Appendix A: Review of Mathematical Background

32 Logic, Sets, Relations, Functions, and Proof Techniques

Throughout this book, we rely on a collection of important mathematical concepts and notations. We summarize them here. For a deeper introduction to these ideas, see any good discrete mathematics text, for example [Epp 2003] or [Rosen 2003].

32.1 Logic

We assume familiarity with the standard systems of both Boolean and quantified logic, so this section is just a review of the definitions and notations that we will use, along with some of the most useful inference rules.

32.1.1 Boolean (Propositional) Logic

A ***proposition*** is a statement that has a truth value. The language of ***well-formed formulas*** (***wffs***) allows us to define propositions whose truth can be determined from the truth of other propositions. A wff is any string that is formed according to the following rules:

- A propositional symbol (e.g., P) is a wff. (Propositional symbols are also called ***variables***, primarily because the term is shorter. We will generally find it convenient to do that, but this use of the term should not be confused with its use in the definition of first-order logic.)
- If P is a wff, then $\neg P$ is a wff.
- If P and Q are wffs, then so are $P \vee Q$, $P \wedge Q$, $P \rightarrow Q$, and $P \leftrightarrow Q$.
- If P is a wff, then (P) is a wff.

Other binary operators, such as XOR (exclusive or) and NAND (not and), can also be defined, but we will not need them.

The definitions of the operators are given by the truth table shown as Table 32.1. It shows how the truth value of a proposition can be computed from the truth values of its components. (Note that the symbol $\vee$ means inclusive or.)

P	Q	$\neg P$	$P \vee Q$	$P \wedge Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
True	*True*	*False*	*True*	*True*	*True*	*True*
True	*False*	*False*	*True*	*False*	*False*	*False*
False	*True*	*True*	*True*	*False*	*True*	*False*
False	*False*	*True*	*False*	*False*	*True*	*True*

Table 32.1 A truth table for the common Boolean operators

We can divide the set of all Boolean wffs into three useful categories, as a function of when they are true:

- A Boolean wff is ***valid*** if and only if it is true for all assignments of truth values to the variables it contains. A valid wff is also called a ***tautology***.
- A Boolean wff is ***satisfiable*** if and only if it is true for at least one assignment of truth values to the variables it contains.
- A Boolean wff is ***unsatisfiable*** if and only if it is false for all assignments of truth values to the variables it contains.

Example 32.1 Using a Truth Table

The wff $P \vee \neg P$ is a tautology (i.e., it is valid). We can easily prove this by extending the truth table shown above and considering the only two possible cases (P is *True* or P is *False*):

P	$\neg P$	$P \vee \neg P$
True	*False*	*True*
False	*True*	*True*

The wff $P \vee \neg Q$ is satisfiable. It is *True* if either P is *True* or Q is *False*. It is not a tautology, however.

The wff $P \wedge \neg P$ is unsatisfiable. It is *False* both in case P is *True* and in case P is *False*.

We'll say that two wffs P and Q are ***equivalent***, which we will write as $P \equiv Q$, iff they have the same truth values regardless of the truth values of the variables they contain. So, for example, $(P \rightarrow Q) \equiv (\neg P \vee Q)$.

In interpreting wffs, we assume that $\neg$ has the highest precedence, followed by $\wedge$, then $\vee$, then $\rightarrow$, then $\leftrightarrow$. So:

$$(P \vee Q \wedge R) \equiv (P \vee (Q \wedge R)).$$

Parentheses can be used to force different interpretations.

The following properties (defined in Section 32.4.3) of the Boolean operators follow from their definitions in the truth table given above:

- The operators $\vee$ and $\wedge$ are commutative and associative.
- The operator $\leftrightarrow$ is commutative but not associative.
- The operators $\vee$ and $\wedge$ are idempotent (e.g., $(P \vee P) \equiv P$).
- The operators $\vee$ and $\wedge$ distribute over each other:
  - $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$.
  - $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$.
- ***Absorption laws***:
  - $P \wedge (P \vee Q) \equiv P$.
  - $P \vee (P \wedge Q) \equiv P$.
- ***Double negation***: $\neg\neg P \equiv P$.
- ***de Morgan's laws***:
  - $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.
  - $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$.

We'll say that a set A of wffs ***logically implies*** or ***entails*** a conclusion Q iff, whenever all of the wffs in A are true, Q is also true.

An ***axiom*** is a wff that is asserted *a priori* to be true. Given a set of axioms, rules of inference can be applied to create new wffs, to which the inference rules can then be applied, and so forth. Any statement so derived is called a ***theorem***. Let A be a set of axioms plus zero or more theorems that have already been derived from those axioms. Then a ***proof*** is a finite sequence of applications of inference rules, starting from A.

A proof is a syntactic object. It is just a sequence of applications of rules. We would like, however, for proofs to tell us something about truth. They can do that if we design our inference rules appropriately. We'll say that an inference rule is ***sound*** iff, whenever it is applied to a set A of axioms, any conclusion that it produces is entailed by A (i.e., it must be true whenever A is). An entire proof is sound iff it consists of a sequence of inference steps each of which was constructed using a sound inference rule. A set of inference rules R is ***complete*** iff, given any set A of axioms, all statements that are entailed by A can be proved by applying the rules in R. If we can define a set of inference rules that is both sound and complete then the set of theorems that can be proved from A will exactly correspond to the set of statements that must be true whenever A is.

The truth table we presented above is the basis for the construction of sound and complete inference rules in Boolean logic. Some useful rules are:

- ***Modus ponens***: From the premises $(P \rightarrow Q)$ and P, conclude Q.
- ***Modus tollens***: From the premises $(P \rightarrow Q)$ and $\neg Q$, conclude $\neg P$.
- ***Or introduction***: From the premise P, conclude $(P \vee Q)$.
- ***And introduction***: From the premises P and Q, conclude $(P \wedge Q)$.
- ***And elimination***: From the premise $(P \wedge Q)$, conclude P or conclude Q.

Any two statements of the form P and $\neg P$ form a ***contradiction***.

32.1.2 First-Order Logic

The primitives in Boolean logic are predicates of no arguments (i.e., Boolean constants). It is useful to extend our logical system to allow predicates of one or more arguments and to allow the use of variables. So, for example, we might like to write $P(China)$ or $Q(x, y)$. ***First-order logic***, often called simply ***FOL*** (or sometimes first-order predicate logic, first-order predicate calculus, or FOPC), allows us to do that.

We will use symbols that start with lower-case letters as variables and symbols that start with upper-case letters as constants, predicates, and functions.

An expression that describes an object is a ***term***. So a variable is a term and an *n*-ary function whose arguments are terms is also a term. Note that if *n* is 0, we have a constant.

We define the language of ***well-formed formulas (wffs)*** in first-order logic to be the set of expressions that can be formed according to the following rules:

- If P is an *n*-ary predicate and each of the expressions $x_1, x_2, \ldots, \mathrm{x}_n$ is a term, then an expression of the form $P(x_1, x_2, \ldots, \mathrm{x}_n)$ is a wff. If any variable occurs in such a wff, then that variable is ***free*** (alternatively, it is not bound).
- If P is a wff, then $\neg P$ is a wff.
- If P and Q are wffs, then so are $P \vee Q$, $P \wedge Q$, $P \rightarrow Q$, and $P \leftrightarrow Q$.
- If P is a wff, then (P) is a wff.
- If P is a wff, then $\forall x\ (P)$ and $\exists x\ (P)$ are wffs. Any free instance of x in P is ***bound*** by the quantifier and is then no longer free. $\forall$ is called the universal quantifier and $\exists$ is called the existential quantifier. In the wff $\forall x\ (P)$ or $\exists x\ (P)$, we'll call P the ***scope*** of the quantifier. It is important to note that when an existentially quantified variable *y* occurs inside the scope of a universally quantified variable *x* (as, for example, in statement 4 below), the meaning of the wff is that for every value of *x* there exists some value of *y* but it need not be the same value of *y* for every value of *x*. So, for example, the following wffs are not equivalent:
    - $\forall x\ (\exists y\ (Father\text{-}of(y, x)))$, and
    - $\exists y\ (\forall x\ (Father\text{-}of(y, x)))$.

For convenience, we will extend this syntax slightly. When no confusion will result, we will allow the following additional forms for wffs:

- $\forall x < c\ (P(x))$ is equivalent to $\forall x\ (x < c \rightarrow P(x))$.
- $\forall x \in S\ (P(x))$ is equivalent to $\forall x\ (x \in S \rightarrow P(x))$.
- $\forall x, y, z\ (P(x, y, z))$ is equivalent to $\forall x\ (\forall y\ (\forall z\ (P(x, y, z))))$.
- $\forall x, y, z \in S\ (P(x, y, z))$ is equivalent to $\forall x \in S\ (\forall y \in S\ (\forall z \in S\ (P(x, y, z))))$.

The logical framework that we have just defined is called ***first-order*** because it allows quantification over variables but not over predicates or functions. It is possible to define higher-order logics that do permit such quantification. For example, in a higher-order logic we might be able to say something like $\forall P\ (P(John) \rightarrow P(Carey))$. In other words, anything that is true of *John* is also true of *Carey*. While it is sometimes useful to be able to make statements such as this, the computational and logical properties of higher-order systems make them very hard to use except in some restricted cases.

A wff with no free variables is called a ***sentence*** or a ***statement***. All of the following are sentences:

1. $Bear(Smoky)$.
2. $\forall x\ (Bear(x) \rightarrow Animal(x))$.
3. $\forall x\ (Animal(x) \rightarrow Bear(x))$.
4. $\forall x\ (Animal(x) \rightarrow \exists y\ (Mother\text{-}of(y, x)))$.
5. $\forall x\ ((Animal(x) \wedge \neg Dead(x)) \rightarrow Alive(x))$.

A ***ground instance*** is a sentence that contains no variables. All of the following are ground instances: *Bear*(*Smoky*), *Animal*(*Smoky*), and *Mother-of*(*BigEyes*, *Smoky*). In computational logic systems, it is common to store the ground instances in a different form than the one that is used for other sentences. They may be contained in a table or a database, for example.

Returning to sentences 1-5 above, 1, 2, and 4, and 5 are true in our everyday world (assuming the obvious referent for the constant Smoky and the obvious meanings of the predicates *Bear*, *Animal*, and *Mother-of*). On the other hand, 3 is not true.

As these examples show, determining whether or not a sentence is true requires appeal to the meanings of the constants, functions, and predicates that it contains. An ***interpretation*** for a sentence w is a pair (D, I). D is a universe of objects. I assigns meaning to the symbols of w: it assigns values, drawn from D, to the constants in w and it assigns functions and predicates (whose domains and ranges are subsets of D) to the function and predicate symbols of w. A ***model*** of a sentence w is an interpretation that makes w true. For example, let w be the sentence, $\forall x\ (\exists y\ (y < x))$. The integers (along with the usual meaning of <) are a model of w since, for any integer, there exists some smaller integer. The positive integers, on the other hand, are an interpretation for w but not a model of it. The sentence w is false for the positive integers since there is no positive integer that is smaller than 1.

A sentence w is ***valid*** iff it is true in all interpretations. In other words, w is valid iff it is true regardless of what the constant, function, and predicate symbols "mean". A sentence w is ***satisfiable*** iff there exists *some* interpretation in which w is true. A sentence is ***unsatisfiable*** iff it is not satisfiable (in other words, there exists no interpretation in which it is true). Any sentence w is valid iff $\neg w$ is unsatisfiable.

Example 32.2 Valid, Satisfiable, and Unsatisfiable Wffs

Let w_1 be the wff:

$$\forall x\ ((P(x) \wedge Q(Smoky)) \rightarrow P(x)).$$

The wff w_1 is valid because it is true regardless of what the predicates P and Q are or what object *Smoky* refers to. It is also satisfiable since it is true in at least one interpretation.

Let w_2 be the wff:

$$\neg(\forall x\ (P(x) \vee \neg(P(x))).$$

The wff w_2 is not valid. It is also unsatisfiable since it is false in all interpretations, which follows from the fact that $\neg w_2$ is valid.

Finally, let w_3 be the wff:

$$\forall x\ (P(x, x)).$$

The wff w_3 is not valid but it is satisfiable. Suppose that the universe is the integers and P is the predicate *LessThanOrEqualTo*. Then P is true for all values of x. But, again with the integers as the universe, suppose that P

is the predicate *LessThan*. Now *P* is false for all values of *x*. Finally, let the universe be the set of all people and let *P* be the predicate *HasConfidenceInTheAbilityOf*. Now *P* is true of some values of *x* (i.e., of those people who have self confidence) and false of others.

A set *A* of axioms ***logically implies*** or ***entails*** a conclusion *c* iff, in every interpretation in which *A* is true (i.e., in every model of *A*), and for all assignments of values to the free variables of *c*, *c* must be true.

As in Boolean logic, a proof in first-order logic starts with a set *A* of axioms and theorems that have already been proved from those axioms. Rules of inference are then applied, creating new statements. Any statement derived in this way is called a ***theorem***. A ***proof*** is a finite sequence of applications of inference rules, starting from the axioms and given theorems.

As in Boolean logic, we will say that an inference rule is ***sound*** iff, whenever it is applied to a set *A* of statements (axioms and given theorems), any conclusion that it produces is entailed by *A* (i.e., it must be true whenever *A* is). A set of inference rules *R* is ***complete*** iff, given any set *A* of statements, all statements that are entailed by *A* can be proved by applying the rules in *R*. As in Boolean logic, we seek a set of inference rules that is both sound and complete.

> Resolution is a single inference rule that is used as the basis for many automatic theorem proving and reasoning programs. It is sound and refutation-complete. By the latter, we mean that if $\neg ST$ is inconsistent with the axioms and if both the axioms and$\neg ST$ have been converted to a restricted syntax called clause form, resolution will find the inconsistency and thus prove *ST*. 𝔅 620.

For Boolean logic, truth tables provide a basis for defining a set of sound and complete inference rules. It is less obvious that such a set exists for first-order logic. But it does, as was first shown by Kurt Gödel in his Completeness Theorem [Gödel 1929]. More specifically, Gödel showed that there exists some set of inference rules *R* such that, given any set of axioms *A* and a sentence *c*, there is a proof of *c*, starting with *A* and applying the rules in *R*, iff *c* is entailed by *A*. Note that all that we are claiming here is that, if there is a proof, there is a procedure for finding it. We are not claiming that there exists a procedure that decides whether or not a proof exists. In fact, as we show in Section 22.4.2, for first-order logic no such decision procedure can exist.

All of the inference rules that we have and will present are sound. The individual inference rules that we have so far considered are not, however, complete. For example, modus ponens is incomplete. But a complete procedure can be constructed by including all of the rules we listed above for Boolean logic, plus new ones, including, among others:

- ***Quantifier exchange***:
  - From $\neg\exists x\,(P)$, conclude $\forall x\,(\neg P)$.
  - From $\forall x\,(\neg P)$, conclude $\neg\exists x\,(P)$.
  - From $\neg\forall x\,(P)$, conclude $\exists x\,(\neg P)$.
  - From $\exists x\,(\neg P)$, conclude $\neg\forall x\,(P)$.
- ***Universal instantiation***: For any constant *C*, from $\forall x\,(P(x))$, conclude $P(C)$.
- ***Existential generalization***: For any constant *C*, from $P(C)$ conclude $\exists x\,(P(x))$.

Example 32.3 A Simple Proof

Assume the following three axioms:

[1] $\forall x\,(P(x) \wedge Q(x) \rightarrow R(x))$.
[2] $P(X_1)$.
[3] $Q(X_1)$.

We prove $R(X_1)$ as follows:

[4] $P(X_1) \wedge Q(X_1) \rightarrow R(X_1)$. (Universal instantiation, [1].)
[5] $P(X_1) \wedge Q(X_1)$. (And introduction, [2], [3].)
[6] $R(X_1)$. (Modus ponens, [5], [4].)

A first-order ***theory*** is a set of axioms and the set of all theorems that can be proved, using a set of sound and complete inference rules, from those axioms. A theory is ***logically complete*** iff, for every sentence *P*, either *P* or $\neg P$ is a theorem. A theory is ***consistent*** iff there is no sentence *P* such that both *P* and $\neg P$ are theorems. If, on the other hand, there is such a sentence, then the theory contains a ***contradiction*** and is ***inconsistent***.

We are often interested in the relationship between a theory and some set of facts that are true in some view we may have of the world (for example the facts of arithmetic or the facts a robot needs in order to move around). Let *W* be a world plus an interpretation (that maps logical objects to objects in the world). Now we can say that a theory is ***sound*** with respect to *W* iff every theorem (in the theory) corresponds to a fact that is true (in *W*). We say that a theory is ***complete*** with respect to *W* iff every fact that is true (in *W*) corresponds to a theorem (in the theory). We will assume that any first-order logic statement in the language of *W* is either true or false in the world that *W* describes. So, if a theory is complete with respect to *W* it must be the case that, for any sentence *P*, either *P* corresponds to a sentence that is true in *W*, in which case it is a theorem, or *P* corresponds to a sentence that is false in *W*, in which case $\neg P$ is a theorem. So any theory that is complete with respect to an interpretation and a set of facts is also logically complete.

By the way, while the language of first-order logic has the property that every statement is either true or false in any world, not all languages share that property. For example, English doesn't. Consider the English sentence, "The king of France has red hair." Is it true or false (in the world as we know it, given the standard meanings of the words)? The answer is neither. It carries the (false) presupposition that there is a king of France and then makes a claim about that individual. This problem disappears, however, when we convert the English sentence into a related sentence in first order logic. We might try:

- $\exists$x (*King-of*(*x*, *France*) $\wedge$ *Haircolor-of*(*x*, *Red*)): This sentence is false in the world.
- $\forall$x (*King-of*(*x*, *France*) $\rightarrow$ *Haircolor-of*(*x*, *Red*)): This sentence is true in the world (trivially, since there are no values of *x* for which *King-of*(*x*, *France*) is true).

There are interesting first-order theories that are both consistent and complete with respect to particular interpretations of interest. One example is Presburger arithmetic, in which the universe is the natural numbers and there is a single function, *plus*, whose properties are axiomatized. There are other theories that are incomplete because we have not yet added enough axioms. But it might be possible, eventually, to find a set of axioms that does the job.

However, many interesting and powerful theories are not both consistent and complete and they will never become so. For example, Gödel's Incompleteness Theorem [Gödel 1931] 💻, one of the most important results in modern mathematics, shows that *any* theory that is derived from a decidable (a notion that we explain in Chapter 17) set of axioms and that characterizes the standard behavior of the constants 0 and 1, plus the functions *plus* and *times* on the natural numbers, cannot be both consistent and complete. In other words, if any such theory is consistent (and it is generally assumed that the standard theory of arithmetic is), then there must be some statements that are true (in arithmetic) but not provable (in the theory). While it is of course possible to add new axioms and thus make more statements provable, there will always remain some true but unprovable statements unless either the set of axioms becomes inconsistent or it becomes infinite and undecidable. In the latter case, the fact that a proof exists is not very useful since it has become impossible to tell whether or not a statement is an axiom and thus can be used in a proof.

