using, we will drop the subscript and write $[x]$.

The phrase "the equivalence class containing x" is not misleading: For every $x \in A$, we have already seen that $x \in [x]$, and we can also check that x belongs to only one equivalence class. Suppose that $x, y \in A$ and $x \in [y]$, so that xRy; we show that $[x] = [y]$. Let z be an arbitrary element of $[x]$, so that zRx. Because zRx, xRy, and R is transitive, it follows that zRy; therefore, $[x] \subseteq [y]$. For the other inclusion we observe that if $x \in [y]$, then $y \in [x]$ because R is symmetric, and the same argument with x and y switched shows that $[y] \subseteq [x]$.

These conclusions are summarized by Theorem 1.14.

Theorem 1.14
If R is an equivalence relation on a set A, the equivalence classes with respect to R form a partition of A, and two elements of A are equivalent if and only if they are elements of the same equivalence class.

Example 1.10 illustrates the extreme case in which every equivalence class contains just one element, and Example 1.11 illustrates the other extreme, in which the single equivalence class A contains all the elements. In the case of congruence mod n for a number $n > 1$, some but not all of the elements of $\mathcal{N}$ other than x are in $[x]$; the set $[x]$ contains all natural numbers that differ from x by a multiple of n.

For an arbitrary equivalence relation R on a set A, knowing the partition determined by R is enough to describe the relation completely. In fact, if we begin with a partition of A, then the relation R on A that is defined by the last statement of Theorem 1.1 (two elements x and y are related if and only if x and y are in the same subset of the partition) is an equivalence relation whose equivalence classes are precisely the subsets of the partition. Specifying a subset of $A \times A$ and specifying a partition on A are two ways of conveying the same information.

Finally, if R is an equivalence relation on A and $S = [x]$, it follows from Theorem 1.14 that every two elements of S are equivalent and no element of S is equivalent to an element not in S. On the other hand, if S is a nonempty subset of A, knowing that S satisfies these two properties allows us to say that S is an equivalence class, even if we don't start out with any particular x satisfying $S = [x]$. If x is an arbitrary element of S, every element of S belongs to $[x]$, because it is equivalent to x; and every element of $[x]$ belongs to S, because otherwise the element x of S would be equivalent to some element not in S. Therefore, for every $x \in S$, $S = [x]$.

1.4 | LANGUAGES

Familar languages include programming languages such as Java and natural languages like English, as well as unofficial "dialects" with specialized vocabularies, such as the language used in legal documents or the language of mathematics. In this book we use the word "language" more generally, taking a language to be any set of strings over an alphabet of symbols. In applying this definition to English, we might take the individual strings to be English words, but it is more common to consider English sentences, for which many grammar rules have been developed. In the case of a language like Java, a string must satisfy certain rules in order to be a legal statement, and a sequence of statements must satisfy certain rules in order to be a legal program.

Many of the languages we study initially will be much simpler. They might involve alphabets with just one or two symbols, and perhaps just one or two basic patterns to which all the strings must conform. The main purpose of this section is to present some notation and terminology involving strings and languages that will be used throughout the book.

An *alphabet* is a finite set of symbols, such as $\{a, b\}$ or $\{0, 1\}$ or $\{A, B, C, \dots, Z\}$. We will usually use the Greek letter Σ to denote the alphabet. A *string* over Σ is a finite sequence of symbols in Σ. For a string x, $|x|$ stands for the *length* (the number of symbols) of x. In addition, for a string x over Σ and an element $\sigma \in \Sigma$,

$$n_\sigma(x) = \text{ the number of occurrences of the symbol } \sigma \text{ in the string } x$$

The *null string* Λ is a string over Σ, no matter what the alphabet Σ is. By definition, $|\Lambda| = 0$.

The set of all strings over Σ will be written Σ^*. For the alphabet $\{a, b\}$, we have

$$\{a, b\}^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

Here we have listed the strings in *canonical* order, the order in which shorter strings precede longer strings and strings of the same length appear alphabetically. Canonical order is different from *lexicographic*, or strictly alphabetical order, in which aa precedes b. An essential difference is that canonical order can be described by making a single list of strings that includes every element of Σ^* exactly once. If we wanted to describe an algorithm that did something with each string in $\{a, b\}^*$, it would make sense to say, "Consider the strings in canonical order, and for each one, ..." (see, for example, Section 8.2). If an algorithm were to "consider the strings of $\{a, b\}^*$ in lexicographic order", it would have to start by considering Λ, a, aa, aaa, ..., and it would never get around to considering the string b.

A language over Σ is a subset of Σ^*. Here are a few examples of languages over $\{a, b\}$:

1. The empty language $\emptyset$.
2. $\{\Lambda, a, aab\}$, another finite language.
3. The language *Pal* of palindromes over $\{a, b\}$ (strings such as aba or $baab$ that are unchanged when the order of the symbols is reversed).
4. $\{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$.
5. $\{x \in \{a, b\}^* \mid |x| \geq 2 \text{ and } x \text{ begins and ends with } b\}$.

The null string Λ is always an element of Σ^*, but other languages over Σ may or may not contain it; of these five examples, only the second and third do.

Here are a few real-world languages, in some cases involving larger alphabets.

6. The language of legal Java identifiers.
7. The language *Expr* of legal algebraic expressions involving the identifier a, the binary operations $+$ and $*$, and parentheses. Some of the strings in the language are a, $a + a * a$, and $(a + a * (a + a))$.
8. The language *Balanced* of balanced strings of parentheses (strings containing the occurrences of parentheses in some legal algebraic expression). Some elements are Λ, ()(()), and ((((())))).
9. The language of numeric "literals" in Java, such as −41, 0.03, and 5.0E−3.
10. The language of legal Java programs. Here the alphabet would include upper- and lowercase alphabetic symbols, numerical digits, blank spaces, and punctuation and other special symbols.

The basic operation on strings is *concatenation*. If x and y are two strings over an alphabet, the concatenation of x and y is written xy and consists of the symbols of x followed by those of y. If $x = ab$ and $y = bab$, for example, then $xy = abbab$ and $yx = babab$. When we concatenate the null string Λ with another string, the result is just the other string (for every string x, $x\Lambda = \Lambda x = x$); and for every x, if one of the formulas $xy = x$ or $yx = x$ is true for some string y, then $y = \Lambda$. In general, for two strings x and y, $|xy| = |x| + |y|$.

Concatenation is an associative operation; that is, $(xy)z = x(yz)$, for all possible strings x, y, and z. This allows us to write xyz without specifying how the factors are grouped.

If s is a string and $s = tuv$ for three strings t, u, and v, then t is a *prefix* of s, v is a *suffix* of s, and u is a *substring* of s. Because one or both of t and u might be Λ, prefixes and suffixes are special cases of substrings. The string Λ is a prefix of every string, a suffix of every string, and a substring of every string, and every string is a prefix, a suffix, and a substring of itself.

Languages are sets, and so one way of constructing new languages from existing ones is to use set operations. For two languages L_1 and L_2 over the alphabet Σ, $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 - L_2$ are also languages over Σ. If $L \subseteq \Sigma^*$, then by the complement of L we will mean $\Sigma^* - L$. This is potentially confusing, because if L is a language over Σ, then L can be interpreted as a language over any larger alphabet, but it will usually be clear what alphabet we are referring to.

We can also use the string operation of concatenation to construct new languages. If L_1 and L_2 are both languages over Σ, the concatenation of L_1 and L_2 is the language

$$L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

For example, $\{a, aa\}\{\Lambda, b, ab\} = \{a, ab, aab, aa, aaab\}$. Because $\Lambda x = x\Lambda$ for every string x, we have

$$\{\Lambda\}L = L\{\Lambda\} = L$$

for every language L.

The language $L = \Sigma^*$, for example, satisfies the formula $LL = L$, and so the formula $LL_1 = L$ does not always imply that $L_1 = \{\Lambda\}$. However, if L_1 is a language such that $LL_1 = L$ for *every* language L, or if $L_1L = L$ for every language L, then $L_1 = \{\Lambda\}$.

At this point we can adopt "exponential" notation for the concatenation of k copies of a single symbol a, a single string x, or a single language L. If $k > 0$, then $a^k = aa \dots a$, where there are k occurrences of a, and similarly for x^k and L^k. In the special case where L is simply the alphabet Σ (which can be interpreted as a set of strings of length 1), $\Sigma^k = \{x \in \Sigma^* \mid |x| = k\}$.

We also want the exponential notation to make sense if $k = 0$, and the correct definition requires a little care. It is desirable to have the formulas

$$a^i a^j = a^{i+j} \qquad x^i x^j = x^{i+j} \qquad L^i L^j = L^{i+j}$$

where a, x, and L are an alphabet symbol, a string, and a language, respectively. In the case $i = 0$, the first two formulas require that we define a^0 and x^0 to be Λ, and the last formula requires that L^0 be $\{\Lambda\}$.

