3
Logic

In which our heroes move carefully through the marsh, making sure that each step follows safely from the one before it.

3.1 *Why You Might Care*

How fondly dost thou reason!

William Shakespeare (1564–1616)
The Comedy of Errors

Logic is the study of truth and falsity, of theorem and proof, of valid reasoning in any context. In this chapter, we focus on *formal logic*, in which it is the "form" of the argument that matters, rather than the "content." This chapter will introduce the two major types of formal logic:

- *propositional logic* (Sections 3.2 and 3.3), in which we will study the truth and falsity of statements, how to construct logical statements from basic logical operators (like **and** and **or**), and how to reason about those statements.

- *predicate logic* (Sections 3.4 and 3.5), which gives us a framework to write logical statements of the form "every x ..." or "there's some x such that ...."

One of our main goals in this chapter will be to define a precise, formal, and unambiguous language to express reasoning—in which writer and reader agree on what each word means.

Logic is the foundation of all of computer science; it's the reasoning that you use when you write the condition of an **if** statement or when you design a circuit to add two 32-bit integers or when you design a program to beat a grandmaster at chess. Because logic is the study of valid reasoning, any endeavor in which one wishes to state and justify claims rigorously—such as that of this book—must at its core rely on logic. Every condition that you write in a loop is a logical statement. When you sit down to write binary search in Python, it is through a (perhaps tacit) use of logical reasoning that you ensure that your code works properly for any input. When you use a search engine to look for web pages on the topic "beatles and not john or paul or george or ringo" you've implicitly used logical reasoning to select this particular query. Solving a Sudoku puzzle is nothing more and nothing less than following logical constraints to their conclusion. The central component of a natural language processing (NLP) system is to take an utterance by a human user that's made in a "natural" language like English and "understand" what it means—and understanding what a sentence means is essentially the same task as understanding the circumstances under which the sentence is true, and thus is a question of logic.

And these are just a handful of examples; for a computer scientist, logic is the basis of the discipline. Indeed, the processor of a computer is built up from almost unthinkably simple logical components: wires and physical implementations of logical operations like **and**, **or**, and **not**. Our main goal in this chapter will be to introduce the basic constructs of logic. But along the way, we will encounter applications of logic to natural language processing, circuits, programming languages, optimizing compilers, and building artificially intelligent systems to play chess and other games.

3.2 *An Introduction to Propositional Logic*

> Everyone wishes to have truth on his side, but not everyone wishes to be on the side of truth.
>
> Richard Whately (1787–1863)

A *proposition* is a statement that is either true or false—*In December 2012, Facebook had over one billion users* or *Java is a programming language that uses indentation to denote block structure,* for example. *Propositional logic* is the study of propositions, including how to formulate statements as propositions, how to evaluate whether a proposition is true or false, and how to manipulate propositions. The goal of this section is to introduce propositions—including related terminology, standard notation, and some techniques for reasoning about propositions.

3.2.1 *Propositions and Truth Values*

We'll begin, briefly, with propositions themselves:

Definition 3.1 (Propositions and Truth Values)
A proposition *is a statement that is either true or false. For a particular proposition p, the* truth value *of p is its truth or falsity.*

A proposition is also sometimes called a *Boolean expression* or a *Boolean formula*. (See Section 2.2.1.) A proposition is written in English as a declarative sentence, the kind of sentence that usually ends with a period. (Questions and demands—like *Did you try binary search?* or *Use quicksort!*—aren't the kinds of things that are true or false, and so they're not propositions.) Here are a few examples:

Example 3.1 (Some sample propositions)
The following statements are all propositions:

1. $2+2=4$.
2. 33 is a prime number.
3. Barack Obama is the 44th person to be president of the United States.
4. Every even integer greater than 2 can be written as the sum of two prime numbers.

(The last of these propositions is called *Goldbach's conjecture*; it's more complicated than the other propositions in this example, and we'll return to it in Section 3.4.)

Let's determine the above propositions' truth values:

Example 3.2 (Determining truth values)
Problem: What are the truth values of the propositions from Example 3.1?

Solution: These propositions' truth values are

1. **True.** It really is the case that $2 + 2$ equals 4.
2. **False.** The integer 33 is not a prime number because $33 = 3 \cdot 11$. (Prime numbers are evenly divisible only by 1 and themselves; 33 is evenly divisible by 3 and 11.)
3. **False.** Although Barack Obama is called president #44, Grover Cleveland was president #22 *and* #24. So Barack Obama is actually the *43rd* person to be president of the United States, not the 44th.
4. **Unknown (!).** Goldbach's conjecture was first made in 1742, but has thus far resisted proof—or disproof! It's easy to check that particular small even integers can be written as the sum of two prime numbers; for example, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 3 + 7$, and so on. But is it true for *all* even integers greater than 2? We simply don't know! Many even integers have been tested, and no violation has been found in any of these tests. But, as far as we know, the next even integer we test *can't* be written as the sum of two primes. See Example 3.47 and Exercises 3.178–3.181.

Before we move on from Example 3.2, there's an important point to make about statements that have an unknown truth value. Even though we don't *know* the truth value of Goldbach's conjecture, it is still a proposition and thus it *has* a truth value. That is, Goldbach's conjecture is indeed either true or false; it's just that we don't know which it is. (Like the proposition *The person currently sitting next to you is wearing clean underwear:* it has a truth value, you just don't know what truth value it has.)

> **Taking it further:** Goldbach's conjecture stands in contrast to declarative sentences whose truth is ill-defined—for example, *This book is boring* and *Logic is fun*. Whether these claims are true or false depends on the (imprecise) definitions of words like *boring* and *fun*. We're going to de-emphasize subtle "shades of truth" questions of this form throughout the book, but see p. 314 for some discussion, including the role of ambiguity in software systems that interact with humans via English language input and output.
>
> There is also a potentially interesting philosophical puzzle that's hiding in questions about the truth values of natural-language utterances. Here's a silly (but obviously true) statement: *The sentence "snow is white" is true if and only if snow is white.* (Of course!) This claim becomes a bit less trivial if the embedded proposition is stated in a different language—Spanish or Dutch, say: *The sentence "La nieve es blanca" is true if and only if snow is white*; or *The sentence "Sneeuw is wit" is true if and only if snow is white.* But there's a troubling paradox lurking here. Surely we would like to believe that the English sentence *x* and the French translation of the English sentence *x* have the same truth value. For example, *Snow is white* and *La neige est blanche* surely are both true, or they're both false. (And, in fact, it's the former.) But this belief leads to a problem with certain self-referential sentences: for example, *This sentence starts with a 'T'* is true, but *Cette phrase commence par un 'T'* is, surely, false.[1]

For more on paradoxes and puzzles of translation, see

3.2.2 *Atomic and Compound Propositions*

We will distinguish between two types of propositions, those that cannot be broken down into conceptually simpler pieces and those that can be:

Definition 3.2 (Atomic and compound propositions)
An atomic proposition *is a proposition that is conceptually indivisible. A* compound proposition *is a proposition that is built up out of conceptually simpler propositions.*

[1] Douglas Hofstadter. *Le Ton Beau de Marot: In Praise of the Music of Language*. Basic Books, 1998; and R. M. Sainsbury. *Paradoxes*. Cambridge University Press, 3rd edition, 2009.

Here's a simple example of the difference:

Example 3.3 (Atomic and compound propositions)
The University of Minnesota's mascot is the Badger is an atomic proposition, because it is not conceptually divisible into any simpler claim.

The University of Washington's mascot is the Duck or the University of Oregon's mascot is the Duck is a compound proposition, because it is conceptually divisible into two simpler claims—namely *The University of Washington's mascot is the Duck* and *The University of Oregon's mascot is the Duck.*

Atomic propositions are also sometimes called *Boolean variables*; see Section 2.2.1. A compound proposition that contains Boolean variables $p_1, \ldots, p_k$ is sometimes called a *Boolean expression* or *Boolean formula over* $p_1, \ldots, p_k$.

Example 3.4 (Password validity as a compound proposition)
A certain small college sends the following instructions to its users when they are required to change their password:

> Your password is valid only if it is at least 8 characters long, you have not previously used it as your password, and it contains at least three different types of characters (lowercase letters, uppercase letters, digits, non-alphanumeric characters).

This compound proposition involves seven different atomic propositions:

- p: the password is valid
- q: the password is at least 8 characters long
- r: the password has been used previously by you
- s: the password contains lowercase letters
- t: the password contains uppercase letters
- u: the password contains digits
- v: the password contains non-alphanumeric characters

The form of the compound proposition is "p, only if q and not r and at-least-three-of $\{s, t, u, v\}$ are true." (Later we'll see how to write this compound proposition in standard logical notation; see Example 3.15.)

3.2.3 *Logical Connectives*

Logical connectives are the glue that creates the more complicated compound propositions from simpler propositions. Here are definitions of our first three of these logical connectives—*not*, *and*, and *or*:

Definition 3.3 (Negation (not): $\neg$)
The proposition $\neg p$ ("not p," called the negation *of the proposition p) is true when the proposition p is false, and is false when p is true.*

Definition 3.4 (Conjunction (and): $\wedge$)
The proposition $p \wedge q$ ("p and q," the conjunction *of the propositions p and q) is true when both of the propositions p and q are true, and is false when one or both of p or q is false.*

Definition 3.5 (Disjunction (or): $\vee$)
The proposition $p \vee q$ ("p or q," the disjunction *of the propositions p and q) is true when one or both of the propositions p or q is true, and is false when both p and q are false.*

In the conjunction $p \wedge q$, the propositions p and q are called *conjuncts*; in $p \vee q$, they are called *disjuncts*. Here's a simple example:

The prefix *con-* means "together" and *dis-* means "apart." (*Junct* means "join.") The *con*junction $p \wedge q$ is true when p and q are true together; the *dis*junction $p \vee q$ is true when p is true "apart from" q, or the other way around.

Example 3.5 (Some simple compound propositions)
Let p denote the proposition *Ohio State's mascot is the Buckeye* and let q denote the proposition *Michigan's mascot is the Wolverine.* Then:

- $\neg q$ denotes the proposition *Michigan's mascot is not the Wolverine.*
- $p \wedge q$ denotes the proposition *Ohio State's mascot is the Buckeye, and Michigan's mascot is the Wolverine.*
- $p \vee q$ denotes the proposition *Ohio State's mascot is the Buckeye, or Michigan's mascot is the Wolverine.*

To help keep the symbols straight, it may be helpful to notice that the symbol $\wedge$ is the angular version of the symbol $\cap$ (intersection), while the symbol $\vee$ is the angular version of the symbol $\cup$ (union). The set $S \cap T$ is the set of all elements contained in S *and* T; the set $S \cup T$ is the set of all elements contained in S *or* T.

Here's an example of translating some English statements that express compound propositions into standard logical notation:

Example 3.6 (From English statements to compound propositions)
Problem: Translate each of the following statements into logical notation. (Name the atomic propositions using appropriate Boolean variables.)

1. Carissa is majoring in computer science and studio art.
2. Either Dave took a formal logic class, or he is a quick learner.
3. Eli broke his hand and didn't take the test as scheduled.
4. Fred knows Python or he has programmed in both C and Java.

Solution: Let's first name the atomic propositions within these English statements:

p = Carissa is majoring in computer science.
q = Carissa is majoring in studio art.
r = Dave took a formal logic class.
s = Dave is a quick learner.
t = Eli broke his hand.
u = Eli took the test as scheduled.
v = Fred knows Python.
w = Fred has programmed in C.
x = Fred has programmed in Java.

We can now translate the four given statements as: (1) $p \wedge q$; (2) $r \vee s$; (3) $t \wedge \neg u$; and (4) $v \vee (w \wedge x)$.

Implication (if/then)

Another important logical connective is $\Rightarrow$, which denotes *implication*. It expresses a familiar idea from everyday life, though one that's not quite captured by a single

English word. Consider the sentence *If you scratch my back, then I'll scratch yours.* It's easiest to think of this sentence as a promise: I've promised that I'll scratch your back *as long as you scratch mine*. I haven't promised anything about what I'll do if you fail to scratch my back—I can abstain from back scratching, or I might generously scratch your back anyway, but I haven't *guaranteed* anything. (You'd justifiably call me a liar if you scratched my back and I failed to scratch yours in return.) This kind of promise is expressed as an *implication* in propositional logic:

Definition 3.6 (Implication: $\Rightarrow$)
The proposition $p \Rightarrow q$ is true when the truth of p implies the truth of q. In other words, $p \Rightarrow q$ is true unless p is true and q is false.

One initially confusing aspect of logical implication is that the word "implies" seems to hint at something about causation—but $p \Rightarrow q$ doesn't actually say anything about p *causing* q, only that p being true *implies that* q is true (or, in other words, p being true *lets us conclude that* q is true).

In the implication $p \Rightarrow q$, the proposition p is called the *antecedent* or the *hypothesis*, and the proposition q is called the *consequent* or the *conclusion*.

Here are a few examples of statements involving implication:

Example 3.7 (Some implications)
The following propositions are all true:

- $1+1=2$ implies that $2+3=5$. ("True implies True" is true.)
- $2+3=4$ implies that $2+2=4$. ("False implies True" is true.)
- $2+3=4$ implies that $2+3=6$. ("False implies False" is true.)

But the following proposition is false:

- $2+2=4$ implies that $2+1=5$. ("True implies False" is false.)

This last proposition is false because $2+2=4$ is true, but $2+1=5$ is false.

There are many different ways to express the proposition $p \Rightarrow q$ in English, including all of those in Figure 3.1.

"p implies q"
"if p, then q"
"p only if q"
"q whenever p"
"q, if p"
"q is necessary for p"
"p is sufficient for q"

Figure 3.1: Some ways of expressing $p \Rightarrow q$ in English.

Here is an example of the same implication being stated in English in many different ways:

Example 3.8 (Expressing implications in English)
According to United States law, people who can legally vote must be American citizens, and they must also satisfy some other various conditions that vary from state to state (for example, registering in advance or not being a felon). Thus the following compound proposition is true:

$$\text{you are a legal U.S. voter} \Rightarrow \text{you are an American citizen.}$$

All of the following sentences express this proposition in English:

If you are a legal U.S. voter, then you are an American citizen.
You being a legal U.S. voter implies that you are an American citizen.
You are a legal U.S. voter only if you are an American citizen.

> You are an American citizen if you are a legal U.S. voter.
> You are an American citizen whenever you are a legal U.S. voter.
> You being an American citizen is necessary for you to be a legal U.S. voter.
> You being a legal U.S. voter is sufficient for you to be an American citizen.

Most of these sentences are reasonably natural ways to express the stated implication, though the last phrasing seems awkward. But it's easier to understand if we slightly rephrase it as "You being a legal U.S. voter *is sufficient for one to conclude that you are* an American citizen."

Here's another example of restating implications:

Example 3.9 (More implications in English)
Consider the proposition

$$\underbrace{\textit{The nondisclosure agreement is valid}}_{p}\ \textit{only if}\ \underbrace{\textit{you signed it}}_{q}.$$

(This statement is *different* from "if you signed, then the agreement is valid": for example, the agreement might not be valid because you're legally a minor and thus not legally allowed to sign away rights.) We can restate $p \Rightarrow q$ as "if p then q":

> *If the nondisclosure agreement is valid, then you signed it.*

We can also restate this implication equivalently—and perhaps more intuitively—using the so-called contrapositive $\neg q \Rightarrow \neg p$ (see Example 3.21):

> *The nondisclosure agreement is invalid if you didn't sign it.*

"Exclusive or" and "if and only if"

The four logical connectives that we have defined so far ($\neg$, $\vee$, $\wedge$, and $\Rightarrow$) are the ones that are most frequently used, but we'll define two other common connectives too. The first is *exclusive or*:

The connective $\oplus$ is usually pronounced like "ex ore" (a former significant other + some rock with high precious-metal content).

Definition 3.7 (Exclusive or: $\oplus$)
The proposition $p \oplus q$ ("p exclusive or q" or, more briefly, "p xor q") is true when one of the propositions p or q is true, but not both. Thus $p \oplus q$ is false when both p and q are true, and when both p and q are false.

When we want to emphasize the distinction between $\vee$ and $\oplus$, we refer to $\vee$ as *inclusive or*. This terminology highlights the fact that $p \vee q$ *includes* the possibility that both p and q are true, while $p \oplus q$ *excludes* that possibility. Unfortunately, the word "or" in English can mean either inclusive or exclusive or, depending on the context in which it's being used. When you see the word "or," you'll have to think carefully about which meaning is intended.

Here's an example of distinguishing inclusive and exclusive or:

Example 3.10 (Inclusive versus exclusive or in English)

Problem: Translate these statements from a cover letter for a job into logical notation:

> You may contact me by email or by phone. I am available for an on-site day-long interview on October 8th in Minneapolis or Hong Kong.

Use the following Boolean variables:

p = you may contact me by phone
q = you may contact me by email
r = I am physically available for an interview in Minneapolis
s = I am physically available for an interview in Hong Kong

Solution: The "or" in "email or phone" is *inclusive*, because you could receive both an email and a call. However, the "or" in "Minneapolis or Hong Kong" is *exclusive*, because it's not physically possible to be simultaneously present in Minneapolis and Hong Kong. Thus a correct translation of these statements is $(p \vee q) \wedge (r \oplus s)$.

We are now ready to define our last logical connective:

Definition 3.8 (If and only if: $\Leftrightarrow$)
The proposition $p \Leftrightarrow q$ ("p if and only if q") is true when the propositions p or q have the same truth value (both p and q are true, or both p and q are false), and false otherwise.

Sometimes you'll see $\Leftrightarrow$ abbreviated in sentences as "iff" as shorthand for "if and only if." We'll avoid this notation in this book, but you should understand it if you see it elsewhere.

The reason that $\Leftrightarrow$ is read as "if and only if" is that $p \Leftrightarrow q$ means the same thing as the compound proposition $(p \Rightarrow q) \wedge (q \Rightarrow p)$. (We'll prove this equivalence in Example 3.23.) Furthermore, the propositions $p \Rightarrow q$ and $q \Rightarrow p$ can be rendered, respectively, as "p only if q" and "p, if q." Thus $p \Leftrightarrow q$ expresses "p if q, and p only if q"—or, more compactly, "p if and only if q." (The connective $\Leftrightarrow$ is also sometimes called the *biconditional,* because an implication can also be called a *conditional.*)

Unfortunately, just like with "or," the word "if" is ambiguous in English. Sometimes "if" is used to express an implication, and sometimes it's used to express an if-and-only-if definition. When you see the word "if" in a sentence, you'll need to think carefully about whether it means $\Rightarrow$ or $\Leftrightarrow$. Here's an example:

Example 3.11 ("If" versus "if and only if" in English)

Problem: Think of a number between 10 and 1,000,000. Let

$p :=$ your number is prime.
$q :=$ your number is even.
$r :=$ your number is evenly divisible by an integer other than 1 and itself.

Now translate the following two sentences into logical notation:

1. If the number you're thinking of is even, then it isn't prime.
2. The number you're thinking of isn't prime if it's evenly divisible by an integer other than 1 and itself.

Solution: The "if" in (1) is an implication, and the "if" in (2) is "if and only if." A correct translation of these sentences is (1) $q \Rightarrow \neg p$; and (2) $\neg p \Leftrightarrow r$.

3.2.4 *Combining Logical Connectives*

The six standard logical connectives that we've defined so far ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\oplus$, and $\Leftrightarrow$) are summarized in Figure 3.2. The logical connective $\neg$ is a *unary operator*, because it builds a compound proposition from a single simpler proposition. The other five connectives are *binary* operators, which build a compound proposition from two simpler propositions. (We'll encounter the full list of binary logical connectives later; see Exercises 4.66–4.71.)

negation	$\neg p$	"not p"	*highest precedence*
conjunction	$p \wedge q$	"p and q"	↓
disjunction	$p \vee q$	"p or q"	
exclusive or	$p \oplus q$	"p xor q"	
implication	$p \Rightarrow q$	"if p, then q" or "p implies q"	
if and only if	$p \Leftrightarrow q$	"p if and only if q"	*lowest precedence*

Figure 3.2: Summary of notation for propositional logic.

> **Taking it further:** The unary-vs.-binary categorization of logical connectives based on how many "arguments" they accept also occurs in other contexts—for example, arithmetic and programming. In arithmetic, for example, one might distinguish between "unary minus" and "binary minus": the former denotes negation, as in -3; the latter subtraction, as in $2 - 3$.
>
> In programming languages, the number of arguments that a function takes is called its *arity*. (The arity of `length` is one; the arity of `equals` is two.) You will sometimes encounter *variable arity* functions that can take a different number of arguments each time they're invoked. Common examples include the print functions in many languages—C's `printf` and Python's `print`, for example, can take any number of arguments—or arithmetic in prefix languages like Scheme, where you can write an expression like `(+ 1 2 3 4)` to denote $1+2+3+4$ $(=10)$.

Order of operations

A full description of the syntax of a programming language always includes a table of the *precedence* of operators, arranged from "binds the tightest" (highest precedence) to "binds the loosest" (lowest precedence). These precedence rules tell us when we have to include parentheses in an expression to make it mean what we want it to mean, and when the parentheses are optional. In the same way, we'll adopt some standard conventions regarding the precedence of our logical connectives:

The word "precedence" (*pre* before, *cede* go) means "what comes first," so precedence rules tell us the order of which the operators "get to go." For example, consider a proposition like $p \wedge q \Rightarrow r$. If $\wedge$ "goes first," the proposition is $(p \wedge q) \Rightarrow r$; if $\Rightarrow$ "goes first," it means $p \wedge (q \Rightarrow r)$. Figure 3.2 says that the former is the correct interpretation.

- Negation ($\neg$) binds the tightest.
- After negation, there is a three-way tie among $\wedge$, $\vee$, and $\oplus$. (We'll always use parentheses in propositions containing more than one of these three operators, just as we should in programs.)
- The trifecta ($\wedge$, $\vee$, and $\oplus$) is followed by $\Rightarrow$.
- $\Rightarrow$ is followed finally by $\Leftrightarrow$.

The horizontal lines in Figure 3.2 separate the logical connectives by their precedence, so that operators closer to the top of the table have higher precedence. For example:

Example 3.12 (Precedence of logical connectives)

The propositions $p \vee \neg q$ and $p \vee q \Rightarrow \neg r \Leftrightarrow p$ mean, respectively,

$$p \vee (\neg q) \qquad \text{and} \qquad \Big((p \vee q) \Rightarrow (\neg r)\Big) \Leftrightarrow p,$$

which we can see by simply applying the relevant precedence rules ("$\neg$ goes first, then $\vee$, then $\Rightarrow$, then $\Leftrightarrow$").

> **Taking it further:** The precedence rules that we've described here match the precedence rules in most programming languages. In Java, for example, the condition `!p && q`—that's "not p and q" in Java syntax—will be interpreted as `(!p) && q`, because not/$\neg$/`!` binds tighter than and/$\wedge$/`&&`.

The precedence rules for operators tell us the order in which two different operators are applied in an expression. For a sequence of applications of the *same* binary operator, we'll use the convention that the operator *associates to the left*. For example, $p \wedge q \wedge r$ will mean $(p \wedge q) \wedge r$ and not $p \wedge (q \wedge r)$.

Example 3.13 (Precedence of logical connectives)

Problem: Fully parenthesize each of the following propositions. (In other words, add parentheses around each operator without changing the meaning.)