Do not be confused by the fact that there exists both a Completeness Theorem and an Incompleteness Theorem. The terminology is unfortunate since it is based on two different notions of completeness. The Completeness Theorem states a fact about the framework of first-order logic itself. It says that there exists a set of inference rules (and, in fact, more than one such set happens to exist) such that, given any set *A* of axioms, the theorems that are provable from *A* are exactly the set of sentences that are entailed by *A*. The Incompleteness Theorem states a fact about theories that can be built within any logical framework. It says that there exist theories (the standard one about arithmetic with *plus* and *times* being one example) that, assuming consistency, are incomplete in the sense that there are sentences that are true in the world but that are not theorems. Such theories are also logically incomplete: there exist sentences *P* such that neither *P* nor $\neg P$ is a theorem.

32.2 Sets

Most of the structures that we will consider are based on the fundamental notion of a set.

32.2.1 What is a Set?

A ***set*** is simply a collection of objects. The objects (which we call the ***elements*** or ***members*** of the set) can be anything: numbers, people, strings, fruits, whatever. For example, all of the following are sets:

- S_1 = {13, 11, 8, 23}.
- S_2 = {8, 23, 11, 13}.
- S_3 = {8, 8, 23, 23, 11, 11, 13, 13}.
- S_4 = {apple, pear, banana, grape}.
- S_5 = {January, February, March, April, May, June, July, August, September, October, November, December}.
- $S_6 = \{x : x \in S_5$ and x has 31 days}.
- S_7 = {January, March, May, July, August, October, December}.
- $\mathbb{N}$ = the nonnegative integers (also called ***the natural numbers***).
- $S_8 = \{i : \exists x \in \mathbb{N}\ (i = 2x)\}$.
- $S_9 = \{0, 2, 4, 6, 8, \ldots\}$.
- S_{10} = the even natural numbers.
- S_{11} = the syntactically valid C programs.
- $S_{12} = \{x : x \in S_{11}$ and x never gets into an infinite loop}.
- S_{13} = finite length strings of `a`'s and `b`'s}.
- Z = the integers (… -3, -2, -1, 0, 1, 2, 3, …).

In the definitions of S_6, S_8, and S_{12}, we have used the colon notation. Read it as "such that". So, for example, read the definition of S_6 as, "the set of all values x such that x is an element of S_5 and x has 31 days". We have used the standard symbol $\in$ for "element of". We will also use $\notin$ for "not an element of". So, for example, $17 \notin S_1$ is true.

Remember that a set is simply a collection of elements. So if two sets contain precisely the same elements (regardless of the way we actually defined the sets), then they are identical. Thus S_6 and S_7 are the same set, as are S_8, S_9, and S_{10}.

Since a set is defined only by what elements it contains, the order in which we list its elements does not matter. Thus S_1 and S_2 are the same set. Also note that a given element is either in a set or it isn't. Duplicates do not matter. So sets S_1, S_2, and S_3 are equal.

One useful technique for describing a set S that is a subset of an existing set D is to define a function (we'll define formally what we mean by a function in Section 32.4) that can be used to determine whether or not a given element is in S. Such a function is called a ***characteristic function***. Formally, a function f with domain D is a characteristic function for a set S iff $f(x)$ = *True* if x is an element of S and *False* otherwise. For example, we used this technique to define set S_6.

We can use programs to define sets. There are two ways to use a program to define a set S:

- Write a program that generates the elements of S. We call the output of such a program an ***enumeration*** of S.
- Write a program that ***decides*** S by implementing the characteristic function of S. Such a program returns *True* if run on some element that is in S and *False* if run on an element that is not in S.

It seems natural to ask, given some set S, "What is the size of S?" or "How many elements does S contain?" We will use the term ***cardinality*** to describe the way we answer such questions. So we'll reply that the cardinality of S, written $|S|$, is n, for some appropriate value of n. For example, $|\{2, 7, 11\}| = 3$. In simple cases, determining the cardinality of a set is straightforward. In other cases, it is more complicated. For our purposes, however, we can get by with three different kinds of answers:

- a natural number (if S is finite),

- “countably infinite” (if S has the same number of elements as there are integers), or
- “uncountably infinite” (if S has more elements than there are integers).

We will formalize these ideas in Section 32.6.8.

The smallest set is the unique set that contains no elements. It is called the ***empty set***, and is written $\varnothing$ or $\{\}$. The cardinality of the empty set, written $|\varnothing|$, is 0.

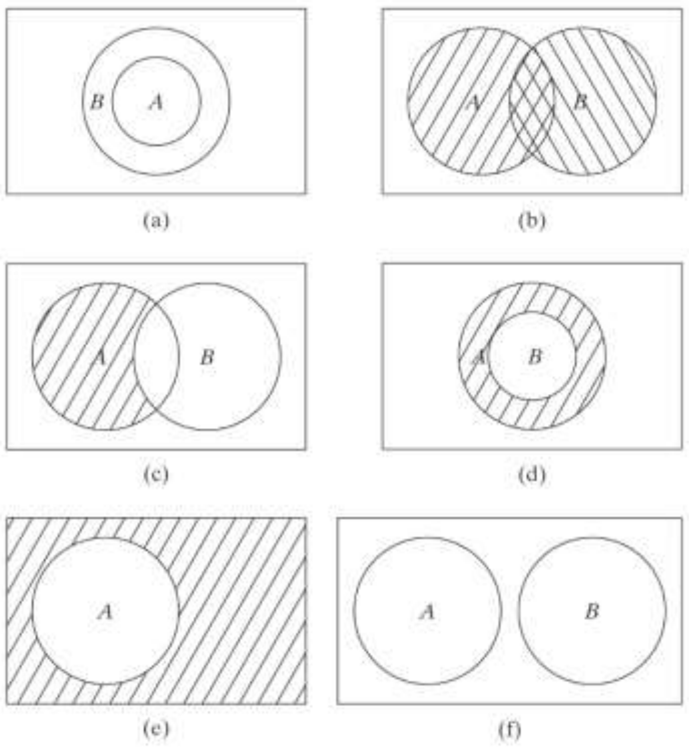
When we are working with sets, it is very important to keep in mind the difference between a set and the elements of a set. Given a set that contains more than one element, this distinction is usually obvious. It is clear that $\{1, 2\}$ is distinct from either the number 1 or the number 2. It sometimes becomes a bit less obvious, though, with ***singleton sets*** (sets that contain only a single element). But it is equally true for them. So, for example, $\{1\}$ is distinct from the number 1. As another example, consider $\{\varnothing\}$. This is a set that contains one element. That element is in turn a set that contains no elements (i.e., the empty set). $\{\{1, 2, 3\}\}$ is also a set that contains one element.

32.2.2 Relating Sets to Each Other

We say that A is a ***subset*** of B (which we write as $A \subseteq B$) iff every element of A is also an element of B. Formally, $A \subseteq B$ iff $\forall x \in A\ (x \in B)$.

The symbol we use for subset ($\subseteq$) looks somewhat like $\leq$. This is no accident. If $A \subseteq B$, then there is a sense in which the set A is “less than or equal to” the set B, since all the elements of A must be in B, but there may be elements of B that are not in A.

Given this definition, notice that every set is a subset of itself. This fact turns out to offer a useful way to prove that two sets A and B are equal: First prove that A is a subset of B. Then prove that B is a subset of A. We will have more to say about this later in Section 32.6.7.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 32.1 Venn diagrams that illustrate relations and functions on sets

We say that A is ***proper subset*** of B (written $A \subset B$) iff $A \subseteq B$ and $A \neq B$. The ***Venn diagram*** shown in Figure 32.1 (a) illustrates the proper subset relationship between A and B. Notice that the empty set is a subset of every set (since, trivially, every element of $\varnothing$, all none of them, is also an element of every other set). And the empty set is a *proper* subset of every set other than itself.

It is useful to define some basic operations that can be performed on sets.

The ***union*** of two sets A and B (written $A \cup B$) contains all elements that are contained in A or B (or both). Formally, $A \cup B = \{x: (x \in A) \vee (x \in B)\}$. We can easily visualize union using a Venn diagram, as shown in Figure 32.1 (b). The union of sets A and B is the entire hatched area in the diagram.

The ***intersection*** of two sets A and B (written $A \cap B$) contains all elements that are contained in both A and B. Formally, $A \cap B = \{x: (x \in A) \wedge (x \in B)\}$. In the Venn diagram shown in Figure 32.1 (b), the intersection of A and B is the double hatched area in the middle.

The ***difference*** of two sets A and B (written A - B or A/B) contains all elements that are contained in A but not in B. Formally, $A/B = \{x: (x \in A) \wedge (x \notin B)\}$. In both of the Venn diagrams shown in Figure 32.1 (c) and (d), the hatched region represents A/B.

The ***complement*** of a set A with respect to a specific universe U (written as $\neg A$) contains exactly those elements of U that are not contained in A (i.e., $\neg A = U - A$). Formally, $\neg A = \{x: (x \in U) \wedge (x \notin A)\}$. For example, if U is the set of residents of Austin and A is the set of Austin residents who like barbeque, then $\neg A$ is the set of Austin residents who don't like barbeque. The complement of A is shown as the hatched region of the Venn diagram shown in Figure 32.1 (e).

Two sets are ***disjoint*** iff they have no elements in common (i.e., their intersection is empty). Formally, A and B are disjoint iff $A \cap B = \varnothing$. In the Venn diagram shown in Figure 32.1 (f), A and B are disjoint:

Given a set A, we can consider the set of all subsets of A. We call this set the ***power set*** of A, and we write it $\mathcal{P}(A)$. For example, let $A = \{1, 2, 3\}$. Then:

$$\mathcal{P}(A) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

The power set of A is interesting because, if we're working with the elements of A, we may well care about all the ways in which we can combine those elements.

Now for one final property of sets. Again consider the set A above. But this time, rather than looking for all possible subsets, let's just look for a single way to carve A up into subsets such that each element of A is in precisely one subset. For example, we might choose any of the following sets of subsets:

$$\{\{1\}, \{2, 3\}\} \quad \text{or} \quad \{\{1, 3\}, \{2\}\} \quad \text{or} \quad \{\{1, 2, 3\}\}.$$

We call any such set of subsets a partition of A. Partitions are very useful. For example, suppose we have a set S of students in a school. We need for every student to be assigned to precisely one lunch period. Thus we must construct a partition of S: a set of subsets, one for each lunch period, such that each student is in precisely one subset. More formally, we say that $\Pi \subseteq \mathcal{P}(A)$ is a ***partition*** of a set A iff:

- no element of Π is empty,
- all pairs of elements of Π are disjoint (alternatively, each element of A is in at most one element of Π), and
- the union of all the elements of Π equals A (alternatively, each element of A is in some element of Π and no element not in A is in any element of Π).

This notion of partitioning a set is fundamental to programming. Every time we analyze the set of possible inputs to a program and consider the various cases that must be dealt with, we are forming a partition of the set of inputs: each input must fall through precisely one path in your program. So it should come as no surprise that, as we build formal models of computational devices, we will rely heavily on the idea of a partition of a set of inputs as an analytical technique.

32.3 Relations

In the last section, we introduced some simple relations that can hold between sets (subset and proper subset) and we defined some operations (functions) on sets (union, intersection, difference, and complement). But we haven't yet defined formally what we mean by a relation or a function. We will do that now. (By the way, the reason we introduced relations and functions on sets in the last section is that we are going to use sets as the basis for our formal definitions of relations and functions and we will need the simple operations we just described as part of our definitions.)

32.3.1 What is a Relation?

An ***ordered pair*** is a sequence of two objects. Given any two objects, x and y, there are two ordered pairs that can be formed. We write them as (x, y) and (y, x). As the name implies, in an ordered pair (as opposed to in a set), order matters (unless x and y happen to be equal).

The ***Cartesian product*** of two sets A and B (written $A \times B$) is the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$. For example, let A be a set of people: {Dave, Sara, Billy}, and let B be a set of desserts: {cake, pie, ice cream}. Then:

$A \times B$ = { (Dave, cake), (Dave, pie), (Dave, ice cream),
(Sara, cake), (Sara, pie), (Sara, ice cream),
(Billy, cake), (Billy, pie), (Billy, ice cream)}.

As you can see from this example, the Cartesian product of two sets contains elements that represent all the ways of pairing some element from the first set with some element from the second. Note that $A \times B$ is not the same as $B \times A$. In our example:

$B \times A$ = { (cake, Dave), (pie, Dave), (ice cream, Dave),
(cake, Sara), (pie, Sara), (ice cream, Sara),
(cake, Billy), (pie, Billy), (ice cream, Billy)}.

If A and B are finite, then the cardinality of their Cartesian product is given by:

$$|A \times B| = |A|\cdot|B|.$$

A ***binary relation*** over two sets A and B is a subset of $A \times B$. For example, let's consider the problem of choosing dessert. We could define a relation that tells us, for each person, what desserts he or she likes. We might write the *Dessert* relation, for example as:

Dessert = {(Dave, cake), (Dave, ice cream), (Sara, pie), (Sara, ice cream)}.

In other words, Dave likes cake and ice cream, Sara likes pie and ice cream, and Billy seems not to like sinful treats.

Not all relations are binary. We define an ***n-ary relation*** over sets $A_1, A_2, \ldots A_n$ to be a subset of $A_1 \times A_2 \times \ldots \times A_n$. The n sets may be different, or they may be the same. For example, let A be a set of people:

A = {Dave, Sara, Billy, Beth, Mark, Cathy, Pete}.

Now suppose that Sara and Dave are the parents of Billy, Beth and Mark are the parents of Cathy, and Billy and Cathy are the parents of Pete. Then we could define a 3-ary (or *ternary*) relation *Child-of*, where the first element of each 3-tuple is the mother, the second is the father, and the third is the child. So we would have the following subset of $A \times A \times A$:

{(Sara, Dave, Billy), (Beth, Mark, Cathy), (Cathy, Billy, Pete)}.

Notice a couple of important properties of relations as we have defined them. First, a relation may be equal to the empty set. For example, if Dave, Sue, and Billy all hate dessert, then the *Dessert* relation would be {} or $\varnothing$.

Second, there are no constraints on how many times a particular element may occur in a relation. In the *Dessert* example, Dave occurs twice, Sue occurs twice, Billy doesn't occur at all, cake occurs once, pie occurs once, and ice cream occurs twice. Given an *n*-ary relation *R*, we'll use the notation $R(x_1, \ldots, x_{n-1})$ for the set that contains those elements with the property that $(x_1, \ldots, x_{n-1}, x_n) \in R$. So, for example *Dessert*(Dave) = {cake, ice cream}.

An *n*-ary relation *R* is a subset of the cross product of *n* sets. The sets may all be different, or some of them may be the same. In the specific case in which all the sets are the same, we will say that *R* is a relation on the set *A*.

Binary relations are particularly useful and are often written in the form $x_1\ R\ x_2$. Common binary relations include = (equality, defined on many domains), < (defined on numbers and some other domains), and ≤ (also defined on numbers and some other domains). For example, the relation < on the integers contains an infinite number of elements drawn from the Cartesian product of the set of integers with itself. For instance, $2 < 7$.

The ***inverse*** of a binary relation *R*, written R^{-1}, is simply the set of ordered pairs in *R* with the elements of each pair reversed. Formally, if $R \subseteq A \times B$, then $R^{-1} \subseteq B \times A = \{(b, a): (a, b) \in R\}$. If a relation is a way of associating with each element of *A* with a corresponding element of *B*, then think of its inverse as a way of associating with elements of *B* their corresponding elements in *A*. Every relation has an inverse. For example, the inverse of < (in the usual sense, defined on numbers) is ≥.

If we have two or more binary relations, we may be able combine them via an operation we'll call composition. For example, if we knew the number of fat grams in a serving of each kind of dessert, we could ask for the number of fat grams in a particular person's dessert choices. To compute this, we first use the *Dessert* relation to find all the desserts each person likes. Next we get the bad news from the *Fatgrams* relation, which probably looks something like this:

{(cake, 30), (pie, 25), (ice cream, 15)}.

Finally, we see that the composed relation that relates people to fat grams is {(Dave, 30), (Dave, 15), (Sara, 25), (Sara, 15)}. Of course, this only worked because, when we applied the first relation, we got back desserts. Then our second relation has desserts as its first component. We couldn't have composed *Dessert* with *Less than*, for example.

Formally, we say that the ***composition*** of two relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, written $R_2 \circ R_1$, is:

$$R_2 \circ R_1 = \{(a, c) : \exists b\ ((a, b) \in R_1 \wedge ((b, c) \in R_2)\}.$$

Note that this definition tells us that, to compute $R_2 \circ R_1$, we first apply R_1, then R_2. In other words we go right to left. Some definitions go the other way. Obviously we can define it either way, but it is important to be consistent. Using the notation we have just defined, we can represent the people to fat grams composition described above as *Fatgrams* ° *Dessert*.

32.3.2 Representing Binary Relations

Binary relations are particularly important. If we're going to work with them, and, in particular, if we are going to compute with them, we need some way to represent them. We have several choices. To represent some binary relation *R*, we could:

1) List the elements of *R*. For example, consider the *Mother-of* relation in a family in which Doreen is the mother of Ann, Ann is the mother of Catherine, and Catherine is the mother of Allison. Then we can write:

   Mother-of = {(Doreen, Ann), (Ann, Catherine), (Catherine, Allison)}.

Clearly, this approach only works for finite relations.

2) Encode *R* as a computational procedure. As with any set, there are at least two ways in which a computational procedure can define *R*. It may:
   - enumerate the elements of *R*, or
   - implement the characteristic function for *R* by returning *True* when given a pair that is in *R* and *False* when given anything else.

3) Encode *R* as an adjacency matrix. Assuming a finite relation $R \subseteq A \times B$, we can build an adjacency matrix to represent *R* as follows:
   3.1. Construct a Boolean matrix *M* (i.e., a matrix all of whose values are *True* or *False*) with $|A|$ rows and $|B|$ columns.
   3.2. Label each row for one element of *A* and each column for one element of *B*.
   3.3. For each element (*p*, *q*) of *R*, set *M*[*p*, *q*] to *True*. Set all other elements of *M* to *False*.

   If we let 1 represent *True* and blank represent *False*, the adjacency matrix shown in Table 32.2 represents the relation *Mother-of* defined above.

	Doreen	Ann	Catherine	Allison
Doreen		1		
Ann			1	
Catherine				1
Allison				

Table 32.2 Representing a relation as an adjacency matrix

4) Encode *R* as a directed graph. If *R* is a relation on the set *A*, we can build a directed graph to represent *R* as follows:
   4.1. Construct a set of vertices (often called nodes), one for each element of *A* that appears in any element of *R*.
   4.2. For each ordered pair in *R*, draw an edge from the first element of the pair to the second.

   The directed graph shown in Figure 32.2 (a) represents our example relation *Mother-of* defined above. If there are two elements *x* and *y*, and both (*x*, *y*) and (*y*, *x*) are in *R*, we will usually draw the graph as shown in Figure 32.2 (b). The directed graph technique can also be used if *R* is a relation over two different sets *A* and *B*. But, in this case, we must construct vertices for elements of *A* and for elements of *B*. So, for example, we could represent the *Fatgrams* relation as shown in Figure 32.2 (c).