Finally, for a language L over an alphabet Σ, we use the notation L^* to denote the language of all strings that can be obtained by concatenating zero or more strings in L. This operation on a language L is known as the *Kleene star*, or Kleene closure, after the mathematician Stephen Kleene. The notation L^* is consistent with the earlier notation Σ^*, which we can describe as the set of strings obtainable by concatenating zero or more strings of length 1 over Σ. L^* can be defined by the formula

$$L^* = \bigcup \{L^k \mid k \in \mathcal{N}\}$$

Because we have defined L^0 to be $\{\Lambda\}$, "concatenating zero strings in L" produces the null string, and $\Lambda \in L^*$, no matter what the language L is.

When we describe languages using formulas that contain the union, concatenation, and Kleene L^* operations, we will use precedence rules similar to the algebraic rules you are accustomed to. The formula $L_1 \cup L_2L_3^*$, for example, means $L_1 \cup (L_2(L_3^*))$; of the three operations, the highest-precedence operation is *, next-highest is concatenation, and lowest is union. The expressions $(L_1 \cup L_2)L_3^*$, $L_1 \cup (L_2L_3)^*$, and $(L_1 \cup L_2L_3)^*$ all refer to different languages.

Strings, by definition, are finite (have only a finite number of symbols). Almost all interesting languages are infinite sets of strings, and in order to use the languages we must be able to provide precise finite descriptions. There are at least two general approaches to doing this, although there is not always a clear line separating them. If we write

$$L_1 = \{ab, bab\}^* \cup \{b\}\{ba\}^*\{ab\}^*$$

we have described the language L_1 by providing a formula showing the possible ways of generating an element: either concatenating an arbitrary number of strings, each of which is either ab or bab, or concatenating a single b with an arbitrary number of copies of ba and then an arbitrary number of copies of ab. The fourth example in our list above is the language

$$L_2 = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$$

which we have described by giving a property that characterizes the elements. For every string $x \in \{a, b\}^*$, we can test whether x is in L_2 by testing whether the condition is satisfied.

In this book we will study notational schemes that make it easy to describe how languages can be *generated*, and we will study various types of algorithms, of increasing complexity, for *recognizing*, or *accepting*, strings in certain languages. In the second approach, we will often identify an algorithm with an abstract machine that can carry it out; a precise description of the algorithm or the machine will effectively give us a precise way of specifying the language.

1.5 | RECURSIVE DEFINITIONS

As you know, recursion is a technique that is often useful in writing computer programs. In this section we will consider recursion as a tool for defining sets: primarily, sets of numbers, sets of strings, and sets of sets (of numbers or strings).

A recursive definition of a set begins with a *basis* statement that specifies one or more elements in the set. The *recursive* part of the definition involves one or more operations that can be applied to elements already known to be in the set, so as to produce new elements of the set.

As a way of defining a set, this approach has a number of potential advantages: Often it allows very concise definitions; because of the algorithmic nature of a typical recursive definition, one can often see more easily how, or why, a particular object is an element of the set being defined; and it provides a natural way of defining functions on the set, as well as a natural way of proving that some condition or property is satisfied by every element of the set.

The Set of Natural Numbers **EXAMPLE 1.15**

The prototypical example of recursive definition is the axiomatic definition of the set $\mathcal{N}$ of natural numbers. We assume that 0 is a natural number and that we have a "successor" operation, which, for each natural number n, gives us another one that is the successor of n and can be written $n + 1$. We might write the definition this way:

1. $0 \in \mathcal{N}$.
2. For every $n \in \mathcal{N}$, $n + 1 \in \mathcal{N}$.
3. Every element of $\mathcal{N}$ can be obtained by using statement 1 or statement 2.

In order to obtain an element of $\mathcal{N}$, we use statement 1 once and statement 2 a finite number of times (zero or more). To obtain the natural number 7, for example, we use statement 1 to obtain 0; then statement 2 with $n = 0$ to obtain 1; then statement 2 with $n = 1$ to obtain 2; . . . ; and finally, statement 2 with $n = 6$ to obtain 7.

We can summarize the first two statements by saying that $\mathcal{N}$ contains 0 and is *closed under the successor operation* (the operation of adding 1).

There are other sets of numbers that contain 0 and are closed under the successor operation: the set of all real numbers, for example, or the set of all fractions. The third statement in the definition is supposed to make it clear that the set we are defining is the one containing *only* the numbers obtained by using statement 1 once and statement 2 a finite number of times. In other words, $\mathcal{N}$ is the *smallest* set of numbers that contains 0 and is closed under the successor operation: $\mathcal{N}$ is a subset of every other such set.

In the remaining examples in this section we will omit the statement corresponding to statement 3 in this example, but whenever we define a set recursively, we will assume that a statement like this one is in effect, whether or not it is stated explicitly.

Just as a recursive procedure in a computer program must have an "escape hatch" to avoid calling itself forever, a recursive definition like the one above must have a basis statement that provides us with at least one element of the set. The recursive statement, that $n + 1 \in \mathcal{N}$ for every $n \in \mathcal{N}$, works in combination with the basis statement to give us all the remaining elements of the set.

EXAMPLE 1.16 Recursive Definitions of Other Subsets of $\mathcal{N}$

If we use the definition in Example 1.15, but with a different value specified in the basis statement:

1. $15 \in A$.
2. For every $n \in A$, $n + 1 \in A$.

then the set A that has been defined is the set of natural numbers greater than or equal to 15.

If we leave the basis statement the way it was in Example 1.15 but change the "successor" operation by changing $n + 1$ to $n + 7$, we get a definition of the set of all natural numbers that are multiples of 7.

Here is a definition of a subset B of $\mathcal{N}$:

1. $1 \in B$.
2. For every $n \in B$, $2 * n \in B$.
3. For every $n \in B$, $5 * n \in B$.

The set B is the smallest set of numbers that contains 1 and is closed under multiplication by 2 and 5. Starting with the number 1, we can obtain 2, 4, 8, ... by repeated applications of statement 2, and we can obtain 5, 25, 125, ... by using statement 3. By using both statements 2 and 3, we can obtain numbers such as $2 * 5$, $4 * 5$, and $2 * 25$. It is not hard to convince yourself that B is the set

$$B = \{2^i * 5^j \mid i, j \in \mathcal{N}\}$$

EXAMPLE 1.17 Recursive Definitions of $\{a,b\}^*$

Although we use $\Sigma = \{a, b\}$ in this example, it will be easy to see how to modify the definition so that it uses another alphabet. Our recursive definition of $\mathcal{N}$ started with the natural number 0, and the recursive statement allowed us to take an arbitrary n and obtain a natural number 1 bigger. An analogous recursive definition of $\{a, b\}^*$ begins with the string of length 0 and says how to take an arbitrary string x and obtain strings of length $|x| + 1$.

1. $\Lambda \in \{a, b\}^*$.
2. For every $x \in \{a, b\}^*$, both xa and xb are in $\{a, b\}^*$.

To obtain a string z of length k, we start with Λ and obtain longer and longer prefixes of z by using the second statement k times, each time concatenating the next symbol onto the right end of the current prefix. A recursive definition that used ax and bx in statement 2 instead of xa and xb would work just as well; in that case we would produce longer and longer suffixes of z by adding each symbol to the left end of the current suffix.

Recursive Definitions of Two Other Languages over $\{a,b\}$ **EXAMPLE 1.18**

We let *AnBn* be the language

$$AnBn = \{a^n b^n \mid n \in \mathcal{N}\}$$

and *Pal* the language introduced in Section 1.4 of all *palindromes* over $\{a, b\}$; a palindrome is a string that is unchanged when the order of the symbols is reversed.

The shortest string in *AnBn* is Λ, and if we have an element $a^i b^i$ of length $2i$, the way to get one of length $2i + 2$ is to add a at the beginning and b at the end. Therefore, a recursive definition of *AnBn* is:

1. $\Lambda \in AnBn$.
2. For every $x \in AnBn$, $axb \in AnBn$.

It is only slightly harder to find a recursive definition of *Pal*. The length of a palindrome can be even or odd. The shortest one of even length is Λ, and the two shortest ones of odd length are a and b. For every palindrome x, a longer one can be obtained by adding the same symbol at both the beginning and the end of x, and every palindrome of length at least 2 can be obtained from a shorter one this way. The recursive definition is therefore

1. Λ, a, and b are elements of *Pal*.
2. For every $x \in Pal$, axa and bxb are in *Pal*.

Both *AnBn* and *Pal* will come up again, in part because they illustrate in a very simple way some of the limitations of the first type of abstract computing device we will consider.

Algebraic Expressions and Balanced Strings of Parentheses **EXAMPLE 1.19**

As in Section 1.4, we let *Expr* stand for the language of legal algebraic expressions, where for simplicity we restrict ourselves to two binary operators, $+$ and $*$, a single identifier a, and left and right parentheses. Real-life expressions can be considerably more complicated because they can have additional operators, multisymbol identifiers, and numeric literals of various types; however, two operators are enough to illustrate the basic principles, and the other features can easily be added by substituting more general subexpressions for the identifier a.

Expressions can be illegal for "local" reasons, such as illegal symbol-pairs, or because of global problems involving mismatched parentheses. Explicitly prohibiting all the features we want to consider illegal is possible but is tedious. A recursive definition, on the other hand, makes things simple. The simplest algebraic expression consists of a single a, and any other one is obtained by combining two subexpressions using $+$ or $*$ or by parenthesizing a single subexpression.