1. $p \vee q \Leftrightarrow p$
2. $p \oplus p \oplus q \oplus q$
3. $\neg p \Leftrightarrow p \Leftrightarrow \neg(p \Leftrightarrow p)$
4. $p \wedge \neg q \Rightarrow r \Leftrightarrow s$
5. $p \Rightarrow q \Rightarrow r \wedge s$

Solution: Using the precedence rules from Figure 3.2 and left associativity, we get:

1. $(p \vee q) \Leftrightarrow p$
2. $((p \oplus p) \oplus q) \oplus q$
3. $((\neg p) \Leftrightarrow p) \Leftrightarrow (\neg(p \Leftrightarrow p))$
4. $((p \wedge (\neg q)) \Rightarrow r) \Leftrightarrow s$
5. $(p \Rightarrow q) \Rightarrow (r \wedge s)$

The choice that logical operators associate to the left (instead of associating to the right) won't matter for most of the logical connectives anyway. For example, the propositions $(p \wedge q) \wedge r$ and $p \wedge (q \wedge r)$ are true under exactly the same circumstances, as we'll see shortly. In fact, of the binary operators $\{\wedge, \vee, \oplus, \Rightarrow, \Leftrightarrow\}$, the only one for which the order of application matters is implication. See Exercises 3.45–3.47.

Writing tip: Because the order of application does matter for implication, it's considered good style to include the optional parentheses so that it's clear what you mean.

3.2.5 Truth Tables

In Section 3.2.3, we described the logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\oplus$, and $\Leftrightarrow$, but we can more systematically define these connectives by using a *truth table* that collects the value yielded by the logical connective under every *truth assignment*.

Definition 3.9 (Truth assignment)

A truth assignment *for a proposition over variables $p_1, p_2, \ldots, p_k$ is a function that assigns a truth value to each p_i.*

For example, the function f where $f(p) = \mathrm{T}$ and $f(q) = \mathrm{F}$ is a truth assignment for the proposition $p \vee \neg q$. (Each "T" abbreviates a truth value of true; each "F" abbreviates a truth value of false.)

For any particular proposition and for any particular truth assignment f for that proposition, we can *evaluate* the proposition under f to figure out the truth value of the entire proposition. In the previous example, the proposition $p \vee \neg q$ is true under the truth assignment with $p = \text{T}$ and $q = \text{F}$ (because $T \vee \neg\text{F}$ is $\text{T} \vee \text{T}$, which is true). A *truth table* displays a proposition's truth value (evaluated in the way we just described) under all truth assignments:

Definition 3.10 (Truth table)
A truth table *for a proposition lists, for each possible truth assignment for that proposition (with one truth assignment per row in the table), the truth value of the entire proposition.*

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Figure 3.3: The truth table for $\wedge$.

For example, the truth table that defines $\wedge$ is shown in Figure 3.3. A few words about this truth table are in order:

- Columns #1 and #2 correspond to the atomic propositions p and q. There is a row in the table corresponding to each possible truth assignment for $p \wedge q$—that is, for every pair of truth values for p and q. (So there are four rows: TT, TF, FT, and FF.)
- The third column corresponds to the compound proposition $p \wedge q$, and it has a T only in the first row. That is, the truth value of $p \wedge q$ is false unless both p and q are true—just as Definition 3.4 said.

The truth tables for the six basic logical connectives (negation, conjunction, disjunction, exclusive or, implication, and "if and only if") are shown in Figure 3.4. It's worth paying special attention to the column for $p \Rightarrow q$: the *only* truth assignment under which $p \Rightarrow q$ is false is when p is true and q is false. *False implies anything! Anything implies true!* For example, both of the following are true propositions:

p	$\neg p$
T	F
F	T

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \oplus q$	$p \Leftrightarrow q$
T	T	T	T	T	F	T
T	F	F	T	F	T	F
F	T	F	T	T	T	F
F	F	F	F	T	F	T

Figure 3.4: Truth tables for the basic logical connectives.

If $2+3=4$*, then you will eat tofu for dinner.* (if false, then anything)
If you are your own mother, then $2+3=5$. (if anything, then true)

To emphasize the point, observe that the first statement is true *even if* you would never eat tofu if it were the last so-called food on earth; the hypothesis "$2+3 = 4$" of the proposition wasn't true, so the truth of the proposition doesn't depend on what your dinner plans are.

For more complicated compound propositions, we can fill in a truth table by repeatedly applying the rules in Figure 3.4. For example, to find the truth table for $(p \Rightarrow q) \wedge (q \vee p)$, we compute the truth tables for $p \Rightarrow q$ and $q \vee p$, and put a "T" in the $(p \Rightarrow q) \wedge (q \vee p)$ column for precisely those rows in which the truth tables for $p \Rightarrow q$ and $q \vee p$ both had "T"s. Here's a simple example, and a somewhat more complicated one:

Example 3.14 (A small truth table)
Here is a truth table for the proposition $(p \wedge q) \Rightarrow \neg q$:

p	q	$p \wedge q$	$\neg q$	$(p \wedge q) \Rightarrow \neg q$
T	T	T	F	F
T	F	F	T	T
F	T	F	F	T
F	F	F	T	T

This truth table shows that the given proposition $(p \wedge q) \Rightarrow \neg q$ is true precisely when at least one of p and q is false.

Example 3.15 (Three (or more) of four, formalized)
In Example 3.4 (on the validity of passwords), we had a sentence of the form

"p, only if q and not r and at-least-three-of $\{s, t, u, v\}$ are true."

Let's translate this sentence into propositional logic. The tricky part will be translating "at least three of $\{s, t, u, v\}$ are true." There are many solutions, but one relatively simple way to do it is to explicitly write out four cases, one corresponding to allowing a different one of the four variables $\{s, t, u, v\}$ to be false:

$$(s \wedge t \wedge u) \vee (s \wedge t \wedge v) \vee (s \wedge u \wedge v) \vee (t \wedge u \wedge v)$$

We can verify that we've gotten this proposition right with a (big!) truth table, shown in Figure 3.5. Indeed, the five rows in which the last column has a "T" are exactly the five rows in which there are three or four "T"s in the columns for s, t, u, and v.

To finish the translation, recall that "x only if y" means $x \Rightarrow y$, so the given sentence can be translated as $p \Rightarrow q \wedge \neg r \wedge$ (the proposition above)—that is,

$$p \Rightarrow q \wedge \neg r \wedge \Big((s \wedge t \wedge u) \vee (s \wedge t \wedge v) \vee (s \wedge u \wedge v) \vee (t \wedge u \wedge v)\Big).$$

Figure 3.5: A truth table for Example 3.15.

s	t	u	v	$s \wedge t \wedge u$	$s \wedge t \wedge v$	$s \wedge u \wedge v$	$t \wedge u \wedge v$	$(s \wedge t \wedge u)$ $\vee\ (s \wedge t \wedge v)$ $\vee\ (s \wedge u \wedge v)$ $\vee\ (t \wedge u \wedge v)$
T	T	T	T	T	T	T	T	T
T	T	T	F	T	F	F	F	T
T	T	F	T	F	T	F	F	T
T	T	F	F	F	F	F	F	F
T	F	T	T	F	F	T	F	T
T	F	T	F	F	F	F	F	F
T	F	F	T	F	F	F	F	F
T	F	F	F	F	F	F	F	F
F	T	T	T	F	F	F	T	T
F	T	T	F	F	F	F	F	F
F	T	F	T	F	F	F	F	F
F	T	F	F	F	F	F	F	F
F	F	T	T	F	F	F	F	F
F	F	T	F	F	F	F	F	F
F	F	F	T	F	F	F	F	F
F	F	F	F	F	F	F	F	F

Taking it further: It's worth pondering why there are five different rows of the truth table in Figure 3.5 in which the last column is true: there are four different truth assignments corresponding to exactly three of $\{s, t, u, v\}$ being true (stu, suv, stv, tuv), and there is one truth assignment corresponding to all four being true ($stuv$). In Chapter 9, on counting, we'll re-encounter this style of question. (And, actually, precisely the same reasoning as in this example will allow us to prove something interesting about error-correcting codes—see Section 4.2.5.)

Computer Science Connections

Natural Language Processing, Ambiguity, and Truth

Our main interest in this book is in developing (and understanding) precise and unambiguous language to express mathematical notions; in this chapter specifically, we're thinking about the truth values of completely precise statements. But thinking about the truth of ambiguous or ill-defined terms is absolutely crucial to any computational system that's designed to interact with users via natural language. (A *natural language* is one like English or French or Xhosa; these languages contrast with *artificial languages* like Java or Python or, arguably, Esperanto or Klingon.)

Natural language processing (*NLP*) (or the roughly similar *computational linguistics*) is the subfield of computer science that lies at the discipline's interface with linguistics.[2] In NLP, we work to develop software systems that can interact with users in a natural language. A necessary step in an NLP system is to take an utterance made by the human user and "understand it." ("Understanding what a sentence means" is more or less the same as "understanding the circumstances under which it is true"—which is fundamentally a question of logic.)

One major reason that NLP is hard is that there is a tremendous amount of ambiguity in natural-language utterances. We can have *lexical ambiguity*, in which two different words are spelled identically but have two different meanings; we have to determine which word is meant in a sentence. Or there's *syntactic ambiguity*, in which a sentence's structure can be interpreted very differently. (See Figure 3.6.) But there are also subtleties about when a statement is true, even if the meaning of each word and the sentence's structure are clear.

A: *Do you prefer coffee or tea?*
B: *Do you prefer cream or sugar?*
C: *We ate cake with walnuts.*
D: *We ate cake with forks.*

Figure 3.6: Examples of *lexical* (A and B) and *syntactic ambiguity* (C and D). The *or* of A/B can be either inclusive or exclusive; simply answering "yes" is a reasonable response to question B, but a bizarre one to question A. The *with* of C/D can either attach to the *cake* or the *eating*; the sentences' structures are consistent with using walnuts as an eating utensil in C, or the cake containing forks as an ingredient in D.

Consider, for example, designing and implementing a conversational system designed to assist with travel planning. (Many airlines or train companies have such systems.) Such a system might engage in a dialogue like the one in Figure 3.7 with a human user. There's no hard-and-fast rule for what other flights should count as "slightly later" and "too much more expensive." This conversational system has to be able to decide the truth of statements like *Delta #2931 is slightly later than Delta #1927* and *Delta #2931 isn't too much more expensive than Delta #1927*, even though the "truth" of these statements depends on heavy use of conversational context and pragmatic reasoning. Of course, even though one cannot unambiguously determine whether these sentences are true or false, they're the kind of statement made continually in natural language. So systems that process natural language must deal with this issue with great frequency.

User: *I want to fly from MSP to BOS on 28 December.*

System: *Delta #1927 is a nonstop flight from MSP to BOS on Delta Airlines for $472 that leaves at 8:45am.*

User: *Is there a slightly later flight that isn't too much more expensive?*

Figure 3.7: A sample dialogue. Suppose that Delta #2931 is a second nonstop flight from MSP to BOS that leaves at 10:33am and costs $529.

One approach for handling these statements whose truth value is ambiguous is called *fuzzy logic*, in which each proposition has a truth value that is a real number between 0 and 1. (So *10:33a is slightly later than 8:45a* is "more true" than *12:19p is slightly later than 8:45a*—so the former might have a truth value of 0.74, while the latter might have a truth value of 0.34. But *7:30a is slightly later than 8:45a* would have a truth value of 0.00, as 7:30a is unambiguously *not* slightly later than 8:45a.)

For more, you can look for a textbook on NLP like

[2] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson Prentice Hall, 2nd edition, 2008.

3.2.6 Exercises

What are the truth values of the following propositions?

3.1 $2^2 + 3^2 = 4^2$

3.2 The number 202 is written 11010010 in binary.

3.3 After executing the C code fragment in Figure 3.8 (shown at right), the variable x has the value 1.

```
int x = 202;
while (x > 2) {
   x = x / 2;
}
```

Figure 3.8: Snippet of C code. Note that x/2 denotes integer division; for example, 7/2 = 3.

Consider the following atomic propositions:

p :	x + y *is valid Python*	u :	x *is a numeric value*
q :	x * y *is valid Python*	v :	y *is a numeric value*
r :	x ** y *is valid Python*	w :	x *is a list*
s :	x * y *is a list*	z :	y *is a list*
t :	x + y *is a list*		

*Using these atomic propositions, translate the following (true!) statements about legal Python programs into logical notation. (Note that these statements do not come close to fully characterizing the set of valid Python statements, for several reasons: first, they're about particular variables—*x *and* y*—rather than about generic variables. And, second, they omit some important common-sense facts—for example, it's not simultaneously possible to be both a list and a numeric value. That is, for example, we have $\neg v \vee \neg z$.)*

3.4 x ** y is valid Python if and only if x and y are both numeric values.

3.5 x + y is valid Python if and only if x and y are both numeric values, or they're both lists.

3.6 x * y is valid Python if and only if x and y are both numeric values, or if one of x and y is a list and the other is numeric.

3.7 x * y is a list if x * y is valid Python and x and y are not both numeric values.

3.8 if x + y is a list, then x * y is not a list.

3.9 x + y and x * y are both valid Python only if x is not a list.

3.10 True story: a 29-year-old friend of mine who does not have an advance care directive was asked the following question on a form at a doctor's office. What should she answer?

If you're over 55 years old, do you have an advance care directive? Circle one: YES NO

In Example 3.15, we constructed a proposition corresponding to "at least three of $\{s, t, u, v\}$ are true." Generalize this construction by building a proposition . . .

3.11 . . . expressing "at least 3 of $\{p_1, \dots, p_n\}$ are true."

3.12 . . . expressing "at least $n - 1$ of $\{p_1, \dots, p_n\}$ are true."

The identity *of a binary operator $\diamond$ is a value i such that, for any x, the expressions $\{x, x \diamond i, i \diamond x\}$ are all equivalent. The* zero *of $\diamond$ is a value z such that, for any x, the expressions $\{z, x \diamond z, z \diamond x\}$ are all equivalent. For an example from arithmetic, the identity of $+$ is 0, because $x + 0 = 0 + x = x$ for any number x. And the zero of multiplication is 0, because $x \cdot 0 = 0 \cdot x = 0$ for any number x. For each of the following, identify the identity or zero of the given logical operator. Justify your answer. Some operators do not have an identity or a zero; if the given operator fails to have the stated identity/zero, explain why it doesn't exist.*

3.13 What is the identity of $\vee$?

3.14 What is the identity of $\wedge$?

3.15 What is the identity of $\Leftrightarrow$?

3.16 What is the identity of $\oplus$?

3.17 What is the zero of $\vee$?

3.18 What is the zero of $\wedge$?

3.19 What is the zero of $\Leftrightarrow$?

3.20 What is the zero of $\oplus$?

Because $\Rightarrow$ is not commutative (that is, because $p \Rightarrow q$ and $q \Rightarrow p$ mean different things), it is not too surprising that $\Rightarrow$ has neither an identity nor a zero. But there are a pair of related definitions that apply to this type of operator:

3.21 The *left identity* of a binary operator $\diamond$ is a value i_ℓ such that, for any x, the expressions x and $i_\ell \diamond x$ are equivalent. The *right identity* of $\diamond$ is a value i_r such that, for any x, the expressions x and $x \diamond i_r$ are equivalent. (Again, some operators may not have left or right identities.) What are the left and right identities of $\Rightarrow$ (if they exist)?

3.22 The *left zero* of a binary operator $\diamond$ is a value z_ℓ such that, for any x, the expressions z_ℓ and $z_\ell \diamond x$ are equivalent; similarly, the *right zero* is a value z_r such that, for any x, the expressions z_r and $x \diamond z_r$ are equivalent. (Again, some operators may not have left or right zeros.) What are the left and right zeros for $\Rightarrow$ (if they exist)?

In many programming languages, the Boolean values True and False are actually stored as the numerical values 1 *and* 0*, respectively. In Python, for example, both* `0 == False` *and* `1 == True` *are True. Thus, despite appearances, we can add or subtract or multiply Boolean values! Furthermore, in many languages (including Python), anything that is not False (in other words, anything other than* 0*) is considered True for the purposes of conditionals. For example, in many programming languages, including Python, code like* `if 2 print "yes" else print "no"` *will print "yes."*

Suppose that x and y are two Boolean variables in a programming language, like Python, where True *and* False *are* 1 *and* 0*, respectively—that is, the values of x and y are both* 0 *or* 1*. Each of the following code snippets includes a conditional statement based on an arithmetic expression using x and y. For each, rewrite the given condition using the standard notation of propositional logic.*

3.23 `if x * y ...`

3.24 `if x + y ...`

3.25 `if 1 - x ...`

3.26 `if (x * (1 - y)) + ((1 - x) * y) ...`

We can use the common programming language features described in in the previous block of exercises to give a simple programming solution to Exercises 3.11–3.12. Assume that $\{p_1, \ldots, p_n\}$ are all Boolean variables in Python—that is, their values are all 0 *or* 1*. Write a Python conditional expressing the condition that . . .*

3.27 . . . at least 3 of $\{p_1, \ldots, p_n\}$ are true.

3.28 . . . at least $n - 1$ of $\{p_1, \ldots, p_n\}$ are true.

In addition to purely logical operations, computer circuitry has to be built to do simple arithmetic very quickly. Here you'll explore some pieces of using propositional logic and binary representation of integers to express arithmetic operations. (It's straightforward to convert your answers into circuits.)

Consider a number $x \in \{0, \ldots, 15\}$ represented as a 4-bit binary number, as shown in Figure 3.9. Denote by x_0 the least-significant bit of x, by x_1 the next bit, and so forth. For example, for the number $x = 12$ (written 1100 *in binary) would have $x_0 = 0$, $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$). For each of the following conditions, give a proposition over the Boolean variables $\{x_0, x_1, x_2, x_3\}$ that expresses the stated condition. (Think of* 0 *as false and* 1 *as true.)*

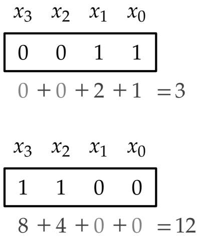


Figure 3.9: Representing $x \in \{0, \ldots, 15\}$ using 4-bits.

3.29 x is greater than or equal to 8.

3.30 x is evenly divisible by 4.

3.31 x is evenly divisible by 5. (*Hint: use a truth table, and then build a proposition from the table.*)

3.32 x is an exact power of two.

3.33 Suppose that we have *two* 4-bit input integers x and y, represented as in Exercises 3.29–3.32. Give a proposition over $\{x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3\}$ that expresses the condition that $x = y$.

3.34 Given two 4-bit integers x and y as in the previous exercise, give a proposition over the Boolean variables $\{x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3\}$ that expresses the condition that $x \leq y$.

3.35 Suppose that we have a 4-bit input integer x, represented by four Boolean variables $\{x_0, x_1, x_2, x_3\}$ as in Exercises 3.29–3.32. Let y be the integer $x + 1$, represented again as a 4-bit value $\{y_0, y_1, y_2, y_3\}$. (For the purposes of this question, treat $15 + 1 = 0$—that is, we're really defining $y = (x + 1) \bmod 16$.) For example, for $x = 11$ (which is 1011 in binary), we have that $y = 12$ (which is 1100 in binary). For each $i \in \{0, 1, 2, 3\}$, give a proposition over the Boolean variables $\{x_0, x_1, x_2, x_3\}$ that expresses the value of y_i.

The remaining problems in this section ask you to build a program to compute various facts about a given proposition φ. To make your life as easy as possible, you should consider a simple representation of φ, based on representing any compound proposition as a list*. In such a list, the first element will be the logical connective, and the remaining elements will be the subpropositions. For example, the proposition $p \Rightarrow (\neg q)$ will be represented as*

```
["implies", ["or", "p", "r"], ["not", "q"]]
```

Now, using this representation of propositions, write a program, in a programming language of your choice, to accomplish the following operations:

We'll occasionally use lowercase Greek letters, particularly φ ("phi") or ψ ("psi"), to denote not-necessarily-atomic propositions.

3.36 (*programming required*) Given a proposition φ, compute the set of all atomic propositions contained within φ. The following recursive formulation may be helpful:

$$\textbf{variables}(p) := \{p\} \qquad \textbf{variables}(\neg\varphi) := \textbf{variables}(\varphi)$$

$$\textbf{variables}(\varphi \diamond \psi) := \textbf{variables}(\varphi) \cup \textbf{variables}(\psi) \qquad \textit{for any connective } \diamond \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow, \oplus, \ldots\}$$

3.37 (*programming required*) Given a proposition φ and a truth assignment for each variable in φ, evaluate whether φ is true or false under this truth assignment.

3.38 (*programming required*) Given a proposition φ, compute the set of all truth assignments for the variables in φ that make φ true. (One good approach: use your solution to Exercise 3.36 to compute all the variables in φ, then build the full list of truth assignments for those variables, and then evaluate φ under each of these truth assignments using your solution to Exercise 3.37.)

3.3 Propositional Logic: Some Extensions

> Against logic there is no armor like ignorance.
>
> Laurence J. Peter (1919–1990)

With the definitions from Section 3.2 in hand, we turn to a few extensions: some special types of propositions, and some special ways of representing propositions.

3.3.1 Tautology and Satisfiability

Several important types of propositions are defined in terms of their truth tables: those that are always true (*tautologies*), sometimes true (*satisfiable* propositions), or never true (*unsatisfiable* propositions). We will explore each of these types in turn.

TAUTOLOGIES

We'll start by considering propositions that are always true:

Etymologically, the word *tautology* comes from *taut* "same" (*to* + *auto*) + *logy* "word." Another meaning for the word "tautology" (in real life, not just in logic) is the unnecessary repetition of an idea: "a canine dog." (The etymology and the secondary street meaning are not totally removed from the usage in logic.)

Definition 3.11 (Tautology)
A proposition is a tautology *if it is true under every truth assignment.*

One reason that tautologies are important is that we can use them to reason about logical statements, which can be particularly valuable when we're trying to prove a claim.

Examples 3.16 and 3.17 illustrate two important tautologies. The first of these tautologies is the proposition $p \vee \neg p$, which is called the *law of the excluded middle*: for any proposition p, either p is true or p is false; there is nothing "in between."

Example 3.16 (Law of the Excluded Middle)
Here is the truth table for the proposition $p \vee \neg p$:

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

The third column is filled with "T"s, so $p \vee \neg p$ is a tautology.

The second tautology is the proposition $p \wedge (p \Rightarrow q) \Rightarrow q$, called *modus ponens*: if we know both that (a) p is true and that (b) the truth of p implies the truth of q, then we can conclude that q is true.

Modus ponens rhymes with "goad us phone-ins"; literally, it means "the mood that affirms" in Latin.

Example 3.17 (Modus Ponens)
Here is the truth table for $p \wedge (p \Rightarrow q) \Rightarrow q$ (with a few extra columns of "scratch work," for each of the constituent pieces of the desired final proposition):

p	q	$p \Rightarrow q$	$p \wedge (p \Rightarrow q)$	$p \wedge (p \Rightarrow q) \Rightarrow q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

There are only "T"s in the last column of this truth table, which establishes that modus ponens is a tautology.

Figure 3.10 contains a number of tautologies that you may find interesting and occasionally helpful. (Exercises 3.60–3.72 ask you to build truth tables to verify that these propositions really are tautologies.)

One terminological note from Figure 3.10: *modus tollens* is the proposition $(p \Rightarrow q) \wedge \neg q \Rightarrow \neg p$, and it's the counterpoint to modus ponens: if we know both that (a) the truth of p implies the truth of q and that (b) q is not true, then we can conclude that p cannot be true either. (Modus tollens means "the mood that denies" in Latin.)