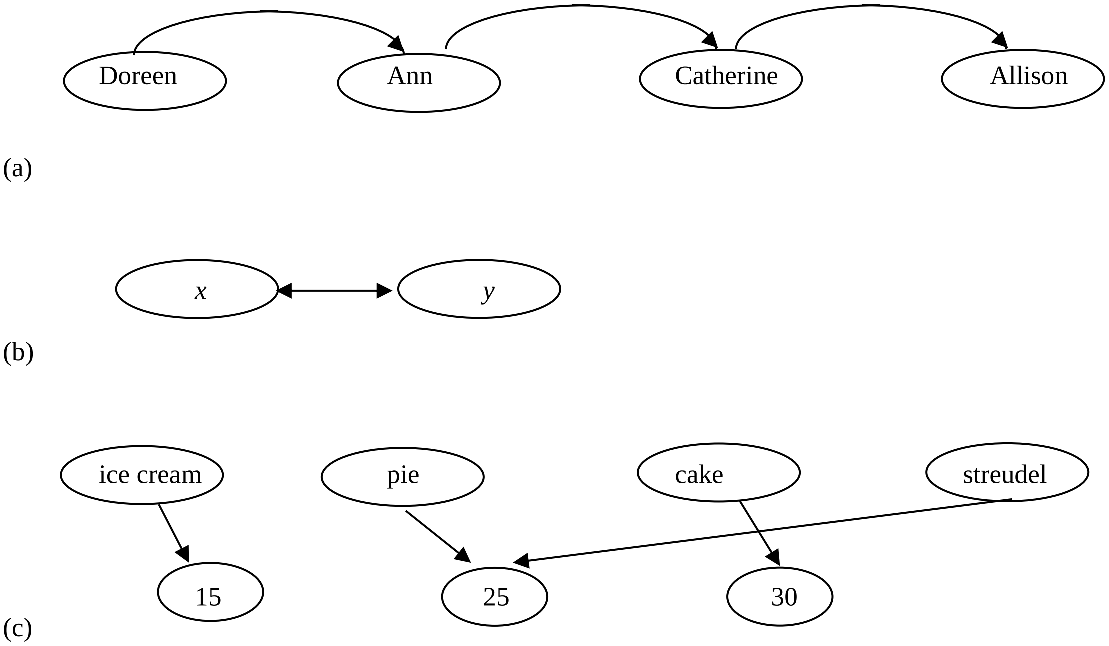
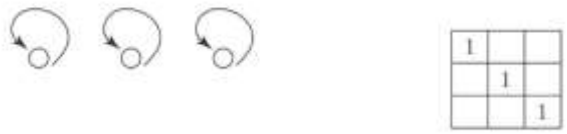


Figure 32.2 Representing relations as graphs

32.3.3 Properties of Binary Relations on Sets

Many useful binary relations have some kind of structure. For example, it might be the case that every element of the underlying set is related to itself. Or it might happen that if x is related to y, then y must necessarily be related to x. There is one special kind of relation, called an equivalence relation that is particularly useful. But before we can define it, we need first to define each of the individual properties that equivalence relations possess.

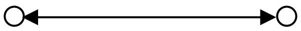
A relation $R \subseteq A \times A$ is ***reflexive*** iff, $\forall x \in A\ ((x, x) \in R)$. For example, consider the relation *Address* defined as "lives at same address as". We will make the simplifying assumption that everyone has only one address. *Address* is a relation over a set of people. Clearly every person lives at the same address as him or herself, so *Address* is reflexive. So is the ≤ relation on the integers. For every integer x, $x \leq x$. But the < relation is not reflexive: in fact, for no integer x, is $x < x$. Both the directed graph and the matrix representations make it easy to tell if a relation is reflexive. In the graph representation, every vertex will have, at a minimum, an edge looping back to itself. In the adjacency matrix representation, there will be ones all along the major diagonal, and possibly elsewhere as well. Figure 32.3 illustrates both cases.



1		
	1	
		1

Figure 32.3 Representing a reflexive relation

A relation $R \subseteq A \times A$ is ***symmetric*** iff $\forall x, y\ ((x, y) \in R \rightarrow (y, x) \in R)$. The *Address* relation we described above is symmetric. If Joanna lives with Ann, then Ann lives with Joanna. The ≤ relation is not symmetric (since, for example, $2 \leq 3$, but it is not true that $3 \leq 2$). The graph representation of a symmetric relation has the property that, between any two vertices, either there is an arrow going in both directions or there is no arrow going in either direction. So we get graphs with components that look like the one shown in Figure 32.4 (a). If we choose the matrix representation, we will end up with a ***symmetric matrix*** (i.e., if you flip it on its major diagonal, you'll get the same matrix back again). In other words, if we have a matrix with 1's as shown in the matrix of Figure 32.4 (b), then there must also be 1's in all the squares marked with an * in that matrix.



(a)

(b)

	*	*		
1				1
1				
	*			

Figure 32.4 Representing a symmetric relation

A relation $R \subseteq A \times A$ is ***antisymmetric*** iff $\forall x, y\ (((x, y) \in R \wedge x \neq y) \rightarrow (y, x) \notin R)$. The *Mother-of* relation we described above is antisymmetric: if Ann is the mother of Catherine, then one thing we know for sure is that Catherine is not also the mother of Ann. Our *Address* relation is clearly not antisymmetric. There are, however, relations that are neither symmetric nor antisymmetric. One example is the *Likes* relation on the set of people: if Joe likes Bob, then it is possible that Bob likes Joe; it is also possible that he doesn't. Note that antisymmetric is not the same as not symmetric. The relation $\varnothing$ is both symmetric and antisymmetric.

A relation $R \subseteq A \times A$ is ***transitive*** iff $\forall x, y, z\ (((x, y) \in R \wedge (y, z) \in R) \rightarrow (x, z) \in R)$. A simple example of a transitive relation is <. *Address* is another one: if Bill lives with Stacy and Stacy lives with Lee, then Bill lives with Lee. *Mother-of* is not transitive. But if we change it slightly to *Ancestor-of*, then we get a transitive relation. If Doreen is an ancestor of Ann and Ann is an ancestor of Catherine, then Doreen is an ancestor of Catherine.

The three properties of reflexivity, symmetry, and transitivity are almost logically independent of each other. We can find simple, potentially useful relations that possess seven of the eight possible combinations of these properties. We show them in Table 32.3 (which we'll extend to include antisymmetry in Exercise 32.10):

Properties	***Domain***	***Example***
None	People	*Mother-of*
Just reflexive	People who can see	*Would-recognize-picture-of*
Just symmetric	People	*Has-ever-been-married-to*
Just transitive	People	*Ancestor-of*
Just reflexive and symmetric	People	*Hangs-out-with* (assuming we can say one hangs out with oneself)
Just reflexive and transitive	Numbers	$\leq$
Just symmetric and transitive	Anything	$\varnothing$
All three	Numbers	=
″	People	*Address*

Table 32.3 Important properties of relations

To see why we can't find a nontrivial (i.e., different from $\varnothing$) example of a relation that is symmetric and transitive but not reflexive, consider a simple relation R on {1, 2, 3, 4}. As soon as R contains a single element that relates two unequal objects (e.g., (1, 2)), it must, for symmetry, contain the matching element (2, 1). So now we have $R' = \{(1, 2), (2, 1)\}$. To make R' transitive, we must add (1, 1) and (2, 2). Call the resulting relation R''. Then R'' would be reflexive, except that neither 3 nor 4 is related to itself. In fact, they are related to nothing. We cannot find an example of a relation *R* that is symmetric and transitive but not reflexive if we insist that all elements of the domain be related under *R* to something.

32.3.4 Equivalence Relations

Although all but one of the combinations we just described are reasonable, one combination is of such great importance that we give it a special name. Given a domain *A*, a relation $R \subseteq A \times A$ is an ***equivalence relation*** iff it is reflexive, symmetric and transitive. *Equality* (for numbers, strings, or whatever) is an equivalence relation (no coincidence, given the name). So is our *Address* (lives at same address) relation.

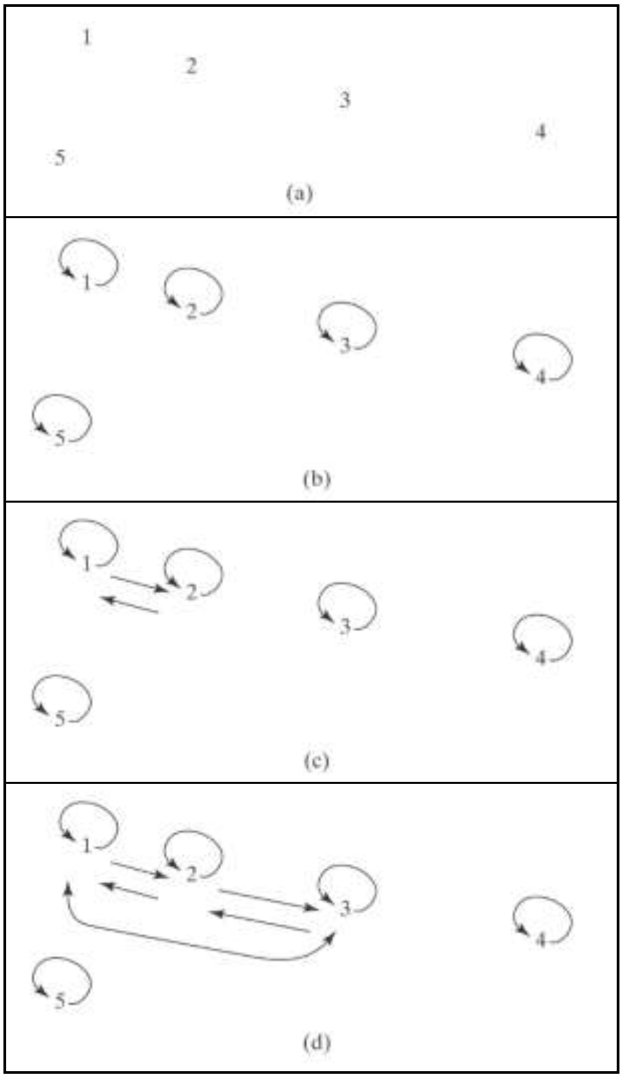


Figure 32.5 Building an equivalence relation

Equality is a very special sort of equivalence relation because it relates an object only to itself. It doesn't help us much to carve up a large set into useful subsets. But other equivalence relations may serve as very useful ways to carve up a set. To see why, consider a set *A*, with five elements, which we can draw as a set of vertices as shown in Figure 32.5 (a). Having done that, we can build an equivalence relation *R* on *A*. First, we'll relate each vertex to itself. That will make the relation reflexive. Once that is done, we'll have the relation shown in Figure 32.5 (b).

Now let's add one additional element, (1, 2), to *R*. As soon as we do that, we must also add (2, 1), since *R* must be symmetric. At this point, we have the relation shown in Figure 32.5 (c). Suppose we now add (2, 3). We must also add (3, 2) to maintain symmetry. In addition, because we have (1, 2) and (2, 3), we must add (1, 3) for transitivity. And then we need (3, 1) to restore symmetry. That gives us the relation shown in Figure 32.5 (d).

Notice what happened here. As soon as we related 3 to 2, we were also forced to relate 3 to 1. If we hadn't, we would no longer have had an equivalence relation. See what happens now if you add (3, 4) to *R*.

What we've seen in this example is that an equivalence relation *R* on a set *S* carves *S* up into a set of clusters or islands, which we'll call ***equivalence classes***. This set of equivalence classes has the following key property:

$$\forall s, t \in S\ ((s \in class_i \wedge (s, t) \in R) \rightarrow t \in class_i).$$

In other words, all elements of *S* that are related under *R* are in the same equivalence class. To describe equivalence classes, we'll use the notation [*x*] to mean the equivalence class to which *x* belongs. Or we may just write [description], where description is some clear property shared by all the members of the class. Notice that, in general, there may be lots of different ways to describe the same equivalence class. In our example, for instance, [1], [2], and [3] are different names for the same equivalence class, which includes the elements 1, 2, and 3. In this example, there are two other equivalence classes as well: [4] and [5].

Recall that Π is a partition of a set *A* iff (a) no element of Π is empty; (b) all members of Π are disjoint; and (c) the union of all the elements of Π equals *A*. If *R* is an equivalence relation on a nonempty set *A*, then the set of equivalence classes of *R* constitutes a partition of *A*. In other words, if we want to take a set *A* and carve it up into a set of subsets, an equivalence relation is a good way to do it.

Example 32.4 Some Equivalence Relations

All of the following relations are equivalence relations:

- The *Address* relation carves up a set of people into subsets of people who live together.

- Let *A* be the set of all strings of letters. Let *Samelength* $\subseteq A \times A$ relate strings whose lengths are the same. *Samelength* is an equivalence relation that carves up the universe of all strings into a collection of subsets, one for each natural number (i.e., strings of length 0, strings of length 1, etc.).

- Let *Z* be the set of integers. Let $\equiv_3 \subseteq Z \times Z$ relate integers that are equivalent modulo 3. In other words, they have the same remainder when divided by 3. $\equiv_3$ is an equivalence relation with three equivalence classes, [0], [1], and [2]. [0] includes 0, 3, 6, etc. [1] includes 1, 4, 7, etc. And [2] includes 2, 5, 8, etc. We will use the notation $\equiv_n$ for positive integer values of *n* to mean equivalent modulo *n*.

- Let *CP* be the set of C programs, each of which accepts an input of variable length. We will call the length of any specific input *n*. Let *Samecomplexity* $\subseteq CP \times CP$ relate two programs iff their running-time complexity is the same. More, precisely, let *Runningtime*(*c, n*) be the maximum time required for program *c* to run on an input of length *n*. Then $(c_1, c_2) \in$ *Samecomplexity* iff there exist natural numbers m_1, m_2, *k* such that:

$$\forall n{>}k\ (Runningtime(c_1, n) \leq m_1 \cdot Runningtime(c_2, n) \wedge Runningtime(c_2, n) \leq m_2 \cdot Runningtime(c_1, n)).$$

Samecomplexity is an equivalence relation. We will have a lot more to say about relations like it in Part V.

Not every relation that connects "similar" things is an equivalence relation. For example, define *Similarcost*(*x, y*) to hold if the price of *x* is within $1 of the price of *y*. Suppose X_1 costs $10, X_2 costs $10.50, and X_3 costs $11.25. Then *Similarcost*(X_1, X_2) and *Similarcost*(X_2, X_3), but not *Similarcost*(X_1, X_3). *Similarcost* is not transitive, although it is reflexive and symmetric. So *Similarcost* is not an equivalence relation.

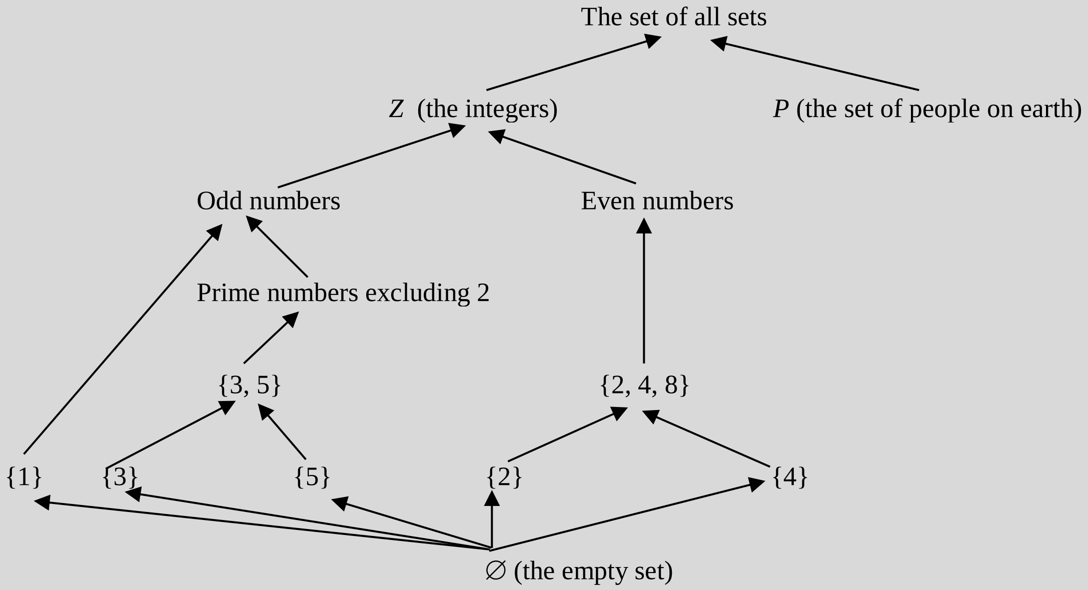
32.3.5 Orderings

Important as equivalence relations are, they are not the only special kind of relation worth mentioning. Let's consider two more.

A ***partial order*** is a relation that is reflexive, antisymmetric, and transitive. Let R be a partial order defined on a set A. Then the pair (A, R) is a ***partially ordered set***. If we write out any partial order as a graph, we'll see a structure like the ones in the following examples. Notice that, to make the graph relatively easy to read, we'll adopt the convention that we don't write in the links that are required by reflexivity and transitivity. But, of course, they are there in the relations themselves.

Example 32.5 Subset-of is a Partial Order

Consider the relation *Subset-of*, defined on the set of all sets. *Subset-of* is a partial order, since it is reflexive (every set is a subset of itself), transitive (if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$) and antisymmetric (if $A \subseteq B$ and $A \neq B$, then it must not be true that $B \subseteq A$). A small piece of *Subset-of* can be drawn as:



Read an arrow from x to y as meaning that (x, y) is an element of *Subset-of*. So, in this example, $\{3\}$ is a subset of $\{3, 5\}$. Note that in a partial order, it is often the case that there are some elements (such as $\{3, 5\}$ and $\{2\}$) that are not related to each other at all (since neither is a subset of the other). Remember in reading this picture that we have omitted the reflexive and transitive arrows.

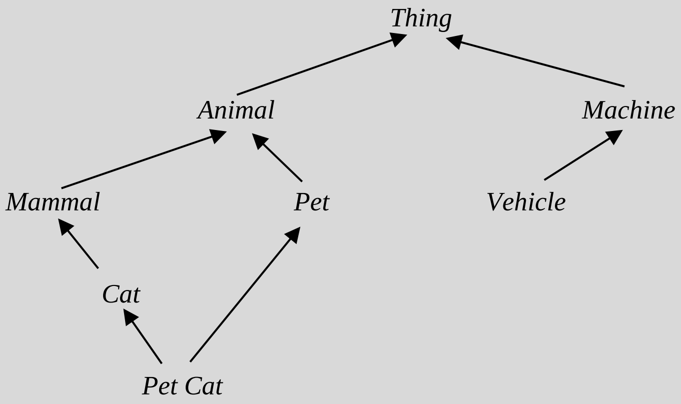
Example 32.6 Proper-subset-of is not a Partial Order

Now consider the relation *Proper-subset-of*. It is not a partial order because it is not reflexive. For example $\{1\} \not\subset \{1\}$.

In many kinds of applications, it is useful to organize the objects we are dealing with by defining a partial order that corresponds to the notion of one object being more or less general than another. Such a relation may be called a ***subsumption relation***.