1. $a \in \mathit{Expr}$.
2. For every x and every y in *Expr*, $x + y$ and $x * y$ are in *Expr*.
3. For every $x \in \mathit{Expr}$, $(x) \in \mathit{Expr}$.

The expression $(a + a * (a + a))$, for example, can be obtained as follows:

$a \in \mathit{Expr}$, by statement 1.

$a + a \in \mathit{Expr}$, by statement 2, where x and y are both a.

$(a + a) \in \mathit{Expr}$, by statement 3, where $x = a + a$.

$a * (a + a) \in \mathit{Expr}$, by statement 2, where $x = a$ and $y = (a + a)$.

$a + a * (a + a) \in \mathit{Expr}$, by statement 2, where $x = a$ and $y = a * (a + a)$.

$(a + a * (a + a)) \in \mathit{Expr}$, by statement 3, where $x = a + a * (a + a)$.

It might have occurred to you that there is a shorter derivation of this string. In the fourth line, because we have already obtained both $a + a$ and $(a + a)$, we could have said

$$a + a * (a + a) \in \mathit{Expr}, \text{ by statement 2, where } x = a + a \text{ and } y = (a + a).$$

The longer derivation takes into account the normal rules of precedence, under which $a + a * (a + a)$ is interpreted as the sum of a and $a * (a + a)$, rather than as the product of $a + a$ and $(a + a)$. The recursive definition addresses only the strings that are in the language, not what they mean or how they should be interpreted. We will discuss this issue in more detail in Chapter 4.

Now we try to find a recursive definition for *Balanced*, the language of balanced strings of parentheses. We can think of balanced strings as the strings of parentheses that can occur within strings in the language *Expr*. The string a has no parentheses; and the two ways of forming new balanced strings from existing balanced strings are to concatenate two of them (because two strings in *Expr* can be concatenated, with either $+$ or $*$ in between), or to parenthesize one of them (because a string in *Expr* can be parenthesized).

1. $\Lambda \in \mathit{Balanced}$.
2. For every x and every y in *Balanced*, $xy \in \mathit{Balanced}$.
3. For every $x \in \mathit{Balanced}$, $(x) \in \mathit{Balanced}$.

In order to use the "closed-under" terminology to paraphrase the recursive definitions of *Expr* and *Balanced*, it helps to introduce a little notation. If we define operations $\circ$, $\bullet$, and $\diamond$ by saying $x \circ y = x + y$, $x \bullet y = x * y$, and $\diamond(x) = (x)$, then we can say that *Expr* is the smallest language that contains the string a and is closed under the operations $\circ$, $\bullet$, and $\diamond$. (This is confusing. We normally think of $+$ and $*$ as "operations", but addition and multiplication are operations on sets of numbers, not sets of strings. In this discussion $+$ and $*$ are simply alphabet symbols, and it would be incorrect to say that *Expr* is closed under addition and multiplication.) Along the same line, if we describe the operation of enclosing a string within parentheses as "parenthesization", we can say that *Balanced* is the smallest language that contains Λ and is closed under concatenation and parenthesization.

A Recursive Definition of a Set of Languages over $\{a,b\}^*$ **EXAMPLE 1.20**

We denote by $\mathcal{F}$ the subset of $2^{\{a,b\}^*}$ (the set of languages over $\{a, b\}$) defined as follows:

1. $\emptyset$, $\{\Lambda\}$, $\{a\}$, and $\{b\}$ are elements of $\mathcal{F}$.
2. For every L_1 and every L_2 in $\mathcal{F}$, $L_1 \cup L_2 \in \mathcal{F}$.
3. For every L_1 and every L_2 in $\mathcal{F}$, $L_1 L_2 \in \mathcal{F}$.

$\mathcal{F}$ is the smallest set of languages that contains the languages $\emptyset$, $\{\Lambda\}$, $\{a\}$, and $\{b\}$ and is closed under the operations of union and concatenation.

Some elements of $\mathcal{F}$, in addition to the four from statement 1, are $\{a, b\}$, $\{ab\}$, $\{a, b, ab\}$, $\{aba, abb, abab\}$, and $\{aa, ab, aab, ba, bb, bab\}$. The first of these is the union of $\{a\}$ and $\{b\}$, the second is the concatenation of $\{a\}$ and $\{b\}$, the third is the union of the first and second, the fourth is the concatenation of the second and third, and the fifth is the concatenation of the first and third.

Can you think of any languages over $\{a, b\}$ that are not in $\mathcal{F}$? For every string $x \in \{a, b\}^*$, the language $\{x\}$ can be obtained by concatenating $|x|$ copies of $\{a\}$ or $\{b\}$, and every set $\{x_1, x_2, \dots, x_k\}$ of strings can be obtained by taking the union of the languages $\{x_i\}$. What could be missing?

This recursive definition is perhaps the first one in which we must remember that elements in the set we are defining are obtained by using the basis statement and one or more of the recursive statements a *finite* number of times. In the previous examples, it wouldn't have made sense to consider anything else, because natural numbers cannot be infinite, and in this book we never consider strings of infinite length. It makes sense to talk about infinite languages over $\{a, b\}$, but none of them is in $\mathcal{F}$. Statement 3 in the definition of $\mathcal{N}$ in Example 1.15 says every element of $\mathcal{N}$ *can be obtained* by using the first two statements—can be obtained, for example, by someone with a pencil and paper who is applying the first two statements in the definition in real time. For a language L to be in $\mathcal{F}$, there must be a sequence of steps, each of which involves statements in the definition, that this person could actually carry out to produce L: There must be languages L_0, L_1, L_2, $\dots$, L_n so that L_0 is obtained from the basis statement of the definition; for each $i > 0$, L_i is either also obtained from the basis statement or obtained from two earlier L_j's using union or concatenation; and $L_n = L$. The conclusion in this example is that the set $\mathcal{F}$ is the set of all *finite* languages over $\{a, b\}$.

One final observation about certain recursive definitions will be useful in Chapter 4 and a few other places. Sometimes, although not in any of the examples so far in this section, a *finite* set can be described most easily by a recursive definition. In this case, we can take advantage of the algorithmic nature of these definitions to formulate an algorithm for obtaining the set.

EXAMPLE 1.21 The Set of Cities Reachable from City *s*

Suppose that C is a finite set of cities, and the relation R is defined on C by saying that for cities c and d in C, cRd if there is a nonstop commercial flight from c to d. For a particular city $s \in C$, we would like to determine the subset $r(s)$ of C containing the cities that can be reached from s, by taking zero or more nonstop flights. Then it is easy to see that the set $r(s)$ can be described by the following recursive definition.

1. $s \in r(s)$.
2. For every $c \in r(s)$, and every $d \in C$ for which cRd, $d \in r(s)$.

Starting with s, by the time we have considered every sequence of steps in which the second statement is used n times, we have obtained all the cities that can be reached from s by taking n or fewer nonstop flights. The set C is finite, and so the set $r(s)$ is finite. If $r(S)$ has N elements, then it is easy to see that by using the second statement $N - 1$ times we can find every element of $r(s)$. However, we may not need that many steps. If after n steps we have the set $r_n(s)$ of cities that can be reached from s in n or fewer steps, and $r_{n+1}(s)$ turns out to be the same set (with no additional cities), then further iterations will not add any more cities, and $r(s) = r_n(s)$. The conclusion is that we can obtain $r(s)$ using the following algorithm.

```
$r_0(s) = \{s\}$
$n = 0$
repeat
    $n = n + 1$
    $r_n(s) = r_{n-1}(s) \cup \{d \in C \mid cRd \text{ for some } c \in r_{n-1}(s)\}$
until $r_n(s) = r_{n-1}(s)$
$r(s) = r_n(s)$
```

In the same way, if we have a finite set C and a recursive definition of a subset S of C, then even if we don't know how many elements C has, we can translate our definition into an algorithm that is guaranteed to terminate and to produce the set S.

In general, if R is a relation on an arbitrary set A, we can use a recursive definition similar to the one above to obtain the *transitive closure* of R, which can be described as the smallest transitive relation containing R.

1.6 | STRUCTURAL INDUCTION

In the previous section we found a recursive definition for a language *Expr* of simple algebraic expressions. Here it is again, with the operator notation we introduced.

1. $a \in$ *Expr*.
2. For every x and every y in *Expr*, $x \circ y$ and $x \bullet y$ are in *Expr*.
3. For every $x \in$ *Expr*, $\diamond(x) \in$ *Expr*.

(By definition, if x and y are elements of *Expr*, $x \circ y = x + y$, $x \bullet y = x * y$, and $\diamond(x) = (x)$.)

Suppose we want to prove that every string x in *Expr* satisfies the statement $P(x)$. (Two possibilities for $P(x)$ are the statements "x has equal numbers of left and right parentheses" and "x has an odd number of symbols".) Suppose also that the recursive definition of *Expr* provides all the information that we have about the language. How can we do it?

The principle of *structural induction* says that in order to show that $P(x)$ is true for every $x \in$ *Expr*, it is sufficient to show:

1. $P(a)$ is true.
2. For every x and every y in *Expr*, if $P(x)$ and $P(y)$ are true, then $P(x \circ y)$ and $P(x \bullet y)$ are true.
3. For every $x \in$ *Expr*, if $P(x)$ is true, then $P(\diamond(x))$ is true.