$(p \Rightarrow q) \wedge p \Rightarrow q$	Modus Ponens
$(p \Rightarrow q) \wedge \neg q \Rightarrow \neg p$	Modus Tollens
$p \vee \neg p$	Law of the Excluded Middle
$p \Leftrightarrow \neg\neg p$	Double Negation
$p \Leftrightarrow p$	
$p \Rightarrow p \vee q$	
$p \wedge q \Rightarrow p$	
$(p \vee q) \wedge \neg p \Rightarrow q$	
$(p \Rightarrow q) \wedge (\neg p \Rightarrow q) \Rightarrow q$	
$(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$	
$(p \Rightarrow q) \wedge (p \Rightarrow r) \Leftrightarrow p \Rightarrow q \wedge r$	
$(p \Rightarrow q) \vee (p \Rightarrow r) \Leftrightarrow p \Rightarrow q \vee r$	
$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$	
$p \Rightarrow (q \Rightarrow r) \Leftrightarrow p \wedge q \Rightarrow r$	

Figure 3.10: Some tautologies.

Satisfiable and unsatisfiable propositions

We now turn to propositions that are sometimes true, and those propositions that are never true:

Definition 3.12 (Satisfiable propositions)
A proposition is satisfiable *if it is true under at least one truth assignment.*

If f is a truth assignment under which a proposition is true, then we say that the proposition is *satisfied by* f.

Definition 3.13 (Unsatisfiable propositions/contradictions)
A proposition is unsatisfiable *if it is not satisfiable. Such a proposition is also called a* contradiction.

Thus a proposition is satisfiable if it is true under at least one truth assignment, and unsatisfiable if it is false under every truth assignment. (And it's a tautology if it is true under every truth assignment.) Here are some examples:

Example 3.18 (Contradiction of $p \Leftrightarrow q$ and $p \oplus q$)
Here is the truth table for $(p \Leftrightarrow q) \wedge (p \oplus q)$:

p	q	$p \Leftrightarrow q$	$p \oplus q$	$(p \Leftrightarrow q) \wedge (p \oplus q)$
T	T	T	F	F
T	F	F	T	F
F	T	F	T	F
F	F	T	F	F

Because the column of the truth table corresponding to the given proposition has no "T"s in it, the proposition $(p \Leftrightarrow q) \wedge (p \oplus q)$ is unsatisfiable.

Though it might not have been immediately apparent when they were defined, the logical connectives $\oplus$ and $\Leftrightarrow$ demand precisely opposite things of their arguments: the proposition $p \oplus q$ is true when p and q have *different* truth values, while $p \Leftrightarrow q$ is true when p and q have the *same* truth values. Because p and q cannot simultaneously have the same and different truth values, the conjunction $(p \Leftrightarrow q) \wedge (p \oplus q)$ is a contradiction.

Example 3.19 (Demanding satisfaction)
Problem: Is the proposition $p \vee q \Rightarrow \neg p \wedge \neg q$ satisfiable?

Solution: We'll answer the question by building a truth table for the given proposition:

p	q	$p \vee q$	$\neg p$	$\neg q$	$\neg p \wedge \neg q$	$p \vee q \Rightarrow \neg p \wedge \neg q$
T	T	T	F	F	F	F
T	F	T	F	T	F	F
F	T	T	T	F	F	F
F	F	F	T	T	T	T

Because there is at least one "T" in the last column in the truth table, the proposition is satisfiable. Specifically, this proposition is satisfied by the truth assignment $p =$ False, $q =$ False. (Under this truth assignment, the hypothesis $p \vee q$ is false; because false implies anything, the entire implication is true.)

Let φ be *any* proposition. Then φ is a tautology exactly when $\neg\varphi$ is unsatisfiable: φ is a tautology when the truth table for φ is all "T"s, which happens exactly when the truth table for $\neg\varphi$ is all "F"s. And that's precisely the definition of $\neg\varphi$ being unsatisfiable!

As we said in Section 3.2.6, we occasionally denote generic propositions by lowercase Greek letters, particularly φ ("phi") or ψ ("psi").

Taking it further: While satisfiability seems like a pretty precise technical definition that wouldn't matter all that much, the *satisfiability problem*—given a proposition φ, determine whether φ is satisfiable—turns out to be at the heart of the biggest open question in computer science today. If you figure out how to solve the satisfiability problem efficiently (or prove that it's impossible to solve efficiently), then you'll be the most famous computer scientist of the century. See the discussion on p. 326.

3.3.2 Logical Equivalence

We'll now turn to a special type of *pairs* of propositions. When two propositions "mean the same thing" (that is, they are true under precisely the same circumstances), they are called *logically equivalent*:

Definition 3.14 (Logical equivalence)
Two propositions φ and ψ are logically equivalent, *written $\varphi \equiv \psi$, if they have exactly identical truth tables (in other words, their truth values are the same under every truth assignment).*

To state it differently: propositions φ and ψ are logically equivalent whenever $\varphi \Leftrightarrow \psi$ is a tautology. Here's a simple example of logical equivalence:

Example 3.20 ($\neg(p \wedge q) \equiv (p \wedge q) \Rightarrow \neg q$)
In Example 3.14, we found that $(p \wedge q) \Rightarrow \neg q$ is true except when p and q are both true. Thus $\neg(p \wedge q)$ is logically equivalent to $(p \wedge q) \Rightarrow \neg q$, as this truth table shows:

p	q	$(p \wedge q) \Rightarrow \neg q$	$\neg(p \wedge q)$
T	T	F	F
T	F	T	T
F	T	T	T
F	F	T	T

Writing tip: Now that we have a reasonable amount of experience in writing truth tables, we will permit ourselves to skip columns when they're both obvious and not central to the point of a particular example. When you're writing *anything*—whether as a food critic or a Shakespeare scholar or a computer scientist—you should always think about the intended audience, and how much detail is appropriate for them.

Implication, converse, contrapositive, inverse, and mutual implication

We'll now turn to an important question of logical equivalence that involves the proposition $p \Rightarrow q$ and three other implications derived from it:

Definition 3.15 (Converse, Contrapositive, and Inverse)
Consider an implication $p \Rightarrow q$. Then:

- *The* converse *of $p \Rightarrow q$ is the proposition $q \Rightarrow p$.*
- *The* contrapositive *of $p \Rightarrow q$ is the proposition $\neg q \Rightarrow \neg p$.*
- *The* inverse *of $p \Rightarrow q$ is the proposition $\neg p \Rightarrow \neg q$.*

These three new implications derived from the original implication $p \Rightarrow q$—particularly the converse and the contrapositive—will arise frequently. Let's compare the three new implications to the original in light of logical equivalence:

		proposition	*converse*	*contrapositive*	*inverse*
p	q	$p \Rightarrow q$	$q \Rightarrow p$	$\neg q \Rightarrow \neg p$	$\neg p \Rightarrow \neg q$
T	T	T	T	T	T
T	F	F	T	F	T
F	T	T	F	T	F
F	F	T	T	T	T

Figure 3.11: The truth table for an implication and its contrapositive, converse, and inverse.

Example 3.21 (Implications, contrapositives, converses, inverses)
Problem: Consider the implication $p \Rightarrow q$. Which of the converse, contrapositive, and inverse of $p \Rightarrow q$ are logically equivalent to the original proposition $p \Rightarrow q$?

Solution: To answer this question, let's build the truth table; see Figure 3.11. Thus the proposition $p \Rightarrow q$ is logically equivalent to its contrapositive $\neg q \Rightarrow \neg p$, but *not* to its inverse or its converse.

Here's a real-world example to make these results more intuitive:

Thanks to Jeff Ondich for Example 3.22.

Example 3.22 (Contrapositives, converses, and inverses)
Consider the following (true!) proposition, of the form $p \Rightarrow q$:

$$\textit{If } \underbrace{\textit{you were President of the U.S. in 2006}}_{p}\textit{, then } \underbrace{\textit{your name is George}}_{q}\textit{.}$$

The contrapositive of this proposition is $\neg q \Rightarrow \neg p$, which is also true:

If your name isn't George, then you weren't President of the U.S. in 2006.

But the converse $q \Rightarrow p$ and the inverse $\neg p \Rightarrow \neg q$ are both blatantly false:

If your name is George, then you were President of the U.S. in 2006.
If you weren't President of the U.S. in 2006, then your name isn't George.

Consider, for example, George Clooney, Saint George, George Lucas, and Curious George—all named George, and none the President in 2006.

For emphasis, let's summarize the results from Example 3.21. Any implication $p \Rightarrow q$ is logically equivalent to its contrapositive $\neg q \Rightarrow \neg p$, but it is *not* logically equivalent to its converse $q \Rightarrow p$ or its inverse $\neg p \Rightarrow \neg q$. You might notice, though, that the inverse of $p \Rightarrow q$ is the contrapositive of the converse of $p \Rightarrow q$ (!), so the inverse and the converse *are* logically equivalent to each other.

Here's another example of the concepts of tautology and satisfiability, as they relate to implications and converses:

Example 3.23 (Mutual implication)

Problem: Consider the conjunction of the implication $p \Rightarrow q$ and its converse: in other words, consider $(p \Rightarrow q) \wedge (q \Rightarrow p)$. Is this proposition a tautology? Satisfiable? Unsatisfiable? Is there a simpler proposition to which it's logically equivalent?

Solution: We can answer this question with a truth table:

p	q	$p \Rightarrow q$	$q \Rightarrow p$	$(p \Rightarrow q) \wedge (q \Rightarrow p)$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

Because there is a "T" in its column, $(p \Rightarrow q) \wedge (q \Rightarrow p)$ *is* satisfiable (and thus isn't a contradiction). But that column does contain an "F" as well, and therefore $(p \Rightarrow q) \wedge (q \Rightarrow p)$ is *not* a tautology.

Notice that the truth table for $(p \Rightarrow q) \wedge (q \Rightarrow p)$ is identical to the truth table for $p \Leftrightarrow q$. (See Figure 3.4.) Thus $p \Leftrightarrow q$ and $(p \Rightarrow q) \wedge (q \Rightarrow p)$ are logically equivalent. (And $\Leftrightarrow$ is called *mutual implication* for this reason: p and q imply each other.)

Some other logically equivalent statements

Figure 3.12 contains a large collection of logical equivalences. These equivalences may use some unfamiliar terminology, which we'll define here. Informally, an operator is *commutative* if the order of its arguments doesn't matter; an operator is *associative* if the way we parenthesize successive applications doesn't matter; and an operator is *idempotent* if applying it to the same argument twice gives that argument back. (In addition to these definitions, there are two other frequently discussed concepts: the *identity* and the *zero* of the operator; logical equivalences involving identities and zeros were left to you, in Exercises 3.13–3.22.) For each equivalence in Figure 3.12, it's worth

Latin: *idem* "same" +*potent* "strength."

Commutativity	$p \vee q \equiv q \vee p$ $p \wedge q \equiv q \wedge p$ $p \oplus q \equiv q \oplus p$ $p \Leftrightarrow q \equiv q \Leftrightarrow p$
Associativity	$p \vee (q \vee r) \equiv (p \vee q) \vee r$ $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$ $p \oplus (q \oplus r) \equiv (p \oplus q) \oplus r$ $p \Leftrightarrow (q \Leftrightarrow r) \equiv (p \Leftrightarrow q) \Leftrightarrow r$
Idempotence	$p \vee p \equiv p$ $p \wedge p \equiv p$

Distribution of $\wedge$ over $\vee$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Distribution of $\vee$ over $\wedge$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
Contrapositive	$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$
	$p \Rightarrow q \equiv \neg p \vee q$ $p \Rightarrow (q \Rightarrow r) \equiv p \wedge q \Rightarrow r$ $p \Leftrightarrow q \equiv \neg p \Leftrightarrow \neg q$
Mutual Implication	$(p \Rightarrow q) \wedge (q \Rightarrow p) \equiv p \Leftrightarrow q$
De Morgan's Laws	$\neg(p \wedge q) \equiv \neg p \vee \neg q$ $\neg(p \vee q) \equiv \neg p \wedge \neg q$

Figure 3.12: Some logically equivalent propositions.

taking a few minutes to think about why the two propositions are logically equivalent. See also Exercises 3.73–3.82.

De Morgan's Laws are named after Augustus De Morgan, a 19th-century British mathematician.

> **Taking it further:** There are at least two ways in which the types of logical equivalences shown in Figure 3.12 play an important role in programming. (See the discussion on p. 327.) First, most modern languages have a feature called *short-circuit evaluation* of logical expressions—they evaluate conjunctions and disjunctions from left to right, and stop as soon as the truth value of the logical expression is known—and programmers can exploit this feature to make their code cleaner or more efficient. Second, in compiled languages, an optimizing compiler can make use of logical equivalences to simplify the machine code that ends up being executed.

3.3.3 *Representing Propositions: Circuits and Normal Forms*

Now that we've established the core concepts of propositional logic, we'll turn to some bigger and more applied questions. We'll spend the rest of this section exploring two specific ways of representing propositions: *circuits*, the wires and connections from which physical computers are built; and two *normal forms*, in which the structure of propositions is restricted in a particular way.

The approach we're taking with normal forms is a commonly used idea to make reasoning about some language L easier: we define a *subset* S of L, with two goals: (1) any statement in L is equivalent to some statement in S; and (2) S is "simple" in some way. Then we can consider any statement from the "full" language L, which we can then "translate" into a simple-but-equivalent statement of S. Defining this subset and its accompanying translation will make it easier to accomplish some task for *all* expressions in L, while still making it easy to write statements clearly.

> **Taking it further:** The idea of translating all propositions into a particular form has a natural analogue in designing and implementing programming languages. For example, every **for** loop can be expressed as a **while** loop instead, but it would be very annoying to program in a language that doesn't have **for** loops. A nice compromise is to allow **for** loops, but behind the scenes to translate each **for** loop into a **while** loop. This compromise makes the language easier for the "user" programmer to use (**for** loops exist!) *and* also makes the job of the programmer of the compiler/interpreter easier (she can worry exclusively about implementing and optimizing **while** loops!).
>
> In programming languages, this translation is captured by the notion of *syntactic sugar*. (The phrase is meant to suggest that the addition of **for** to the language is a bonus for the programmer—"sugar on top," maybe—that adds to the syntax of the language.) The programming language Scheme is perhaps the pinnacle of syntactic sugar; the core language is almost unbelievably simple. Here's one illustration: `(and x y)` (Scheme for "$x \wedge y$") is syntactic sugar for `(if x y #f)` (that's "if x then y else false"). So a Scheme programmer can use `and`, but there's no "real" `and` that has to be handled by the interpreter.

Circuits

We'll introduce the idea of circuits by using the proposition $(p \wedge \neg q) \vee (\neg p \wedge q)$ as an example. (Note, by the way, that this proposition is logically equivalent to $p \oplus q$.)

Observe that the stated proposition is a disjunction of two smaller propositions, $p \wedge \neg q$ and $\neg p \wedge q$. Similarly, $p \wedge \neg q$ is a conjunction of two even simpler propositions, namely p and $\neg q$. A representation of a proposition called a *tree* continues to break down every compound proposition embedded within it. (We'll talk about trees in detail in Chapter 11.) The tree for $(p \wedge \neg q) \vee (\neg p \wedge q)$ is shown in Figure 3.13. The tree-based view isn't much of a change from our usual notation $(p \wedge \neg q) \vee (\neg p \wedge q)$; all we've done is use the parentheses and order-of-operation rules to organize the logical connectives. But this representation is closely related to a very important way of viewing logical propositions: *circuits.*

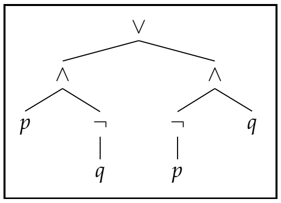


Figure 3.13: A tree-based view of $(p \wedge \neg q) \vee (\neg p \wedge q)$.

Figure 3.14 shows the same proposition redrawn as a collection of *wires* and *gates*. Wires carry a truth value from one physical location to another; gates are physical implementations of logical connectives. We can think of truth values "flowing in" as inputs to the left side of each gate, and a truth value "flowing out" as output from the right side of the gate. (The only substantive difference between Figures 3.13 and 3.14—aside from which way is up—is whether the two p inputs come from the same wire, and likewise whether the two q inputs do.)

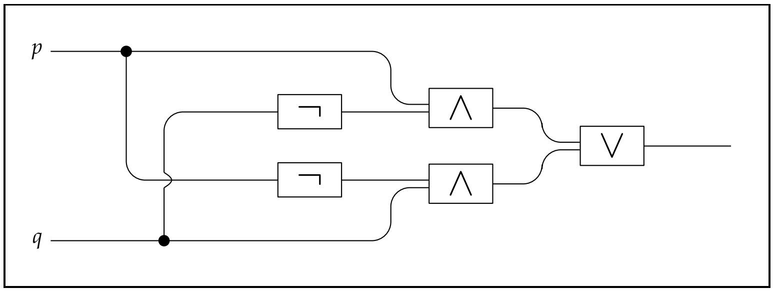
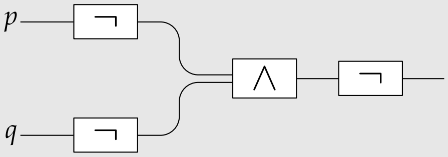


Figure 3.14: A circuit-based view.

Example 3.24 (Using and and not for or)

Problem: Build a circuit for $p \vee q$ using only $\wedge$ and $\neg$ gates.

Solution: We'll use one of De Morgan's Laws, which says that $p \vee q \equiv \neg(\neg p \wedge \neg q)$:



This basic idea—of replacing one logical connective by another one (or by multiple other ones)—is a crucial part of the construction of computers themselves; we'll return to this idea in Section 4.4.1.

Conjunctive and Disjunctive Normal Forms

In the rest of this section, we'll consider a way to simplify propositions: *conjunctive* and *disjunctive normal forms*, which constrain propositions to have a particular format. To define these restricted types of propositions, we need a basic definition: a *literal* is a Boolean variable (a.k.a. an atomic proposition) or the negation of a Boolean variable. (So p and $\neg p$ are both literals.)

Definition 3.16 (Conjunctive normal form)
A proposition is in conjunctive normal form (CNF) *if it is the conjunction of one or more* clauses, *where each* clause *is the disjunction of one or more literals.*

Definition 3.17 (Disjunctive normal form)
A proposition is in disjunctive normal form (DNF) *if it is the disjunction of one or more* clauses, *where each* clause *is the conjunction of one or more literals.*

Less formally, a proposition in conjunctive normal form is "the and of a bunch of ors," and a proposition in disjunctive normal form is "the or of a bunch of ands."

> **Taking it further:** In computer architecture and digital electronics, people usually refer to a proposition in CNF as being a *product of sums*, and a proposition in DNF as being a *sum of products*. (There is a deep way of thinking about formal logic based on $\wedge$ as multiplication, $\vee$ as addition, 0 as False, and 1 as True; see Exercises 3.23–3.26.)

Here is a simple example of both CNF and DNF:

Example 3.25 (Simple propositions in CNF and DNF)
The proposition $(\neg p \vee q \vee r) \wedge (\neg q \vee \neg r) \wedge (r)$ is in conjunctive normal form. It has three clauses: $\neg p \vee q \vee r$ and $\neg q \vee \neg r$ and r.

The proposition $(\neg p \wedge q \wedge r) \vee (\neg q \wedge \neg r) \vee (r)$ is in disjunctive normal form, again with three clauses: $\neg p \wedge q \wedge r$ and $\neg q \wedge \neg r$ and r.

While conjunctive and disjunctive normal forms seem like heavy restrictions on the format of propositions, it turns out that *every* proposition is logically equivalent to a CNF proposition and to a DNF proposition:

Theorem 3.1 (All propositions are expressible in CNF)
For any proposition φ, there is a proposition φ_{cnf} over the same Boolean variables and in conjunctive normal form such that $\varphi \equiv \varphi_{cnf}$.

Theorem 3.2 (All propositions are expressible in DNF)
For any proposition φ, there is a proposition ψ_{dnf} over the same Boolean variables and in disjunctive normal form such that $\varphi \equiv \psi_{dnf}$.

These two theorems are perhaps the first results that we've encountered that are unexpected, or at least unintuitive. There's no particular reason for it to be clear that they're true—let alone how we might prove them. But we can, and we will: we'll prove both theorems in Section 4.4.1 and again in Section 5.4.3, after we've introduced some relevant proof techniques. But, for now, here are a few examples of translating propositions into DNF/CNF.

Problem-solving tip: A good strategy when you're trying to prove a not-at-all-obvious claim is to test out some small examples, and then try to start to figure a general pattern.

Example 3.26 (Translating basic connectives into DNF)

Problem: Give propositions in disjunctive normal form that are logically equivalent to each of the following:

1. $p \vee q$
2. $p \wedge q$
3. $p \Rightarrow q$
4. $p \Leftrightarrow q$

Solution: 1 & 2. These questions are boring: both propositions are already in DNF, with 2 clauses (p and q) and 1 clause ($p \wedge q$), respectively.

3. Figure 3.12 tells us that $p \Rightarrow q \equiv \neg p \vee q$, and $\neg p \vee q$ is in DNF.
4. The proposition $p \Leftrightarrow q$ is true when p and q are either both true or both false, and false otherwise. So we can rewrite $p \Leftrightarrow q$ as $(p \wedge q) \vee (\neg p \wedge \neg q)$. We can check that we've gotten this proposition right with a truth table:

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

And here's the task of translating basic logical connectives into CNF:

Example 3.27 (Translating basic connectives into CNF)

Problem: Give propositions in conjunctive normal form that are logically equivalent to each of the following:

1. $p \Rightarrow q$
2. $p \Leftrightarrow q$
3. $p \oplus q$

(Note that, as with DNF, both $p \vee q$ and $p \wedge q$ are already in CNF.)

Solution: 1. As above, we know that $p \Rightarrow q \equiv \neg p \vee q$, and $\neg p \vee q$ is also in CNF.

2. We can rewrite $p \Leftrightarrow q$ as follows:

$$\begin{aligned} p \Leftrightarrow q &\equiv (p \Rightarrow q) \wedge (q \Rightarrow p) && \textit{mutual implication (Example 3.23)} \\ &\equiv (\neg p \vee q) \wedge (\neg q \vee p) && x \Rightarrow y \equiv \neg x \vee y \textit{ (Figure 3.12), used twice} \end{aligned}$$

The proposition $(\neg p \vee q) \wedge (\neg q \vee p)$ is in CNF.

3. Because $p \oplus q$ is true as long as one of $\{p, q\}$ is true and one of $\{p, q\}$ is false, it's easy to verify via truth table that $p \oplus q \equiv (p \vee q) \wedge (\neg p \vee \neg q)$, which is in CNF.

We've only given some examples of converting a (simple) proposition into a new proposition, logically equivalent to the original, that's in either CNF or DNF. We will figure out how to generalize this technique to *any* proposition in Section 4.4.1.

Computer Science Connections

Computational Complexity, Satisfiability, and $1,000,000

Complexity theory is the subfield of computer science devoted to understanding the resources—time and memory, usually—necessary to solve particular problems. It's the subject of a great deal of fascinating current research in theoretical computer science.[3] Here is a central problem of complexity theory, the *satisfiability problem:*

Given: A Boolean formula φ over variables $p_1, p_2, \ldots, p_n$.
Output: Is φ satisfiable?