Example 32.7 Concepts Form a Subsumption Relation

Consider a set of concepts, each of which corresponds to some significant set of entities in the world. Some concepts are more general than others. We'll say that a concept x is ***subsumed*** by a concept y (written $x \leq y$) iff every instance of x is also an instance of y. Alternatively, y is at least as general as x. A small piece of this subsumption relation for some concepts that might be used to model the meanings of common English words is:

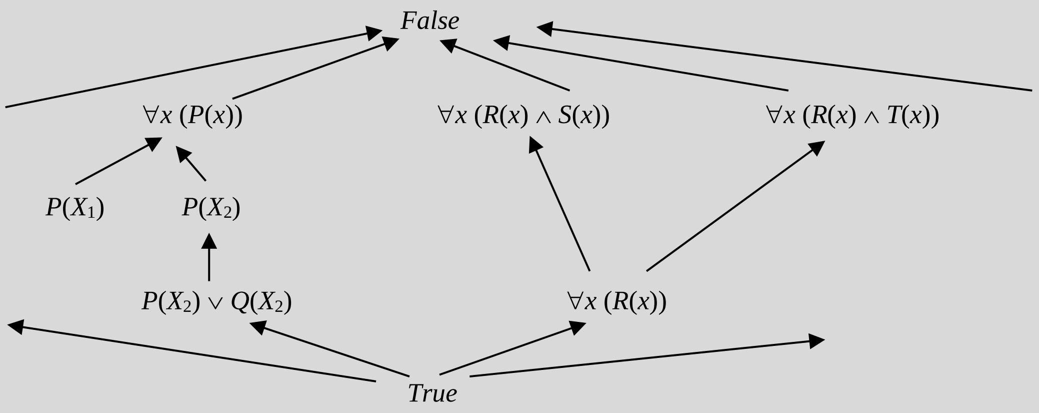


Concept subsumption is a partial order. It is very similar to the *Subset-of* relation except that it is defined only on the specific subsets that have been defined as concepts.

Subsumption relations are useful because they tell us when we have new information. If we already know that some object X_1 is a cat, we learn nothing new when told that it is an animal.

Example 32.8 Logical Statements Form a Subsumption Lattice

A first-order logic sentence P is subsumed by another sentence Q (written $P \leq Q$) iff, whenever Q is true P must be true, regardless of the values assigned to the variables, functions, and predicates of P and Q. For example: $\forall x\ (P(x))$ subsumes $P(X_1)$, since, regardless of what the predicate P is and what axioms we have about it, and regardless of what object X_1 represents, if $\forall x\ (P(x))$ is true, then $P(X_1)$ must be true. Why is this a useful notion? Suppose that we are building a theorem-proving or reasoning program. If we already know $\forall x\ P(x)$, and we are then told $P(X_1)$, we can throw away this new fact. It doesn't add to our knowledge (except perhaps to focus our attention on the object X_1), since it is subsumed by something we already knew. A small piece of the subsumption relation on sentences is shown in the following graph:



The subsumption relation on sentences is a partial order.

The symbol $\leq$ is often used to denote a partial order. Let $\leq$ be an arbitrary partial order defined on some domain A. Any element x of A such that $\forall y \in A\ ((y \leq x) \rightarrow (y = x))$ is a ***minimal element*** of A with respect to $\leq$. In other words, x is a minimal element if there are no other elements less than or equal to it. Similarly, any element x of A such that $\forall y \in A\ (x \leq y \rightarrow y = x)$ is a ***maximal element*** of A with respect to $\leq$. There may be more than one minimal (or maximal) element in a partially ordered set. For example, the partially ordered set of concepts in Example 32.7 has two minimal elements, *Pet Cat* and *Vehicle*. If there is a unique minimal element it is called the ***least element***. If there is a unique maximal element it is called the ***greatest element***. The set of logical sentences ordered by subsumption has a greatest element, *False*, which subsumes everything. It makes the strongest, and in fact, unsatisfiable claim. There is also a least element, *True*, which makes the weakest possible claim, and is subsumed by all other sentences.

A ***total order*** $R \subseteq A \times A$ is a partial order that has the additional property that $\forall x, y \in A\ ((x, y) \in R \vee (y, x) \in R)$. In other words, every pair of elements must be related to each other one way or the other. The classic example of a total order is ≤ (or ≥, if you prefer) on the integers. The ≤ relation is a partial order and, given any two integers x and y, either $x \leq y$ or $y \leq x$. If we draw any total order as a graph, we'll get something that looks like the graph in Figure 32.6 (again without the reflexive and transitive links shown).

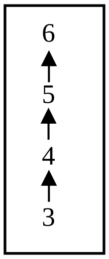


Figure 32.6 Drawing a total order as a graph

This is only a tiny piece of the graph, of course. It continues infinitely in both directions. But notice that, unlike our earlier examples of partial orders, there is no splitting in this graph. For every pair of elements, one is above and one is below. If R is a total order defined on a set A, then the pair (A, R) is a ***totally ordered set***. Of course, not all partial orders are also total. For example, the *Subset-of* relation we described in Example 32.5 is not a total order.

Given a partially ordered set (A, R), an ***infinite descending chain*** is a totally ordered, with respect to R, subset B of A that has no minimal element. If (A, R) contains no infinite descending chains then it is called a ***well-founded set***. An equivalent definition is the following: A partially ordered set (A, R) is a well-founded set iff every subset of A has at least one minimal element with respect to R. If (A, R) is a well-founded set and R is a total order, then (A, R) is called a ***well-ordered set***. Every well-ordered set has a least element. For example, consider the sets ℕ (the natural numbers) and Z (the integers). The totally ordered set (ℕ, ≤) is well-founded and well-ordered. Its least element is 0. The totally ordered set $(Z, \leq)$ is neither well-founded nor well-ordered, since it contains an infinite number of infinite descending chains, such as 3, 2, 1, 0, -1, -2, ….

Table 32.4 reviews some of our examples.

(A, R*)*	***Well-founded?***	***Well-ordered?***
The set of sets with respect to the *subset-of* relation	Yes	No
The set of concepts with respect to *subsumption*	Yes	No
The set of first-order sentences with respect to *subsumption*	Yes	No
The set of natural numbers under ≤	Yes	Yes
The set of integers under ≤	No	No

Table 32.4 Checking for well-foundedness and well-orderedness

Well-founded and well-ordered sets are important. Well-ordered sets provide the basis for proofs by induction (as we'll see in Section 32.6.6). Well-founded sets (that are often also well-ordered) provide the basis for proofs that loops and recursively defined functions halt (as we'll see in Section 32.7.1).

32.4 Functions

Relations are very general. They allow an object to be related to any number of other objects at the same time (as they are, for example, in our *Dessert* relation). Sometimes we want a more restricted notion, in which each object is related to a unique other object. For example, (at least in an ideal world without criminals or incompetent bureaucrats) each American resident is related to a unique social security number. To capture this idea we need functions.

32.4.1 What is a Function?

We begin with the common definition of a function: A ***function*** f from a set A to a set B is a binary relation that is a subset of $A \times B$, with the additional property that:

$$\forall x \in A\ ((((x, y) \in f \wedge (x, z) \in f) \rightarrow y = z) \wedge \exists y \in B\ ((x, y) \in f\ )).$$

In other words, each element of A is related to exactly one element of B.

The *Dessert* relation we defined earlier is not a function since Dave and Sara each occur twice. We haven't restricted each person to precisely one dessert. A simple relation that *is* a function is the successor function *succ* defined on the integers:

$$succ(n) = n + 1.$$

Of course, we cannot write out all the elements of *succ* (since there are an infinite number of them), but *succ* includes:
$\{\ldots, (-3, -2), (-2, -1), (-1, 0), (0, 1), (1, 2), (2, 3)\ldots\}$.

It is useful to define some additional terms to make it easy to talk about functions. We start by writing:

$$f : A \rightarrow B,$$

which means that f is a function from the set A to the set B. We call A the ***domain*** of f and B the ***codomain*** or ***range*** of f. We may also say that f is a function from A to B. Using this notation, we can write function definitions that have two parts, the first of which specifies the domain and range and the second of which defines the way in which the elements of the range are related to the elements of the domain. So, for example, we define the successor function on the integers (denoted as Z) by writing:

$$succ: Z \rightarrow Z,$$
$$succ(n) = n + 1.$$

If $x \in A$, then we write:

$$f(x),$$

which we read as, "f of x", to indicate the element of B to which x is related. We call this element the ***image*** of x under f or the ***value*** of f for x. Note that, given the definition of a function, there must be exactly one such element. We will also call x the ***argument*** of f. For example we have that:

$$succ(1) = 2,\ \ succ\ (2) = 3, \ldots$$

Thus 2 is the image (or the value) of the argument 1 under *succ*.

We will also use the notation $f(x)$ to refer to the function f (as opposed to f's value at a specific point x) whenever we need a way to refer to f's argument. So, for example, we'll write, as we did above, $succ(n) = n + 1$.

The function *succ* is a ***unary function***. It maps from a single element (a number) to another element. We are also interested in functions that map from ordered pairs of elements to a value. We call such functions ***binary functions***. For example, integer addition is a binary function:

$$+: (Z \times Z) \rightarrow Z.$$

Thus + includes elements such as ((2, 3), 5), since 2 + 3 is 5. We could also write:

$$+((2, 3)) = 5.$$

We have used double parentheses here because we are using the outer set to indicate function application (as we did above without confusion for *succ*) and the inner set to define the ordered pair to which the function is being applied. But this is confusing. So, generally, when the domain of a function is the Cartesian product of two or more sets, as it is here, we drop the inner set of parentheses and simply write:

$$+(2, 3) = 5.$$

The ***prefix notation*** that we have used so far, in which we write the name of the function first, followed by its arguments, can be used for functions of any number of arguments. For the specific, common case of binary functions, it is often convenient to use an alternative: ***infix notation*** in which the function name (often called the operator) is written between its two arguments.

$$2 + 3 = 5.$$

So far, we have considered unary functions and binary functions. But just as we could define *n*-ary relations for arbitrary values of *n*, we can define *n*-ary functions. For any positive integer *n*, an ***n*-ary function** *f* is a function that is defined as:

$$f : (S_1 \times S_2 \ldots \times S_n) \rightarrow R.$$

For example, let Z be the set of integers. Then:

$$quadraticequation : (Z \times Z \times Z) \rightarrow F.$$

is a function whose domain is an ordered triple of integers and whose range is a set of functions. The definition of *quadraticequation* is:

$$quadraticequation(a, b, c) = ax^2 + bx + c.$$

What we did here is typical of function definition. First we specify the domain and the range of the function. Then we define how the function is to compute its value (an element of the range) given its arguments (an element of the domain).

Whenever the domain of a function *f* is an ordered *n*-tuple of elements drawn from a single set *S*, we may (loosely) say that the domain of *f* is *S*. In this case, we may also say that *f* is a function of *n* arguments. So, for example, we may talk about the binary function + on the domain $\mathbb{N}$ (when, properly, its domain is $\mathbb{N} \times \mathbb{N}$).

Recall that, in the last section, we said that we could compose binary relations to derive new relations. Clearly, since functions are just special kinds of binary relations, if we can compose binary relations we can certainly compose binary functions. Because a function returns a unique value for each argument, it generally makes a lot more sense to compose functions than it does relations, and you'll see that although we rarely compose relations that aren't functions, we compose functions all the time. So, following our definition above for relations, we define the ***composition*** of two functions $f \subseteq A \times B$ and $g \subseteq B \times C$, written $g \circ f$, as:

$$g \circ f = \{(a, c) : \exists b\ ((a, b) \in f \text{ and } (b, c) \in g)\}.$$

Notice that the composition of two functions must necessarily also be a function. We mentioned above that there is sometimes confusion about the order in which relations (and now functions) should be applied when they are composed. To avoid this problem, we will introduce a new notation $g(f(x))$. We use the parentheses here to indicate function application, just as we did above. So $g \circ f(x) = g(f(x))$. This notation is clear. Apply g to the result of first applying f to x. This notation reads right to left as does our definition of the ° notation.

32.4.2 Properties of Functions

Some functions possess properties that may make them particularly useful for certain tasks.

The definition that we gave for a function at the beginning of this section required that, for $f : A \rightarrow B$ to be a function, it must be defined for every element of A (i.e., every element of A must be related to some element of B). This is the standard mathematical definition of a function. But, as we pursue the idea of "computable functions" (i.e., functions that can be implemented on some reasonable computing platform), we'll see that there are functions whose domains cannot be effectively defined. For example, consider a function *steps* whose input is a Java program and whose result is the number of steps that are executed by the program on the input 0. This function is undefined for programs that do not halt on the input 0. As we'll see in Chapter 19, there can exist no program that can check a Java program and determine whether or not it will halt on the input 0. So there is no program that can look at a possible input to *steps* and determine whether that input is in *steps*'s domain. In Chapter 25, we will consider two approaches to fixing this problem. One is to extend the range of *steps*, for example by adding a special value, *Error*, that can be the result of applying *steps* to a program that doesn't halt on input 0. The difficulty with this approach is that *steps* becomes uncomputable since there exists no algorithm that can know when to return *Error*. Our alternative is to expand the domain of *steps*, for example to the set of all Java programs. Then we must acknowledge that if *steps* is applied to certain elements of its domain (i.e., programs that don't halt), its value will be undefined.

In order to be able to talk about functions like *steps*, we'll introduce two new terms. We'll say that $f : A \rightarrow B$ is a ***total function*** on A iff it is a function that is defined on all elements of A (i.e., it is a function in the standard mathematical sense). We'll say that $f : A \rightarrow B$ is a ***partial function*** on A iff f is a subset of $A \times B$ and every element of A is related to no more than one element of B. In Chapter 25 we will return to a discussion of partial functions. Until then, when we say that f is a function, we will mean that it is a total function.

A function $f : A \rightarrow B$ is ***one-to-one*** iff no two elements of A map to the same element of B. *Succ* is one-to-one. For example, the only number to which we can apply *succ* and derive 2 is 1. *Quadraticequation* is also one-to-one. But + isn't. For example, both +(2, 3) and +(4, 1) equal 5.

A function $f : A \rightarrow B$ is ***onto*** iff every element of B is the value of some element of A. Another way to think of this is that a function is onto iff all of the elements of B are "covered" by the function. As we defined it above, *succ* is onto. But let's define a different function *succ'* on $\mathbb{N}$ (the natural numbers) (rather than the integers). So we define:

$$succ' : \mathbb{N} \rightarrow \mathbb{N}.$$
$$succ'(n) = n + 1.$$

Succ' is not onto because there is no natural number i such that $succ'(i) = 0$.

The easiest way to envision the differences between an arbitrary relation, a function, a one-to-one function, and an onto function is to make two columns (the first for the domain and the second for the range) and think about the kind of matching problems you probably had on tests in elementary school.

Consider the six matching problems shown in Figure 32.7. In each, we'll consider ways of relating the elements of the first column (the domain) to the elements of the second column (the range). Example 1 describes a relation that is not a (total) function because C is an element of its domain that is not related to any element of its range. Example 2 describes a relation that is not a function because there are three values associated with A. The third example is a function since, for each object in the first column, there is a single value in the second column. But this function is neither one-to-one (because X is derived from both A and B) nor onto (because Z is not the image of anything). The fourth example is a function that is one-to-one (because no element of the second column is related to more than one element of the first column). But it still isn't onto because Z has been skipped: nothing in the first column derives it. The fifth example is a function that is onto (since every element of column two has an arrow coming into it), but it isn't one-to-one, since Z is derived from both C and D. The sixth and final example is a function that is both one-to-one and onto. By the way, see if you can modify either example 4 or example 5 to make it both one-to-one and onto. You're not allowed to change the number of elements in either column, just the arrows. You'll notice that you can't do it. In order for a function to be both one-to-one and onto, there must be equal numbers of elements in the domain and the range.

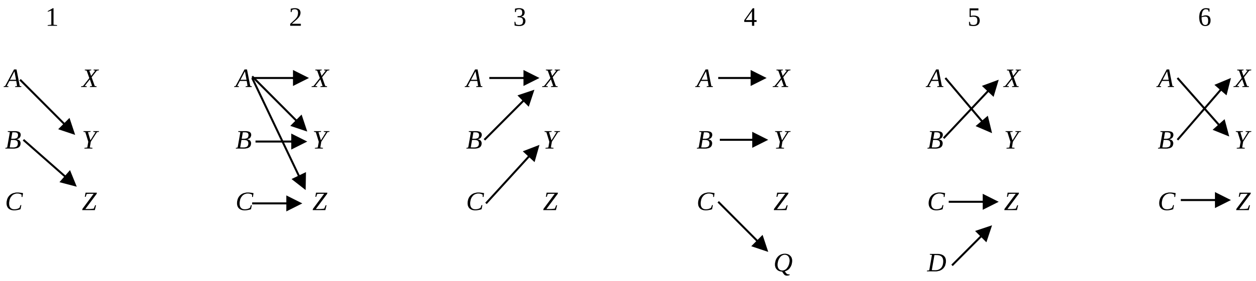


Figure 32.7 Kinds of relations and functions

The ***inverse*** of a binary function f is the relation that contains exactly the ordered pairs in f with the elements of each pair reversed. We'll write the inverse of f as f^{-1}. Formally, if $f \subseteq A \times B$, then $f^{-1} \subseteq B \times A = \{(b, a): (a, b) \in f\}$. Since every function is a relation, every function has a relational inverse, but that relational inverse may or may not also be a function. For example, look again at example 3 of the matching problem above. Although it is a function, its inverse is not. Given the argument X, should we return the value A or B? Now consider example 4. Its inverse is also not a function, since there is no value to be returned for the argument Z. Example 5 has the same problem example 3 does. Now look at example 6. Its inverse is a function. Whenever a function is both one-to-one and onto, its inverse will also be a function and that function will be both one-to-one and onto. Such functions are called ***bijections***. Bijections are useful because they enable us to move back and forth between two sets without loss of information. Look again at example 6. We can think of ourselves as operating in the $\{A, B, C\}$ universe or in the $\{X, Y, Z\}$ universe interchangeably since we have a well defined way to move from one to the other. And if we move from column one to column two and then back, we'll be exactly where we started.

It is sometimes useful to talk about functions that map one object to another but, in so doing, do not fundamentally change the way that the objects behave with respect to some structure (i.e., some set of functions that we care about). A ***homomorphism*** is a function that maps the elements of its domain to the elements of its range in such a way that some structure of the original set is preserved. So, considering just binary functions, if f is a homomorphism and # is a binary function in the structure that we are considering, then it must be case that $\forall x, y\ (f(x) \# f(y) = f(x \# y))$. The structure of unary and higher order functions must also be preserved in a similar way.

Given a particular function f, whether or not it is a homomorphism depends on the structure that we are considering. So, for example, consider the integers, along with one function, addition. Then the function $f(x) = 2x$ is a homomorphism because $2x + 2y = 2(x + y)$. But, if the structure we are working with also contains a second function, multiplication, then f is no longer a homomorphism because, unless x or y is 0, $2x \cdot 2y \neq 2(x \cdot y)$.

If a homomorphism f is also a bijection, then it is called an ***isomorphism***. If two objects are isomorphic to each other, then they are indistinguishable with respect to the defining structure. For example, consider the set of undirected graphs, along with all of the standard graph operations that determine size and paths. If G is an arbitrary graph, let $f(G)$ be exactly G except that the symbol # is appended to the name of every vertex. This function f is an isomorphism. The two graphs shown in Figure 30.8 are isomorphic.

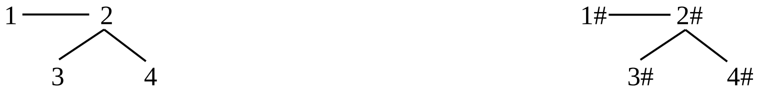


Figure 32.8 Two isomorphic graphs

When the intersection of the domain and the range of a function f is not empty, it is sometimes useful to find elements of the domain that are unchanged by the application of f. A ***fixed point*** of a function f is an element x of f's domain with the property that $f(x) = x$. For example, 1 and 2 are fixed points of the factorial function since $1! = 1$ and $2! = 2$. The factorial function has no other fixed points.