It's not hard to believe that this principle is correct. If the element a of *Expr* that we start with has the property we want, and if all the operations we can use to get new elements *preserve* the property (that is, when they are applied to elements having the property, they produce elements having the property), then there is no way we can ever use the definition to produce an element of *Expr* that does not have the property.

Another way to understand the principle is to use our paraphrase of the recursive definition of *Expr*. Suppose we denote by L_P the language of all strings satisfying P. Then saying every string in *Expr* satisfies P is the same as saying that *Expr* $\subseteq L_P$. If *Expr* is indeed the smallest language that contains a and is closed under the operations $\circ$, $\bullet$, and $\diamond$, then *Expr* is a subset of every language that has these properties, and so it is enough to show that L_P itself has them—i.e., L_P contains a and is closed under the three operations. And this is just what the principle of structural induction says.

The feature to notice in the statement of the principle is the close resemblance of the statements 1–3 to the recursive definition of *Expr*. The outline of the proof is provided by the structure of the definition. We illustrate the technique of structural induction by taking P to be the second of the two properties mentioned above and proving that every element of *Expr* satisfies it.

A Proof by Structural Induction That Every Element of *Expr* Has Odd Length

EXAMPLE 1.22

To simplify things slightly, we will combine statements 2 and 3 of our first definition into a single statement, as follows:

1. $a \in$ *Expr*.
2. For every x and every y in *Expr*, $x + y$, $x * y$, and (x) are in *Expr*.

The corresponding statements that we will establish in our proof are these:

1. $|a|$ is odd.
2. For every x and y in *Expr*, if $|x|$ and $|y|$ are odd, then $|x + y|$, $|x * y|$, and $|(x)|$ are odd.

The *basis* statement of the proof is the statement that $|a|$ is odd, which corresponds to the basis statement $a \in$ *Expr* in the recursive definition of *Expr*. When we prove the conditional statement, in the *induction step* of the proof, we will assume that x and y are elements of *Expr* and that $|x|$ and $|y|$ are odd. We refer to this assumption as the *induction hypothesis*. We make no other assumptions about x and y; they are arbitrary elements of *Expr*. This is confusing at first, because it seems as though we are assuming what we're trying to prove (that for arbitrary elements x and y, $|x|$ and $|y|$ are odd). It's important to say it carefully. We are not assuming that $|x|$ and $|y|$ are odd for every x and y in *Expr*. Rather, we are considering two arbitrary elements x and y and assuming that the lengths of those two strings are odd, in order to prove the conditional statement

$$\text{If } |x| \text{ and } |y| \text{ are odd, then } |x \circ y|, |x \bullet y| \text{ and } |\diamond(x)| \text{ are odd.}$$

Each time we present a proof by structural induction, we will be careful to state explicitly what we are trying to do in each step. We say first what we are setting out to prove, or what the ultimate objective is; second, what the statement is that needs to be proved in the basis step, and why it is true; third, what the induction hypothesis is; fourth, what we are trying to prove in the induction step; and finally, what the steps of that proof are. Surprisingly often, if we are able to do the first four things correctly and precisely, the proof of the induction step turns out to be very easy.

Here is our proof.

To Prove: For every element x of *Expr*, $|x|$ is odd.

Basis step. We wish to show that $|a|$ is odd. This is true because $|a| = 1$.

Induction hypothesis. x and y are in *Expr*, and $|x|$ and $|y|$ are odd.

Statement to be proved in the induction step. $|x + y|$, $|x * y|$, and $|(x)|$ are odd.

Proof of induction step. The numbers $|x + y|$ and $|x * y|$ are both $|x| + |y| + 1$, because the symbols of $x + y$ include those in x, those in y, and the additional "operator" symbol. The number $|(x)|$ is $|x| + 2$, because two parentheses have been added to the symbols of x. The first number is odd because the induction hypothesis implies that it is the sum of two odd numbers plus 1, and the second number is odd because the induction hypothesis implies that it is an odd number plus 2.

EXAMPLE 1.23 Mathematical Induction

Very often, the easiest way to prove a statement of the form "For every integer $n \geq n_0$, $P(n)$" is to apply the principle of structural induction, using the recursive definition given in Example 1.11 of the subset $\{n \in \mathcal{N} \mid n \geq n_0\}$. Such a proof is referred to as a proof by mathematical induction, or simply a proof by induction. The statement $P(n)$ might be a numerical fact or algebraic formula involving n, but in our subject it could just as easily be a statement about a set with n elements, or a string of length n, or a sequence of n steps.

The basis step is to establish the statement $P(n)$ for n_0, the smallest number in the set. The induction hypothesis is the assumption that k is a number in the set and that $P(n)$ is true when $n = k$, or that $P(k)$ is true; and the induction step is to show using this assumption that $P(k+1)$ is true. Here is an example, in which $n_0 = 0$, so that the set is simply $\mathcal{N}$.

To prove: For every $n \in \mathcal{N}$, and every set A with n elements, 2^A has exactly 2^n elements.

Basis step. The statement to be proved is that for every set A with 0 elements, 2^A has $2^0 = 1$ element. This is true because the only set with no elements is $\emptyset$, and $2^\emptyset = \{\emptyset\}$, which has one element.

Induction hypothesis. $k \in \mathcal{N}$ and for every set A with k elements, 2^A has 2^k elements.

Statement to be proved in the induction step. For every set A with $k+1$ elements, 2^A has 2^{k+1} elements.

Proof of induction step. If A has $k+1$ elements, then because $k \geq 0$, it must have at least one. Let a be an element of A. Then $A - \{a\}$ has k elements. By the induction hypothesis, $A - \{k\}$ has exactly 2^k subsets, and so A has exactly 2^k subsets that do not contain a. Every subset B of A that contains a can be written $B = B_1 \cup \{a\}$, where B_1 is a subset of A that doesn't contain a, and for two different subsets containing a, the corresponding subsets not containing a are also different; therefore, there are precisely as many subsets of A that contain a as there are subsets that do not. It follows that the total number of subsets is $2 * 2^k = 2^{k+1}$.

Strong Induction **EXAMPLE 1.24**

We present another proof by mathematical induction, to show that every positive integer 2 or larger can be factored into prime factors. The proof will illustrate a variant of mathematical induction that is useful in situations where the ordinary induction hypothesis is not quite sufficient.

To prove: For every natural number $n \geq 2$, n is either a prime or a product of two or more primes.

For reasons that will be clear very soon, we modify the statement to be proved, in a way that makes it seem like a stronger statement.

To prove: For every natural number $n \geq 2$, every natural number m satisfying $2 \leq m \leq n$ is either a prime or a product of two or more primes.

Basis step. When $n = 2$, the modified statement is that every number m satisfying $2 \leq m \leq 2$ is either a prime or a product of two or more primes. Of course the only such number m is 2, and the statement is true because 2 is prime.

Induction hypothesis. $k \geq 2$, and for every m satisfying $2 \leq m \leq k$, m is either a prime or a product of primes.

Statement to be proved in the induction step. For every m satisfying $2 \leq m \leq k+1$, m is either prime or a product of primes.

Proof of induction step. For every m with $2 \leq m \leq k$, we already have the conclusion we want, from the induction hypothesis. The only additional statement we need to prove is that $k+1$ is either prime or a product of primes.

If $k+1$ is prime, then the statement we want is true. Otherwise, by the definition of a prime, $k+1 = r * s$, for some positive integers r and s, neither of which is 1 or $k+1$. It follows that $2 \leq r \leq k$ and $2 \leq s \leq k$, and so the induction hypothesis implies that each of the two is either prime or a product of primes. We may conclude that their product $k+1$ is the product of two or more primes.

As we observed in the proof, the basis step and the statement to be proved in the induction step were not changed at all as a result of modifying the original statement. The purpose of the modification is simply to allow us to use the stronger induction hypothesis: not only that the statement $P(n)$ is true when $n = k$, but that it is true for every n satisfying $2 \leq n \leq k$. This was not necessary in Example 1.23, but often you will find when you reach the proof of the induction step that the weaker hypothesis doesn't provide enough information. In this example, it tells us that k is a prime or a product of primes—but we need to know that the numbers r and s have this property, and neither of these is k.

Now that we have finished this example, you don't have to modify the statement to be proved when you encounter another situation where the stronger induction hypothesis is necessary; declaring that you are using *strong induction* allows you to assume it.

In Example 1.19 we gave a recursive definition of the language *Balanced*, the set of balanced strings of parentheses. When you construct an algebraic expression or an expression in a computer program, and you need to end up with a balanced string of parentheses, you might check your work using an algorithm like the following. Go through the string from left to right, and keep track of the number of excess left parentheses; start at 0, add 1 each time you hit a left parenthesis, and subtract 1 each time you hit a right parenthesis; if the number is 0 when you reach the end and has never dropped below 0 along the way, the string is balanced. We mentioned at the beginning of this section that strings in *Expr* or *Balanced* have equal numbers of left and right parentheses; strengthening the condition by requiring that no prefix have more right parentheses than left produces a condition that characterizes balanced strings and explains why this algorithm works.