The satisfiability problem is pretty simple to solve. In fact, we've implicitly described an algorithm for this problem already:

- construct the truth table for the n-variable proposition φ; and
- check to see whether there are any "T"s in φ's column of the table.

But this algorithm is not very fast, because the truth table for φ has lots and lots of rows—2^n rows, to be precise. (We've already seen this for $n = 1$, for negation, and $n = 2$, for all the binary connectives, with $2^1 = 2$ and $2^2 = 4$ rows each; in Chapter 9, we'll address this counting issue formally.) And then even a moderate value of n means that this algorithm will not terminate in your lifetime; 2^{300} exceeds the number of particles in the known universe.

So, it's clear that there is an algorithm that solves the SAT problem. What's not clear is whether there is a substantially more efficient algorithm to solve the SAT problem. It's so unclear, in fact, that nobody knows the answer, and this question is one of the biggest open problems in computer science and mathematics today. (Arguably, it's *the* biggest.) The Clay Mathematics Institute will even give a $1,000,000 prize to anyone who solves it.

Why is this problem so important? The reason is that, in a precise technical sense, SAT is *just as hard* as a slew of other problems that have a plethora of unspeakably useful applications: the traveling salesman problem, protein folding, optimally packing the trunk of a car with suitcases. This slew is a class of computational problems known as NP ("nondeterministic polynomial time"), for which it is easy to "verify" correct answers. In the context of SAT, that means that whenever you've got a satisfiable proposition φ, it's very easy for you to (efficiently) convince me that φ is satisfiable. Here's how: you'll simply tell me a truth assignment under which φ evaluates to true. And I can make sure that you didn't try to fool me by plugging and chugging: I substitute your truth assignment in for every variable, and then I make sure that the final truth value of φ is indeed True.

One of the most important results in theoretical computer science in the 20th century—that's saying something for a field that was founded in the 20th century!—is the *Cook–Levin Theorem:*[4] *if one can solve SAT efficiently, then one can solve* any *problem in* NP *efficiently.* The major open question is what's known as the *P-versus-NP question*. A problem that's in P is easy to solve from scratch. A problem that's in NP is easy to verify (in the way described above). So the question is: does P = NP? Is verifying an answer to a problem no easier than solving the problem from scratch? (It seems intuitively "clear" that the answer is no—but nobody has been able to prove it!)

You can read more about complexity theory in general, and the P-versus-NP question addressed here in particular, in most books on algorithms or the theory of computing. Some excellent places to read more are:

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009; Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison–Wesley, 2006; and Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 3rd edition, 2012.

[4] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971; and Leonid Levin. Universal search problems. *Problems of Information Transmission*, 9(3):265–266, 1973. In Russian.

Computer Science Connections

Short-Circuit Evaluation, Optimization, and Modern Compilers

The logical equivalences in Figure 3.12 may seem far removed from "real" programming, but logical equivalences are actually central in modern programming. Here are two ways in which they play an important role:

Short-circuit evaluation: In most modern programming languages, a logical expression involving **and**s and **or**s will only be evaluated until the truth value of the expression can be determined. For an example in Java, see Figure 3.15. Like most modern languages, Java evaluates an $\wedge$ expression from left to right and stops as soon as it finds a false conjunct. Similarly, Java evaluates an $\vee$ expression from left to right and stops as soon as it finds a true disjunct, because $\text{True} \vee \textit{anything} \equiv \text{True}$. This style of evaluation is called *short-circuit evaluation*.

```
if (2 > 3 && x + y < 9) {
  ...
} else {
  ...
}
```

Figure 3.15: A snippet of Java code. In Java, && denotes $\wedge$ and `||` denotes $\vee$. The second conjunct of the `if` condition will actually never be evaluated, because `2 > 3` is false, and $\text{False} \wedge \textit{anything} \equiv \text{False}$.

Two slick ways in which programmers can take advantage of short-circuit evaluation are shown in Figure 3.16.

- Lines 1–4 use short-circuit evaluation to avoid deeply nested **if** statements to handle exceptional cases. When $x = 0$, evaluating the second disjunct would cause a divide-by-zero error—but the second disjunct isn't evaluated when $x = 0$ because the first disjunct was true!
- Lines 6–9 use short-circuit evaluation to make code faster. If the second conjunct typically takes much longer to evaluate (or if it is much more frequently true) than the first conjunct, then careful ordering of conjuncts avoids a long and usually fruitless computation.

```
1  if (x == 0
2      || (x-1) / x > 0.5) {
3    ...
4  }
5
6  if (simpleOrOftenFalse(x)
7      && complexOrOftenTrue(x)) {
8    ...
9  }
```

Figure 3.16: Two handy ways to rely on short-circuit evaluation.

Compile-time optimization: For a program written in a compiled language like C, the source code is translated into machine-readable form by the *compiler*. But this translation is not verbatim; instead, the compiler streamlines your code (when it can!) to make it run faster.

One of the simplest types of compiler optimizations is *constant folding*: if some of the values in an arithmetic or logical expression are constants—known to the compiler at "compile time," and thus unchanged at "run time"—then the compiler can "fold" those constants together. Using the rules of logical or arithmetic equivalence broadens the types of code that can be folded in this way. For example, in C, when you write an assignment statement like `y = x + 2 + 3`, most compilers will translate it into `y = x + 5`. But what about `z = 7 * x * 8`? A modern compiler *will* optimize it into `z = x * 56`, using the commutativity of multiplication. Because the compiler can reorder the multiplicands without affecting the value, and this reordering allows the 7 and 8 to be folded into 56, the compiler does the reordering and the folding.

An example using logical equivalences is shown in Figure 3.17. Because $p \vee \neg p$ is a tautology—the law of the excluded middle—*no matter what the value of* p, the "then" clause is executed, not the "else" clause. Thus the compiler doesn't even have to waste time checking whether p is true or false, and this optimization can be applied.

```
if (p || !p) {  /* "p or not p" */
    x = 51;
} else {
    x = 63;
}
```

```
x = 51;
```

Figure 3.17: Two snippets of C code. When this code is compiled on a modern optimizing compiler (gcc 4.3.4, with optimization turned on), the machine code that is produced is *exactly* identical for both snippets.

3.3.4 Exercises

The operators $\wedge$ and $\vee$ are idempotent (see Figure 3.12)—that is, $p \wedge p \equiv p \vee p \equiv p$. But $\Rightarrow$, $\oplus$, and $\Leftrightarrow$ are not idempotent. Simplify—that is, give as-simple-as-possible propositions that are logically equivalent to—the following:

3.39 $p \Rightarrow p$

3.40 $p \oplus p$

3.41 $p \Leftrightarrow p$

Consider the proposition $p \Rightarrow \neg p \Rightarrow p \Rightarrow q$. Add parentheses to this proposition so that the resulting proposition …

3.42 … is logically equivalent to True (that is, the result is a tautology).

3.43 … is logically equivalent to q.

3.44 Give as simple as possible a proposition logically equivalent to the (unparenthesized) original.

Unlike the binary connectives $\{\wedge, \vee, \oplus, \Leftrightarrow\}$, implication is not associative. In other words, $p \Rightarrow (q \Rightarrow r)$ and $(p \Rightarrow q) \Rightarrow r$ are not *logically equivalent. The next few exercises explore the non-associativity of $\Rightarrow$.*

3.45 Prove that implication is not associative by giving a truth assignment in which $p \Rightarrow (q \Rightarrow r)$ and $(p \Rightarrow q) \Rightarrow r$ have different truth values.

3.46 Consider the propositions $p \Rightarrow (q \Rightarrow q)$ and $(p \Rightarrow q) \Rightarrow q$. One of these is a tautology; one of them is not. Which is which? Prove your answer.

3.47 Consider the propositions $p \Rightarrow (p \Rightarrow q)$ and $(p \Rightarrow p) \Rightarrow q$. Is either one a tautology? Satisfiable? Unsatisfiable? What is the simplest proposition to which each is logically equivalent?

On an exam, I once asked students to write a proposition logically equivalent to $p \oplus q$ using only the logical connectives $\Rightarrow$, $\neg$, and $\wedge$. Here are some of the students' answers. Which ones are right?

3.48 $\neg(p \wedge q) \Rightarrow (\neg p \wedge \neg q)$

3.49 $(p \Rightarrow \neg q) \wedge (q \Rightarrow \neg p)$

3.50 $(\neg p \Rightarrow q) \wedge \neg(p \wedge q)$

3.51 $\neg\left[(p \wedge \neg q \Rightarrow \neg p \wedge q) \wedge (\neg p \wedge q \Rightarrow p \wedge \neg q)\right]$

3.52 Write a proposition logically equivalent to $p \oplus q$ using only the logical connectives $\Rightarrow$, $\neg$, and $\vee$.

The following code uses nested conditionals, or compound propositions as conditions. Simplify each as much as possible. (For example, if $p \Rightarrow q$, it's a waste of time to test whether q holds in a block where p is known to be true.)

3.53

```
if (x > 20
      or (x <= 20 and y < 0))
then foo(x,y)
else bar(x,y)
```

3.54

```
if (y >= 0
      or y <= x
      or (x - y) * y >= 0)
  then foo(x,y)
  else bar(x,y)
```

3.55

```
if (x % 12 == 0):
  then if not (x % 4 == 0):
          then foo(x)
          else bar(x)
  else if (x == 17):
          then baz(x)
          else quz(x)
```

(Note that `x % k == 0` is true when $x \bmod k = 0$, also known as when $k \mid x$.)

Simplify the following propositions as much as possible.

3.56 $(\neg p \Rightarrow q) \wedge (q \wedge p \Rightarrow \neg p)$

3.57 $(p \Rightarrow \neg p) \Rightarrow ((q \Rightarrow (p \Rightarrow p)) \Rightarrow p)$

3.58 $(p \Rightarrow p) \Rightarrow (\neg p \Rightarrow \neg p) \wedge q$

3.59 Is the following claim true or false? Prove your answer.

Claim: Every proposition over the single variable p is either logically equivalent to p or it is logically equivalent to $\neg p$.

Show using truth tables that these propositions from Figure 3.10 are tautologies:

3.60 $(p \Rightarrow q) \wedge \neg q \Rightarrow \neg p$ (Modus Tollens)

3.61 $p \Rightarrow p \vee q$

3.62 $p \wedge q \Rightarrow p$

3.63 $(p \vee q) \wedge \neg p \Rightarrow q$

3.64 $(p \Rightarrow q) \wedge (\neg p \Rightarrow q) \Rightarrow q$

3.65 $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$

3.66 $(p \Rightarrow q) \wedge (p \Rightarrow r) \Leftrightarrow p \Rightarrow q \wedge r$

3.67 $(p \Rightarrow q) \vee (p \Rightarrow r) \Leftrightarrow p \Rightarrow q \vee r$

3.68 $p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$

3.69 $p \Rightarrow (q \Rightarrow r) \Leftrightarrow p \wedge q \Rightarrow r$

Show that the following propositions are tautologies:

3.70 $p \vee (p \wedge q) \Leftrightarrow p$

3.71 $p \wedge (p \vee q) \Leftrightarrow p$

3.72 $p \oplus q \Rightarrow p \vee q$

Prove De Morgan's Laws:

3.73 $\neg(p \wedge q) \equiv \neg p \vee \neg q$

3.74 $\neg(p \vee q) \equiv \neg p \wedge \neg q$

Show the following logical equivalences regarding associativity using truth tables:

3.75 $p \vee (q \vee r) \equiv (p \vee q) \vee r$

3.76 $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$

3.77 $p \oplus (q \oplus r) \equiv (p \oplus q) \oplus r$

3.78 $p \Leftrightarrow (q \Leftrightarrow r) \equiv (p \Leftrightarrow q) \Leftrightarrow r$

Show using truth tables that the following logical equivalences hold:

3.79 $p \Rightarrow q \equiv \neg p \vee q$

3.80 $p \Rightarrow (q \Rightarrow r) \equiv p \wedge q \Rightarrow r$

3.81 $p \Leftrightarrow q \equiv \neg p \Leftrightarrow \neg q$

3.82 $\neg(p \Rightarrow q) \equiv p \wedge \neg q$

3.83 On p. 327, we discussed the use of tautologies in optimizing compilers. In particular, these compilers will perform the following optimization, transforming the first block of code into the second:

```
if (p || !p) {  /* "p or not p" */
   x = 51;
} else {
   x = 63;
}
```

```
x = 51;
```

The compiler performs this transformation because $p \vee \neg p$ is a tautology—no matter what the truth value of p, the proposition $p \vee \neg p$ is true. But there *are* situations in which this code translation actually changes the behavior of the program, *if* `p` *can be an arbitrary expression* (rather than just a Boolean variable)! Describe such a situation. (*Hint: why do (some) people watch auto racing?*)

The unknown circuit in Figure 3.18 takes three inputs $\{p, q, r\}$, and either turns on a light bulb (output of the circuit = true) or leaves it off (output = false). For each of the following, draw a circuit—using at most three $\wedge$, $\vee$, and $\neg$ gates—that is consistent with the listed behavior. The light's status is unknown for unlisted inputs. (If multiple circuits are consistent with the given behavior, draw any one them.)

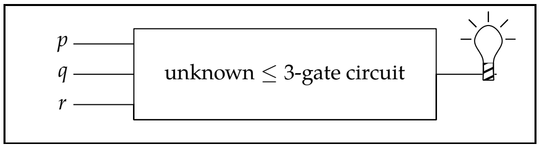


Figure 3.18: A circuit with at most 3 gates.

3.84 The light is on when the true inputs are $\{q\}$ or $\{r\}$. The light is off when the true inputs are $\{p\}$ or $\{p, q\}$ or $\{p, q, r\}$.

3.85 The light is on when the true inputs are $\{p, q\}$ or $\{p, r\}$. The light is off when the true inputs are $\{p\}$ or $\{q\}$ or $\{r\}$.

3.86 The light is off when the true inputs are $\{p\}$ or $\{q\}$ or $\{r\}$ or $\{p, q, r\}$.

3.87 The light is off when the true inputs are $\{p, q\}$ or $\{p, r\}$ or $\{q, r\}$ or $\{p, q, r\}$.

3.88 Consider a simplified class of circuits like those from Exercises 3.84–3.87: there are *two* inputs $\{p, q\}$ and at most *two* gates, each of which is $\wedge$, $\vee$, or $\neg$. There are a total of $2^4 = 16$ distinct propositions over inputs $\{p, q\}$: four different input configurations, each of which can turn the light on or leave it off. Which, if any, of these 16 propositions *cannot* be expressed using up to two $\{\wedge, \vee, \neg\}$ gates?

3.89 (*programming required*) Consider the class of circuits from Exercises 3.84–3.87: inputs $\{p, q, r\}$, and at most three gates chosen from $\{\wedge, \vee, \neg\}$. There are a total of $2^8 = 256$ distinct propositions over inputs $\{p, q, r\}$: eight different input configurations, each of which can turn the light on or leave it off. Write a program to determine how many of these 256 propositions can be represented by a circuit of this type. (If you design it well, your program will let you check your answers to Exercises 3.84–3.88.)

3.90 Consider a set $S = \{p, q, r, s, t\}$ of Boolean variables. Let $\varphi = p \oplus q \oplus r \oplus s \oplus t$. Describe *briefly* the conditions under which φ is true. Use English and, if appropriate, standard (nonlogical) mathematical notation. (*Hint: look at the symbol $\oplus$ itself. What's $p + q + r + s + t$, treating true as 1 and false as 0 as in Exercises 3.23–3.26?*)

3.91 *Dithering* is a technique for converting grayscale images to black-and-white images (for printed media like newspapers). The classic dithering algorithm proceeds as follows. For every pixel in the image, going from top to bottom ("north to south"), and from left to right ("west to east"):

- "Round" the current pixel to black or white. (If it's closer to black, make it black; if it's closer to white, make it white.)
- This alteration to the current pixel has created "rounding error" x (in other words, we have added $x > 0$ "whiteness units" by making it white, or $x < 0$ "whiteness units" by making it black). We compensate for this adding a total of $-x$ "whiteness units," distributed among the neighboring pixels to the "east" (add $-7x/16$ to the eastern neighboring pixel) "southwest" ($-3x/16$), "south" ($-5x/16$) and "southeast" ($-x/16$). If any of these neighboring pixels don't exist (because the current pixel is on the border of the image), simply ignore the corresponding fraction of $-x$ (and don't add it anywhere).

I assigned a dithering exercise in an introductory CS class, and I got, more or less, the code in Figure 3.19 from one student. This code is correct, but it is very repetitious. Reorganize this code so that it's not so repetitive. In particular, rewrite lines 7–63 ensuring that each "distribute the error" line (9, 11, 12, and 13) appears *only once* if your solution.

```
1  for y = 1 ... height:
2    for x = 1 ... width:
3      if P[x,y] is more white than black:
4        $error$ = "white" - P[x,y]
5        P[x,y] = "white"
6
7        if x > 1:
8          if x < width and not (y < height):
9            add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
10         else if x < width and y < height:
11           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
12           add $\frac{3}{16} \cdot error$ to P[x+1,y+1]  (SE)
13           add $\frac{1}{16} \cdot error$ to P[x-1,y+1]  (SW)
14           add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
15         else if y < height
16                   and not (x < width):
17           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
18           add $\frac{1}{16} \cdot error$ to P[x-1,y+1]  (SW)
19         else:
20           do nothing
21       else:
22         if x < width and not (y < height):
23           add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
24         else if x < width and y < height:
25           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
26           add $\frac{3}{16} \cdot error$ to P[x+1,y+1]  (SE)
27           add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
28         else if y < height
29                   and not (x < width):
30           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
31         else:
32           do nothing
33
34     else:  # P[x,y] is closer to "black"
35       $error$ = "black" - P[x,y]
36       P[x,y] = "black"
37
38       if x > 1:
39         if x < width and not (y < height):
40           add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
41         else if x < width and y < height:
42           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
43           add $\frac{3}{16} \cdot error$ to P[x+1,y+1]  (SE)
44           add $\frac{1}{16} \cdot error$ to P[x-1,y+1]  (SW)
45           add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
46         else if y < height
47                   and not (x < width):
48           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
49           add $\frac{1}{16} \cdot error$ to P[x-1,y+1]  (SW)
50         else:
51           do nothing
52       else:
53         if x < width and not (y < height):
54           add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
55         else if x < width and y < height:
56           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
57           add $\frac{3}{16} \cdot error$ to P[x+1,y+1]  (SE)
58           add $\frac{7}{16} \cdot error$ to P[x+1,y]    (E)
59         else if y < height
60                   and not (x < width):
61           add $\frac{5}{16} \cdot error$ to P[x,y+1]    (S)
62         else:
63           do nothing
```

Figure 3.19: Some dithering code.

Recall Definition 3.16: a proposition φ is in conjunctive normal form (CNF) if φ is the conjunction of one or more clauses, where each clause is the disjunction of one or more literals, and where a literal is an atomic proposition or its negation. Further, recall Definition 3.17: φ is in disjunctive normal form (DNF) if φ is the disjunction of one or more clauses, where each clause is the conjunction of one or more literals.

Give a proposition in disjunctive normal form that's logically equivalent to . . .

3.92 $\neg(p \wedge q) \Rightarrow r$

3.93 $p \wedge (q \vee r) \Rightarrow (q \wedge r)$

3.94 $p \vee \neg(q \Leftrightarrow p \wedge r)$

3.95 $p \oplus (\neg p \Rightarrow (q \Rightarrow r) \wedge \neg r)$

Give a proposition in conjunctive normal form that's logically equivalent to . . .

3.96 $\neg(p \wedge q) \Rightarrow r$

3.97 $p \wedge (q \Rightarrow (r \Rightarrow q \oplus r))$

3.98 $(p \Rightarrow q) \Rightarrow (q \Rightarrow r \wedge p)$

3.99 $p \Leftrightarrow (q \vee r \vee \neg p)$

A CNF proposition φ is in 3CNF *if each clause contains* exactly three *distinct literals. (Note that p and $\neg p$ are distinct literals.) In terms of the number of clauses, what's the smallest 3CNF formula . . .*

3.100 . . . that's a tautology?

3.101 . . . that's not satisfiable?

Consider the set of 3CNF propositions over the variables $\{p, q, r\}$ for which no clause appears more than once. (Exercises 3.102–3.104 turn out to be boring without the restriction of no repeated clauses; we could repeat the same clause as many times as we please: $(p \vee q \vee r) \wedge (p \vee q \vee r) \wedge (p \vee q \vee r) \cdots$.) Two clauses that contain precisely the same literals (in any order) do not count as distinct. (But recall that a single clause can *contain a variable in both negated and unnegated form.) In terms of the number of clauses, what's the largest* 3*-variable distinct-clause 3CNF proposition . . .*

3.102 . . . at all (with no further restrictions)?

3.103 . . . that's a tautology?

3.104 . . . that's satisfiable?

A proposition φ is in 3DNF *if it is the disjunction of one or more clauses, each of which is the conjunction of exactly three distinct literals. In terms of the number of clauses, what's the smallest 3DNF formula . . .*

3.105 . . . that's a tautology?

3.106 . . . that's not satisfiable?

3.4 An Introduction to Predicate Logic

> But the fact that some geniuses were laughed at does not imply that all who are laughed at are geniuses. They laughed at Columbus, they laughed at Fulton, they laughed at the Wright brothers. But they also laughed at Bozo the Clown.
>
> Carl Sagan (1934–1996)
> *Broca's Brain: Reflections on the Romance of Science* (1979)

Propositional logic, which we have been discussing thus far, gives us formal notation to encode Boolean expressions. But these expressions are relatively simple, a sort of "unstructured programming" style of logic. *Predicate logic* is a more general type of logic that allows us to write function-like logical expressions called *predicates*, and to express a broader range of notions than in propositional logic.

3.4.1 Predicates

Informally, a predicate is a property that a particular entity might or might not have; for example, *being a vowel* is a property that some letters do have (`A`, `E`, ...) and some letters do not have (`B`, `C`, ...). A predicate isn't the kind of thing that's true or false, so predicates are different from propositions; rather, a predicate is like a "proposition with blanks" waiting to be filled in. For example:

Example 3.28 (Some predicates)

- "The integer ___ is prime."
- "The string ___ is a palindrome."
- "The person ___ costarred in a movie with Kevin Bacon."
- "The string ___ is alphabetically after the string ___."
- "The integer ___ evenly divides the integer ___."

Once the blanks of a predicate are filled in, the resulting expression is a proposition. Here are some examples of propositions—some true, some false—derived from the predicates in Example 3.28:

Example 3.29 (Some propositions derived from Example 3.28)

- "The integer 57 is prime."
- "The string `TENET` is a palindrome."
- "The person Sean Connery costarred in a movie with Kevin Bacon."
- "The string `PYTHON` is alphabetically after the string `PYTHAGOREAN`."
- "The integer 17 evenly divides the integer 42."

We can now give a formal definition of predicates:

Definition 3.18 (Predicate)
A predicate *P is a Boolean-valued function—that is, P is a function* $P : U \to \{\text{True}, \text{False}\}$ *for a set U. The set U is called the* universe *or the* domain of discourse, *and we say that P is a predicate* over U.

When the universe U is clear from context, we will allow ourselves to be sloppy with notation by leaving U implicit.