32.4.3 Properties of Binary Functions

Any relation that uniquely maps from each element of its domain to some element of its range is a function. The two sets involved can be anything and the mapping can be arbitrary. However, most of the functions we actually care about behave in some sort of regular fashion. It is useful to articulate a set of properties that many of the functions that we'll study have. When these properties are true of a function, or a set of functions, they give us techniques for proving additional properties of the objects involved. In the following definitions, we will consider an arbitrary binary function # defined over a set A. As examples, we'll consider functions whose actual domains are ordered pairs of sets, integers, strings, and Boolean expressions.

A binary function # is ***commutative*** iff $\forall x, y \in A\ (x \# y = y \# x)$.

Examples:	$i + j = j + i.$	(integer addition)
	$A \cap B = B \cap A.$	(set intersection)
	$P \wedge Q \equiv Q \wedge P.$	(Boolean *and*)

A binary function # is ***associative*** iff $\forall x, y, z \in A\ ((x \# y) \# z = x \# (y \# z))$.

Examples:	$(i + j) + k = i + (j + k).$	(integer addition)
	$(A \cap B) \cap C = A \cap (B \cap C).$	(set intersection)
	$(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R).$	(Boolean *and*)
	$(s \parallel t) \parallel w = s \parallel (t \parallel w).$	(string concatenation)

A binary function # is ***idempotent*** iff $\forall x \in A\ (x \# x = x)$.

Examples:	$min(i, i) = i.$	(integer minimum)
	$A \cap A = A.$	(set intersection)
	$P \wedge P \equiv P.$	(Boolean *and*)

The ***distributivity*** property relates two binary functions: A function # distributes over another function % iff $\forall x, y, z \in A\ (x \# (y \% z) = (x \# y) \% (x \# z))$.

Examples:	$i \cdot (j + k) = (i \cdot j) + (i \cdot k).$	(integer multiplication over addition)
	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$	(set union over intersection)
	$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge Q).$	(Boolean *and* over *or*)

Absorption laws also relate two binary functions to each other: A function # absorbs another function % iff $\forall x, y \in A\ (x \# (x \% y) = x))$.

Examples:	$A \cap (A \cup B) = A.$	(Set intersection absorbs union.)
	$P \vee (P \wedge Q) \equiv P.$	(Boolean *or* absorbs *and*.)
	$P \wedge (P \vee Q) \equiv P.$	(Boolean *and* absorbs *or*.)

It is often the case that when a function is defined over some set A, there are special elements of A that have particular properties with respect to that function. In particular, it is worth defining what it means to be an identity and to be a zero:

An element a is an ***identity*** for the function # iff $\forall x \in A\ ((x \# a = x) \wedge (a \# x = x))$.

Examples:	$i \cdot 1 = i$.	(1 is an identity for integer multiplication.)
	$i + 0 = i$.	(0 is an identity for integer addition.)
	$A \cup \varnothing = A$.	($\varnothing$ is an identity for set union.)
	$P \vee False \equiv P$.	(*False* is an identity for Boolean *or*..)
	$s \,\|\, "" = s$.	("" is an identity for string concatenation)

Sometimes it is useful to differentiate between a ***right identity*** (one that satisfies the first requirement above) and a ***left identity*** (one that satisfies the second requirement above). But for all the functions we'll be concerned with, if there is a left identity, it is also a right identity and vice versa, so we will talk simply about an identity.

An element *a* is a ***zero*** for the function # iff $\forall x \in A\ ((x \# a = a) \wedge (a \# x = a))$.

Examples:	$i \cdot 0 = 0$.	(0 is a zero for integer multiplication.)
	$A \cap \varnothing = \varnothing$.	($\varnothing$ is a zero for set intersection.)
	$P \wedge False \equiv False$.	(*False* is a zero for Boolean *and*.)

Just as with identities, it is sometimes useful to distinguish between left and right zeros, but we won't need to.

Although we're focusing here on binary functions, there's one important property that unary functions may have that is worth mentioning here:

A unary function \$ is a ***self inverse*** iff $\forall x\ (\$(\$(x)) = x)$. In other words, if we compose the function with itself (apply it twice), we get back the original argument.

Examples:	$-(-(i)) = i$.	(Multiplying by -1 is a self inverse for integers.)
	$1/(1/i) = i$ if $i \neq 0$.	(Dividing into 1 is a self inverse for integers.)
	$\neg\neg A = A$.	(Complement is a self inverse for sets.)
	$\neg(\neg P) = P$.	(Negation is a self inverse for Booleans.)
	$(s^R)^R = s$.	(Reversal is a self inverse for strings.)

32.4.4 Properties of Functions on Sets

The functions that we have defined on sets satisfy most of the properties that we have just considered. Further, as we saw above, some set functions have a zero or an identity. We'll summarize here (without proof) the most useful properties that hold for the functions we have defined on sets:

Commutativity:	$A \cup B = B \cup A$. $A \cap B = B \cap A$.
Associativity:	$(A \cup B) \cup C = A \cup (B \cup C)$. $(A \cap B) \cap C = A \cap (B \cap C)$.
Idempotency:	$A \cup A = A$. $A \cap A = A$.
Distributivity:	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.
Absorption:	$(A \cup B) \cap A = A$. $(A \cap B) \cup A = A$.
Identity:	$A \cup \varnothing = A$.
Zero:	$A \cap \varnothing = \varnothing$.

Self Inverse: $\neg\neg A = A$.

In addition, we will want to make use of the following theorems that can be proven to apply specifically to sets and their operations (as well as to Boolean expressions, with $\vee$ substituted for $\cup$ and $\wedge$ substituted $\cap$):

De Morgan's laws: $\neg(A \cup B) = \neg A \cap \neg B$.
$\neg(A \cap B) = \neg A \cup \neg B$.

32.5 Closures

We say that a binary relation R on a set A is ***closed under*** property P iff R possesses P. For example, the relation $\leq$ as generally defined on the integers is closed under *transitivity*.

Sometimes, if a relation R is not closed under P we may want to ask what elements would have to be added to R so that it would possess P. So, let R be a binary relation on a set A. A relation R' is a ***closure*** of R with respect to some property P iff:

- $R \subseteq R'$,
- R' is closed under P, and
- There is no smaller relation R'' that contains R and is closed under P. (One relation R_1 is smaller than another relation R_2 iff $|R_1| < |R_2|$.

In other words, to form the closure of R with respect to P we add to R the minimum number of elements required to establish P. So, for example, the ***transitive closure*** of a binary relation R is the smallest relation R' that contains R but is transitive. Thus, if R contains the elements (x, y) and (y, z), the transitive closure of R must also contain the element (x, z).

Example 32.9 Forming Transitive and Reflexive Closures

Let $R = \{(1, 2), (2, 3), (3, 4)\}$.

- The transitive closure of R is $\{(1, 2), (2, 3), (3, 4), (1, 3), (1, 4), (2, 4)\}$.
- The reflexive closure of R is $\{(1, 2), (2, 3), (3, 4), (1, 1), (2, 2), (3, 3), (4, 4)\}$.

Example 32.10 The Transitive Closure of Parent-of

The transitive closure of *Parent-of* is *Ancestor-of.*

Under some conditions (which will hold in all the cases we consider), it is possible to prove that a relation R has a unique closure under the property P. (See Section 32.8 for a discussion of this issue.)

We can define the closure of a set with respect to a function in a similar manner. Let f be a function of n arguments. We say that a set A is ***closed under*** f iff, whenever all n of f's arguments are elements of A, the value of f is also in A. For example, the positive integers are closed under addition. The positive integers are not closed under subtraction since, for example 7-10 = -3.

As we did for relations, we may again want to consider, whenever a set A is not closed under some function $f: X \rightarrow Y$, how A could be augmented (with additional elements drawn from X) so that it would be closed. Let f be function of n arguments drawn from a set A. A set A' is a ***closure*** of A under f iff:

- $A \subseteq A'$,
- A' is closed under f, and
- There is no smaller set A'' that contains A and is closed under f.

Example 32.11 Closures under Functions

- {0} is not closed under the successor function *succ*, since *succ*(0) = 1. The closure of {0} under *succ* is ℕ (the natural numbers).
- ℕ is closed under addition (since the sum of any two natural numbers is a natural number). So the closure of ℕ under addition is simply ℕ.
- ℕ is not closed under subtraction. For example, 5 - 7 is not a natural number. The closure of ℕ under subtraction is *Z* (the integers).
- *Z* is not closed under division. Its closure under division is *Q* (the rational numbers) plus a special element that is the result of dividing by 0.
- *Q* is not closed under limits. Its closure under limits is *R* (the real numbers).
- *R* is not closed under square root. Its closure under square root is *C* (the complex numbers).
- The set of even length strings of a's and b's is closed under concatenation.
- The set of odd length strings of a's and b's is not closed under concatenation. For example the string aaa concatenated with the string aaa is aaaaaa, whose length is not odd. The closure of the odd length strings of a's and b's under concatenation is the set of all strings of a's and b's.
- Let *A* be the set of all strings of a's. So *A* = {a, aa, aaa, aaaa, aaaaa, …}. Let *S* be the set that contains all subsets *SS* of *A* where *SS* contains an odd number of elements. So *S* = {{a}, {aa}, {aaa}, …, {a, aa, aaa}, …}. *S* is not closed under union, since, for example {a} ∪ {aa} = {a, aa}, which is not in *S*, since it contains an even number of elements. The closure of *S* under union is the set of all finite sets in $\mathcal{P}(A)$.

Given a set *S* and a property *P*, we may want to compute the closure of *S* with respect to *P*. For example, we will often want to compute the transitive closure of a binary relation *R* on a set *A*. This is harder than it seems. We can't just add a fixed number of elements to *R* and then quit. Every time we add a new element, such as (*x*, *y*), we have to look to see whether there is some element (*y*, *z*). If so, we also have to add (*x*, *z*). And, similarly, we must check for any element (*w*, *x*) that would force us to add (*w*, *y*). If *R* is infinite, there is no guarantee that this process will ever terminate. Theorem 32.5 (presented in Section 32.8) guarantees that a unique closure exists but it does not guarantee that the closure will contain a finite number of elements and thus be computable in a finite amount of time.

We can, however, guarantee that the transitive closure of any *finite* binary relation is computable. How? A very simple approach is the following algorithm for computing the transitive closure of a binary relation *R* with *n* elements on a set *A*. If *t* is an ordered pair, then *t.first* will refer to the first element of the pair and *t.second* will refer to the second element.

```
computetransitiveclosure(R: relation) =
     1.  trans = R.                                /* Initially trans is just the original relation.
     /* We need to find all cases where (x, y) and (y, z) are in trans. Then we must insert (x, z) into trans if
     /*  it isn't already there.
     2.  addedSomething = True.                    /* Keep going until we make one whole pass.
                                                      without adding any new elements to trans.
     3.  While addedSomething = True do:
         3.1.  addedSomething = False.
         3.2.  For each element t1 of trans do:
                   For each element t2 of trans do:    /* Compare t1 to every other element of trans.
                       If t1.second = t2.first then do:            /* We have(x, y) and (y, z).
                           If (t1.first, t2.second) ∉ trans then do:   /* We have to add (x, z).
                               Insert(trans, (t1.first, t2.second)).
                               addedSomething = True.
```

This algorithm is straightforward and correct, but it may be inefficient. There are more efficient algorithms. In particular, if we represent a relation as an adjacency matrix, we can do better using Warshall's algorithm 💻, which finds the transitive closure of a relation over a set of *n* elements using approximately $2n^3$ bit operations.

In Section 32.8, we present a more general definition of closure that includes, as special cases, the two specific definitions presented here. We also elaborate on some of the claims that we have just made.

32.6 Proof Techniques

In this section we summarize the most important proof techniques that we will use in the rest of this book.

32.6.1 Proof by Construction

Suppose that we want to prove an assertion of the form $\exists x\ (Q(x))$ or $\forall x\ (\exists y\ (P(x, y)))$. One way to prove such a claim is to show a (provably correct) algorithm that finds the value that we claim must exist. We call that technique ***proof by construction***.

For example, we might wish to prove that every pair of integers has a greatest common divisor. We could prove that claim by exhibiting a correct greatest common divisor algorithm. In exhibiting such an algorithm, we show not only that the greatest common divisor exists (since the algorithm provably finds one for every input pair), we show something more: a method to determine the greatest common divisor for any pair of integers. While this is a stronger claim than the one we started with, it is often the case that such stronger claims are easier to prove.

32.6.2 Proof by Contradiction

Suppose that we want to prove some assertion P. One approach is to assume, to the contrary, that $\neg P$ were true. We then show, with that assumption, that we can derive a contradiction. The law of the excluded middle says that ($P \vee \neg P$). If we accept it, and we shall, then, since $\neg P$ cannot be true, P must be.

Example 32.12 There is an Infinite Number of Primes

Consider the claim that there is an infinite number of prime numbers. Following Euclid, we prove this claim by assuming, to the contrary, that the set P of prime numbers is finite. So there exists some value of n such that $P = \{p_1, p_2, p_3, \dots p_n\}$. Let:

$$q = (p_1p_2p_3 \dots p_n) + 1.$$

Since q is greater than each p_i, it is not on the list of primes. So it must be composite. In that case, it must have at least one prime factor, which must then be an element of P. Suppose that factor is p_k, for some $k \leq n$. Then q must have at least one other factor, some integer i such that:

$$\begin{aligned} q &= ip_k. \\ (p_1p_2p_3 \dots p_n) + 1 &= ip_k. \\ (p_1p_2p_3 \dots p_n) - ip_k &= -1. \end{aligned}$$

Now observe that p_k divides both terms on the left since it is prime and so must be in the set $\{p_1, p_2, p_3, \dots p_n\}$. Factoring it out, we get:

$$\begin{aligned} p_k(p_1p_2p_{k-1}p_{k+1} \dots p_n - i) &= -1. \\ p_k &= -1/(p_1p_2p_{k-1}p_{k+1} \dots p_n - i). \end{aligned}$$

But, since $(p_1p_2p_{k-1}p_{k+1} \dots p_n - i)$ is an integer, this means that $|p_k| < 1$. But that cannot be true since p_k is prime and thus greater than 1. So q is not composite. Since q is greater than 1 and not composite, it must be prime, contradicting the assumption that all primes are in the set $\{p_1, p_2, p_3, \dots p_n\}$.

■

Notice that this proof, in addition to being a proof by contradiction, is constructive. It exhibits a specific example that contradicts the initial assumption.

Example 32.13 $\sqrt{2}$ is Irrational

Consider the claim that $\sqrt{2}$ is irrational. We prove this claim by assuming, to the contrary, that $\sqrt{2}$ is rational. In that case, it is the quotient of two integers, *i* and *j*. So we have:

$$\sqrt{2} = i/j.$$

If *i* and *j* have any common factors, then reduce them by those factors. Now we have:

$$\sqrt{2} = k/n, \text{ where } k \text{ and } n \text{ have no common factors.}$$
$$2 = k^2/n^2.$$
$$2n^2 = k^2.$$

Since 2 is a factor of k^2, k^2 must be even and so k is even. Since k is even, we can rewrite it as $2m$ for some integer m. Substituting $2m$ for k, we get:

$$2n^2 = (2m)^2.$$
$$2n^2 = 4m^2.$$
$$n^2 = 2m^2.$$

So n^2 is even and thus n is even. But now both k and n are even and so have 2 as a common factor. But we had reduced them until they had no common factors. The assumption that $\sqrt{2}$ is rational has led to a contradiction. So $\sqrt{2}$ cannot be rational.

■

32.6.3 *Proof by Counterexample*

Consider any claim of the form $\forall x\ (P(x))$. Such a claim is false iff $\exists x\ (\neg P(x))$. We can prove that it is false by finding such an x.

Example 32.14 Mersenne Primes

Let M be the set of numbers of the form $2^n - 1$ for some positive integer n. M is also called the set of ***Mersenne numbers*** 💻. Now consider only those cases in which n is prime. (In fact, some authors restrict the term Mersenne number only to those cases.) Consider two statements:

1. If n is prime, then $2^n - 1$ is prime.
2. If $2^n - 1$ is prime, then n is prime.

Statement 2 is true 💻. But what about statement 1? Hundreds of years ago, some mathematicians believed that it was true, although they had no proof of it. Then, in 1536, Hudalricus Regius refuted the claim by showing a counterexample: 2^{11}-1 = 2047 is not prime. But that was not the end of false conjectures about these numbers. The elements of M that are also prime are called ***Mersenne primes*** 💻, after the monk Marin Mersenne, who, in 1644, made the claim that numbers of the form $2^n - 1$ are prime if n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127 and 257, but are composite for all other positive integers $n \leq 257$. Mersenne's claim was shown to be false by counterexample, over two hundred years later, when it was discovered that 2^{61}-1 is also prime. Later discoveries showed other ways in which Mersenne was wrong. The correct list of values of $n \leq 257$ such that $2^n - 1$ is prime is 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107 and 127.

Example 32.15 All it Takes is One Counterexample

Consider the following claim:

> Let A, B, and C be any sets. If $A - C = A - B$ then $B = C$.

We show that this claim is false with a counterexample: Let $A = \varnothing$, $B = \{1\}$, and $C = \{2\}$. $A - C = A - B = \varnothing$. But $B \neq C$.

■

32.6.4 Proof by Case Enumeration

Consider a claim of the form $\forall x \in A\ (P(x))$. Sometimes the most straightforward way to prove that P holds for all elements of A is to divide A into two or more subsets and then to prove P separately for each subset.

Example 32.16 The Postage Stamp Problem

Suppose that the postage required to mail a letter is always at least 6¢. Prove that it is possible to apply any required postage to a letter given only 2¢ and 7¢ stamps.

We prove this general claim by dividing it into two cases, based on the value of n, the required postage:

1. If n is even (and 6¢ or more), apply $n/2$ 2¢ stamps.
2. If n is odd (and 6¢ or more), then $n \geq 7$ and n-7 ≥ 0 and is even. 7¢ can be applied with one 7¢ stamp. Apply one 7¢ stamp and $(n$-$7)/2$ 2¢ stamps.

■

32.6.5 Mathematical Induction

The ***principle of mathematical induction*** states:

If: $P(b)$ is true for some integer base case b, and
For all integers $n \geq b$, $P(n) \rightarrow P(n+1)$
Then: For all integers $n \geq b$, $P(n)$

A proof using mathematical induction, of an assertion P about some set of positive integers greater than or equal to some specific value b, has three parts:

1) A clear statement of the assertion P.
2) A proof that that P holds for some base case b, the smallest value with which we are concerned. Often, $b = 0$ or 1, but sometimes P may hold only once we get past some initial unusual cases,
3) A proof that, for all integers $n \geq b$, if $P(n)$ then it is also true that $P(n+1)$. We'll call the claim $P(n)$ the ***induction hypothesis***.

Example 32.17 The Sum of the First n Odd Positive Integers is n^2

Consider the claim that that the sum of the first n odd positive integers is n^2. We first check for plausibility:

$(n = 1)$ 1 = 1 = 1^2.
$(n = 2)$ 1 + 3 = 4 = 2^2.
$(n = 3)$ 1 + 3 + 5 = 9 = 3^2.
$(n = 4)$ 1 + 3 + 5 + 7 = 16 = 4^2, and so forth.

The claim appears to be true, so we should prove it. Let $Odd_i = 2(i - 1) + 1$ denote the i^{th} odd positive integer. Then we can rewrite the claim as:

$$\forall n \geq 1 \quad (\sum_{i=1}^{n} Odd_i = n^2).$$

The proof of the claim is by induction on n:

- Base case: take 1 as the base case. $1 = 1^2$.
- Prove: $\forall n \geq 1((\sum_{i=1}^{n} Odd_i = n^2) \rightarrow (\sum_{i=1}^{n+1} Odd_i = (n+1)^2))$.