EXAMPLE 1.25 Another Characterization of Balanced Strings of Parentheses

The language *Balanced* was defined as follows in Example 1.19.

1. $\Lambda \in$ *Balanced*.
2. For every x and every y in *Balanced*, both xy and (x) are elements of *Balanced*.

We wish to show that a string x belongs to this language if and only if the statement $B(x)$ is true:

> $B(x)$: x contains equal numbers of left and right parentheses, and no prefix of x contains more right than left.

For the first part, showing that every string x in *Balanced* makes the condition $B(x)$ true, we can use structural induction, because we have a recursive definition of *Balanced*. The basis step is to show that $B(\Lambda)$ is true, and it is easy to see that it is. The induction hypothesis is that x and y are two strings in *Balanced* for which $B(x)$ and $B(y)$ are true, and the statement to be proved in the induction step is that $B(xy)$ and $B((x))$ are both true. We will show the first statement, and the second is at least as simple.

Because x and y both have equal numbers of left and right parentheses, the string xy does also. If z is a prefix of xy, then either z is a prefix of x or $z = xw$ for some prefix w of y. In the first case, the induction hypothesis tells us $B(x)$ is true, which implies that z cannot have more right parentheses than left. In the second case, the induction hypothesis tells us that x has equal numbers of left and right parentheses and that w cannot have more right than left; therefore, xw cannot have more right than left.

For the second part of the proof, we can't use structural induction based on the recursive definition of *Balanced*, because we're trying to prove that *every* string of parentheses, not just every string in *Balanced*, satisfies some property. There is not a lot to be gained by trying to use structural induction based on a recursive definition of the set of strings of parentheses, and instead we choose strong induction, rewriting the statement so that it involves an integer explicitly:

To prove: For every $n \in \mathcal{N}$, if x is a string of parentheses so that $|x| = n$ and $B(x)$ is true, then $x \in$ *Balanced*.

Basis step. The statement in the basis step is that if $|x| = 0$ and $B(x)$, then $x \in$ *Balanced*. We have more assumptions than we need; if $|x| = 0$, then $x = \Lambda$, and so $x \in$ *Balanced* because of statement 1 in the definition of *Balanced*.

Induction hypothesis. $k \in \mathcal{N}$, and for every string x of parentheses, if $|x| \leq k$ and $B(x)$, then $x \in$ *Balanced*. (Writing "for every string x of parentheses, if $|x| \leq k$" says the same thing as "for every $m \leq k$, and every string x of parentheses with $|x| = m$" but involves one fewer variable and sounds a little simpler.)

Statement to be proved in induction step. For every string x of parentheses, if $|x| = k + 1$ and $B(x)$, then $x \in$ *Balanced*.

Proof of induction step. We suppose that x is a string of parentheses with $|x| = k + 1$ and $B(x)$. We must show that $x \in$ *Balanced*, which means that x can be obtained from statement 1 or statement 2 in the definition. Since $|x| > 0$, statement 1 won't help; we must show x can be obtained from statement 2. The trick here is to look at the two cases in statement 2 and work backward.

If we want to show that $x = yz$ for two shorter strings y and z in *Balanced*, the way to do it is to show that $x = yz$ for two shorter strings y and z satisfying $B(y)$ and $B(z)$; for then the induction hypothesis will tell us that y and z are in *Balanced*. However, this may not be possible, because the statements $B(y)$ and $B(z)$ require that

y and z both have equal numbers of left and right parentheses. The string ((())), for example, cannot be expressed as a concatenation like this. We must show that these other strings can be obtained from statement 2.

It seems reasonable, then, to consider two cases. Suppose first that $x = yz$, where y and z are both shorter than x and have equal numbers of left and right parentheses. No prefix of y can have more right parentheses than left, because every prefix of y is a prefix of x and $B(x)$ is true. Because y has equal numbers of left and right, and because no prefix of x can have more right than left, no prefix of z can have more right than left. Therefore, both the statements $B(y)$ and $B(z)$ are true. Since $|y| \leq k$ and $|z| \leq k$, we can apply the induction hypothesis to both strings and conclude that y and z are both elements of *Balanced*. It follows from statement 2 of the definition that $x = yz$ is also.

In the other case, we assume that $x = (y)$ for some string y of parentheses and that x cannot be written as a concatenation of shorter strings with equal numbers of left and right parentheses. This second assumption is useful, because it tells us that no prefix of y can have more right parentheses than left. (If some prefix did, then some prefix y_1 of y would have exactly one more right than left, which would mean that the prefix $(y_1$ of x had equal numbers; but this would contradict the assumption.) The string y has equal numbers of left and right parentheses, because x does, and so the statement $B(y)$ is true. Therefore, by the induction hypothesis, $y \in$ *Balanced*, and it follows from statement 2 that $x \in$ *Balanced*.

EXAMPLE 1.26 Sometimes Making a Statement Stronger Makes It Easier to Prove

In this example we return to the recursive definition of *Expr* that we have already used in Example 1.22 to prove a simple property of algebraic expressions. Suppose we want to prove now that no string in *Expr* can contain the substring ++.

In the basis step we observe that a does not contain this substring. If we assume in the induction hypothesis that neither x nor y contains it, then it is easy to conclude that $x * y$ and (x) also don't. Trying to prove that $x + y$ doesn't, however, presents a problem. Neither x nor y contains ++ as a substring, but if x ended with + or y started with +, then ++ would occur in the concatenation. The solution is to prove the stronger statement that for every $x \in$ *Expr*, x doesn't begin or end with + *and* doesn't contain the substring ++. In both the basis step and the induction step, there will be a few more things to prove, but they are not difficult and the induction hypothesis now contains all the information we need to carry out the proof.

EXAMPLE 1.27 Defining Functions on Sets Defined Recursively

A recursive definition of a set suggests a way to prove things about elements of the set. By the same principle, it offers a way of defining a function at every element of the set. In the case of the natural numbers, for example, if we define a function at 0, and if for every natural number n we say what $f(n + 1)$ is, assuming that we know what $f(n)$ is, then we have effectively defined the function at every element of $\mathcal{N}$. There are many familiar functions commonly defined this way, such as the factorial function f:

$$f(0) = 1;\ \text{for every } n \in \mathcal{N},\ f(n+1) = (n+1) * f(n)$$

and the function $u : \mathcal{N} \to 2^A$ defined by

$$u(0) = S_0;\ \text{for every } n \in \mathcal{N}, u(n+1) = u(n) \cup S_{n+1}$$

where S_0, S_1, ... are assumed to be subsets of A. Writing nonrecursive definitions of these functions is also common:

$$f(n) = n * (n-1) * (n-2) * \cdots * 2 * 1$$
$$u(n) = S_0 \cup S_1 \cup S_2 \cup \cdots \cup S_n$$

In the second case, we could avoid "..." by writing

$$u(n) = \bigcup_{i=0}^{n} S_i = \bigcup \{S_i \mid 0 \le i \le n\}$$

although the recursive definition also provides concise definitions of both these notations.

It is easy to see that definitions like these are particularly well suited for induction proofs of properties of the corresponding functions, and the exercises contain a few examples. We consider a familiar function $r : \{a, b\}^* \to \{a, b\}^*$ that can be defined recursively by referring to the recursive definition of $\{a, b\}^*$ in Example 1.17.

$$r(\Lambda) = \Lambda;\ \text{for every } x \in \{a, b\}^*,\ r(xa) = ar(x) \text{ and } r(xb) = br(x).$$

If it is not obvious what the function r is, you can see after using the definition to compute $r(aaba)$, for example,

$$r(aaba) = ar(aab) = abr(aa) = abar(a) = abar(\Lambda a) = abaar(\Lambda) = abaa\Lambda = abaa$$

that it is the function that reverses the order of the symbols of a string. We will often use the notation x^r instead of $r(x)$, in this example as well as several places where this function makes an appearance later.

To illustrate the close relationship between the recursive definition of $\{a, b\}^*$, the recursive definition of r, and the principle of structural induction, we prove the following fact about the reverse function.

$$\text{For every } x \text{ and every } y \text{ in } \{a, b\}^*,\ (xy)^r = y^r x^r$$

In planning the proof, we are immediately faced with a potential problem, because the statement has the form "for every x and every y, ..." rather than the simpler form "for every x, ..." that we have considered before. The first step in resolving this issue is to realize that the quantifiers are nested; we can write the statement in the form $\forall x(P(x))$, where $P(x)$ is itself a quantified statement, $\forall y(\dots)$, and so we can attempt to use structural induction on x.

In fact, although the principle here is reasonable, you will discover if you try this approach that it doesn't work. It will be easier to see why after we have completed the proof using the approach that does work.

The phrase "for every x and every y" means the same thing as "for every y and every x", and now the corresponding formula looks like $\forall y(\dots)$. As we will see, this turns out to be better because of the order in which x and y appear in the expression $r(xy)$.

To prove: For every y in $\{a, b\}^*$, $P(y)$ is true, where $P(y)$ is the statement "for every $x \in \{a, b\}^*$, $(xy)^r = y^r x^r$".

Basis step. The statement to be proved in the basis step is this: For every $x \in \{a, b\}^*$, $(x\Lambda)^r = \Lambda^r x^r$. This statement is true, because for every x, $x\Lambda = x$; Λ^r is defined to be Λ; and $\Lambda x^r = x^r$.