Although we didn't use the name at the time, we've already encountered predicates, in Chapter 2. Definition 2.18 introduced the notation $\{x \in U : P(x)\}$ to denote the set of those objects $x \in U$ for which P is true. The set abstraction notation "selects" the elements of U for which the predicate P is true.

Example 3.30 (Some example predicates)
Here are a few more sample predicates based on arithmetic:

1. $\mathit{isPrime}(n)$: the positive integer n is a prime number.
2. $\mathit{isPowerOf}(n,k)$: the integer n is an exact power of k: $n = k^i$ for some $i \in \mathbb{Z}^{\geq 0}$.
3. $\mathit{onlyPowersOfTwo}(S)$: every element of the set S is a power of two.
4. $Q(n,a,b)$: positive integer n satisfies $n = a + b$, and integers a and b are both prime.
5. $\mathit{sumOfTwoPrimes}(n)$: positive integer n is equal to the sum of two prime numbers.

(To reiterate Definition 3.18, the *isPrime* predicate, for example, is a function $\mathit{isPrime} : \mathbb{Z}^{>0} \to \{\text{True}, \text{False}\}$.)

Deriving propositions from predicates

Again, by plugging particular values into the predicates from Example 3.30, we get propositions, each of which has a truth value:

Example 3.31 (Propositions derived from predicates)
Using the predicates in Example 3.30, let's figure out the truth values of the propositions $\mathit{isPrime}(261)$, $\mathit{isPrime}(262)$, $Q(8,3,5)$, and $Q(9,3,6)$. For each, we'll simply plug the given arguments into the definition of the predicate and figure out the truth value of the resulting proposition.

- A little arithmetic shows that $261 = 3 \cdot 87$; thus $\mathit{isPrime}(261) = \text{False}$.
- Similarly, we have $262 = 2 \cdot 131$, so $\mathit{isPrime}(262) = \text{False}$.
- To compute the truth value of $Q(8,3,5)$, we simply plug $n = 8$, $a = 3$, and $b = 5$ into the definition of $Q(n,a,b)$. The proposition $Q(8,3,5)$ requires that *the positive integer* 8 *satisfies* $8 = 3 + 5$, *and the integers* 3 *and* 5 *are both prime.* All of the requirements are met, so $Q(8,3,5) = \text{True}$.
- On the other hand, $Q(9,3,6) = \text{False}$ because $Q(9,3,6)$ requires that $9 = 3 + 6$, *and* that the integers 3 and 6 are both prime. But 6 isn't prime.

Just like the propositional logical connectives, each predicate takes a fixed number of arguments. So a predicate might be *unary* (taking one argument, like the predicate *isPrime*); or *binary* (taking two arguments, like *isPowerOf*); or *ternary* (taking three arguments, like Q from Example 3.30); and so forth. Here are a few more examples:

Example 3.32 (More propositions derived from predicates)

Problem: Using the predicates in Example 3.30, find the truth values of these propositions:

1. *sumOfTwoPrimes*(17) and *sumOfTwoPrimes*(34)
2. *isPowerOf* (16, 2) and *isPowerOf* (2, 16)
3. *onlyPowersOfTwo*($\{1, 2, 8, 128\}$)

Solution: As before, we just plug the given arguments into the definition:

1. *sumOfTwoPrimes*(17) = False: the only way to get an odd number n by adding two prime numbers is for one of those prime numbers to be 2—but $17 - 2 = 15$, and 15 isn't prime. But *sumOfTwoPrimes*(34) = True, because $34 = 17 + 17$, and 17 is prime. (And the other 17 is prime, too.)
2. *isPowerOf* (16, 2) = True because $2^4 = 16$ (and the exponent 4 is an integer), but *isPowerOf* (2, 16) = False because $16^{1/4} = 2$ (and $1/4$ is not an integer).
3. *onlyPowersOfTwo*($\{1, 2, 8, 128\}$) = True because every element of $\{1, 2, 8, 128\}$ is a power of two: $\{1, 2, 8, 128\} = \{2^0, 2^1, 2^3, 2^7\}$.

These brief examples may already be enough to begin to give you a sense of the power of logical abstraction that predicates grant us: we can now consider the same logical "condition" applied to two different "arguments." In a sense, propositional logic is like programming without functions; letting ourselves use predicates allows us to write two related propositions using related notation, and to reason simultaneously about multiple propositions—just like writing a function in Java allows you to think simultaneously about the same function applied to different arguments.

> **Taking it further:** Predicates give a convenient way of representing the state of play of multiplayer games like Tic-Tac-Toe, checkers, and chess. The basic idea is to define a predicate $P(B)$ that expresses "Player 1 will win from board position B if both players play optimally." For more on this idea, and on the application of logic (both predicate and propositional) to playing these kinds of games, see the discussion on p. 344.

3.4.2 Quantifiers

We've seen that we can form a proposition from a predicate by applying that predicate to a particular argument. But we can also form a proposition from a predicate using *quantifiers*, which allow us to formalize statements like *every Java program contains at least four* **for** *loops* (false!) or *there is a proposition that cannot be expressed using only the connectives* $\wedge$ *and* $\vee$ (true! See Exercise 4.71).

These types of statements are expressed by the two standard quantifiers, the *universal* ("every") and *existential* ("some") quantifiers (see Figure 3.20):

$\forall x \in S : P(x)$	"for all" (universal quantifier)	true if $P(x)$ is true for *every* $x \in S$.
$\exists x \in S : P(x)$	"there exists" (existential quantifier)	true if $P(x)$ is true for *at least one* $x \in S$.

Figure 3.20: Summary of notation for predicate logic.

Definition 3.19 (Universal quantifier ("for all"): $\forall$)
Let P be a predicate over the universe S. The proposition $\forall x \in S : P(x)$ ("for all x in S, $P(x)$") is true if, for every possible $x \in S$, $P(x)$ is true.

Definition 3.20 (Existential quantifier ("there exists"): $\exists$)
Let P be a predicate over the universe S. The proposition $\exists x \in S : P(x)$ ("there exists an x in S such that $P(x)$") is true if, for at least one possible $x \in S$, we have that $P(x)$ is true.

The *for all* notation is $\forall$, an upside-down 'A' as in "all"; the *exists* notation is $\exists$, a backward 'E' as in "exists." (Annoyingly, they had to be flipped in different directions: a backward 'A' is still an 'A,' and an upside-down 'E' is still an 'E.')

Here's an example of two simple numerical propositions using these quantifiers:

Example 3.33 (Simple propositions using quantifiers)
Problem: What are the truth values of the following two propositions?

1. $\forall n \in \mathbb{Z}^{\geq 2} : \mathit{isPrime}(n)$
2. $\exists n \in \mathbb{Z}^{\geq 2} : \mathit{isPrime}(n)$

Solution: 1. **False.** This proposition says "every integer $n \geq 2$ is prime." This statement is false because, for example, the integer 32 is greater than or equal to 2 and is not prime.

2. **True.** The proposition says "there exists an integer $n \geq 2$ that is prime." This statement is true because, for example, the integer 31 (which is greater than or equal to 2) *is* prime.

In addition, we can make precise many intuitive statements using quantifiers. For example, we can use quantifiers to formalize the predicates from Example 3.30. (See Figure 3.21 for a reminder.)

isPrime(n): $n \in \mathbb{Z}^{>0}$ is a prime number. *isPowerOf*(n,k): $n \in \mathbb{Z}$ is an exact power of k.	*onlyPowersOfTwo*(S): every element of S is a power of two. $Q(n,a,b)$: $n \in \mathbb{Z}^{>0}$ satisfies $n = a + b$, and $a, b \in \mathbb{Z}$ are both prime.	*sumOfTwoPrimes*(n): $n \in \mathbb{Z}^{>0}$ is equal to the sum of two prime numbers.

Figure 3.21: Reminder of the predicates from Example 3.30.

Example 3.34 (Some example predicates, formalized)
isPrime(n): An integer $n \in \mathbb{Z}^{>0}$ is prime if and only if $n \geq 2$ and the only integers that evenly divide n are 1 and n itself. Thus we are really expressing a condition on every candidate divisor d: either $d \in \{1, n\}$, or d doesn't evenly divide n. Using the "divides" notation from Definition 2.10, we can formalize *isPrime*(n) as

$$n \geq 2 \wedge \left[\forall d \in \mathbb{Z}^{\geq 1} : \left(d \mid n \;\Rightarrow\; d = 1 \vee d = n \right) \right].$$

isPowerOf(n,k): We can formalize this predicate as $\exists i \in \mathbb{Z}^{\geq 0} : n = k^i$.

onlyPowersOfTwo(S): Because *isPowerOf*$(n, 2)$ expresses the condition that n is a power of two, we can formalize this predicate as $\forall x \in S : \mathit{isPowerOf}(x, 2)$.

$Q(n, a, b)$: Formalizing Q actually doesn't require a quantifier at all; we can simply write $Q(n, a, b)$ as $(n = a + b) \wedge \mathit{isPrime}(a) \wedge \mathit{isPrime}(b)$.

sumOfTwoPrimes(n): This predicate requires that *there exist* prime numbers a and b that sum to n. Given our definition of Q, we can write *sumOfTwoPrimes*(n) as

$$\exists \langle a, b \rangle \in \mathbb{Z} \times \mathbb{Z} : Q(n, a, b).$$

("There exists a pair of integers $\langle a, b \rangle$ such that $Q(n, a, b)$.") Or we could write *sumOfTwoPrimes*(n) as $\exists a \in \mathbb{Z} : [\exists b \in \mathbb{Z} : Q(n, a, b)]$, by *nesting* one quantifier within the other. (See Section 3.5.)

Here's one further example, regarding the *prefix* relationship between two strings:

Example 3.35 (Prefixes, formalized)
A binary string $x \in \{0,1\}^k$ is a *prefix* of the binary string $y \in \{0,1\}^n$, for $n \geq k$, if y is x with some extra bits added on at the end. For example, `01` and `0110` are both prefixes of `01101010`, but `1` is not a prefix of `01101010`. If we write $|x|$ and $|y|$ to denote the length of x and y, respectively, then we can formalize *isPrefixOf*(x, y) as

$$|x| \leq |y| \quad \wedge \quad \Big[\forall i \in \{i \in \mathbb{Z} : 1 \leq i \leq |x|\} \;:\; x_i = y_i\Big].$$

In other words, y must be no shorter than x, and the first $|x|$ characters of y must equal their corresponding characters in x.

Quantifiers as loops

One useful way of thinking about these quantifiers is by analogy to loops in programming. If we ever encounter an $x \in S$ for which $\neg P(x) = \text{True}$, then we immediately know that $\forall x \in S : P(x)$ is false. Similarly, any $x \in S$ for which $Q(x) = \text{True}$ is enough to demonstrate that $\exists x \in S : Q(x)$ is true. But if we "loop through" all candidate values of x and fail to encounter an x with $\neg P(x)$ or $Q(x)$, we know that $\forall x \in S : P(x)$ is true or $\exists x \in S : Q(x)$ is false. By this analogy, we might think of the two standard quantifiers as executing the programs in Figure 3.22(a) for $\forall$, and Figure 3.22(b) for $\exists$.

```
for $x$ in $S$:
    if not $P(x)$ then
        return False
return True
```

(a) A loop corresponding to $\forall x \in S : P(x)$.

```
for $x$ in $S$:
    if $Q(x)$ then
        return True
return False
```

(b) A loop corresponding to $\exists x \in S : Q(x)$.

Figure 3.22: Two **for** loops that return the value of $\forall x \in S : P(x)$ and $\exists x \in S : Q(x)$.

Another intuitive and useful way to think about these quantifiers is as a supersized version of $\wedge$ and $\vee$:

$$\forall x \in \{x_1, x_2, \ldots, x_n\} : P(x) \quad \equiv \quad P(x_1) \wedge P(x_2) \wedge \cdots \wedge P(x_n)$$
$$\exists x \in \{x_1, x_2, \ldots, x_n\} : P(x) \quad \equiv \quad P(x_1) \vee P(x_2) \vee \cdots \vee P(x_n)$$

The first of these propositions is true only if *every one* of the $P(x_i)$ terms is true; the second is true if *at least one* of the $P(x_i)$ terms is true.

There is one way in which these analogies are loose, though: just as for $\sum$ (summation) and $\prod$ (product) notation (from Section 2.2.7), the loop analogy only makes sense when the domain of discourse is finite! The Figure 3.22(a) "program" for a true proposition $\forall x \in \mathbb{Z} : P(x)$ would have to complete an infinite number of iterations before returning True. But the intuition may still be helpful.

Precedence and parenthesization

As in propositional logic, we'll adopt standard conventions regarding order of operations so that we don't overdose on parentheses. We treat the quantifiers $\forall$ and $\exists$ as binding tighter than the propositional logical connectives. Thus

$$\forall x \in S : P(x) \;\; \Rightarrow \;\; \exists y \in S : P(y)$$

will be understood to mean

$$\Big[\forall x \in S : P(x)\Big] \;\; \Rightarrow \;\; \Big[\exists y \in S : P(y)\Big].$$

To express the other reading (which involves nested quantifiers; see Section 3.5), we can use parentheses explicitly, by writing $\forall x \in S : \big[P(x) \Rightarrow \exists y \in S : P(y)\big]$.

Free and bound variables

Consider the variables x and y in the expressions

$$3 \mid x \qquad\qquad \text{and} \qquad\qquad \forall y \in \mathbb{Z} : 3 \mid y.$$

Understanding the first of these expressions requires knowledge of what x means, whereas the second is a self-contained statement that can be understood without any outside knowledge. The variable x is called a *free* or *unbound variable*: its value is not fixed by the expression. In contrast, the variable y is a *bound variable*: its value is defined within the expression itself. We say that the quantifier *binds* the variable y, and the *scope* or *body* of the quantifier is the part of the expression in which it has bound y. (We've encountered bound variables before; they arise whenever a variable name is assigned a value within an expression. For example, the variable i is bound in the arithmetic expression $\sum_{i=1}^{10} i^2$, as is the variable n in $\{n \in \mathbb{Z} : |n| \leq |n^2|\}$.)

A single expression can contain both free and bound variables: for example, the expression $\exists y \in \mathbb{Z}^{\geq 0} : x \geq y$ contains a bound variable y and a free variable x. Here's another example:

Example 3.36 (Free and bound variables)

Problem: Which variables are free in the following expression?

$$\Big[\forall x \in \mathbb{Z} : x^2 \geq y\Big] \wedge \Big[\forall z \in \mathbb{Z} : y = z \vee z^y = 1\Big]$$

Solution: The variable y doesn't appear as the variable bound by either of the quantifiers in this expression, so y is a free variable. Both x and z are bound by the universal quantifiers. (Incidentally, this expression is true if and only if $y = 0$.)

To test whether a particular variable x is free or bound in an expression, we can (consistently) replace x by a different name in that expression. If the meaning stays the same, then x is bound; if the meaning changes, then x is free. For example:

Example 3.37 (Testing for free and bound variables)
Consider the following pairs of propositions:

$$\exists x \in S : x > 251 \quad \text{and} \quad \exists y \in S : y > 251 \qquad \text{(A)}$$

$$x \geq 42x \quad \text{and} \quad y \geq 42y \qquad \text{(B)}$$

The expressions in (A) express precisely the same condition, namely: *some element of S is greater than* 251. Thus, the variables x and y in these two expressions are *bound*.

But the expressions in (B) mean different things, in the sense that we can construct a context in which these two statements have different truth values (for example, $x = 3$ and $y = -2$). The first expression states a condition on the value of x, and the latter states a condition on the value of y. So x is a free variable in "$x \geq 42x$."

Taking it further: The free-versus-bound-variable distinction is also something that may be familiar from programming, at least in some programming languages. There are some interesting issues in the design and implementation of programming languages that center on how free variables in a function definition, for example, get their values. See the discussion on p. 345.

An expression of predicate logic that contains no free variables is called *fully quantified.* For expressions that are not fully quantified, we adopt a standard convention that any unbound variables in a stated claim are *implicitly* universally quantified. For example, consider these claims:

Claim A: If $x \geq 1$, then $x^2 \leq x^3$.
Claim B: For all $x \in \mathbb{R}$, if $x \geq 1$, then $x^2 \leq x^3$.

When we write a (true) claim like Claim A, we will implicitly interpret it to mean Claim B. (Note that Claim B also explicitly notes $\mathbb{R}$ as the domain of discourse, which was left implicit in Claim A.)

3.4.3 Theorem and Proof in Predicate Logic

Recall that a *tautology* is a proposition that is always true—in other words, it is true no matter what each Boolean variable p in the proposition "means" (that is, whether p is true or false). In this section, we will be interested in the corresponding notion of always-true statements of predicate logic, which are called *theorems.* A statement of predicate logic is "always true" when it's true no matter what its predicates mean. (Formally, the "meaning" of a predicate P is the set of elements of the universe U for which the predicate is true—that is, $\{x \in U : P(x)\}$.)

Definition 3.21 (Theorems in predicate logic)
A fully quantified expression of predicate logic is a theorem *if and only if it is true for every possible meaning of each of its predicates.*

Analogously, two fully quantified expressions are *logically equivalent* if, for every possible meaning of their predicates, the two expressions have the same truth values.

We'll begin with a simple example of a theorem and a nontheorem:

Example 3.38 (A theorem of predicate logic)
Let S be any set. The following claim is true *regardless of what the predicate P denotes:*

$$\forall x \in S : \Big[P(x) \vee \neg P(x) \Big].$$

Indeed, this claim simply says that every $x \in S$ either makes $P(x)$ true or $P(x)$ false. And that assertion is true if the predicate $P(x)$ is "$x \geq 42$" or "x has red hair" or "x prefers programming in Python to playing Parcheesi"—indeed, it's true for any predicate P.

Example 3.39 (A nontheorem)
Let's show that the following proposition is not a theorem:

$$\Big[\forall x \in S : P(x) \Big] \vee \Big[\forall x \in S : \neg P(x) \Big].$$

A theorem must be true regardless of P's meaning, so we can establish that this proposition isn't a theorem by giving an example predicate that makes it false. Here's one: let P be *isPrime* (where S is $\mathbb{Z}$). Observe that $\forall x \in \mathbb{Z} : \mathit{isPrime}(x)$ is false because $\mathit{isPrime}(4) = \text{False}$; and $\forall x \in \mathbb{Z} : \neg \mathit{isPrime}(x)$ is false because $\neg \mathit{isPrime}(5) = \text{False}$. Thus the given proposition is false when P is *isPrime*, and so it is not a theorem.

Note the crucial difference between Example 3.38, which states that *every element of S either makes P true or makes P false,* and Example 3.39, which states that *either every element of S makes P true, or every element of S makes P false.* (Intuitively, it's the difference between "Every letter is either a vowel or a consonant" and "Every letter is a vowel or every letter is a consonant." The former is true; the latter is false.)

Example 3.39 establishes that the proposition $[\forall x \in S : P(x)] \vee [\forall x \in S : \neg P(x)]$ isn't true for *every* meaning of the predicate P, but it may be true for *some* meanings. For example, if $P(x)$ is the predicate $x^2 \geq 0$ and S is the set $\mathbb{R}$, then this disjunction is true (because $\forall x \in \mathbb{R} : x^2 \geq 0$ is true).

The challenge of proofs in predicate logic

The remainder of this section states some theorems of predicate logic, along with an initial discussion of how we might prove that they're theorems. (A *proof* of a statement is simply a convincing argument that the statement is a theorem.) Much of the rest of the book will be devoted to developing and writing proofs of theorems like these, and Chapter 4 will be devoted exclusively to some techniques and strategies for proofs. (This section will preview some of the ideas we'll see there.) Some theorems of predicate logic are summarized in Figure 3.23; we'll prove a few of them here, and you'll return to some of the others in the exercises.

$\forall x \in S : \Big[P(x) \vee \neg P(x) \Big]$	
$\neg \Big[\forall x \in S : P(x) \Big] \Leftrightarrow \Big[\exists x \in S : \neg P(x) \Big]$	De Morgan's Laws (quantified form)
$\neg \Big[\exists x \in S : P(x) \Big] \Leftrightarrow \Big[\forall x \in S : \neg P(x) \Big]$	
$\Big[\forall x \in S : P(x) \Big] \Rightarrow \Big[\exists x \in S : P(x) \Big]$	*if the set S is nonempty*
$\forall x \in \varnothing : P(x)$	Vacuous quantification
$\neg \exists x \in \varnothing : P(x)$	
$\exists x \in S : \Big[P(x) \vee Q(x) \Big] \quad \Leftrightarrow \quad \Big[\exists x \in S : P(x) \Big] \vee \Big[\exists x \in S : Q(x) \Big]$	
$\forall x \in S : \Big[P(x) \wedge Q(x) \Big] \quad \Leftrightarrow \quad \Big[\forall x \in S : P(x) \Big] \wedge \Big[\forall x \in S : Q(x) \Big]$	
$\exists x \in S : \Big[P(x) \wedge Q(x) \Big] \quad \Rightarrow \quad \Big[\exists x \in S : P(x) \Big] \wedge \Big[\exists x \in S : Q(x) \Big]$	
$\forall x \in S : \Big[P(x) \vee Q(x) \Big] \quad \Leftarrow \quad \Big[\forall x \in S : P(x) \Big] \vee \Big[\forall x \in S : Q(x) \Big]$	
$\Big[\forall x \in S : P(x) \Rightarrow Q(x) \Big] \wedge \Big[\forall x \in S : P(x) \Big] \Rightarrow \Big[\forall x \in S : Q(x) \Big]$	
$\Big[\forall x \in \{y \in S : P(y)\} : Q(x) \Big] \quad \Leftrightarrow \quad \Big[\forall x \in S : P(x) \Rightarrow Q(x) \Big]$	
$\Big[\exists x \in \{y \in S : P(y)\} : Q(x) \Big] \quad \Leftrightarrow \quad \Big[\exists x \in S : P(x) \wedge Q(x) \Big]$	
$\varphi \wedge \Big[\exists x \in S : P(x) \Big] \Leftrightarrow \Big[\exists x \in S : \varphi \wedge P(x) \Big]$	*if x does not appear as a free variable in φ*
$\varphi \vee \Big[\forall x \in S : P(x) \Big] \Leftrightarrow \Big[\forall x \in S : \varphi \vee P(x) \Big]$	*if x does not appear as a free variable in φ*

Figure 3.23: A few theorems involving quantification.

While predicate logic allows us to express claims that we couldn't state without quantifiers, that extra expressiveness comes with a cost! For a quantifier-free proposition (like all propositions in Sections 3.2–3.3), there is a straightforward—if tedious—algorithm to decide whether a given proposition is a tautology: first, build a truth table for the proposition; and, second, check to make sure that the proposition is true in every row. It turns out that the analogous question for predicate logic is much more difficult—in fact, *impossible* to solve in general: there's no algorithm that's guaranteed to figure out whether a given fully quantified expression is a theorem! Demonstrating that a statement in predicate logic is a theorem will require you to *think* in a way that demonstrating that a statement in propositional logic is a tautology did not.

> **Taking it further:** See the discussion on p. 346 for more about the fact that there's no algorithm guaranteed to determine whether a given proposition is a theorem. The absence of such an algorithm sounds like bad news; it means that proving predicate-logic statements is harder, because you can't just plug-and-chug into a simple algorithm to figure out whether a given statement is actually always true. But this fact is also precisely the reason that creativity plays a crucial role in proofs and in theoretical computer science more generally—and why, arguably, proving things can be fun! (For me, this difference is exactly why I find Sudoku less interesting than crossword puzzles: when there's no algorithm to solve a problem, we have to embrace the creative challenge in attacking it.)