Observe that the sum of the first n + 1 odd integers is the sum of the first n of them plus the n+1st, so:

$$\sum_{i=1}^{n+1} Odd_i = \sum_{i=1}^{n} Odd_i + Odd_{n+1}.$$

$= n^2 + Odd_{n+1}$. (Using the induction hypothesis.)

$= n^2 + 2n + 1$. (Since Odd_{n+1} is $2(n + 1 - 1) + 1 = 2n + 1$.)

$= (n + 1)^2$. ■

Mathematical induction lets us prove properties of positive integers. But it also lets us prove properties of other things if the properties can be described in terms of integers. For example, we could talk about the cardinality of a finite set, or the length of a finite string.

Example 32.18 The Cardinality of the Power Set of a Finite Set

Let A be any finite set. We prove the following claim about the cardinality of the power set of A:

$$|\mathcal{P}(A)| = 2^{|A|}.$$

The proof is by induction on $|A|$, the cardinality of A.

- Base case: take 0 as the base case. $|A| = 0$, $A = \varnothing$, and $\mathcal{P}(A) = \{\varnothing\}$, whose cardinality is $1 = 2^0 = 2^{|A|}$.

- Prove: $\forall n \geq 0$ (($|\mathcal{P}(A)| = 2^{|A|}$ for all sets A of cardinality n) → ($|\mathcal{P}(A)| = 2^{|A|}$ for all sets A of cardinality n + 1)).

  We do this as follows. Consider any value of $n \geq 0$ and any set A with n + 1 elements. Since $n \geq 0$, A must have at least one element. Pick one and call it a. Now consider the set B that we get by removing a from A. $|B|$ must be n. So, by the induction hypothesis, $|\mathcal{P}(B)| = 2^{|B|}$. Now return to $\mathcal{P}(A)$. It has two parts: those subsets of A that include a and those that don't. The second part is exactly $\mathcal{P}(B)$, so we know that it has $2^{|B|} = 2^n$ elements. The first part (all the subsets that include a) is exactly all the subsets that don't include a with a added in). Since there are 2^n subsets that don't include a and there are the same number of them once we add a to each, we have that the total number of subsets of our original set A is 2^n (for the ones that don't include a) plus another 2^n (for the ones that do include a), for a total of $2(2^n) = 2^{n+1}$, which is exactly $2^{|A|}$.

■

Mathematical induction can be used to prove properties of a linear sequence of objects by assigning to each object its index in the sequence.

Example 32.19 Generalized Modus Tollens

Recall the inference rule we call modus tollens: From ($P \rightarrow Q$) and $\neg Q$, conclude $\neg P$. We can use mathematical induction to prove a generalization of modus tollens to an arbitrary chain of implications. Suppose that we know, for any value of $n \geq 2$, two things:

$\forall i$, where $1 \leq i < n$-1, $(P_i \rightarrow P_{i+1})$ /* In a chain of n propositions, each implies the next.
$\neg P_n$ /* The last proposition is known to be false.

Then generalized modus tollens will let us conclude that all the preceding propositions are also false, and so, in particular, it must be the case that:

$$\neg P_1.$$

We can use induction to prove this rule. To make it easy to describe the rule as we work, we'll introduce the notation P |- Q to mean that, from P, we can derive Q. Using this notation, we can state concisely the rule we are trying to prove:

$\forall n \geq 2\ (((\forall i < n\ (P_i \rightarrow P_{i+1})) \wedge \neg P_n) \mid\!\!- \neg P_1)$.

The proof is by induction on *n*, the number of propositions.

- Base case: take 2 as the base case. We have $P_1 \rightarrow P_2$ and $\neg P_2$. So, using modus tollens, we conclude $\neg P_1$.

- Prove that if the claim is true for *n* propositions it must be true for *n*+1 of them:

$$(((\forall i < n\ (P_i \rightarrow P_{i+1})) \wedge \neg P_n) \mid\!\!- \neg P_1) \rightarrow (((\forall i < n + 1\ (P_i \rightarrow P_{i+1})) \wedge \neg P_{n+1}) \mid\!\!- \neg P_1)$$

$((P_n \rightarrow P_{n+1}) \wedge \neg P_{n+1}) \mid\!\!- \neg P_n$	(Modus tollens)
$((\forall i < n\ (P_i \rightarrow P_{i+1})) \wedge \neg P_n) \mid\!\!- \neg P_1$	(Induction hypothesis)
$((\forall i < n\ (P_i \rightarrow P_{i+1})) \wedge (P_n \rightarrow P_{n+1}) \wedge \neg P_{n+1}) \mid\!\!- \neg P_1$	(Chaining)
$((\forall i < n + 1\ (P_i \rightarrow P_{i+1})) \wedge \neg P_{n+1}) \mid\!\!- \neg P_1$	(Simplification)

■

Mathematical induction relies on the fact that any subset of the nonnegative integers forms a well-ordered set (as defined in Section 32.3.5) under the relation ≤. Once we have done an induction proof, we know that *A*(*b*) (where *b* is typically 0 or 1, but it could be some other starting value) is true and we know that $\forall n \geq b\ (A(n) \rightarrow A(n + 1))$. Then we claim that $\forall n \geq b\ (A(n))$. Suppose that the principle of mathematical induction were not sound and there existed some set *S* of nonnegative integers ≥ *b* for which *A*(*n*) is false. Then, since *S* is well-ordered, it has a least element, which we can call *x*. By definition of *S*, *x* must be equal to or greater than *b*. But it cannot actually be *b* because we proved *A*(*b*). So it must be greater than *b*. Now consider *x* - 1. Since *x* - 1 is less than *x*, it cannot be in *S* (since we chose *x* to be the smallest value in *S*). If *x* - 1 is not in *S*, then we know *A*(*x* - 1). But we proved that $\forall n \geq 0\ (A(n) \rightarrow A(n+1))$, so $A(x - 1) \rightarrow A(x)$. But we assumed $\neg A(x)$. So that assumption led us to a contradiction and thus must be false.

Sometimes the principle of mathematical induction is stated in a slightly different but formally equivalent way:

If: *A*(*b)* is true for some integer value *b*, and
For all integers $n \geq b$ ((*A*(*k*) is true for all integers *k* where $b \leq k \leq n) \rightarrow A(n+1)$),
Then: For all integers $n \geq b\ (A(x))$.

This form of mathematical induction is sometimes called ***strong induction***. To use it, we prove that whenever *A* holds for all nonnegative integers starting with *b*, up to and including *n*, it must also hold for *n* + 1. We can use whichever form of the technique is easiest for a particular problem.

32.6.6 The Pigeonhole Principle

Suppose that we have *n* pigeons and *k* holes. Each pigeon must fly into a hole. If *n* > *k*, then there must be at least one hole that contains more than one pigeon. We call this obvious observation the ***pigeonhole principle***. More formally, consider any function $f: A \rightarrow B$. The pigeonhole principle says:

If |*A*| > |*B*| then *f* is not one-to-one.

The pigeonhole principle is a useful technique for proving relationships between sets. For example, suppose that set *A* is the set of all students who live in the dorm. Set *B* is the set of rooms in the dorm. The function *f* maps each student to a dorm room. So, if |*A*| > |*B*|, we can use the pigeonhole principle to show that some students have roommates. As another everyday use of the principle, consider: If there are more than 366 people in a class, then at least two of them must share a birthday. The pigeonhole principle is also useful in proving less obvious claims.

Example 32.20 The Coins and Balance Problem

Consider the following problem: You have three coins. You know that two are of equal weight; the third is different. You do not know which coin is different and you do not know whether it is heavier or lighter than the other two. Your task is to identify the different coin and to say whether it is heavier or lighter than the others. The only tool you have is a balance, with two pans, onto which you may place one or more objects. The balance has three possible outputs: left pan heavier than right pan, right pan heavier than left pan, both pans the same weight. Show that you cannot solve this problem in a single weighing.

There are six possible situations: there are three coins, any one of which could be different, and the different coin can be either heavier or lighter. But a single weighing (no matter how you choose to place coins on pans) has only three possible outcomes. So there is at least one outcome that corresponds to at least two situations. Thus one weighing cannot be guaranteed to determine the situation uniquely.

32.6.7 Showing That Two Sets Are Equal

A great deal of what we do when we build a theory about some domain is to prove that various sets of objects in that domain are equal. For example, in our study of automata theory, we are going to want to prove assertions such as these:

- The set of strings defined by some regular expression α is identical to the set of strings defined by some second regular expression β.
- The set of strings that will be accepted by some given finite state machine M is the same as the set of strings that will be accepted by some new finite state machine M' that has fewer states than M has.
- The set of languages that can be defined using regular expressions is the same as the set of languages that can be accepted by a finite state machine.
- The set of problems that can be solved by a Turing Machine with a single tape is the same as the set of problems that can be solved by a Turing Machine with any finite number of tapes.

So we become very interested in the question, “How does one prove that two sets are identical?” There are lots of ways and many of them require special techniques that apply in specific domains. But it is worth mentioning two very general approaches here.

Sometimes we want to compare apples to apples. We may, for example, want to prove that two sets of strings are identical, even though they may have been derived differently. In this case, one approach is to use the set identity theorems that we enumerated in the last section. Suppose, for example, that we want to prove that:

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

We can prove this as follows:

$$\begin{aligned} A \cup (B \cap (A \cap C)) &= (A \cup B) \cap (A \cup (A \cap C)). && \text{(Distributivity)}\\ &= (A \cup B) \cap ((A \cap C) \cup A). && \text{(Commutativity)}\\ &= (A \cup B) \cap A. && \text{(Absorption)}\\ &= A. && \text{(Absorption)} \end{aligned}$$

Sometimes, even when we're comparing apples to apples, the theorems we’ve listed aren’t enough. In these cases, we need to use the definitions of the operators. Suppose, for example, that we want to prove that:

$$A - B = A \cap \neg B.$$

We can prove this as follows (where U stands for the universe with respect to which we take complement):

$$\begin{aligned} A - B &= \{x : x \in A \text{ and } x \notin B\}.\\ &= \{x : x \in A \text{ and } (x \in U \text{ and } x \notin B)\}.\\ &= \{x : x \in A \text{ and } x \in U - B\}.\\ &= \{x : x \in A \text{ and } x \in \neg B\}.\\ &= A \cap \neg B. \end{aligned}$$

Sometimes, though, our problem is more complex. We may need to compare apples to oranges. In other words, we may need to compare sets that aren't even defined in the same terms. For example, we will want to be able to prove that A: {the set of languages that can be defined using regular expressions} is the same as B: {the set of languages that can be accepted by a finite state automaton}. This seems very hard: regular expressions, which we describe in Chapter 6, are strings that look like:

```
a* (b ∪ ba)*
```

Finite state machines, which we describe in Chapter 5, are a collection of states and rules for moving from one state to another. How can we possibly prove that A (defined in terms of regular expressions) and B (defined in terms of finite state machines) are the same set? The answer is that we can show that any two sets are equal by showing that each is a subset of the other. So, to prove that $A = B$, we will show first that, given a regular expression, we can construct a finite state machine that accepts exactly the strings that the regular expression describes. That gives us $A \subseteq B$. But there might still be some finite state machines that don't correspond to any regular expressions. So we then show that, given a finite state machine, we can construct a regular expression that defines exactly the same strings that the machine accepts. That gives us $B \subseteq A$. In Section 6.2, we describe both of these proofs and use them to prove the claim, called Kleene's Theorem, that $A = B$. We will use the same technique several more times throughout the book.

32.6.8 Showing That a Set is Finite or Countably Infinite

Next, let's return briefly to the question, "What is the cardinality of a set?" In this book, we will be concerned with three cases:

- finite sets,
- countably infinite sets, and
- uncountably infinite sets.

We will use the following definitions for the terms "finite" and "infinite":[14]

A set A is ***finite*** and has cardinality $n \in \mathbb{N}$ (the natural numbers) iff either $A = \varnothing$ or there is a bijection from $\{1, 2, \ldots n\}$ to A, for some value of n. Alternatively, a set is finite if we can count its elements and finish. The cardinality of a finite set is simply a natural number whose value is the number of elements in the set.

A set is ***infinite*** iff it is not finite. The first infinite set we'll consider is $\mathbb{N}$, the natural numbers. Following Cantor, we'll call the cardinality of $\mathbb{N}$ $\aleph_0$. (Read this as "aleph null". Aleph is the first symbol of the Hebrew alphabet.)

Now consider an arbitrary set A. We'll say that A is ***countably infinite*** and also has cardinality $\aleph_0$ iff there exists some bijection f: $\mathbb{N} \rightarrow A$. And we need one more definition: a set is ***countable*** iff it is either finite or countably infinite. We use the term "countable" because the elements of a countable set can be counted with the integers.

To prove that a set A is countably infinite, it suffices to find a bijection from $\mathbb{N}$ to it.

Example 32.21 There is a Countably Infinite Number of Even Numbers

The set E of even natural numbers is countably infinite. To prove this, we offer the bijection:

$$Even : \mathbb{N} \rightarrow E,$$
$$Even(x) = 2x.$$

So we have the following mapping from $\mathbb{N}$ to E:

[14] An alternative is to begin by saying that a set A is infinite iff there exists a one-to-one mapping from A into a proper subset of itself. Then a set is finite iff it is not infinite. With the axiom of choice, these two defintions are equivalent.

$\mathbb{N}$	E
0	0
1	2
2	4
3	6
…	…

The last example was easy. The bijection was obvious. Sometimes it is less so. In harder cases, a good way to think about the problem of finding a bijection from $\mathbb{N}$ to some set A, is to turn it into the problem of finding an enumeration of A.

An ***enumeration*** of a set A is simply a list of the elements of A in some order. Each element of A must occur in the enumeration exactly once. Of course, if A is infinite, the enumeration will be infinite. But as long as we can guarantee that every element of A will show up eventually, we have an enumeration.

Theorem 32.1 Infinite Enumeration and Countable Infinity

Theorem: A set A is countably infinite iff there exists an infinite enumeration of it.

Proof: We prove the if and only-if parts separately.

If* A *is countably infinite, then there exists an infinite enumeration of it: Since A is countably infinite, there exists a bijection f from $\mathbb{N}$ to it. We construct an infinite enumeration of A as follows (where the only slight issue is that we number the elements of an enumeration starting with 1 and the natural numbers start with 0): For all $i \geq 1$, the i^{th} element of the enumeration of A will be $f(i - 1)$. So the first element of the enumeration will be the element that 0 maps to, the second element of the enumeration will be the element that 1 maps to, and so forth.

If there exists an infinite enumeration* E *of* A*, then* A *is countably infinite: Define f: $\mathbb{N} \rightarrow A$, where $f(i)$ is the $(i+1)^{st}$ element of the list E. The function f is a bijection from $\mathbb{N}$ to A, so A is countably infinite.

■

We can use Theorem 32.1 both to show that a set is countably infinite (by exhibiting an infinite enumeration of it) and to show that a set is not countably infinite (by showing that no infinite enumeration of it exists).

Theorem 32.2 Finite Union

Theorem: The union U of a finite number of countably infinite sets is countably infinite.

Proof: The proof is by enumeration of the elements of U. We need a technique for producing that enumeration. The simplest thing to do would be to start by enumerating all the elements of the first set, then all the elements of the second, etc. But, since the first set is infinite, we will never get around to considering any of the elements of the other sets. So we need another technique. We take the first element from each of the sets, then the second element from each, and so forth, checking before inserting each element to make sure that it is not already there.

■

Using a technique similar to the one we just used to prove Theorem 32.2, it is easy to show that, for any fixed n, the set of ordered n-tuples of elements drawn from a countably infinite set must also be countably infinite. So, for example, the rational numbers are countably infinite.

Theorem 32.3 Countably Infinite Union

Theorem: The union U of a countably infinite number of countably infinite sets is countably infinite.

Proof: The proof is by enumeration of the elements of U. Now we cannot use the simple enumeration technique that we used in the proof of Theorem 32.2. Since we are now considering an infinite number of sets, if we tried that technique we'd never get to the second element of any of the sets. So we follow the arrows as shown in Figure 32.9.

The numbers in the squares indicate the order in which we select elements for the enumeration. This process goes on forever, but it is systematic and it guarantees that, if we wait long enough, any element of any of the sets will eventually be enumerated. Note that, before we actually enter any element into the enumeration, we must check to make sure that it has not already been generated.

	Set 1	Set 2	Set 3	Set 4	…
Element 1	1	2	4	7	
Element 2	3	5	8		
Element 3	6	9			
…	10				

Figure 32.9 Systematically enumerating the elements of an infinite number of infinite sets

■

It turns out that there are a lot of countably infinite sets. Some of them, like the even natural numbers, appear at first to contain fewer elements than $\mathbb{N}$ does. Some of them, like the union of a countable number of countable sets, appear at first to be bigger. But in both cases there is a bijection from $\mathbb{N}$ to the elements of the set, so the cardinality of the set is $\aleph_0$.

32.6.9 Showing That a Set is Uncountably Infinite: Diagonalization

But not all infinite sets are countably infinite. There are sets with more than $\aleph_0$ elements. There are more than $\aleph_0$ real numbers, for example. As another case, consider an arbitrary set S with cardinality $\aleph_0$. Now consider $\mathcal{P}(S)$ (the power set of S). $\mathcal{P}(S)$ has cardinality greater than $\aleph_0$. To prove this, we need to show that, although $\mathcal{P}(S)$ is infinite, there exists no bijection from $\mathbb{N}$ to it. To do this, we will use a technique called ***diagonalization***.

Diagonalization is a kind of proof by contradiction. To show that a set A is not countably infinite, we assume that it is, in which case there would be some enumeration of it. Every element of A would have to be on that list somewhere. But we show how to construct an element of A that cannot be on the list, no matter how the list was constructed. Thus there exists no enumeration of A. So A is not countably infinite.

Theorem 32.4 The Cardinality of the Power Set

Theorem: If S is a countably infinite set, $\mathcal{P}(S)$ (the power set of S) is infinite but not countably infinite.

Proof: $\mathcal{P}(S)$ must be infinite because, for each of the infinitely many elements s of S, the set $\{s\}$ is an element of $\mathcal{P}(S)$.

But now we must prove that $\mathcal{P}(S)$ is not countably infinite. The proof is by diagonalization. Since S is countably infinite, by Theorem 32.1, there exists an infinite enumeration of it. We can use that enumeration to construct a representation of each subset SS of S as an infinite binary vector that contains one element for each element of the original set S. If SS contains element 1 of S, then the first element of its vector will be 1, otherwise 0 (which we'll show as blank to make our tables easy to read). Similarly for all the other elements of S. Of course, since S is countably infinite, the length of each vector will also be countably infinite. Thus we might represent a particular subset SS of S as the infinite vector shown in Figure 32.10 (a).

Now, assume that $\mathcal{P}(S)$ is countably infinite. Then there is some enumeration of it. Pick any such enumeration, and write it as shown in Figure 32.10 (b) (where each row represents one element of $\mathcal{P}(S)$ as described above. Ignore for the moment the numbers enclosed in parentheses.) This table is infinite in both directions. Since it is an enumeration of $\mathcal{P}(S)$, it must contain one row for each element of $\mathcal{P}(S)$. But it doesn't. To prove that it doesn't, we will construct L, an element of $\mathcal{P}(S)$ that is not on the list. To do this, consider the numbers in parentheses along the diagonal of the matrix of Figure 32.10 (b). Using them, we can construct L so that it corresponds to the vector shown in Figure 32.10 (c). What we mean by $\neg(1)$ is that if the square labeled (1) is a 1 then 0; if the square labeled (1) is a 0, then 1.