Induction hypothesis. $y \in \{a, b\}^*$, and for every $x \in \{a, b\}^*$, $(xy)^r = y^r x^r$.

Statement to be proved in induction step. For every $x \in \{a, b\}^*$,
$(x(ya))^r = (ya)^r x^r$ and $(x(yb))^r = (yb)^r x^r$.

Proof of induction step. We will prove the first part of the statement, and the proof in the second part is almost identical. The tools that we have available are the recursive definition of $\{a, b\}^*$, the recursive definition of r, and the induction hypothesis; all we have to do is decide which one to use when, and avoid getting lost in the notation.

$$\begin{aligned}
(x(ya))^r &= ((xy)a)^r && \text{(because concatenation is associative—i.e., } x(ya) = (xy)a) \\
&= a(xy)^r && \text{(by the second part of the definition of } r\text{, with } xy \text{ instead of } x) \\
&= a(y^r x^r) && \text{(by the induction hypothesis)} \\
&= (ay^r)x^r && \text{(because concatenation is associative)} \\
&= (ya)^r x^r && \text{(by the second part of the definition of } r\text{, with } y \text{ instead of } x)
\end{aligned}$$

Now you can see why the first approach wouldn't have worked. Using induction on x, we would start out with $((xa)y)^r$. We can rewrite this as $(x(ay))^r$, and the induction hypothesis in this version would allow us to rewrite it again as $(ay)^r x^r$. But the a is at the wrong end of the string ay, and the definition of r gives us no way to proceed further.

EXERCISES

1.1. In each case below, construct a truth table for the statement and find another statement with at most one operator ($\vee$, $\wedge$, $\neg$, or $\rightarrow$) that is logically equivalent.

a. $(p \rightarrow q) \wedge (p \rightarrow \neg q)$

b. $p \vee (p \rightarrow q)$

c. $p \wedge (p \rightarrow q)$

d. $(p \rightarrow q) \wedge (\neg p \rightarrow q)$

e. $p \leftrightarrow (p \leftrightarrow q)$

f. $q \wedge (p \rightarrow q)$

1.2. A principle of classical logic is *modus ponens*, which asserts that the proposition $(p \wedge (p \rightarrow q)) \rightarrow q$ is a tautology, or that $p \wedge (p \wedge q)$ logically implies q. Is there any way to define the conditional statement $p \rightarrow q$, other than the way we defined it, that makes it false when p is true and q is false and makes the modus ponens proposition a tautology? Explain.

1.3. Suppose m_1 and m_2 are integers representing months ($1 \leq m_i \leq 12$), and d_1 and d_2 are integers representing days (d_i is at least 1 and no larger than the number of days in month m_i). For each i, the pair (m_i, d_i) can be thought of as representing a date; for example, (9, 18) represents September 18. We wish to write a logical proposition involving the four integers that says (m_1, d_1) comes before (m_2, d_2) in the calendar.

a. Find such a proposition that is a disjunction of two propositions (i.e., combines them using $\vee$).

b. Find such a proposition that is a conjunction of two propositions (combines them using $\wedge$).

1.4. In each case below, say whether the statement is a tautology, a contradiction, or neither.

a. $p \vee \neg(p \rightarrow p)$

b. $p \wedge \neg(p \rightarrow p)$

c. $p \rightarrow \neg p$

d. $(p \rightarrow \neg p) \vee (\neg p \rightarrow p)$

e. $(p \rightarrow \neg p) \wedge (\neg p \rightarrow p)$

f. $(p \wedge q) \vee (\neg p) \vee (\neg q)$

1.5. In the nine propositions $p \wedge q \vee r$, $p \vee q \wedge r$, $\neg p \wedge q$, $\neg p \vee q$, $\neg p \rightarrow q$, $p \vee q \rightarrow r$, $p \wedge q \rightarrow r$, $p \rightarrow q \vee r$, and $p \rightarrow q \wedge r$, the standard convention if no parentheses are used is to give $\neg$ the highest precedence, $\wedge$ the next-highest, $\vee$ the next-highest after that, and $\rightarrow$ the lowest. For example, $\neg p \vee r$ would normally be interpreted $(\neg p) \vee r$ and $p \rightarrow q \vee r$ would normally be interpreted $p \rightarrow (q \vee r)$. Are there any of the nine whose truth value would be unchanged if the precedence of the two operators involved were reversed? If so, which ones?

1.6. Prove that every string of length 4 over the alphabet $\{a, b\}$ contains the substring xx, for some nonnull string x. One way is to consider all sixteen cases, but try to reduce the number of cases as much as possible.

1.7. Describe each of the following infinite sets using the format $\{\_\_\_\_\_\_\_ \mid n \in \mathcal{N}\}$, without using "..." in the expression on the left side of the vertical bar.

a. $\{0, -1, 2, -3, 4, -5, \dots\}$

b. $\{\{0\}, \{1\}, \{2\}, \dots\}$

c. $\{\{0\}, \{0, 1\}, \{0, 1, 2\}, \{0, 1, 2, 3\}, \dots\}$

d. $\{\{0\}, \{0, 1\}, \{0, 1, 2, 3\}, \{0, 1, 2, 3, 4, 5, 6, 7\}, \{0, 1, \dots, 15\}, \{0, 1, 2, \dots, 31\}, \dots\}$

1.8. In each case below, find an expression for the indicated set, involving A, B, C, and any of the operations $\cup$, $\cap$, $-$, and $'$.

a. $\{x | x \in A$ or $x \in B$ but not both$\}$

b. $\{x | x$ is an element of exactly one of the three sets A, B, and $C\}$

c. $\{x | x$ is an element of at most one of the three sets A, B, and $C\}$

d. $\{x | x$ is an element of exactly two of the three sets A, B, and $C\}$

1.9. For each integer n, denote by C_n the set of all real numbers less than n, and for each positive number n let D_n be the set of all real numbers less than $1/n$. Express each of the following unions or intersections in a simpler way. For example, the answer to (a) is C_{10}. The answer is not always one of the sets C_i or D_i, but there is an equally simple answer in each case. Since ∞ is not a number, the expressions C_∞ and D_∞ do not make sense and should not appear in your answers.

a. $\bigcup\{C_n \mid 1 \le n \le 10\}$
b. $\bigcup\{D_n \mid 1 \le n \le 10\}$
c. $\bigcap\{C_n \mid 1 \le n \le 10\}$
d. $\bigcap\{D_n \mid 1 \le n \le 10\}$
e. $\bigcup\{C_n \mid 1 \le n\}$
f. $\bigcup\{D_n \mid 1 \le n\}$
g. $\bigcap\{C_n \mid 1 \le n\}$
h. $\bigcap\{D_n \mid 1 \le n\}$
i. $\bigcup\{C_n \mid n \in \mathcal{Z}\}$
j. $\bigcap\{C_n \mid n \in \mathcal{Z}\}$

1.10. List the elements of $2^{2^{\{0,1\}}}$, and number the items in your list.

1.11. In each case below, say whether the given statement is true for the universe $(0, 1) = \{x \in \mathcal{R} \mid 0 < x < 1\}$, and say whether it is true for the universe $[0, 1] = \{x \in \mathcal{R} \mid 0 \le x \le 1\}$. For each of the four cases, you should therefore give two true-or-false answers.

a. $\forall x(\exists y(x > y))$
b. $\forall x(\exists y(x \ge y))$
c. $\exists y(\forall x(x > y))$
d. $\exists y(\forall x(x \ge y))$

1.12. a. How many elements are there in the set

$$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}\}\}?$$

b. Describe precisely the algorithm you used to answer part (a).

1.13. Simplify the given set as much as possible in each case below. Assume that all the numbers involved are real numbers.

a. $\bigcap\{\{x \mid |x - a| < r\} \mid r > 0\}$
b. $\bigcup\{\{x \mid |x - a| \le r\} \mid r > 0$

1.14. Suppose that A and B are nonempty sets and $A \times B \subseteq B \times A$. Show that $A = B$. Suggestion: show that $A \subseteq B$ and $B \subseteq A$, using proof by contradiction in each case.

1.15. Suppose that A and B are subsets of a universal set U.

a. What is the relationship between $2^{A \cup B}$ and $2^A \cup 2^B$? (Under what circumstances are they equal? If they are not equal, is one necessarily a subset of the other, and if so, which one?) Give reasons for your answers.

b. Same question for $2^{A \cap B}$ and $2^A \cap 2^B$.

c. Same question for $2^{(A')}$ and $(2^A)'$ (both subsets of 2^U).

1.16. Suppose A and B are finite sets, A has n elements, and $f : A \to B$.

a. If f is one-to-one, what can you say about the number of elements of B?

b. If f is onto, what can you say about the number of elements of B?

1.17. In each case below, say whether the indicated function is one-to-one and what its range is.

a. $m : \mathcal{N} \to \mathcal{N}$ defined by $m(x) = \min(x, 2)$

b. $M : \mathcal{N} \to \mathcal{N}$ defined by $M(x) = \max(x, 2)$

c. $s : \mathcal{N} \to \mathcal{N}$ defined by $s(x) = m(x) + M(x)$

d. $f : \mathcal{N} - \{0\} \to 2^{\mathcal{N}}$, where $f(n)$ is the set of prime factors of n

e. (Here A is the set of all finite sets of primes and B is the set $\mathcal{N} - \{0\}$.) $g : A \to B$, where $g(S)$ is the product of the elements of S. (The product of the elements of the empty set is 1.)