3.4.4 *A Few Examples of Theorems and Proofs*

In the rest of this section, we will see a few further theorems of predicate logic, with proofs. As we've said, there's no formulaic approach to prove these theorems; we'll need to employ a variety of strategies in this endeavor.

Negating quantifiers: a first example

Suppose that your egomaniacal, overconfident partner from Intro CS wanders into the lab and says *For any array A that you give me, partner, my implementation of insertion sort correctly sorts A.* You know, though, that your partner is wrong. (You spot a bug in his egomaniacal code.) What would that mean? Well, you might reply, gently but firmly: *There's an array A for which your implementation of insertion sort does not correctly sort A.* The equivalence that you're using is a theorem of predicate logic:

Example 3.40 (Negating universal quantifiers)
Let's prove the equivalence you're using to debunk your partner's claim:

$$\neg\big[\forall x \in S : P(x)\big] \Leftrightarrow \big[\exists x \in S : \neg P(x)\big].$$

Perhaps the easiest way to view this claim is as a quantified version of the tautology $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$, which was one of De Morgan's Laws from propositional logic. If we think of $\forall x \in S : P(x)$ as $P(x_1) \wedge P(x_2) \wedge P(x_3) \wedge \cdots$, then

$$\begin{aligned}\neg\big[\forall x \in S : P(x)\big] &\approx \neg\big[P(x_1) \wedge P(x_2) \wedge P(x_3) \wedge \cdots\big] \\ &\equiv \big[\neg P(x_1) \vee \neg P(x_2) \vee \neg P(x_3) \vee \cdots\big] \\ &\approx \exists x \in S : \neg P(x),\end{aligned}$$

where the second line follows by the propositional version of De Morgan's Laws. There is something slightly more subtle to our claim because the set S might be infinite, but the idea is identical. If there's an $a \in S$ such that $P(a) =$ False, then $\exists x \in S : \neg P(x)$ is true (because a is an example) and $\forall x \in S : P(x)$ is false (because a is a counterexample). And if every $a \in S$ has $P(a) =$ True, then $\exists x \in S : \neg P(x)$ is false and $\forall x \in S : P(x)$ is true.

The analogous claim for the negation of $\exists x \in S : P(x)$ is also a theorem:

Example 3.41 (Negating existential quantifiers)
Let's prove that this claim is a theorem, too:

$$\neg\big[\exists x \in S : P(x)\big] \Leftrightarrow \big[\forall x \in S : \neg P(x)\big].$$

To see that this claim is true for an arbitrary predicate P, we start with the claim from Example 3.40, but using the predicate $Q(x) := \neg P(x)$. (Note that Q is also a predicate—so Example 3.40 holds for Q too!) Thus we know that

$$\neg\big[\forall x \in S : Q(x)\big] \Leftrightarrow \big[\exists x \in S : \neg Q(x)\big],$$

and, because $p \Leftrightarrow q \equiv \neg p \Leftrightarrow \neg q$, we therefore also know that

$$\big[\forall x \in S : Q(x)\big] \Leftrightarrow \neg\big[\exists x \in S : \neg Q(x)\big].$$

But $Q(x)$ is just $\neg P(x)$ and $\neg Q(x)$ is just $P(x)$, by definition of Q, and so we have

$$\big[\forall x \in S : \neg P(x)\big] \Leftrightarrow \neg\big[\exists x \in S : P(x)\big].$$

Thus we've now shown that the desired claim is true for any predicate P, so it is a theorem.

All implies some: a proof of an implication

The entirety of Chapter 4 is devoted to proofs and proof techniques; there's lots more there about how to approach proving or disproving new claims. But here we'll preview a particularly useful proof strategy for proving an implication, and use it to establish another theorem of predicate logic. Here's the method of proof:

Definition 3.22 (Proof by assuming the antecedent)
Suppose that we must prove an implication $\varphi \Rightarrow \psi$. Because the only way for $\varphi \Rightarrow \psi$ to fail *to be true is for φ to be true and ψ to be false, to prove that the implication $\varphi \Rightarrow \psi$ is always true, we will rule out the one scenario in which it wouldn't be. Specifically, we* assume that *φ is true, and then* prove that *ψ must be true too, under this assumption.*

(Recall from the truth table of ⇒ that the only way for the implication $\varphi \Rightarrow \psi$ to be false is when φ is true but ψ is false. Also recall that the proposition φ is called the *antecedent* of the implication $\varphi \Rightarrow \psi$; hence this proof technique is called *assuming the antecedent*.) Here are two examples of proofs that use this technique, one from propositional logic and one from arithmetic:

- Let's prove that $p \Rightarrow p \lor q$ is a tautology: we assume that the antecedent p is true, and we must prove that the consequent $p \lor q$ is true too. But that's obvious, because p is true (by our assumption), and True $\lor\, q \equiv$ True.
- Let's prove that *if x is a perfect square, then $4x$ is a perfect square*: assume that x is a perfect square, that is, assume that $x = k^2$ for an integer k. Then $4x = 4k^2 = (2k)^2$ is a perfect square too, because $2k$ is also an integer.

Finally, here's a theorem of predicate logic that we can prove using this technique:

Problem-solving tip: When you're facing a statement that contains a lot of mathematical notation, try to understand it by rephrasing it as an English sentence. Restating the assertion from Example 3.42 in English makes it pretty obvious that it's true: *if everyone in S satisfies P—and there's actually someone in S—then of course someone in S satisfies P!*

Example 3.42 (If everybody's doing it, then somebody's doing it)
Consider the following proposition, for an arbitrary nonempty set S:

$$\Big[\forall x \in S : P(x)\Big] \;\Rightarrow\; \Big[\exists x \in S : P(x)\Big].$$

We'll prove this claim by assuming the antecedent. Specifically, we assume $\forall x \in S : P(x)$, and we need to prove that $\exists x \in S : P(x)$.

Because the set S is nonempty, we know that there's at least one element $a \in S$. By our assumption, we know that $P(a)$ is true. But because $P(a)$ is true, then it's immediately apparent that $\exists x \in S : P(x)$ is true too—because we can just pick $x := a$.

Vacuous quantification

Consider the proposition *All even prime numbers greater than* 12 *have a* 3 *as their last digit*. Write P to denote the set of all even prime numbers greater than 12; formalized, then, this claim can be written as $\forall n \in P : n \bmod 10 = 3$. Is this claim true or false? It has to be true! The point is that P actually contains no elements (there *are* no even prime numbers other than 2, because an even number is by definition divisible by 2). Thus this claim says: for every $n \in \varnothing$, something-or-other is true of n. But there is no n in $\varnothing$, so the claim has to be true! The general statement of the theorem is

$$\forall x \in \varnothing : P(x).$$

Quantification over the empty set is called *vacuous quantification*; this proposition is said to be *vacuously true*.

Here's another way to see that $\forall x \in \varnothing : P(x)$ is a theorem, using the De Morgan–like view of quantification. The negation of $\forall x \in \varnothing : P(x)$ is $\exists x \in \varnothing : \neg P(x)$, but there never exists *any* element $x \in \varnothing$, let alone an element $x \in \varnothing$ such that $\neg P(x)$. Thus $\exists x \in \varnothing : \neg P(x)$ is false, and therefore its negation $\neg \exists x \in \varnothing : \neg P(x)$, which is equivalent to $\forall x \in \varnothing : P(x)$, is true.

Disjunctions and quantifiers

Here's one last example, where we'll figure out when the "or" of two quantified statements can be expressed as one single quantified statement:

Example 3.43 (Disjunctions and quantifiers)
Consider the following two propositions, for an arbitrary set S:

$$\forall x \in S : \Big[P(x) \vee Q(x)\Big] \quad \Leftrightarrow \quad \Big[\forall x \in S : P(x)\Big] \vee \Big[\forall x \in S : Q(x)\Big] \tag{A}$$

$$\exists x \in S : \Big[P(x) \vee Q(x)\Big] \quad \Leftrightarrow \quad \Big[\exists x \in S : P(x)\Big] \vee \Big[\exists x \in S : Q(x)\Big] \tag{B}$$

Problem: Is either (A) or (B) a theorem? Prove your answers.

Solution: Claim (B) is a theorem. To prove it, we'll show that the left-hand side implies the right-hand side, and vice versa. (That is, we're proving $p \Leftrightarrow q$ by proving both $p \Rightarrow q$ and $q \Rightarrow p$, which is a legitimate proof because $p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$.) Both proofs will use the technique of assuming the antecedent.

- First, suppose that $\exists x \in S : [P(x) \vee Q(x)]$ is true. Then there is some particular $x^* \in S$ for which either $P(x^*)$ or $Q(x^*)$. But in either case, we're done: if $P(x^*)$ then $\exists x \in S : P(x)$—in particular, x^* satisfies the condition; if $Q(x^*)$ then $\exists x \in S : Q(x)$.
- Conversely, suppose that $[\exists x \in S : P(x)] \vee [\exists x \in S : Q(x)]$ is true. Thus either there's an $x^* \in S$ such that $P(x^*)$ or an $x^* \in S$ such that $Q(x^*)$. That x^* suffices to make the left-hand side true.

Problem-solving tip: In thinking about a question like whether (A) from Example 3.43 is a theorem, it's often useful to get intuition by plugging in a few sample values for S, P, and Q.

On the other hand, (A) is not a theorem, for much the same reason as in Example 3.39. (In fact, if $Q(x) := \neg P(x)$, then Examples 3.38 and 3.39 precisely show that (A) is not a theorem.) The set $\mathbb{Z}$ and the predicates *isOdd* and *isEven* make (A) false: the left-hand side is true ("all integers are either even or odd") but the right-hand side is false ("either (i) all integers are even, or (ii) all integers are odd").

Although (A) from this example is not a theorem, one direction of it is; we'll prove this implication as another example:

Example 3.44 (Disjunction, quantifiers, and one-way implications)
The $\Leftarrow$ direction of (A) from Example 3.43 is a theorem:

$$\forall x \in S : \Big[P(x) \vee Q(x)\Big] \quad \Leftarrow \quad \Big[\forall x \in S : P(x)\Big] \vee \Big[\forall x \in S : Q(x)\Big].$$

To convince yourself of this claim, observe that if $P(x)$ is true for an arbitrary $x \in S$, then it's certainly true that $P(x) \vee Q(x)$ is true for an arbitrary $x \in S$ too. And if $Q(x)$ is true for every $x \in S$, then, similarly, $P(x) \vee Q(x)$ is true for every $x \in S$.

To prove this claim, we assume the antecedent $[\forall x \in S : P(x)] \vee [\forall x \in S : Q(x)]$. Thus either $[\forall x \in S : P(x)]$ or $[\forall x \in S : Q(x)]$, and, in either case, we've argued that $P(x) \vee Q(x)$ is true for all $x \in S$.

You'll have a chance to consider a number of other theorems of predicate logic in the exercises, including the $\wedge$-analogy to Examples 3.43–3.44 (in Exercises 3.130–3.131).

Computer Science Connections

Game Trees, Logic, and Winning Tic-Tac(-Toe)

In 1997, Deep Blue, a chess-playing program developed by IBM,[5] beat the chess Grandmaster Garry Kasparov in a six-match series. This event was a turning point in the public perception of computation and artificial intelligence; it was the first time that a computer had outperformed the best humans at something that most people tended to identify as a "human endeavor." Ten years later, a research group developed a program called Chinook, a perfect checkers-playing system: from any game position arising in its games, Chinook chooses *the* best possible legal move.[6]

While chess and checkers are very complicated games, the basic ideas of playing them—ideas based on logic—are shared with simpler games. Consider *Tic-Tac*, a 2-by-2 version of Tic-Tac-Toe. Two players, O and X, make alternate moves, starting with O; a player wins by occupying a complete row or column. Diagonals don't count, and if the board is filled without O or X winning, then the game is a draw. Note that—unless O is tremendously dull—O will win the game, but we will use a *game tree* (Figure 3.24), which represents all possible moves, to systematize this reasoning.

Here's the basic idea. Define $P(B)$ to be the predicate

$$P(B) := \text{"Player O wins under optimal play starting from board } B\text{."}$$

For example, $P(\begin{array}{c|c} \text{X} & \\ \hline \text{O} & \text{O} \end{array}) = \text{True}$ because O has already won; and $P(\begin{array}{c|c} \text{O} & \text{X} \\ \hline \text{X} & \text{O} \end{array}) = \text{False}$ because it's a draw. The answer to the question "does O win Tic-Tac if both players play optimally?" is the truth value of $P(\begin{array}{c|c} & \\ \hline & \end{array})$. If it's O's turn in board B, then $P(B)$ is true if and only if *there exists* a possible move for O leading to a board B' in which $P(B')$; if it's X's turn, then $P(B)$ is true if and only if *every* possible move made by X leads to a board B' in which $P(B')$. So

$$\begin{aligned} P(\begin{array}{c|c} & \text{O} \\ \hline & \end{array}) &= P(\begin{array}{c|c} \text{X} & \text{O} \\ \hline & \end{array}) \wedge P(\begin{array}{c|c} & \text{O} \\ \hline \text{X} & \end{array}) \wedge P(\begin{array}{c|c} & \text{O} \\ \hline & \text{X} \end{array}) \\ \text{and } P(\begin{array}{c|c} & \\ \hline & \end{array}) &= P(\begin{array}{c|c} \text{O} & \\ \hline & \end{array}) \vee P(\begin{array}{c|c} & \text{O} \\ \hline & \end{array}) \vee P(\begin{array}{c|c} & \\ \hline \text{O} & \end{array}) \vee P(\begin{array}{c|c} & \\ \hline & \text{O} \end{array}). \end{aligned}$$

The game tree, labeled appropriately, is shown in Figure 3.25. If we view the truth values from the leaves as "bubbling up" from the bottom of the tree, then a board B gets assigned the truth value True if and only if Player O can guarantee a win from the board B.

Some serious complications arise in writing a program to play more complicated games like checkers or chess. Here are just a few of the issues that one must confront in building a system like Deep Blue or Chinook:[7]

- There are $\approx$ 500,000,000,000,000,000,000 different checkers positions—and $\approx 10^{40}$ chess positions!—so we can't afford to represent them all. (Luckily, we can choose moves so most positions are never reached.)
- Approximately one bit per trillion is written incorrectly *merely in copying data on current hard disk technologies.* So a program constructing a massive structure like the checkers game tree must "check its work."
- For a game as big as chess, we can't afford to compute all the way to the bottom of the tree; instead, we *estimate* the quality of each position after computing a handful of layers deep in the game tree.

[5] Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134:57–83, 2002.

[6] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Rob Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 14 September 2007.

Thanks to Jon Kleinberg for suggesting this game.

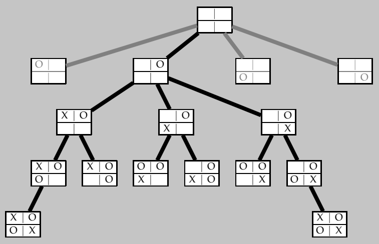


Figure 3.24: 25% of the Tic-Tac game tree. (The missing 75% is rotated, but otherwise identical.)

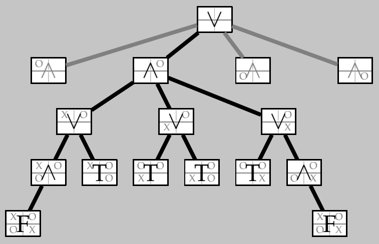


Figure 3.25: The game tree, with each win for O labeled by T, each loss/draw by F, ∨ if it's Player O's turn, and ∧ if it's Player X's turn.

For more on game trees and algorithms for exploring large search spaces, see a good artificial intelligence (AI) text like

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3rd edition, 2009.

Computer Science Connections

Nonlocal Variables and Lexical vs. Dynamic Scoping

In a function f written in a programming language—say, C or Python—we can use several different types of variables that store values:

- *local variables*, whose values are defined completely within the body of f;
- *parameters*, inputs to f whose value is specified when f is invoked;
- *nonlocal variables*, which get their value from other contexts. The most common type of these "other" variables is a *global variable*, which persists throughout the execution of the entire program.

```
int addB(int a) {
  return a + b;
}
```

```
def addB(a):
  return a + b
```

Figure 3.26: A function addB written in C and analogous function addB written in Python. Here addB takes one (integer) parameter a, accesses a nonlocal variable b, and returns a + b.

For an example function (written in C and Python as illustrative examples) that uses both a parameter and a nonlocal variable, see Figure 3.26. In the body of this function, the variable a is a *bound* variable; specifically, it is bound when the function is invoked with an actual parameter. But the variable b is *unbound*. (Just as with a quantified expression, an unbound variable is one for which the meaning of the function could change if we replaced that variable with a different name. If we changed the a to an x in both lines 1 and 2, then the function would behave identically, but if we changed the b to a y, then the function would behave differently.)

In this function, the variable b has to somehow get a value from somewhere if we are going to be able to invoke the function addB without causing an error. Often b will be a global variable, but it is also possible in Python or C (with appropriate compiler settings) to *nest* function definitions—just as quantifiers can be nested. (See Section 3.5.)

For more about lexical versus dynamic scope, and other related issues, see a textbook on programming languages. (One of the other interesting issues is that there are actually multiple paradigms for passing parameters to a function; we're discussing *call-by-value* parameter passing, which probably is the most common.) Some good books on programming languages include

One fundamental issue in the design and implementation in programming languages is illustrated in Figure 3.27.[8] Suppose x is an unbound variable in the definition of a function f. Generally, programming languages either use *lexical scope*, where x's value is found by looking "outward" where f is *defined*; or *dynamic scope*, where x's value is found by looking where f is *called*. Almost all modern programming languages use lexical scope, though *macros* in C and other languages use dynamic scope. While we're generally used to lexical scope and therefore it feels more intuitive, there are some circumstances in which macros can be tremendously useful and convenient.

```
int b = 17;

int addB(int a) { return a + b; }
    /* a FUNCTION in C finds values for unbound */
    /* variables in the *defining* environment */

int test() {
  int b = 128;
  return addB(3);
}

test(3);        /* returns 20 */
```

```
int b = 17;

#define addB(a)     a + b
    /* a MACRO in C finds values for unbound */
    /* variables in the *calling* environment */

int test() {
  int b = 128;
  return addB(3);
}

test(3);        /* returns 131 */
```

Figure 3.27: Two C snippets defining addB, where the nonlocal variable b gets its value from different places.

[8] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 3rd edition, 2009; and Kenneth C. Louden and Kenneth A. Lambert. *Programming Languages: Principles and Practices*. Course Technology, 3rd edition, 2011.

Computer Science Connections

Gödel's Incompleteness Theorem

Given a fully quantified proposition φ, is φ a theorem? This apparently simple question drove the development of some of the most profound and mind-numbing results of the last hundred years. In the early 20th century, there was great interest in the "formalist program," advanced especially by the German mathematician David Hilbert. The formalist approach aimed to turn all of mathematical reasoning into a machine: one could feed in a mathematical statement φ as input, turn a hypothetical crank, and the machine would spit out a proof or disproof of φ as output. But this program was shattered by two closely related results—two of the greatest intellectual achievements of the 20th century.

The first blow to the formalist program was the proof by Kurt Gödel, in 1931, of what became known as *Gödel's Incompleteness Theorem*. Gödel's incompleteness theorem is based on the following two important and desirable properties of logical systems:

- A logical system is *consistent* if only true statements can be proven. (In other words, if there is a proof of φ in the system, then φ is true.)
- A logical system is *complete* if every true statement can be proven. (In other words, if φ is true, then there is a proof of φ in the system.)

Gödel's Incompleteness Theorem is the following troubling result:

Theorem 3.3 (Gödel's (First) Incompleteness Theorem)
Any sufficiently powerful logical system is either inconsistent or incomplete.

(Here "sufficiently powerful" just means "capable of expressing multiplication"; predicate logic as described here is certainly "sufficiently powerful.")

If the system is inconsistent, then there is a false statement φ that can be proven (which means that anything can be proven, as false implies anything!). And if the system is incomplete, then there is a true statement φ that cannot be proven. Gödel's proof proceeds by constructing a self-referential logical expression φ that means "φ is not provable." (So if φ is true, then the system is incomplete; and if φ is false, then the system is inconsistent.)

The second strike against the formalist program was the proof of the *undecidability of the halting problem*, shown independently by Alan Turing and Alonzo Church in the 1930s. We can think of the halting problem as asking the following question: given a function f written in Python and an input x, does running $f(x)$ get stuck in an infinite loop? (Or does it eventually terminate?) The *undecidability* of this problem means that *there is no algorithm that solves the halting problem.* A corollary of this result is that our problem—given a fully quantified proposition φ, is φ a theorem?—is also undecidable. We'll discuss uncomputability in more detail in Chapter 4.

Undecidability, incompleteness, and their profound consequences are the focus of a number of excellent textbooks on the theory of computation[9]—and also Douglas Hofstadter's fascinating masterpiece *Gödel, Escher, Bach,*[10] which is all-but-required reading for computer scientists.

See, for example:

[9] Dexter Kozen. *Automata and Computability*. Springer, 1997; and Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 3rd edition, 2012.

[10] Douglas Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Vintage, 1980.

3.4.5 Exercises

Figure 3.28 lists some well-known programming languages, with some characteristics. Using these characteristics, define a predicate that's true for each of the following lists of languages, and false for every other language in the table. For example, the predicate $P(x)$ = "x has strong typing and x is not functional" makes P(Pascal) and P(Java) true, and makes $P(x)$ false for every $x \in$ {C, C++, LaTeX, ML, Perl, Scheme}.

3.107 Java

3.108 ML, Perl

3.109 Pascal, Scheme, Perl

3.110 LaTeX, Java, C++, Perl

3.111 C, Pascal, ML, C++, LaTeX, Scheme, Perl

	paradigm	*typing*	*scope*
C	*imperative*	*weak*	*lexical*
C++	*object-oriented*	*weak*	*lexical*
Java	*object-oriented*	*strong*	*lexical*
LaTeX	*scripting*	*weak*	*dynamic*
ML	*functional*	*strong*	*lexical*
Pascal	*imperative*	*strong*	*lexical*
Perl	*scripting*	*weak*	*either*
Scheme	*functional*	*weak*	*either*

Figure 3.28: Some programming languages.

Examples 3.4 and 3.15 construct a proposition corresponding to "the password contains at least three of four character types (digits, lowercase letters, uppercase letters, other)." In that example, we took "the password contains at least one digit" (and its analogues for the other character types) as an atomic proposition. But we could give a lower-level characterization of valid passwords. Let isDigit, isLower, *and* isUpper *be predicates that are true of single characters of the appropriate type. Use standard arithmetic notation and these predicates to formalize the following conditions on a password $x = \langle x_1, \ldots, x_n \rangle$, where x_i is the ith character in the password:*

3.112 x is at least 8 characters long.

3.113 x contains at least one lowercase letter.

3.114 x contains at least one non-alphanumeric character. (Remember that *isDigit*, *isLower*, and *isUpper* are the only predicates available!)

3.115 (*Inspired by a letter to the editor in* The New Yorker *by Alexander George from 24 December 2007.*) Steve Martin, the great comedian, reports in *Born Standing Up: A Comic's Life* that, inspired by Lewis Carroll, he started closing his shows with the following line.[11] (It got big laughs.)