Elem 1 of *S*	Elem 2 of *S*	Elem 3 of *S*	Elem 4 of *S*	Elem 5 of *S*	Elem 6 of *S*	……
1			1	1		……

(a)

	Elem 1 of *S*	Elem 2 of *S*	Elem 3 of *S*	Elem 4 of *S*	Elem 5 of *S*	Elem 6 of *S*	…….
Elem 1 of $\mathcal{P}(S)$	1 (1)						…..
Elem 2 of $\mathcal{P}(S)$		1 (2)					…..
Elem 3 of $\mathcal{P}(S)$	1	1	(3)				…..
Elem 4 of $\mathcal{P}(S)$			1	(4)			…..
Elem 5 of $\mathcal{P}(S)$	1		1		(5)		…..
Elem 6 of $\mathcal{P}(S)$	0	1	1			(6)	…..
…							…..

(b)

¬(1)	¬(2)	¬(3)	¬(4)	¬(5)	¬(6)	…..

(c)

Figure 32.10 Using diagonalization to show the uncountability of a power set

So we've constructed the representation for an element of $\mathcal{P}(S)$. It must be an element of $\mathcal{P}(S)$ since it describes a possible subset of *S*. But we've built it so that it differs from the first element in the list of Figure 32.10 (b) by whether or not it includes element 1 of *S*. It differs from the second element in the list by whether or not it includes element 2 of *S*. And so forth. In the end, it must differ from every element in the list in at least one place. Yet it represents an element of $\mathcal{P}(S)$. Thus we have a contradiction. The list was not an enumeration of $\mathcal{P}(S)$. But since we made no assumptions about it except that it was an enumeration of $\mathcal{P}(S)$, no such enumeration can exist. In particular, if we try to fix the problem by simply adding our new element *L* to the list, we can just turn around and do the same thing again and create yet another element that is not on the list. Thus there are more than $\aleph_0$ elements in $\mathcal{P}(S)$.

■

If a set *S* is infinite but not countably infinite then we will say that it is ***uncountably infinite***. So, for example, $\mathcal{P}(\mathbb{N})$ is uncountably infinite, since, by Theorem 32.4, the power set of any countably infinite set is infinite but not countably infinite. The real numbers are uncountably infinite, which can be shown with a proof that is very similar to the one we just did for the power set except that it is a bit tricky because, when we write out each number as an infinite sequence of digits (just as we wrote out each set above as an infinite sequence of 0's and 1's), we have to consider the fact that several distinct sequences may represent the same number.

Not all uncountably infinite sets have the same cardinality. There are more elements in the power set of the real numbers than there are real numbers, for example.

32.7 Reasoning about Programs

An ***algorithm*** is a detailed procedure that accomplishes some clearly specified task. A ***program*** is an executable encoding of an algorithm. Not all algorithms halt. For example, a monitoring system might be designed never to halt but to run constantly, looking for some pattern of events to which some sort of response is required. So not all programs are designed to halt. However, we will focus on the class of programs whose job is to accept input, compute, and halt, having produced appropriate output. Useful programs of this sort possess two kinds of properties:

- correctness properties, including:
  - The program eventually halts.
  - When it halts, it has produced the desired output.
- performance properties, including:
  - time requirements, and
  - space requirements.

Entire books have been written on techniques for proving these properties. We summarize here just the few techniques that we will find the most useful in the rest of this book.

32.7.1 Proving Correctness Properties

We will first consider the problem of proving that a program halts. Then we'll look at techniques that can be used to show that a program's result satisfies its specification.

Proving that a Program Halts

When we describe a program to solve a problem, we would like to be able to prove that the program always halts. One of the main results of the theory that we will develop in this book is that there can exist no algorithm to solve the halting problem, which we can state as, "Answer the following question: Given the text of some program *M* and some input *w*, does *M* halt on input *w*?" So there can exist no general purpose algorithm that considers an arbitrary program and determines whether or not it halts on even one input, much less on all inputs. However, that does not mean that there are not particular programs that can be shown to halt on all inputs.

Any program that has no loops and no recursive function calls halts when it reaches the end of its code. So we focus our attention on proving that loops and recursive functions halt. In a nutshell, any such proof must show that the loop or the recursion executes some finite number of steps. Sometimes, particularly in the case of for loops, we can simply state the maximum number of steps.

Example 32.22 Termination of a For Loop

Consider the following very simple program *P*:

```
$P$(some arguments) =
    For $i$ = 1 to 10 do:
            Compute something.
```

As long as the compute step of *P* does not modify *i*, we can safely claim that this loop executes at most 10 times. (It could possibly execute fewer if it exits prematurely.)

When dealing with while and until loops and with recursive functions, it may not be possible to make such a straightforward statement. In proving that any such program *P* halts, we will generally rely on the existence of some well-founded set (*S*, *R*) such that:

- There exists some bijection between each step of *P* and some element of the set *S*,
- The first step of *P* corresponds to a maximal (with respect to *R*) element of *S*,
- Each successive step of *P* corresponds to a smaller (with respect to *R*) element of *S*, and
- *P* halts on or before it executes a step that corresponds to a minimal (with respect to *R*) element of *S*.

Example 32.23 Choosing a Well-Founded Set

Consider the following simple program P that acts on a finite-length string:

```
$P(s$: string) =
    While $length(s) > 0$ do:
            Remove the first character from $s$ and call it $c$.
            if $c$ = a return $True$.
    Return $False$.
```

Let $S = \{0, 1, 2, \ldots, |s|\}$. $(S, \leq)$ is a well-founded set whose least element is 0. Associate each step of the loop with $|s|$ as the step is about to be executed. The first pass through the loop is associated the initial length of s, which is the maximum value of $|s|$ throughout the computation. $|s|$ is decremented by one each time through the loop. P halts when $|s|$ is 0 or before (if it finds the character a). So the maximum number of times the loop can be executed is the initial value of $|s|$.

If we cannot find a well-founded set that corresponds to the steps of a loop or a recursively defined function, then it is likely that that program fails to halt on at least some inputs.

Example 32.24 When We Can't Find a Well-Founded Set

Consider the following program P, along with the claim that, given some positive integer n, P always halts and finds and prints the square root of n:

```
$P(n$: positive integer) =
    $r = 0$.
    Until $r * r = n$ do:
            $r = r + 1$.
    Print($r$).
```

We could try to prove that P always halts by using the well-founded set $(\mathbb{N}, \leq)$. Associate each step of the loop with $n - r^2$. On entrance to the loop, this difference must be in $\mathbb{N}$ since n is in $\mathbb{N}$ and $r = 0$. The difference decreases at each step through the loop, as r increases. If r ever equals the square root of n, the difference will be 0 and the loop will terminate. But, if n is not a perfect square, there is no guarantee that the difference will not simply become more and more negative. So there is no bijection between $n - r^2$ and $\mathbb{N}$. There is one between $n - r^2$ and Z (the integers), but Z has no minimal element and so is not well-founded. As it turns out, there is no well-founded set that can be put in one-to-one correspondence with the steps of this loop, which cannot be guaranteed to halt.

Proving that a Program Computes the Correct Result

Given that a program halts, does it halt with the correct result? We will find two techniques particularly useful for proving that it does:

- loop invariants, which we will introduce briefly here.
- induction, which we reviewed in Section 32.6.6.

Often the most straightforward way to analyze any sort of iterated process is to focus not on what the process does but rather on what it doesn't do. So we'll describe some key property that does not change at any step of the process's execution.

Example 32.25 The Coffee Can Problem

Consider the following problem, which we'll call the ***coffee can problem*** [Gries 1989]: we have a coffee can that contains some white beans and some black beans. We perform the following operation on the beans:

```
Until no further beans can be removed do:
    Randomly choose two beans.
```

If the two beans are the same color then throw both of them away and add a new black bean.
If the two beans are different colors then throw away the black one and return the white one to the can.

It is easy to show that this process must halt. After each step, the number of beans in the can decreases by one. When only one bean remains, no further beans can be removed.

But what can we say about the one remaining bean? Is it white or black? The answer is that if the original number of white beans is odd, the remaining bean is white. Otherwise the remaining bean is black. To see why this is true, we note that our bean culling process preserves white bean parity. In other words, if the number of white beans starts out even, it stays even. If the number of white beans starts out odd, it stays odd. To prove that this is so, we consider each action that the culling process can perform. There are three:

- Two white beans are removed and one black bean is added.
- Two black beans are removed and one black bean is added.
- One black bean is removed.

In each of these, an even number of white beans is removed and white bean parity is preserved. So, if the number of white beans is initially odd, the number of white beans can never become zero and a white bean must be the sole survivor. If, on the other hand, the number of white beans is initially even, it can never become one. Thus any sole survivor must be black.

The white bean parity property that we just described is an example of a ***loop invariant***: a predicate I that describes a property that doesn't change during the execution of an iterative process. To use a loop invariant to prove the correctness of a program, we must prove each of the following:

- I is true on entry to the loop.
- The truth of I is maintained at each pass through the loop. By this we mean that, if I is true at the beginning of a particular pass through the loop, then it must also be true at the end of that pass. Note, however, that I may fail to hold at some point partway through the loop.
- I, together with the loop termination condition, imply whatever property we wish to prove is true on exit from the loop.

Example 32.26 Finding a Loop Invariant

Consider the following program P:

```
P(s: string) =
        count = 0.
        For i = 1 to length(s) do:
            If the $i^{th}$ character of s is a then count = count + 1.
        Print (count).
```

Prove that the value of *count*, on exit from the loop, is the number of a's in s. Call this claim C. We will use a loop invariant to prove C.

We'll use the notation $\#_a(s)$ to mean the number of a's in the string s. Let:

$$I = [\#_a(s) = count + \#_a(\text{the last } (length(s) + 1 - i) \text{ characters of } s)].$$

In other words, the total number of a's in s is equal to the current value of *count* plus the number of a's in that part of s that has not so far been examined by the loop.

We show:

- *I* is true on entry to the loop: *i* = 1 and *count* = 0. So we have:

  $\#_{\mathrm{a}}(s) = 0 + \#_{\mathrm{a}}$(the last (*length*(*s*)) characters of *s*), which is true.

- *I* is maintained at each step through the loop: If the i^{th} character of *s* is an a, then *count* is incremented by 1. But *i* is also incremented, so the number of a's in the last (*length*(*s*) +1 – *i*) characters of *s* is decremented by 1, leaving *count* + $\#_{\mathrm{a}}$(the last (*length*(*s*) +1 – *i*) characters of *s*) unchanged. If the i^{th} character of *s* is not an a, then the value of both *count* and the number of a's in the last (*length*(*s*) +1 – *i*) remains unchanged.

- *I*, together with the loop termination condition, imply *C*: on exit from the loop, *i* = *length*(*s*) + 1. So we have:

$$\begin{aligned} I \wedge [i = \mathit{length}(s) + 1] &\equiv [\#_{\mathrm{a}}(s) = \mathit{count} + \#_{a}(\text{the last } (\mathit{length}(s) + 1 - i) \text{ characters of } s)] \wedge [i = \mathit{length}(s) + 1]. \\ &\equiv [\#_{\mathrm{a}}(s) = \mathit{count} + \#_{a}(\text{the last } (\mathit{length}(s) + 1 - (\mathit{length}(s) + 1) \text{ characters of } s]. \\ &\equiv [\#_{\mathrm{a}}(s) = \mathit{count} + \#_{a}(\text{the last 0 characters of } s]. \\ &\equiv \#_{\mathrm{a}}(s) = \mathit{count}. \end{aligned}$$

So, on exit from the loop, *count* is equal to the number of a's in *s*. Thus *C* is true. Note that a separate proof is required to show that the loop does in fact terminate.

Example 32.27 Finding a Loop Invariant for a Program that Doesn't Halt

Consider the following program *P*, which differs from our other examples since it is not designed to halt:

```
P() =
    s = "".
    Loop:
        Print(s).
        s = s || a.
```

Prove that *P* will print all and only the finite length strings composed of 0 or more a's (and no other characters). We will use a loop invariant to prove that *P* prints only strings composed exclusively of a's. We will use induction to prove that *P* will eventually print all strings composed only of a's.

The loop invariant we need is *I* = [*s* contains no characters other than a].

We show:

- *I* is true on entry to the loop the first time: *s* is the empty string and so contains no characters that are not a.

- *I* is maintained at each step through the loop: *s* is unchanged through the loop except to have a single a added to the end of it. So if it contained only a's at the top of the loop, it will contain only a's at the bottom.

- We are not concerned with what happens when the loop in *P* terminates, since it doesn't. So we can skip the step in which we show that some statement is true on exit from the loop.

Since *I* must be true at the top of the loop, it is true when the print statement executes, so only strings composed exclusively of a's will be printed.

Now we need to show that *P* will eventually print any string *s* that is composed of no characters other than a. We do this by induction on |*s*|:

- Base step: Let |*s*| = 0. *P* prints *s* the first time through the loop.

- Induction hypothesis: P prints all strings of a's of length n. Note that, for any value of n, there is only one such string. Call it $\mathtt{a}^n$.

- Prove that P prints all strings of a's of length $n + 1$. There is only one such string, namely $\mathtt{a}^n\mathtt{a}$. By the induction hypothesis, P generates $\mathtt{a}^n$. When it does that, the variable s is equal to $\mathtt{a}^n$. The next thing P does is to concatenate one more a onto s, which then equals $\mathtt{a}^n\mathtt{a}$, and print it.

So, for all $n \geq 0$, P prints the string composed exactly of n a's.

32.7.2 Analyzing Complexity

Whenever we present a program P, we may want to ask the question, "How long will it take P to run?" Generally the answer depends on the size of the input. So our answer will usually be stated as a function of some number that corresponds to a reasonable measure of the size of the input. If the input is a string, we can use the length of the string. If the input is a structure like a list or an array, we can use the number of elements in the structure. If the input is a number, we will typically use the length of the binary or decimal encoding of the number.

In Part V of this book we present a formal theory of complexity. Here we present an informal treatment of the approach that we will describe there.

We will describe the time complexity of a program P as a function of the size of its input, which we'll call n. We are typically not interested in how long it takes P to run on small inputs. Rather we are concerned with how quickly execution time grows as n grows. While in some cases we are concerned with an exact count of the number of steps that P must execute, we are often willing to ignore constant factors and instead to concentrate on whether P's execution time:

- is constant (i.e., it is independent of n),
- grows linearly with n,
- grows faster than n but at a rate that can be described by some polynomial function of n (for example, n^2), or
- grows at a rate that is faster than any polynomial function of n (for example 2^n).

Suppose that we have a program that, on input of length n, executes $n^3 + 2n + 3$ steps. As n increases, the n^3 term dominates the other two. So we would like to ignore the slower growing terms of the function $n^3 + 2n + 3$ and characterize the time required to execute this program as the simpler function n^3. To do that, we introduce the notion of asymptotic dominance of one function by another.

Let $f(n)$ and $g(n)$ be functions from the natural numbers to the positive reals. Then we'll say that the function $g(n)$ ***asymptotically dominates*** the function $f(n)$ iff there exists a positive integer k and a positive constant c such that:

$$\forall n \geq k\ (f(n) \leq c\ g(n)).$$

In other words, ignoring some number of small cases (all those of size less than k), and ignoring some constant factor c, $f(n)$ is bounded from above by $g(n)$.

We will use the symbol $\mathcal{O}$ to denote the asymptotic dominance relation, so $\mathcal{O}(g(n))$ is the set of all functions that are asymptotically dominated by $g(n)$. Thus, if $g(n)$ asymptotically dominates $f(n)$, we will write:

$$f(n) \in \mathcal{O}(g(n)).$$

This claim is read, "f is big-O of g". It is also often written $f(n) = \mathcal{O}(g(n))$, although that statement is not literally correct since $\mathcal{O}(g(n))$ is a set of functions, not a function.

Example 32.28 O

$n^3 + 2n + 3 \in \mathcal{O}(n^3)$, since we can let $k = 2$ and $c = 2$ and observe that for all $n \geq 2$, $n^3 + 2n + 3 \leq 2n^3$.

Now we can return to the problem of characterizing the execution time of a program P. Let $f(n)$ be a function that describes the time required to execute P as a function of n, where n is some reasonable measure of the size of P's input. We'll say that P runs in time $\mathcal{O}(g(n))$ iff $f(n) \in \mathcal{O}(g(n))$.

Example 32.29 Using $\mathcal{O}$ to Measure Time Complexity: A Linear Example

Consider again the program P from Example 32.26:

```
P(s: string) =
        count = 0.
        For i = 1 to length(s) do:
            If the $i^{th}$ character of s is a then count = count + 1.
        Print (count).
```

Let n = length(s). The number of program steps that P executes is at most $2 + 2n \in \mathcal{O}(n)$. So the execution time of P grows linearly in the length of its input.

Example 32.30 Using $\mathcal{O}$ to Measure Time Complexity: A Quadratic Example

Consider the following program P, which returns *True* if any two elements of its input vector are the same and *False* otherwise:

```
P(v: vector of integers) =
    For i = 1 to length(v) do:
        For j = i + 1 to length(v) do:
            If v[i] = v[j] then return True.
    Return False.
```

Let n = length(v). In the worst case, P goes through the outer loop n times. At each pass, unless it finds a match, it goes through the inner loop on average $n/2$ times. So the number of program steps that P executes is at most $1 + n(1 + 2n/2) = 1 + n + n^2 \in \mathcal{O}(n^2)$. So the execution time of P grows as the square of the length of its input.

Suppose that a program P, on input of size n, runs in time $f(n) = 2 + 4n$. Then $f(n) \in \mathcal{O}(n)$. But notice that it is also true that $f(n) \in \mathcal{O}(n^2)$ and $f(n) \in \mathcal{O}(2n)$, since both n^2 and $2n$ also asymptotically dominate $2 + 4n$. In Chapter 27 we will define Θ, a relation that is similar to $\mathcal{O}$ except that it is stricter. Specifically:

$$f(n) \in \Theta(g(n)) \text{ iff } f(n) \in \mathcal{O}(g(n)) \text{ and } g(n) \in \mathcal{O}(f(n)).$$

So $2 + 4n \in \Theta(n)$, but $2 + 4n \notin \Theta(n^2)$ because $n^2 \notin \mathcal{O}(n)$.

Discussions of the complexity of algorithms should use Θ, whenever possible, since we want the tightest bound we can find. But that is not the convention. As we did in both Example 32.29 and Example 32.30, we will use the standard convention of writing $f(n) \in \mathcal{O}(g(n))$ instead of $f(n) \in \Theta(g(n))$, but, whenever we can, we will choose values for $g(n)$ such that the claim that $f(n) \in \Theta(g(n))$ would also be true.

In analyzing the algorithms that we will consider in Parts II through IV of this book, we will use the $\mathcal{O}$ relation as we have just defined it. In Chapter 27, we will have more to say about $\mathcal{O}$ and similar relations such as Θ.