1.18. Find a formula for a function from $\mathcal{Z}$ to $\mathcal{N}$ that is a bijection.

1.19. In each case, say whether the function is one-to-one and whether it is onto.

a. $f : \mathcal{Z} \times \mathcal{Z} \to \mathcal{Z} \times \mathcal{Z}$, defined by $f(a, b) = (a + b, a - b)$

b. $f : \mathcal{R} \times \mathcal{R} \to \mathcal{R} \times \mathcal{R}$, defined by $f(a, b) = (a + b, a - b)$

1.20. Suppose A and B are sets and $f : A \to B$. For a subset S of A, we use the notation $f(S)$ to denote the set $\{f(x) \mid x \in S\}$. Let S and T be subsets of A.

a. Is the set $f(S \cup T)$ a subset of $f(S) \cup f(T)$? If so, give a proof; if not, give a counterexample (i.e., say what the sets A, B, S, and T are and what the function f is).

b. Is the set $f(S) \cup f(T)$ a subset of $f(S \cup T)$? Give either a proof or a counterexample.

c. Repeat part (a) with intersection instead of union.

d. Repeat part (b) with intersection instead of union.

e. In each of the first four parts where your answer is no, what extra assumption on the function f would make the answer yes? Give reasons for your answer.

1.21. Let E be the set of even natural numbers, S the set of nonempty subsets of E, T the set of nonempty subsets of $\mathcal{N}$, and $\mathcal{P}$ the set of partitions of $\mathcal{N}$ into two nonempty subsets.

a. Suppose $f : T \to \mathcal{P}$ is defined by the formula $f(A) = \{A, \mathcal{N} - A\}$ (in other words, for a nonempty subset A of $\mathcal{N}$, $f(A)$ is the partition of $\mathcal{N}$ consisting of the two subsets A and $\mathcal{N} - A$). Is f a bijection from T to $\mathcal{P}$? Why or why not?

b. Suppose that $g : S \to \mathcal{P}$ is defined by $g(A) = \{A, \mathcal{N} - A\}$. Is g a bijection from S to $\mathcal{P}$? Why or why not?

1.22. Suppose U is a set, $\circ$ is a binary operation on U, and S_0 is a subset of U. Define the subset A of U recursively as follows:

$$S_0 \subseteq A; \text{ for every } x \text{ and } y \text{ in } A, x \circ y \in A$$

(In other words, A is the smallest subset of A that contains the elements of S_0 and is closed under $\circ$.) Show that

$$A = \bigcap \{S \mid S_0 \subseteq S \subseteq U \text{ and } S \text{ is closed under } \circ\}$$

1.23. In each case below, a relation on the set $\{1, 2, 3\}$ is given. Of the three properties, reflexivity, symmetry, and transitivity, determine which ones the relation has. Give reasons.

a. $R = \{(1, 3), (3, 1), (2, 2)\}$

b. $R = \{(1, 1), (2, 2), (3, 3), (1, 2)\}$

c. $R = \emptyset$

1.24. For each of the eight lines of the table below, construct a relation on $\{1, 2, 3\}$ that fits the description.

reflexive	**symmetric**	**transitive**
true	true	true
true	true	false
true	false	true
true	false	false
false	true	true
false	true	false
false	false	true
false	false	false

1.25. Each case below gives a relation on the set of all nonempty subsets of $\mathcal{N}$. In each case, say whether the relation is reflexive, whether it is symmetric, and whether it is transitive.

a. R is defined by: ARB if and only if $A \subseteq B$.

b. R is defined by: ARB if and only if $A \cap B \neq \emptyset$.

c. R is defined by: ARB if and only if $1 \in A \cap B$.

1.26. Let R be a relation on a set S. Write three quantified statements (the domain being S in each case), which say, respectively, that R is not reflexive, R is not symmetric, and R is not transitive.

1.27. Suppose S is a nonempty set, $A = 2^S$, and the relation R on A is defined as follows: For every X and every Y in A, XRY if and only if there is a bijection from X to Y.

a. Show that R is an equivalence relation.

b. If S is a finite set with n elements, how many equivalence classes does the equivalence relation R have?

c. Again assuming that S is finite, describe a function $f : A \to \mathcal{N}$ so that for every X and Y in A, XRY if and only if $f(X) = f(Y)$.

1.28. Suppose A and B are sets, $f : A \to B$ is a function, and R is the relation on A so that for $x, y \in A$, xRy if and only if $f(x) = f(y)$.

a. Show that R is an equivalence relation on A.

b. If $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $B = \mathcal{N}$, and $f(x) = (x - 3)^2$ for every $x \in A$, how many equivalence classes are there, and what are the elements of each one?

c. Suppose A has p elements and B has q elements. If the function f is one-to-one (not necessarily onto), how many equivalence classes does the equivalence relation R have? If the function f is onto (not necessarily one-to-one), how many equivalence classes does R have?

1.29. Show that for every set A and every equivalence relation R on A, there is a set B and a function $f : A \to B$ such that R is the relation described in Exercise 1.28.

1.30. For a positive integer n, find a function $f : \mathcal{N} \to \mathcal{N}$ so that the equivalence relation $\equiv_n$ on $\mathcal{N}$ can be described as in Exercise 1.28.

1.31. Show that for every language L, $LL^* = L^*$ if and only if $\Lambda \in L$.

1.32. For a finite language L, let $|L|$ denote the number of elements of L. For example, $|\{\Lambda, a, ababb\}| = 3$. This notation has nothing to do with the length $|x|$ of a string x. The statement $|L_1L_2| = |L_1||L_2|$ says that the number of strings in the concatenation L_1L_2 is the same as the product of the two numbers $|L_1|$ and $|L_2|$. Is this always true? If so, give reasons, and if not, find two finite languages $L_1, L_2 \subseteq \{a, b\}^*$ such that $|L_1L_2| \neq |L_1||L_2|$.

1.33. Let L_1 and L_2 be subsets of $\{a, b\}^*$.

a. Show that if $L_1 \subseteq L_2$, then $L_1^* \subseteq L_2^*$.

b. Show that $L_1^* \cup L_2^* \subseteq (L_1 \cup L_2)^*$.

c. Give an example of two languages L_1 and L_2 such that $L_1^* \cup L_2^* \neq (L_1 \cup L_2)^*$.

d. †One way for the two languages $L_1^* \cup L_2^*$ and $(L_1 \cup L_2)^*$ to be equal is for one of the two languages L_1 and L_2 to be a subset of the other, or more generally, for one of the two languages L_1^* and L_2^* to be a subset of the other. Find an example of languages L_1 and L_2 for which neither of L_1^*, L_2^* is a subset of the other, but $L_1^* \cup L_2^* = (L_1 \cup L_2)^*$.

1.34. †Suppose that $x, y \in \{a, b\}^*$ and neither is Λ. Show that if $xy = yx$, then for some string z and two integers i and j, $x = z^i$ and $y = z^j$.

1.35. †Consider the language $L = \{yy \mid y \in \{a, b\}^*\}$. We know that $L = L\{\Lambda\} = \{\Lambda\}L$, because every language L has this property. Is there any other way to express L as the concatenation of two languages? Prove your answer.

1.36. a. Consider the language L of all strings of a's and b's that do not end with b and do not contain the substring bb. Find a finite language S such that $L = S^*$.

b. Show that there is no language S such that S^* is the language of all strings of a's and b's that do not contain the substring bb.

1.37. Let L_1, L_2, and L_3 be languages over some alphabet Σ. In each case below, two languages are given. Say what the relationship is between them. (Are they always equal? If not, is one always a subset of the other?) Give reasons for your answers, including counterexamples if appropriate.

a. $L_1(L_2 \cap L_3)$, $L_1L_2 \cap L_1L_3$

b. $L_1^* \cap L_2^*$, $(L_1 \cap L_2)^*$

c. $L_1^*L_2^*$, $(L_1L_2)^*$

1.38. In each case below, write a quantified statement, using the formal notation discussed in the chapter, that expresses the given statement. In both cases the set A is assumed to be a subset of the domain, not necessarily the entire domain.

a. There are at least two distinct elements in the set A satisfying the condition P (i.e., for which the proposition $P(x)$ holds).

b. There is exactly one element x in the set A satisfying the condition P.

1.39. Consider the following 'proof' that every symmetric, transitive relation R on a set A must also be reflexive:

> Let a be any element of A. Let b be any element of A for which aRb. Then since R is symmetric, bRa. Now since R is transitive, and since aRb and bRa, it follows that aRa. Therefore R is reflexive.

Your answer to Exercise 1.24 shows that this proof cannot be correct. What is the first incorrect statement in the proof, and why is it incorrect?

1.40. †Suppose A is a set having n elements.

a. How many relations are there on A?

b. How many reflexive relations are there on A?

c. How many symmetric relations are there on A?

d. How many relations are there on A that are both reflexive and symmetric?