> *I'm not going home tonight; I'm going to Bananaland, a place where only two things are true, only two things: One, all chairs are green; and two, no chairs are green.*

Steve Martin describes the joke as a contradiction—but, in fact, these two true things are not contradictory! Describe how it is possible for both "all chairs in Bananaland are green" and "no chairs in Bananaland are green" to be simultaneously true.

As a rough approximation, we can think of a database *as a two-dimensional table, where rows correspond to individual entities, and columns correspond to fields (data about those entities). A database* query *defines a predicate $Q(x)$ that consists of tests of the values from various columns, joined by the basic logical connectives. The database system then returns a list of rows/entities for which the predicate is true. We can think of this type of database access as involving predicates: in response to query Q, the system returns the list of all rows x for which $Q(x)$ is true.*

See Figure 3.29 for an example; here, to find a list of all students with grade point averages over 3.4 *who have taken at least one CS course if and only if they're from Hawaii, we could query* $\text{GPA}(x) \geq 3.4 \wedge (\text{CS?}(x) = \textit{yes} \Leftrightarrow \text{home}(x) = \textit{Hawaii})$. *For this database, this query would return Charlie (and not Alice, Bob, or Dave).*

Each of the following predicates $Q(x)$ uses tests on particular columns in x's row. For each, give a logically equivalent predicate in which each column's name appears at most once. *You may also use the symbols* $\{\text{True}, \text{False}, \wedge, \vee, \neg, \Rightarrow\}$ *as many times as you please. Use a truth table to prove that your answer is logically equivalent to the given predicate.*

name	*GPA*	*CS?*	*home*	⋯
Alice	4.0	*yes*	*Alaska*	⋯
Bob	3.14	*yes*	*Bermuda*	⋯
Charlie	3.54	*no*	*California*	⋯
Dave	3.8	*yes*	*Delaware*	⋯
		⋮		

Figure 3.29: A sample database.

3.116 $[age(x) < 18] \vee (\neg[age(x) < 18] \wedge [gpa(x) \geq 3.0])$

3.117 $cs(x) \Rightarrow \neg(hawaii(x) \Rightarrow (hawaii(x) \wedge cs(x)))$

3.118 $(hasMajor(x) \wedge \neg junior(x) \wedge oncampus(x)) \vee (hasMajor(x) \wedge \neg junior(x) \wedge \neg oncampus(x))$
$\vee (hasMajor(x) \wedge junior(x) \wedge \neg oncampus(x))$

3.119 Following the last few exercises, you might begin to think that any query can be rewritten without duplication. Can it? Consider a unary predicate that is built up from the predicates $P(x)$ and $Q(x)$ and the propositional symbols $\{\text{True}, \text{False}, \wedge, \vee, \neg, \Rightarrow\}$. Decide whether the following claim is true or false, and prove your answer:

Claim: Every such predicate is logically equivalent to a predicate that uses only the following symbols: (i) $\{\text{True}, \text{False}, \wedge, \vee, \neg, \Rightarrow\}$, all of which can be used as many times as you please; and (ii) the predicates $\{P(x), Q(x)\}$, which can appear *only one time each.*

[11] Steve Martin. *Born Standing Up: A Comic's Life*. Simon & Schuster, 2008.

Modern web search engines allow users to specify Boolean conditions in their queries. For example, "social OR networks" will return only web pages containing either the word "social" or the word "networks." You can view a query as a predicate Q; the search engine returns (in some order) the list of all pages p for which $Q(p)$ is true. Consider the following queries:

A: "java AND program AND NOT computer"
B: "(computer OR algorithm) AND java"
C: "java AND NOT (computer OR algorithm OR program)"

Give an example of a web page—or a sentence—that would be returned . . .

3.120 . . . by query A but not by B or C.

3.121 . . . by query B but not by A or C.

3.122 . . . by query C but not by A or B.

3.123 Prove or disprove: $\forall n \in \mathbb{Z} : \mathit{isPrime}(n) \Rightarrow \frac{n}{2} \notin \mathbb{Z}$.

3.124 Translate this Groucho Marx quote into logical notation: *It isn't necessary to have relatives in Kansas City in order to be unhappy.* Let $P(x)$ be "x has relatives in Kansas City" and $Q(x)$ be "x is unhappy," and view the statement as implicitly making a claim that a particular kind of person exists.

Write an English sentence that expresses the logical negation of each given sentence. (Don't just say "It is not the case that ..."; give a genuine negation.) Some of the given sentences are ambiguous in their meaning; if so, describe all of the interpretations of the sentence that you can find, then choose one and give its negation.

3.125 Every entry in the array A is positive.

3.126 Every decent programming language denotes block structure with parentheses or braces.

3.127 There exists an odd number that is evenly divisible by a different odd number.

3.128 There is a point in Minnesota that is farther than ten miles from a lake.

3.129 Every sorting algorithm takes at least $n \log n$ steps on some n-element input array.

In Examples 3.43 and 3.44, we proved that

$$\exists x \in S : \Big[P(x) \lor Q(x)\Big] \;\Leftrightarrow\; \Big[\exists x \in S : P(x)\Big] \lor \Big[\exists x \in S : Q(x)\Big]$$

$$\forall x \in S : \Big[P(x) \lor Q(x)\Big] \;\Leftarrow\; \Big[\forall x \in S : P(x)\Big] \lor \Big[\forall x \in S : Q(x)\Big]$$

are theorems. Argue that the following $\land$-analogies to these statements are also theorems:

3.130 $\exists x \in S : \Big[P(x) \land Q(x)\Big] \;\Rightarrow\; \Big[\exists x \in S : P(x)\Big] \land \Big[\exists x \in S : Q(x)\Big]$

3.131 $\forall x \in S : \Big[P(x) \land Q(x)\Big] \;\Leftrightarrow\; \Big[\forall x \in S : P(x)\Big] \land \Big[\forall x \in S : Q(x)\Big]$

Explain why the following are theorems of predicate logic:

3.132 $\Big[\forall x \in S : P(x) \Rightarrow Q(x)\Big] \land \Big[\forall x \in S : P(x)\Big] \Rightarrow \Big[\forall x \in S : Q(x)\Big]$

3.133 $\Big[\forall x \in \{y \in S : P(y)\} : Q(x)\Big] \;\Leftrightarrow\; \Big[\forall x \in S : P(x) \Rightarrow Q(x)\Big]$

3.134 $\Big[\exists x \in \{y \in S : P(y)\} : Q(x)\Big] \;\Leftrightarrow\; \Big[\exists x \in S : P(x) \land Q(x)\Big]$

Explain why the following propositions are theorems of predicate logic, assuming that x does not appear as a free variable in the expression φ (and assuming that S is nonempty):

3.135 $\varphi \Leftrightarrow \Big[\forall x \in S : \varphi\Big]$

3.136 $\varphi \lor \Big[\forall x \in S : P(x)\Big] \Leftrightarrow \Big[\forall x \in S : \varphi \lor P(x)\Big]$

3.137 $\varphi \land \Big[\exists x \in S : P(x)\Big] \Leftrightarrow \Big[\exists x \in S : \varphi \land P(x)\Big]$

3.138 $\Big(\varphi \Rightarrow \Big[\exists x \in S : P(x)\Big]\Big) \Leftrightarrow \Big[\exists x \in S : \varphi \Rightarrow P(x)\Big]$

3.139 $\Big(\Big[\exists x \in S : P(x)\Big] \Rightarrow \varphi\Big) \Leftrightarrow \Big[\forall x \in S : P(x) \Rightarrow \varphi\Big]$

3.140 Give an example of a predicate P, a nonempty set S, and an expression φ containing x as a free variable such that the proposition from Exercise 3.136 is false. Because x has to get its meaning from somewhere, we will imagine a universal quantifier for x wrapped around the entire expression. Specifically, give an example of P, φ, and S for which

$$\forall x \in S : \Big[\varphi \lor [\forall x \in S : P(x)]\Big] \quad \text{is not logically equivalent to} \quad \forall x \in S : \Big[[\forall x \in S : \varphi \lor P(x)]\Big].$$

3.5 *Predicate Logic: Nested Quantifiers*

> Everybody hates me because I'm so universally liked.
>
> Peter De Vries (1910–1993)

Just as we can place one loop inside another in a program, we can place one quantified statement inside another in predicate logic. In fact, the most interesting quantified statements almost always involve more than one quantifier. (For example: *during every semester, there's a computer science class that every student on campus can take.*) In formal notation, such a statement typically involves *nested quantifiers*—that is, multiple quantifiers in which one quantifier appears inside the scope of another.

We've encountered statements involving nested quantification before, although so far we've discussed them using English rather than mathematical notation. The definition of a partition of a set (Definition 2.30) or of an onto function (Definition 2.49) are two examples. (To make the latter definition's quantifiers more explicit: an onto function $f : A \to B$ is one where, for every element of B, there's an element of A such that $f(a) = b$: that is, $\forall b \in B : \big[\exists a \in A : f(a) = b\big]$.) Here are two other examples:

Example 3.45 (No unmatched elements in an array)

Let's express the condition that every element of an array $A[1 \ldots n]$ is a "double"—that is, appears at least twice in A. (For example, the array $[3, 2, 1, 1, 4, 4, 2, 3, 1]$ satisfies this condition.) This condition requires that, for every index i, there exists another index j such that $A[i] = A[j]$. We can express the requirement as follows:

$$\forall i \in \{1, 2, \ldots, n\} : \Big[\exists j \in \{1, 2, \ldots, n\} : i \neq j \wedge A[i] = A[j]\Big].$$

Example 3.46 (Alphabetically later)

Let's formalize the predicate "The string ___ is alphabetically after the string ___" from Example 3.28. For two letters $a, b \in \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}$, write $a < b$ if a is earlier in the alphabet than b; we'll use this ordering on *letters* to define an ordering on *strings*. Let x and y be strings over $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}$. There are two ways for x to be alphabetically later than y:

- y is a (proper) prefix of x. (See Example 3.35.) For example, $\underline{\texttt{FORT}}\texttt{RAN}$ is after $\underline{\texttt{FORT}}$.
- x and y share an initial prefix of identical letters, and the first i for which $x_i \neq y_i$ has x_i later in the alphabet than y_i. For example, $\underline{\texttt{PAS}\underline{\texttt{T}}}\texttt{OR}$ comes after $\underline{\texttt{PAS}\underline{\texttt{C}}}\texttt{AL}$.

Formally, then, $x \in \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}^n$ is alphabetically after $y \in \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}^m$ if

$$\Big[m < n \ \wedge \ [\forall j \in \{1, 2 \ldots, m\} : x_j = y_j]\Big] \qquad \textit{y is a proper prefix of x} \ldots$$

$$\vee \Big[\exists i \in \{1, \ldots, \min(n, m)\} : \ x_i > y_i \ \wedge \ [\forall j \in \{1, 2 \ldots, i-1\} : x_i = y_i]\Big]$$

$$\ldots \textit{or } x_{1,\ldots,i-1} = y_{1,\ldots,i-1} \textit{ and } x_i > y_i.$$

"Sorting alphabetically" is usually called *lexicographic ordering* in computer science. This ordering reflects the way that words are listed in the dictionary (also known as the *lexicon*).

Here is one more example of a statement that we've already seen—Goldbach's conjecture—that implicitly involves nested quantifiers; we'll formalize it in predicate logic. (Part of the point of this example is to illustrate how complex even some apparently simple concepts are; there's a good deal of complexity hidden in words like "even" and "prime," which at this point seem pretty intuitive!)

Writing tip: Just as with nested loops in programs, the deeper the nesting of quantifiers, the harder an expression is for a reader to follow. Using well-chosen predicates (like *isPrime*, for example) in a logical statement can make it much easier to read—just like using well-chosen (and well-named) functions makes your software easier to read!

Example 3.47 (Goldbach's Conjecture)

Problem: Recall Goldbach's conjecture, from Example 3.1:

> Every even integer greater than 2 can be written as the sum of two prime numbers.

Formalize this proposition using nested quantifiers.

Solution: Using the *sumOfTwoPrimes* predicate from Example 3.34, we can write this statement as either of the following:

$$\forall n \in \{n \in \mathbb{Z} : n > 2 \land 2 \mid n\} : \mathit{sumOfTwoPrimes}(n) \tag{A}$$

$$\forall n \in \mathbb{Z} : \Big[n > 2 \land 2 \mid n \;\Rightarrow\; \mathit{sumOfTwoPrimes}(n) \Big] \tag{B}$$

In (B), we quantify over *all* integers, but the implication $n > 2 \land 2 \mid n \Rightarrow \mathit{sumOfTwoPrimes}(n)$ is trivially true for an integer n that's not even or not greater than 2, because false implies anything! Thus the only instantiations of the quantifier in which the implication has any "meat" is for even integers greater than 2. As such, these two formulations are equivalent. (See Exercise 3.133.) Expanding the definition of *sumOfTwoPrimes*(n) from Example 3.34, we can also rewrite (B) as

$$\left[\begin{array}{l} \forall n \in \mathbb{Z} : n > 2 \land 2 \mid n \;\;\Rightarrow \\ \quad \exists p \in \mathbb{Z} : \exists q \in \mathbb{Z} : \; \big[\mathit{isPrime}(p) \land \mathit{isPrime}(q) \land n = p + q\big] \end{array} \right] \tag{C}$$

We've also already seen that the predicate *isPrime* implicitly contains quantifiers too ("for all potential divisors d, it is not the case that d evenly divides p")—and, for that matter, so does the "evenly divides" predicate $\mid$. In Exercises 3.178, 3.179, and 3.180, you'll show how to rewrite Goldbach's Conjecture in a few different ways, including using yet further layers of nested quantifiers.

3.5.1 *Order of Quantification*

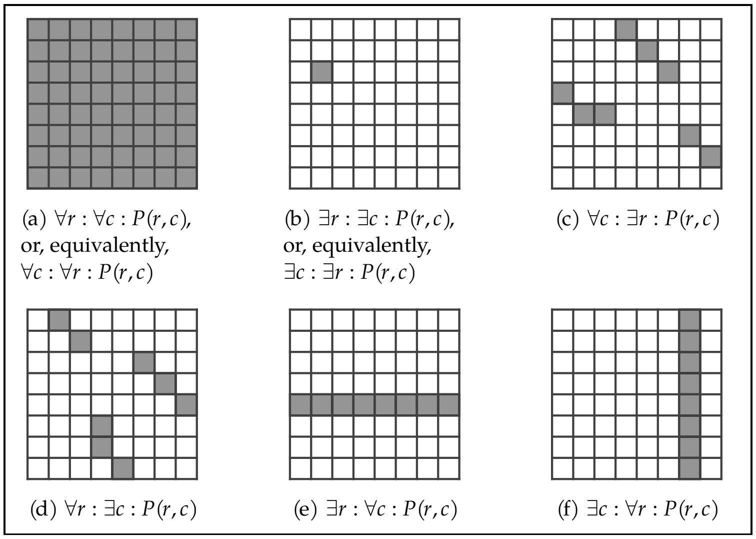
In expressions that involve nested quantifiers, the order of the quantifiers matters! As a frivolous example, take the title of the 1947 hit song "Everybody Loves Somebody" (sung by Dean Martin). There are two plausible interpretations of the title:

$$\forall x : \exists y : \; x \text{ loves } y \qquad\qquad \text{and} \qquad\qquad \exists y : \forall x : \; x \text{ loves } y.$$

The former is the more natural reading; it says that every person x has someone that he or she loves, but each different x can love a different person. (As in: "every child loves his or her mother.") The latter says that there is one single person loved by *every* x. (As in: "Everybody loves Raymond.") These claims are different!

> **Taking it further:** Disambiguating the order of quantification in English sentences is one of the most daunting challenges in natural language processing (NLP) systems. (See p. 314.) Compare *Every student received a diploma* and *Every student heard a commencement address*: there are, surely, many diplomas and only one address, but building a software system that understands that fact is tremendously challenging! There are many other vexing types of ambiguity in NLP systems, too. A classic example of ambiguity in natural language is the sentence *I saw the man with the telescope*. Is the man holding a telescope? Or did I use one to see him? Human listeners are able to use pragmatic knowledge about the world to disambiguate, but doing so properly in an NLP system is very difficult.

Figure 3.30 shows a visual representation of the importance of this order of quantification. Compare Figure 3.30(d) and Figure 3.30(f), for example: $\forall r : \exists c : P(r,c)$ is true if every row has at least one column with a filled cell in it, whereas $\exists c : \forall r : P(r,c)$ requires that there be a *single* column so that every row has that column's cell filled. The proposition $\exists c : \forall r : P(r,c)$ is *not* true in Figure 3.30(d)—though the proposition $\forall r : \exists c : P(r,c)$ is true in *both* Figure 3.30(d) and Figure 3.30(f).



(a) $\forall r : \forall c : P(r,c)$, or, equivalently, $\forall c : \forall r : P(r,c)$

(b) $\exists r : \exists c : P(r,c)$, or, equivalently, $\exists c : \exists r : P(r,c)$

(c) $\forall c : \exists r : P(r,c)$

(d) $\forall r : \exists c : P(r,c)$

(e) $\exists r : \forall c : P(r,c)$

(f) $\exists c : \forall r : P(r,c)$

Figure 3.30: An illustration of order of quantification. Let r index a *row* of the grid, and let c index a *column*. If $P(r,c)$ is true in each filled cell, then the corresponding proposition is true.

Here's a mathematical example that illustrates the difference even more precisely.

Example 3.48 (The largest real number)

Problem: One of the following propositions is true; the other is false. Which is which?

$$\exists y \in \mathbb{R} : \forall x \in \mathbb{R} : x < y \tag{A}$$

$$\forall x \in \mathbb{R} : \exists y \in \mathbb{R} : x < y \tag{B}$$

Solution: Translating these propositions into English helps resolve this question.

(A) says that there is a real number y for which the following property holds: every real number is less than y. ("There is a largest real number.") But there isn't a largest real number! So (A) is false. (If someone tells you that y^* satisfies $\forall x \in \mathbb{R} : x < y^*$, then you can convince him he's wrong by choosing $x = y^* + 1$.) On the other hand, (B) says that, for every real number x, there is a real number greater than x. And that's true: for any $x \in \mathbb{R}$, the number $x + 1$ is greater than x.

In fact, (B) is nearly the negation of (A). (Before you read through the derivation, can you figure out why we had to say "nearly" in the last sentence?)

$$\begin{aligned}
&\equiv \neg\big[\exists y \in \mathbb{R} : \forall x \in \mathbb{R} : x < y\big] && \textit{negation of (A)}\\
&\equiv \forall y \in \mathbb{R} : \neg\big[\forall x \in \mathbb{R} : x < y\big] && \textit{De Morgan's Laws (quantified form)}\\
&\equiv \forall y \in \mathbb{R} : \exists x \in \mathbb{R} : \neg(x < y) && \textit{De Morgan's Laws (quantified form)}\\
&\equiv \forall y \in \mathbb{R} : \exists x \in \mathbb{R} : y \leq x && \neg(x < y) \Leftrightarrow y \leq x\\
&\equiv \forall x \in \mathbb{R} : \exists y \in \mathbb{R} : x \leq y. && \textit{renaming the bound variables}
\end{aligned}$$

So (B) and the negation of (A) are almost—but not quite—identical: the latter has a $\leq$ where the former has a $<$. But both (B) and $\neg$(A) are theorems!

Although the order of quantifiers does matter when universal and existential quantifiers both appear in a proposition, the order of consecutive universal quantifiers doesn't matter, nor does the order of consecutive existential quantifiers. (Using our previously defined terminology—see Figure 3.12—these quantifiers are *commutative.*) Thus the following statements are theorems of predicate logic:

$$\forall x \in S : \forall y \in T : P(x,y) \;\Leftrightarrow\; \forall y \in T : \forall x \in S : P(x,y) \qquad (*)$$

$$\exists x \in S : \exists y \in T : P(x,y) \;\Leftrightarrow\; \exists y \in T : \exists x \in S : P(x,y) \qquad (**)$$

The point is simply that the left- and right-hand sides of $(*)$ are both true if and only if $P(x,y)$ is true for every pair $\langle x,y \rangle \in S \times T$, and the left- and right-hand sides of $(**)$ are both true if and only if $P(x,y)$ holds for at least one pair $\langle x,y \rangle \in S \times T$. See Figure 3.30(a) and Figure 3.30(b): both sides of $(*)$ require that all the cells be filled and both sides of $(**)$ require that at least one cell be filled. Because of these equivalences, as notational shorthand we'll sometimes write $\forall x, y \in S : P(x,y)$ instead of writing $\forall x \in S : \forall y \in S : P(x,y)$. We'll use $\exists x, y \in S : P(x,y)$ analogously.

Nested quantification and nested loops

Just as it can be helpful to think of a quantifier in terms of a corresponding loop, it can be helpful to think of nested quantifiers in terms of nested loops. And a useful way to think about the importance of the order of quantification is through the way in which changing the order of nested loops changes what they compute. In Exercises 3.191–3.196, you'll get a chance to do some translations between quantified statements and nested loops.

Here's one example about how thinking about the nested-loop analogy for nested quantifiers can be helpful. Imagine writing a nested loop to examine every element of a 2-dimensional array. As long as iterations don't depend on each other, it doesn't matter whether we proceed through the array in *row-major order* ("for each row, look at all columns' entries") or *column-major order* ("for each column, look at all rows' entries"). Figure 3.31 illustrates a loop-based view of the logical equivalence expressed by $(**)$, above: both code segments always have the same return value. (The graphical view is that both check every cell of the "grid" of possible inputs to A, as in Figure 3.30(b), just in different orders.)

```
for $j = 1$ to $m$:
    for $i = 1$ to $n$:
        if $A[i,j]$ then
            return True
return False
```

```
for $i = 1$ to $n$:
    for $j = 1$ to $m$:
        if $A[i,j]$ then
            return True
return False
```

Figure 3.31: Two nested **for** loops that return the value of $\exists i : \exists j : A[i,j] \equiv \exists j : \exists i : A[i,j]$, by looping in row- or column-major orders.

3.5.2 *Negating Nested Quantifiers*

Recall the rules for negating quantifiers found earlier in the chapter:

$$\neg\forall x \in S : P(x) \;\Leftrightarrow\; \exists x \in S : \neg P(x)$$

$$\neg\exists x \in S : P(x) \;\Leftrightarrow\; \forall x \in S : \neg P(x)$$

Informally, these theorems say that "*everybody is P* is false" is equivalent to "*somebody isn't P*"; and, similarly, "*somebody is P* is false" is equivalent to "*everybody isn't P*."

Here we will consider negating a sequence of *nested* quantifiers. Negating nested quantifiers proceeds in precisely the same way as negating a single quantifier, just acting on one quantifier at a time. (We already saw this idea in Example 3.48, where we repeatedly applied these quantified versions of De Morgan's Laws to a sequence of nested quantifiers.) For example:

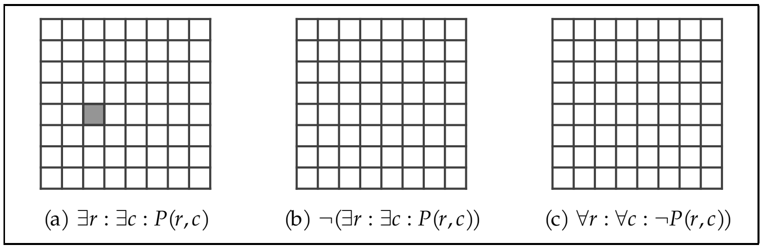

(a) $\exists r : \exists c : P(r,c)$
(b) $\neg(\exists r : \exists c : P(r,c))$
(c) $\forall r : \forall c : \neg P(r,c))$

Figure 3.32: Negating nested quantifiers: what it means for (a) a filled cell to exist; (b) it not to be the case that a filled cell exists; and (c) that every cell is unfilled.