32.8 *A General Definition of Closure* ✸

In Section 32.5 we introduced closures. We elaborate on that discussion here. We begin by reviewing what we said there. Imagine some set *S* and some property *P*. If we care about making sure that *S* has property *P*, we could do the following:

1. Examine *S* for *P*. If it has property *P*, we quit.
2. If it doesn't, then add to *S* the smallest number of additional elements required to satisfy *P*.

We will say that *S* is closed with respect to *P* iff it possesses *P*. And, if we have to add elements to *S* in order to satisfy *P*, we'll call a smallest such expanded *S* that does satisfy *P* a closure of *S* with respect to *P*.

Example 32.31 Some Relations and Their Closures

1. Let *S* be a set of friends we are planning to invite to a party. Let *P* be, "*S* should include everyone who is likely to find out about the party" (since we don't want to offend anyone). Let's assume that if you invite Bill and Bill has a friend Bob, then Bill may tell Bob about the party. This means that if you want *S* to satisfy *P*, then you have to invite not only your friends, but your friends' friends, and their friends, and so forth. If you move in a fairly closed circle, you may be able to satisfy *P* by adding a few people to the guest list. On the other hand, it's possible that you'd have to invite the whole city before *P* would be satisfied. It depends on the connectivity of the *Friendof* relation in your social setting. The problem is that whenever you add a new person to *S*, you have to turn around and look at that person's friends and consider whether there are any of them who are not already in *S*. If there are, they must be added, and so forth. There is one positive feature of this problem, however. Notice that there is a unique set that does satisfy *P*, given the initial set *S*. There aren't any choices to be made.

2. Let *S* be a set of 6 people. Let *P* be, "*S* can enter a baseball tournament". This problem is different from the previous one in two important ways. First, there is a clear limit on how many elements we have to add to *S* in order to satisfy *P*. We need 9 people and when we've got them we can stop. But notice that there is not a unique way to satisfy *P* (assuming that we know more than 9 people). Any way of adding 3 people to *S* will work.

3. Let *S* be the *Address* relation (which we defined earlier as "lives at same address as"). Since relations are sets, we should be able to treat *address* just as we've treated the sets of people in our last two examples. We know that *Address* is an equivalence relation. So we'll let *P* be the property of being an equivalence relation (i.e., reflexive, symmetric, and transitive). But suppose we are only able to collect facts about living arrangements in a piecemeal fashion. For example, we may learn that *Address* contains {(Dave, Stacy), (Jen, Pete), (John, Bill)}. Immediately we know, because *Address* must be reflexive, that it must also contain {(Dave, Dave), (Stacy, Stacy), (Jen, Jen), (Pete, Pete), (John, John), (Bill, Bill)}. And, since *Address* must also be symmetric, it must contain {(Stacy, Dave), (Pete, Jen), (Bill, John)}. Now suppose that we discover that Stacy lives with Jen. We add {(Stacy, Jen)}. To make *Address* symmetric again, we must add {(Jen, Stacy)}. But now we also have to make it transitive by adding {(Dave, Jen), (Jen, Dave)}.

4. Let *S* be the set of positive integers. Let *P* be, "The sum of any two elements of *S* is also in *S*". Now we've got a property that is already satisfied. The sum of any two positive integers is a positive integer. This time, we don't have to add anything to *S* to establish *P*.

5. Let *S* again be the set of positive integers. Let *P* be, "The quotient of any two elements of *S* is also in *S*". This time we have a problem. 3/5 is not a positive integer. We can add elements to *S* to satisfy *P*. If we do, we end up with exactly the positive rational numbers.

To use closures effectively, we need to define precisely what we mean when we say that a set *S* is closed under *P* or that the closure of *S* under *P* is *T*. We present here a set of definitions that include all but one of the specific cases that we just described. The definitions of closure that we presented in Section 32.5 are special cases of the ones presented here. The one requirement that must be met in order to apply these definitions to a closure problem is that we must be able to describe the property *P* that is to be maintained as a relation.

Let *n* be an integer greater than or equal to 1. Let *R* be an *n*-ary relation on a set *A*. Thus elements of *R* are of the form (d_1, d_2, …, d_n). We say that a subset *S* of *A* is ***closed under*** *R* iff, whenever:

- $d_1, d_2, \ldots d_{n-1} \in S$ (all of the first n-1 elements are already in the set S), and
- $(d_1, d_2, \ldots d_{n-1}, d_n) \in R$ (the last element is related to the n-1 other elements via R).

it is also true that $d_n \in S$.

A set S' is a ***closure*** of S with respect to R (defined on A) iff:

- $S \subseteq S'$,
- S' is closed under R, and
- $\forall T$ (($S \subseteq T$ and T is closed under R) $\rightarrow |S'| \leq |T|$).

In other words, S' is a closure of S with respect to R if it is an extension (i.e., a superset) of S that is closed under R and if there is no smaller set that also meets both of those requirements. Note that we cannot say that S' must be the smallest set that will do the job, since we do not yet have any guarantee that there is a unique such smallest set (recall the softball example above).

These definitions of closure are a very natural way to describe our first example above. Drawing from a set A of people, you start with S equal to your friends. Then, to compute your invitee list S', you simply take the closure of S with respect to the relation *fFriendof*, which will force you to add to S' your friends' friends, their friends, and so forth.

These definitions also apply naturally to our fifth example, the positive integers under division. The smallest set that contains the positive integers and that is closed under division is the positive rationals. So the closure under division of the positive integers is the positive rationals.

Now consider our second example, the case of the baseball team. Here there is no relation R that specifies, if one or more people are already on the team, that some specific other person must also be on. The property we care about is a property of the team (set) as a whole and not a property of patterns of individuals (elements). Thus this example, although similar, is not formally an instance of closure as we have just defined it. This turns out to be significant and leads us to the following definition:

Any property that asserts that a set S is closed under some relation R is called a ***closure property*** of S.

Theorem 32.5 Closures Exist and are Unique

Theorem: If R is a closure property, as just defined, on a set A and S is a subset of A, then the closure of S with respect to R exists and is unique.

Proof: Omitted.

■

Stating the theorem another way, if its conditions are met then there exists a unique minimal set S' that contains S and is closed under R. Of all of our examples above, the baseball example is the only one that cannot be described in the terms of this definition of a closure property. The theorem that we have just stated (without proof) guarantees, therefore, that it will be the only one that does not have a unique minimal solution.

The definitions that we have just provided also work to describe our third example, in which we want to compute the closure of a relation (since, after all, a relation is a set). All we have to do is to come up with relations that describe the properties of being reflexive, symmetric, and transitive. To help us see what those relations need to be, let's recall our definitions of symmetry, reflexivity, and transitivity:

- A binary relation $R \subseteq A \times A$ is ***reflexive*** iff, for each $a \in A$, $(a, a) \in R$.
- A binary relation $R \subseteq A \times A$ is ***symmetric*** iff, whenever $(a, b) \in R$, so is (b, a).
- A binary relation $R \subseteq A \times A$ is ***transitive*** iff, whenever $(a, b) \in R$ and $(b, c) \in R$, $(a, c) \in R$.

Looking at these definitions, we can come up with three relations, *Reflexivity*, *Symmetry*, and *Transitivity*. All three are relations on relations, and they will enable us to define these three properties using the closure definitions we've given so far. All three definitions assume a base set A on which the relation that we are interested in is defined:

- For any a in A, $((a, a)) \in$ *Reflexivity* and no other elements are. Notice the double parentheses here. *Reflexivity* is a unary relation, where each element is itself an ordered pair. It doesn't really "relate" two elements. It is simply a list of ordered pairs. To see how it works to define reflexive closure, imagine a set $A = \{x, y\}$. Now suppose we start with a relation R on $A = \{(x, y)\}$. Clearly R isn't reflexive: the *Reflexivity* relation on A is $\{((x, x)), ((y, y))\}$. *Reflexivity* is a unary relation. So n, in the definition of closure, is 1. Consider the first element $((x, x))$. We consider all the components before the n^{th} (i.e., first) and see if they are in A. This means we consider the first zero components. Trivially, all zero of them are in A. So the n^{th} (the first) must also be. This means that (x, x) must be in R. But it isn't. So to compute the closure of R under *Reflexivity*, we add it. Similarly for (y, y).

- For any a and b in A, $a \neq b \rightarrow [((a, b), (b, a)) \in$ *Symmetry*] and no other elements are. This one is a lot easier. Again, suppose we start with a set $A = \{x, y\}$ and a relation R on $A = \{(x, y)\}$. Clearly R isn't symmetric: *Symmetry* on $A = \{((x, y), (y, x)), ((y, x), (x, y))\}$. But look at the first element of *Symmetry*. It tells us that for R to be closed under *Symmetry*, whenever (x, y) is in R, (y, x) must also be. But it isn't. To compute the closure of R under *Symmetry*, we must add it.

- For any a, b and c and in A, $[a \neq b \wedge b \neq c] \rightarrow [((a, b), (b, c), (a, c)) \in$ *Transitivity*] and no other elements are. Now we will exploit a ternary relation. Whenever the first two elements of it are present in some relation R, then the third must also be if R is transitive. This time, let's start with a set $A = \{x, y, z\}$ and a relation R on $A = \{(x, y), (y, z)\}$. Clearly R is not transitive: The *Transitivity* relation on A is $\{((x, y), (y, z), (x, z)), ((x, z), (z, y), (x, y)), ((y, x), (x, z), (y, z)), ((y, z), (z, x), (y, x)), ((z, x), (x, y), (z, y)), ((z, y), (y, x), (z, x))\}$. Look at the first element of it. Both of the first two components of it are in R. But the third isn't. To make R transitive, we must add it.

We can also describe the closure of the positive integers under division with a closure property: let A be the positive rationals, let S be the positive integers and let R be *Quotientclosure*, defined as:

- For any a, b and c and in A, $[a/b = c] \rightarrow [(a, b, c) \in$ *Quotientclosure*].

So there exists a unique closure of S with respect to *Quotientclosure*. In this case, that closure is A.

We now have a general definition of closure that makes it possible to prove the existence of a unique closure for any set and any relation R. The only constraint is that this definition works only if we can define the property we care about as an n-ary relation for some finite n. There are cases of closure where this is not possible, as we saw above in the baseball team example, but we will not consider them further.

32.9 Exercises

1) Prove each of the following:
   a) $((A \wedge B) \rightarrow C) \leftrightarrow (\neg A \vee \neg B \vee C)$.
   b) $(A \wedge \neg B \wedge \neg C) \rightarrow (A \vee \neg(B \wedge C))$.

2) List the elements of each of the following sets:
   a) $\mathcal{P}$({apple, pear, banana}).
   b) $\mathcal{P}(\{\texttt{a}, \texttt{b}\}) - \mathcal{P}(\{\texttt{a}, \texttt{c}\})$.
   c) $\mathcal{P}(\varnothing)$.
   d) $\{\texttt{a}, \texttt{b}\} \times \{1, 2, 3\} \times \varnothing$.
   e) $\{x \in \mathbb{N}: (x \leq 7 \wedge x \geq 7)\}$.
   f) $\{x \in \mathbb{N}: \exists y \in \mathbb{N}\ (y < 10 \wedge (y + 2 = x))\}$ (where $\mathbb{N}$ is the set of nonnegative integers).
   g) $\{x \in \mathbb{N}: \exists y \in \mathbb{N}\ (\exists z \in \mathbb{N}\ ((x = y + z) \wedge (y < 5) \wedge (z < 4)))\}$.

3) Prove each of the following:
   a) $A \cup (B \cap C \cap D) = (A \cup B) \cap (A \cup D) \cap (A \cup C)$.
   b) $A \cup (B \cap C \cap A) = A$.
   c) $(B \cap C) - A \subseteq C$.

4) Consider the English sentence, “If some bakery sells stale bread and some hotel sells flat soda, then the only thing everyone likes is tea.” This sentence has at least two meanings. Write two (logically different) first-order-logic sentences that correspond to meanings that could be assigned to this sentence. Use the following predicates: $B(x)$ is *True* iff x is a bakery; $S_B(x)$ is *True* iff x sells stale bread; $H(x)$ is *True* iff x is a hotel; $S_S(x)$ is *True* iff x sells flat soda; $L(x, y)$ is *True* iff x likes y; and $T(x)$ is *True* iff x is tea.

5) Let P be the set of positive integers. Let L = {A, B, …, Z} (i.e., the set of upper case characters in the English alphabet). Let T be the set of strings of one or more upper case English characters. Define the following predicates over those sets:

- For $x \in L$, $V(x)$ is *True* iff x is a vowel. (The vowels are A, E, I, O, and U.)
- For $x \in L$ and $n \in P$, $S(x, n)$ is *True* iff x can be written in n strokes.
- For $x \in L$ and $s \in T$, $O(x, s)$ is *True* iff x occurs in the string s.
- For $x, y \in L$, $B(x, y)$ is *True* iff x occurs before y in the English alphabet.
- For $x, y \in L$, $E(x, y)$ is *True* iff $x = y$.

   Using these predicates, write each of the following statements as a sentence in first-order logic:
   a) A is the only upper case English character that is a vowel and that can be written in three strokes but does not occur in the string STUPID.
   b) There is an upper case English character strictly between K and R that can be written in one stroke.

6) Choose a set A and predicate P and then express the set {1, 4, 9, 16, 25, 36, …} in the form:

   $\{x \in A : P(x)\}$.

7) Find a set that has a subset but no proper subset.

8) Give an example, other than one of the ones in the book, of a reflexive, symmetric, intransitive relation on the set of people.

9) Not equal (defined on the integers) is (circle all that apply): reflexive, symmetric, transitive.

10) In Section 32.3.3, we showed a table that listed the eight possible combinations of the three properties: reflexive, symmetric and transitive. Add antisymmetry to the table. There are now 16 possible combinations. Which combinations could some nontrivial binary relation posses? Justify your answer with examples to show the combinations that are possible and proofs of the impossibility of the others.

11) Using the definition of $\equiv_p$ (equivalence modulo p) that is given in Example 32.4, let R_p be a binary relation on $\mathbb{N}$, defined as follows, for any $p \geq 1$:
      $R_p = \{(a, b): a \equiv_p b\}$
   So, for example R_3 contains (0, 0), (6, 9), (1, 4), etc., but does not contain (0, 1), (3, 4), etc.
   a) Is R_p an equivalence relation for every $p \geq 1$? Prove your answer.
   b) If R_p is an equivalence relation, how many equivalence classes does it induce for a given value of p? What are they? (Any concise description is fine.)
   c) Is R_p a partial order? A total order? Prove your answer.

12) Let $S = \{w \in \{\texttt{a}, \texttt{b}\}^*\}$. Define the relation *Substr* on the set S to be $\{(s, t) : s \text{ is a substring of } t\}$.
   a) Choose a small subset of *Substr* and draw it as a graph (in the same way that we drew the graph of Example 32.5.
   b) Is *Substr* a partial order?

13) Let P be the set of people. Define the function:

   father-of: $P \rightarrow P$.
   father-of(x) = the person who is x's father

   a) Is *father-of* one-to-one?
   b) Is it onto?

14) Are the following sets closed under the following operations? If not, give an example that proves that they are not and then specify what the closure is.
   a) The negative integers under subtraction.
   b) The negative integers under division.
   c) The positive integers under exponentiation.
   d) The finite sets under Cartesian product.
   e) The odd integers under remainder, mod 3.
   f) The rational numbers under addition.

15) Give examples to show that
   a) The intersection of two countably infinite sets can be finite.
   b) The intersection of two countably infinite sets can be countably infinite.
   c) The intersection of two uncountable sets can be finite.
   d) The intersection of two uncountable sets can be countably infinite.
   e) The intersection of two uncountable sets can be uncountable.

16) Let $R = \{(1, 2), (2, 3), (3, 5), (5, 7), (7, 11), (11, 13), (4, 6), (6, 8), (8, 9), (9, 10), (10, 12)\}$. Draw a directed graph representing R^*, the reflexive, transitive closure of R.

17) Let $\mathbb{N}$ be the set of nonnegative integers. For each of the following sentences in first-order logic, state whether the sentence is valid, is not valid but is satisfiable, or is unsatisfiable. Assume the standard interpretation for < and >. Assume that f could be any function on the integers. Prove your answer.
   a) $\forall x \in \mathbb{N}\ (\exists y \in \mathbb{N}\ (y < x))$.
   b) $\forall x \in \mathbb{N}\ (\exists y \in \mathbb{N}\ (y > x))$.
   c) $\forall x \in \mathbb{N}\ (\exists y \in \mathbb{N}\ f(x) = y)$.

18) Let $\mathbb{N}$ be the set of nonnegative integers. Let A be the set of nonnegative integers x such that $x \equiv_3 0$. Show that $|\mathbb{N}| = |A|$.

19) What is the cardinality of each of the following sets? Prove your answer
   a) $\{n \in \mathbb{N} : n \equiv_3 0\}$.
   b) $\{n \in \mathbb{N} : n \equiv_3 0\} \cap \{n \in \mathbb{N} : n \text{ is prime}\}$.
   c) $\{n \in \mathbb{N} : n \equiv_3 0\} \cup \{n \in \mathbb{N} : n \text{ is prime}\}$.

20) Prove that the set of rational numbers is countably infinite.

21) Use induction to prove each of the following claims:
   a) $\forall n>0\ (\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6})$.

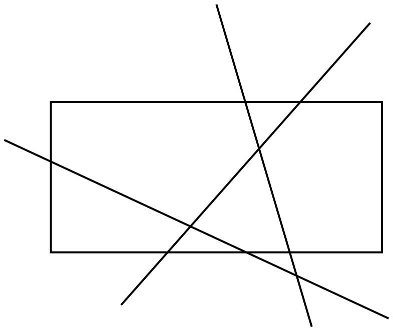
b) $\forall n>0$ ($n! \geq 2^{n-1}$). Recall that $0! = 1$ and $\forall n > 0$ ($n! = n(n-1)(n-2) \ldots 1$).

c) $\forall n>0$ ($\sum_{k=0}^{n} 2^k = 2^{n+1}-1$).

d) $\forall n \geq 0$ ($\sum_{k=0}^{n} r^k = \frac{r^{n+1}-1}{r-1}$), given $r \neq 0,1$.

e) $\forall n\geq 0$ ($\sum_{k=0}^{n} f_k^2 = f_n \cdot f_{n+1}$), where f_n is the n^{th} element of the Fibonacci sequence, as defined in Example 24.4.

22) Consider a finite rectangle in the plane. We will draw some number of (infinite) lines that cut through the rectangle. So, for example, we might have:



In Section 28.7.6, we define what we mean when we say that a map can be colored using two colors. Treat the rectangle that we just drew as a map, with regions defined by the lines that cut through it. Use induction to prove that, no matter how many lines we draw, the rectangle can be colored using two colors.

23) Let $div_2(n) = \lfloor n/2 \rfloor$ (i.e., the largest integer that is less than or equal to $n/2$). Alternatively, think of it as the function that performs division by 2 on a binary number by shifting right one digit. Prove that the following program correctly multiplies two natural numbers. Clearly state the loop invariant that you are using.

```
mult(n, m: natural numbers) =
        result = 0.
        While $m \neq 0$ do
                If odd(m) then result = result + n.
                $n = 2n$.
                $m = div_2(m)$.
```

24) Prove that the following program computes the function *double*(*s*) where, for any string *s*, *double*(*s*) = *True* if *s* contains at least one pair of adjacent characters that are identical and *False* otherwise. Clearly state the loop invariant that you are using.

```
double(s: string) =
        found = False.
        for i = 1 to length(s) - 1 do
                if s[i] = s[i + 1] then found = True.
        return(found).
```