1.41. Suppose R is a relation on a nonempty set A.

a. Define $R^s = R \cup \{(x, y) \mid yRx\}$. Show that R^s is symmetric and is the smallest symmetric relation on A containing R (i.e., for any symmetric relation R_1 with $R \subseteq R_1$, $R^s \subseteq R_1$).

b. Define R^t to be the intersection of all transitive relations on A containing R. Show that R^t is transitive and is the smallest transitive relation on A containing R.

c. Let $R^u = R \cup \{(x, y) \mid \exists z(xRz \text{ and } zRy)\}$. Is R^u equal to the set R^t in part (b)? Either prove that it is, or give an example in which it is not.

The relations R^s and R^t are called the symmetric closure and transitive closure of R, respectively.

1.42. Suppose R is an equivalence relation on a set A. A subset $S \subseteq A$ is *pairwise inequivalent* if no two distinct elements of S are equivalent. S is a *maximal* pairwise inequivalent set if S is pairwise inequivalent and for every element of A, there is an element of S equivalent to it. Show that a set S is a maximal pairwise inequivalent set if and only if it contains exactly one element of each equivalence class.

1.43. Suppose R_1 and R_2 are equivalence relations on a set A. As discussed in Section 1.3, the equivalence classes of R_1 and R_2 form partitions P_1 and P_2, respectively, of A. Show that $R_1 \subseteq R_2$ if and only if the partition P_1 is *finer* than P_2 (i.e., every subset in the partition P_2 is the union of one or more subsets in the partition P_1).

1.44. Each case below gives, a recursive definition of a subset L of $\{a, b\}^*$. Give a simple nonrecursive definition of L in each case.

a. $a \in L$; for any $x \in L$, xa and xb are in L.

b. $a \in L$; for any $x \in L$, bx and xb are in L.

c. $a \in L$; for any $x \in L$, ax and xb are in L.

d. $a \in L$; for any $x \in L$, xb, xa, and bx are in L.

e. $a \in L$; for any $x \in L$, xb, ax, and bx are in L.

f. $a \in L$; for any $x \in L$, xb and xba are in L.

1.45. Prove using mathematical induction that for every nonnegative integer n,

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$$

(If $n = 0$, the sum on the left is 0 by definition.)

1.46. Suppose r is a real number other than 1. Prove using mathematical induction that for every nonnegative integer n,

$$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

1.47. Prove using mathematical induction that for every nonnegative integer n,

$$1 + \sum_{i=1}^{n} i * i! = (n+1)!$$

1.48. Prove using mathematical induction that for every integer $n \geq 4$, $n! > 2^n$.

1.49. Suppose x is any real number greater than -1. Prove using mathematical induction that for every nonnegative integer n, $(1 + x)^n \geq 1 + nx$. (Be sure you say in your proof exactly how you use the assumption that $x > -1$.)

1.50. Prove using mathematical induction that for every positive integer n,

$$\sum_{i=1}^{n} i * 2^i = (n-1) * 2^{n+1} + 2$$

1.51. Prove using mathematical induction that for every nonnegative integer n, n is either even or odd but not both. (By definition, an integer n is even if there is an integer i so that $n = 2 * i$, and n is odd if there is an integer i so that $n = 2 * i + 1$.)

1.52. Prove that for every language $L \subseteq \{a, b\}^*$, if $L^2 \subseteq L$, then $LL^* \subseteq L$.

1.53. Suppose that Σ is an alphabet, and that $f : \Sigma^* \to \Sigma^*$ has the property that $f(\sigma) = \sigma$ for every $\sigma \in \Sigma$ and $f(xy) = f(x)f(y)$ for every $x, y \in \Sigma^*$. Prove that for every $x \in \Sigma^*$, $f(x) = x$.

1.54. Prove that for every positive integer n, there is a nonnegative integer i and an odd integer j so that $n = 2^i * j$.

1.55. Show using mathematical induction that for every $x \in \{a, b\}^*$ such that x begins with a and ends with b, x contains the substring ab.

1.56. Show using mathematical induction that every nonempty subset A of $\mathcal{N}$ has a smallest element. (Perhaps the hardest thing about this problem is finding a way of formulating the statement so that it involves an integer n and can therefore be proved by induction. Why is it *not* feasible to prove that for every integer $n \geq 1$, every subset A of $\mathcal{N}$ containing at least n elements has a smallest element?)

1.57. Some recursive definitions of functions on $\mathcal{N}$ don't seem to be based directly on the recursive definition of $\mathcal{N}$ in Example 1.15. The *Fibonacci* function f is usually defined as follows.

$$f(0) = 0;\ \ f(1) = 1;\ \ \text{for every } n > 1,\ f(n) = f(n-1) + f(n-2).$$

Here we need to give both the values $f(0)$ and $f(1)$ in the first part of the definition, and for each larger n, $f(n)$ is defined using both $f(n-1)$ and $f(n-2)$. If there is any doubt in your mind, you can use strong induction to verify that for every $n \in \mathcal{N}$, $f(n)$ is actually defined. Use strong induction to show that for every $n \in \mathcal{N}$, $f(n) \leq (5/3)^n$. (Note that in the induction step, you can use the recursive formula only if $n > 1$; checking the case $n = 1$ separately is comparable to performing a second basis step.)

1.58. The numbers a_n, for $n \geq 0$, are defined recursively as follows.

$$a_0 = -2;\ \ a_1 = -2;\ \ \ \text{for } n \geq 2,\ a_n = 5a_{n-1} - 6a_{n-2}$$

Use strong induction to show that for every $n \geq 0$, $a_n = 2 * 3^n - 4 * 2^n$. (Refer to Example 1.24.)

1.59. Show that the set B in Example 1.11 is precisely the set $S = \{2^i * 5^j \mid i, j \in \mathcal{N}\}$.

1.60. Suppose the language $L \subseteq \{a, b\}^*$ is defined recursively as follows:

$$\Lambda \in L;\qquad \text{for every } x \in L, \text{ both } ax \text{ and } axb \text{ are elements of } L.$$

Show that $L = L_0$, where $L_0 = \{a^i b^j \mid i \geq j\}$. To show that $L \subseteq L_0$ you can use structural induction, based on the recursive definition of L. In the other direction, use strong induction on the length of a string in L_0.

1.61. Find a recursive definition for the language $L = \{a^i b^j \mid i \leq j \leq 2i\}$, and show that it is correct (i.e., show that the language described by the recursive definition is precisely L). In order to come up with a recursive definition, it may be helpful to start with recursive definitions for each of the languages $\{a^i b^i \mid i \geq 0\}$ and $\{a^i b^{2i} \mid i \geq 0\}$.

1.62. In each case below, find a recursive definition for the language L and show that it is correct.

a. $L = \{a^i b^j \mid j \geq 2i\}$

b. $L = \{a^i b^j \mid j \leq 2i\}$

1.63. For a string x in the language *Expr* defined in Example 1.19, $n_a(x)$ denotes the number of a's in the string, and we will use $n_{\text{op}}(x)$ to stand for the number of operators in x (the number of occurrences of $+$ or $*$). Show that for every $x \in \mathit{Expr}$, $n_a(x) = 1 + n_{\text{op}}(x)$.

1.64. For a string x in *Expr*, show that x does not start or end with $+$ and does not contain the substring $++$. (In this case it would be feasible to prove, first, that strings in *Expr* do not start or end with $+$, and then that they don't contain the substring $++$; as Example 1.26 suggests, however, it is possible to prove both statements in the same induction proof.)

1.65. Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$$\Lambda \in L; \quad \text{for every } x \in L, \text{ both } xa \text{ and } xba \text{ are in } L.$$

Show that for every $x \in L$, both of the following statements are true.

a. $n_a(x) \geq n_b(x)$.

b. x does not contain the substring bb.

1.66. Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$$\Lambda \in L; \quad \text{for every } x \text{ and } y \text{ in } L, \text{ the strings } axb, bxa, \text{ and } xy \text{ are in } L.$$

Show that $L = \mathit{AEqB}$, the language of all strings x in $\{a, b\}^*$ satisfying $n_a(x) = n_b(x)$.

1.67. †Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$$\Lambda \in L; \quad \text{for every } x \text{ and } y \text{ in } L, \text{ the strings } axby \text{ and } bxay \text{ are in } L.$$

Show that $L = \mathit{AEqB}$, the language of all strings x in $\{a, b\}^*$ satisfying $n_a(x) = n_b(x)$.

1.68. †Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$$a \in L; \quad \text{for every } x \text{ and } y \text{ in } L, \text{ the strings } ax, bxy, xby, \text{ and } xyb \text{ are in } L.$$

Show that $L = L_0$, the language of all strings x in $\{a, b\}^*$ satisfying $n_a(x) > n_b(x)$.

1.69. For a relation R on a set S, the *transitive closure* of R is the relation R^t defined as follows: $R \subseteq R^t$; for every x, every y, and every z in S, if $(x, y) \in R^t$ and $(y, z) \in R^t$, then $(x, z) \in R^t$. (We can summarize the definition by saying that R^t is the smallest transitive relation containing R.) Show that if R_1 and R_2 are relations on S satisfying $R_1 \subseteq R_2$, then $R_1^t \subseteq R_2^t$.