Example 3.49 (No cell is filled ≡ every cell is empty)

Observe that $\exists r : \exists c : P(r,c)$ is true if any r and c makes $P(r,c)$ true—that is, visually, that any cell in the grid in Figure 3.32(a) is filled. For $\exists r : \exists c : P(r,c)$ to be false (Figure 3.32(b)), then we need:

$$\neg(\exists r : \exists c : P(r,c)) \equiv \forall r : \neg(\exists c : P(r,c)) \equiv \forall r : \forall c : \neg P(r,c).$$

That is, $P(r,c)$ *is false for every r and c*—that is, visually, every cell in the grid is unfilled (Figure 3.32(c)). Similarly,

$$\neg\exists r : \forall c : P(r,c) \equiv \forall r : \neg\forall c : P(r,c) \equiv \forall r : \exists c : \neg P(r,c).$$

Thus $\neg\exists r : \forall c : P(r,c)$ expresses that *it's not the case that there's a row with all columns filled*; using the above equivalence, we can rephrase the condition as *every row has at least one unfilled column.*

Example 3.50 (Triple negations)

Here's an example of negating a sequence of triply nested quantifiers:

$$\begin{aligned}\neg\exists x : \forall y : \exists z : P(x,y,z) &\equiv \forall x : \neg\forall y : \exists z : P(x,y,z)\\ &\equiv \forall x : \exists y : \neg\exists z : P(x,y,z)\\ &\equiv \forall x : \exists y : \forall z : \neg P(x,y,z).\end{aligned}$$

Here's a last example, which requires translation from English into logical notation:

Example 3.51 (Negating nested quantifiers)

Problem: Negate the following sentence:

> *For every iPhone user, there's an iPhone app that every one of that user's iPhone-using friends has downloaded.*

Solution: First, let's reason about how the given statement would be false: there would be some iPhone user—we'll call him Steve—such that, for every iPhone app, Steve has a friend who didn't download that app.

Write U and A for the sets of iPhone users and apps, respectively. In (pseudo)logical notation, the original claim looks like

$$\forall u \in U : \exists a \in A : \forall v \in U : \big[(u, v \text{ friends}) \Rightarrow (v \text{ downloaded } a)\big].$$

To negate this statement, we apply the quantified De Morgan's laws, once per quantifier:

$$\begin{aligned}
&\neg\forall u \in U : \exists a \in A : \forall v \in U : [(u, v \text{ friends}) \Rightarrow (v \text{ downloaded } a)]\\
&\equiv \exists u \in U : \neg\exists a \in A : \forall v \in U : [(u, v \text{ friends}) \Rightarrow (v \text{ downloaded } a)]\\
&\equiv \exists u \in U : \forall a \in A : \neg\forall v \in U : [(u, v \text{ friends}) \Rightarrow (v \text{ downloaded } a)]\\
&\equiv \exists u \in U : \forall a \in A : \exists v \in U : \neg[(u, v \text{ friends}) \Rightarrow (v \text{ downloaded } a)].
\end{aligned}$$

Using $\neg(p \Rightarrow q) \equiv p \wedge \neg q$ (Exercise 3.82), we can further write this expression as:

$$\equiv \exists u \in U : \forall a \in A : \exists v \in U : [(u, v \text{ friends}) \wedge \neg(v \text{ downloaded } a)].$$

This last proposition, translated into English, matches the informal description above as to why the original claim would be false: *there's some person such that, for every app, that person has a friend who hasn't downloaded that app.*

3.5.3 *Two New Ways of Considering Nested Quantifiers*

We'll close this section with two different but useful ways to think about nested quantification. As a running example, consider the following (true!) proposition

$$\forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : x = y + 1, \qquad (\dagger)$$

which says that the number that's one less than every integer is an integer too. We'll discuss two ways of thinking about propositions like (†) with nested quantifiers: as a "game with a demon" in which you battle against an all-knowing demon to try to make the innermost quantifier's body true;[12] and as a single quantifier whose body is a predicate, but a predicate that just happens to be expressed using quantifiers.

Thanks to Dexter Kozen for teaching me this way of thinking of nested quantifiers. See:

NESTED QUANTIFIERS AS DEMON GAMES

One way to think about any proposition involving nested quantifiers is as a "game" played between you and a demon. Here are the rules of the game:

- Your goal is to make the innermost statement—$x = y + 1$ for our running example (†)—turn out to be true; the demon's goal is to make that statement false.
- Every "for all" quantifier in the expression is a choice that the demon makes; every "there exists" quantifier in the expression is a choice that you get to make. (That

[12] Dexter Kozen. *Automata and Computability*. Springer, 1997.

is, in the expression $\forall a \in S : \cdots$, the demon chooses a particular value of $a \in S$, and the game continues in the "$\cdots$" part of the expression. And in the expression $\exists b \in S : \cdots$, you choose a particular value of $b \in S$, and, again, the game continues in the "$\cdots$" part.)

- Your choices and the demon's choices are made in the left-to-right order (from the outside in) of the quantifiers.

- You win the game—in other words, the proposition in question is true—if, no matter how cleverly the demon plays, you make the innermost statement true.

Here are two examples of viewing quantified statement as demon games, one for a true statement and one for a false statement:

Example 3.52 (Showing that (†) is true)
We'll use a "game with the demon" to argue that $\forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : x = y + 1$ is true.

1. The outermost quantifier is $\forall$, so the demon picks a value for $x \in \mathbb{Z}$.
2. Now you get to pick a value $y \in \mathbb{Z}$. A good choice for you is $y := x - 1$.
3. Because you chose $y = x - 1$, indeed $x = y + 1$. You win!

(For example, if the demon picks 16, you pick 15. If the demon picks -19, you pick -20. And so forth.) No matter what the demon picks, your strategy will make you win—and therefore (†) is true!

By contrast, consider (†) with the order of quantification reversed:

Example 3.53 (A losing demon game)
Consider playing a demon game for the proposition

$$\exists y \in \mathbb{Z} : \forall x \in \mathbb{Z} : x = y + 1.$$

Unfortunately, the $\exists$ is first, which means that you have to make the first move. But when you pick a number y, the demon *then* gets to pick an x—and there are an infinitude of x values that the demon can choose so that $x \neq y + 1$. (You pick 42? The demon picks 666. You pick 17? The demon picks 666. You pick 665? The demon picks 616.) Therefore you can't guarantee that you win the game, so we haven't established this claim.

By the way, you *could* win a demon game to prove the negation of the claim in Example 3.53:

$$\neg(\text{the claim from Example 3.53}) \equiv \forall y \in \mathbb{Z} : \exists x \in \mathbb{Z} : x \neq y + 1.$$

First, the demon picks some unknown $y \in \mathbb{Z}$. Then you have to pick an $x \in \mathbb{Z}$ such that $x \neq y + 1$—but that's easy: for any y the demon picks, you pick $x = y$. You win!

Nested quantifiers as single quantifiers

In our running example—$\forall x \in \mathbb{Z} : \underline{\exists y \in \mathbb{Z} : x = y + 1}$—what kind of thing is the underlined piece of the expression? It can't be a proposition, because x is a free variable in it. But once we plug in a value for x, the expression becomes true or false. In other words, the expression $\exists y \in \mathbb{Z} : x - 1 = y$ is itself a (unary) predicate: once we are given a value of x, we can compute the truth value of the expression. Similarly, the expression $x - 1 = y$ is also a predicate—but a binary predicate, taking both an x and y as arguments. Let's define two predicates:

- $P(x, y)$ denotes the predicate $x - 1 = y$.
- $\mathit{hasIntPredecessor}(x)$ denotes the predicate $\exists y \in \mathbb{Z} : x - 1 = y$.

Using this notation, we can write (†) as

$$\begin{aligned}
\forall x \in \mathbb{Z} : \overbrace{\exists y \in \mathbb{Z} : \overbrace{x - 1 = y}^{P(x,y)}}^{\mathit{hasIntPredecessor}(x)} &\equiv \forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : P(x, y) \\
&\equiv \forall x \in \mathbb{Z} : \mathit{hasIntPredecessor}(x). \qquad (\ddagger)
\end{aligned}$$

One implication of this view is that negating nested quantifiers is really just the same as negating non-nested quantifiers. For example:

Example 3.54 (Negating nested quantifiers)
We can view the negation of (†), as written in (‡), as follows:

$$\begin{aligned}
\neg(\dagger) &\equiv \neg\forall x \in \mathbb{Z} : \mathit{hasIntPredecessor}(x) \\
&\equiv \exists x \in \mathbb{Z} : \neg\mathit{hasIntPredecessor}(x).
\end{aligned}$$

And, re-expanding the definition of *hasIntPredecessor* and again applying the quantified De Morgan's Law, we have that

$$\begin{aligned}
\neg\mathit{hasIntPredecessor}(x) &\equiv \neg\exists y \in \mathbb{Z} : P(x, y) \\
&\equiv \forall y \in \mathbb{Z} : \neg P(x, y) \\
&\equiv \forall y \in \mathbb{Z} : x - 1 \neq y.
\end{aligned}$$

Together, these two negations show

$$\begin{aligned}
\neg(\dagger) &\equiv \exists x \in \mathbb{Z} : \neg\mathit{hasIntPredecessor}(x) \\
&\equiv \exists x \in \mathbb{Z} : \forall y \in \mathbb{Z} : \neg P(x, y) \\
&\equiv \exists x \in \mathbb{Z} : \forall y \in \mathbb{Z} : x - 1 \neq y.
\end{aligned}$$

Taking it further: This view of nested quantifiers as a single quantifier whose body just happens to express its condition using quantifiers has a close analogy with writing a particular kind of function in a programming language. If we look at a two-argument function in the right light, we can see it as a function that takes one argument *and returns a function that takes one argument*. This approach is called *Currying*; see p. 357 for some discussion.

Computer Science Connections

Currying

Consider a binary predicate $P(x,y)$, as used in a quantified expression like $\forall y : \forall x : P(x,y)$. As we discussed, we can think of this expression as first plugging in a value for y, which then yields a unary predicate $\forall x : P(x,y)$ which then takes the argument x.

There's an interesting parallel between this view of nested quantifiers and a way of writing functions in some programming languages. For concreteness, let's think about a very simple function that takes two arguments and returns their sum. Figure 3.33 shows implementations of this function in three different programming languages: ML, Python, and Scheme. A few notes about syntax:

- For ML: `fun` is a keyword that says we're defining a function; `sum` is the name of it; `a b` is the list of arguments; and that function is defined to return the value of `a + b`.
- For Scheme: `(lambda args body)` denotes the function that takes arguments `args` and returns the value of the function body `body`. Applying the function `f` to arguments `arg1`, `arg2`, ..., `argN` is expressed as `(f arg1 arg2 ... argN)`. For example, `(+ 1 2)` has the value 3.

```
fun sum a b = a + b;        (* ML *)

def sum(a,b):               # Python
   return a + b

(define sum                 ; Scheme
   (lambda (a b)
      (+ a b)))
```

Figure 3.33: Implementations of sum$(a,b) = a + b$ in three languages.

We can then use the function `sum` to actually add numbers; see Figure 3.34.

```
sum 2 3;            (* returns 5 *)
sum 99 12;          (* returns 111 *)
```

```
sum(2,3)            # returns 5
sum(99,12)          # returns 111
```

```
(sum 2 3)           ; returns 5
(sum 99 12)         ; returns 111
```

Figure 3.34: Using sum in all three languages.

But now suppose that we wanted to write a new function that takes one argument and adds 3 to it. Can we make use of the `sum` function to do so? (The analogy to predicates is that taking a two-argument predicate and applying it to one argument gives one-argument predicate; here we're trying to take a two-argument function in a programming language and apply it to one argument to yield a one-argument function.) It turns out that creating the "add 3" function using `sum` is very easy in ML: we simply apply `sum` to one argument, and the result is a function that "still wants" one more argument. See Figure 3.35.

```
(* define a "value" add3 as sum
   applied to 3, making add3 a
   1-argument function *)
val add3 = sum 3;

add3 0;            (* returns 3 *)
add3 108;          (* returns 111 *)
add3 199;          (* returns 202 *)
```

Figure 3.35: Applying `sum` to one of two arguments in ML, and then applying the resulting function to a second argument.

A function like `sum` in ML, which takes its multiple arguments "one at a time," is said to be *Curried*—in honor of Haskell Curry, a 20th-century American logician. (The programming language Haskell is also named in his honor.) Thinking about Curried functions is a classical topic in the study of the structure of programming languages.[13] While writing Curried functions is almost automatic in ML, one can also write Curried functions in other programming languages, too. Examples of a Curried version of `sum` in Python and Scheme are in Figure 3.36; it's even possible to write Curried functions in C or Java, though it's much less natural than in ML/Python/Scheme.

```
def sum(a):
  def sumA(b):
     return a + b
  return sumA

sum(3)(2)        # returns 5
add3 = sum(3)
add3(2)          # returns 5
```

```
(define sum              ;; define sum as
  (lambda (a)            ;; the function taking argument a and returning
    (lambda (b) (+ a b))) ;;  [the function taking argument b and returning a+b]

((sum 3) 2)              ;; returns 5
(define add3 (sum 3))
(add3 2)                 ;; returns 5
```

Figure 3.36: Python/Scheme Currying.

For more, see the classic text
[13] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, 2nd edition, 1996.

3.5.4 Exercises

Let F denote the set of all functions $f : \mathbb{R} \to \mathbb{R}$ taking real numbers as input and producing real numbers as output. (For one example, $\mathsf{plusone}(x) = x + 1$ *is a function* $\mathsf{plusone} : \mathbb{R} \to \mathbb{R}$*, so* $\mathsf{plusone} \in F$*.) Are the following propositions true or false? Justify your answers, including a description of the function(s) if they exist.*

3.141 $\forall c \in \mathbb{R} : \big[\exists f \in F : f(0) = c\big]$

3.142 $\exists f \in F : \big[\forall c \in \mathbb{R} : f(0) = c\big]$

3.143 $\forall c \in \mathbb{R} : \big[\exists f \in F : f(c) = 0\big]$

3.144 $\exists f \in F : \big[\forall c \in \mathbb{R} : f(c) = 0\big]$

Under many operating systems, users can schedule a task to be run at a specified time in the future. In Unix-like operating systems, this type of scheduled job is called a cron job. *(For example, a backup might run nightly at 2:00am, and a scratch drive might be emptied out weekly on Friday night at 11:50pm.)*

Greek: *chron-* "time."

Let $T = \{1, 2, \ldots, t_{\max}\}$ be a set of times (measured in minutes, let's say), and let J be a set of jobs. Let scheduledAt *be a predicate so that* scheduledAt(j, t) *is true if and only if job j is scheduled at time t. (Assume that jobs do not last more than one minute.) Formalize the following conditions using only standard quantifiers, arithmetic operators, logical connectives, and the* scheduledAt *predicate.*

3.145 There is never more than one job scheduled at the same time.

3.146 Every job is scheduled at least once.

3.147 Job A is never run twice within two minutes.

3.148 Job B is run at least three times.

3.149 Job C is run at most twice.

3.150 Job D is run sometime after the last time that Job E is run.

3.151 Job F is run at least once between consecutive executions of Job G.

3.152 Job H is run at most once between consecutive executions of Job I.

Let $P[1 \ldots n, 1 \ldots m]$ be a 2-dimensional array of the pixels of a black-and-white image: for every x and y, the value of $P[x, y] = 0$ if the $\langle x, y \rangle$th pixel is black, and $P[x, y] = 1$ if it's white. Translate these statements into predicate logic:

3.153 Every pixel in the image is black.

3.154 There is at least one white pixel.

3.155 Every row has at least one white pixel.

3.156 There are never two consecutive white pixels in the same column.

A standard American crossword puzzle is a 15-by-15 grid, which can be represented as a two-dimensional 15-by-15 array G, where $G[i, j] =$ True *if and only if the cell in the ith row and jth column is "open" (a.k.a. unfilled, a.k.a. not a black square). Maximal contiguous horizontal or vertical sequences of two or more open squares are called* words. *For any $i \leq 0$, $i > 15$, $j \leq 0$, or $j > 15$, treat $G[i, j] =$* False.

> **Taking it further:** *The assumption that the $\langle i, j \rangle$th cell of G is False except when $1 \leq i, j \leq 15$ can be re-expressed as us pretending that our real grid is surrounded by black squares. In CS, this style of structure is called a* sentinel, *wherein we introduce boundary values to avoid having to write out verbose special cases.*

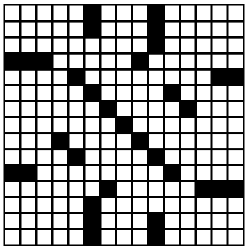


Figure 3.37: A sample American crossword puzzle.

There are certain customs that G must obey to be a standard American puzzle. (See Figure 3.37, for example.) Rewrite the informally stated conditions that follow as fully formal definitions.

3.157 *no unchecked letters:* every open cell appears in both a down word and an across word.

3.158 *no two-letter words:* every word has length at least 3.

3.159 *rotational symmetry:* if the entire grid is rotated by 180°, then the rotated grid is identical to the original grid.

3.160 *overall interlock:* for any two open squares, there is a path of open squares that connects the first to the second. (That is, we can get from *here* to *there* through words.) Your answer should formally define a predicate $P(i, j, x, y)$ that is true exactly when there exists is a path from $\langle i, j \rangle$ to $\langle x, y \rangle$: "there exists a sequence of open squares starting with $\langle i, j \rangle$ such that . . .".)

3.161 Definition 2.30 defines a *partition* of a set S as a set $\{A_1, A_2, \ldots, A_k\}$ of sets such that (i) $A_1, A_2, \ldots, A_k$ are all nonempty; (ii) $A_1 \cup A_2 \cup \cdots \cup A_k = S$; and (iii) for any distinct $i, j \in \{1, \ldots, k\}$, the sets A_i and A_j are disjoint. Formalize this definition using nested quantifiers and basic set notation.

3.162 Consider the "maximum" problem: given an array of numbers, return the maximum element of that array. Complete the formal specification for this problem by finishing the specification under "output":

Input: An array $A[1 \ldots n]$, where each $A[i] \in \mathbb{Z}$.
Output: An integer $x \in \mathbb{Z}$ such that . . .

Let $T = \{1, \ldots, 12\} \times \{0, 1, \ldots, 59\}$ denote the set of numbers that can be displayed on a digital clock in twelve-hour mode. (A clock actually displays a colon between the two numbers.) We can think of a clock as a function $c : T \to T$, so that when the real time is $t \in T$, then the clock displays the time $c(t)$. (For example, if fastby7 *runs seven minutes fast, then* fastby7*(12:00) = 12:07.)*

For several of these questions, it may be helpful to make use of the function add : $T \times \mathbb{Z}^{\geq 0} \to T$ *so that* add(t, x) *denotes the time that's x minutes later than t. See Exercise 2.243.*

Formalize each of the following predicates using only the standard quantifiers and equality symbols.

3.163 A clock is *right* if it always displays the correct time. Formalize the predicate *right*.

3.164 A clock *keeps time* if there's some fixed offset by which it is always off from being right. (For example, *fastby7* above correctly keeps time.) Formalize the predicate *keepsTime*.

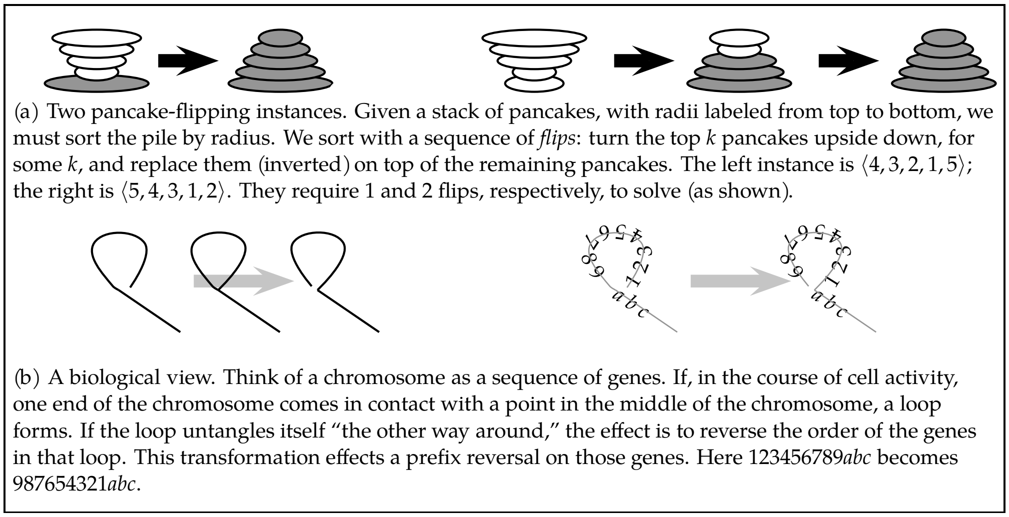
3.165 A clock is *close enough* if it always displays a time that's within two minutes of the correct time. Formalize the predicate *closeEnough*.

3.166 A clock is *broken* if there's some fixed time that it always displays, regardless of the real time. Formalize the predicate *broken*.

3.167 "Even a broken clock is right twice a day," they say. (They mean: "even a broken clock displays the correct time at least once per T.") Formalize the adage and prove it true.

A classic topic of study for computational biologists is genomic distance measures*: given two genomes, we'd like to report a single number that represents how different those two genomes are. These distance computations are useful in, for example, reconstructing the evolutionary tree of a collection of species.*

Consider two genomes A and B of bacterium. Let's label the n genes that appear in A's chromosome, in order, as $\pi_A = 1, 2, \ldots, n$. The same genes appear in a different order in B—say, in the order $\pi_B = r_1, r_2, \ldots r_n$. A particular model of genomic distance will define a specific way in which this list of numbers can mutate; the question at hand is to find the minimum-length sequence of these mutations that are necessary to explain the difference between the orders π_A and π_B. One type of biologically motivated mutation is the prefix reversal*—in which some prefix of π_B is reversed, as in $\langle \underline{3, 2, 1}, 4, 5 \rangle \to \langle \underline{1, 2, 3}, 4, 5 \rangle$. It turns out that this model is exactly the* pancake-flipping problem*, the subject of the lone academic paper with Bill Gates as an author.*[14] *(See Figure 3.38.)*



(a) Two pancake-flipping instances. Given a stack of pancakes, with radii labeled from top to bottom, we must sort the pile by radius. We sort with a sequence of *flips*: turn the top k pancakes upside down, for some k, and replace them (inverted) on top of the remaining pancakes. The left instance is $\langle 4, 3, 2, 1, 5 \rangle$; the right is $\langle 5, 4, 3, 1, 2 \rangle$. They require 1 and 2 flips, respectively, to solve (as shown).

(b) A biological view. Think of a chromosome as a sequence of genes. If, in the course of cell activity, one end of the chromosome comes in contact with a point in the middle of the chromosome, a loop forms. If the loop untangles itself "the other way around," the effect is to reverse the order of the genes in that loop. This transformation effects a prefix reversal on those genes. Here 123456789*abc* becomes 987654321*abc*.

Figure 3.38: The pancake-flipping problem, and its biological significance.

Suppose that you are given a stack of pancake radii $r_1, r_2, \ldots, r_n$, arranged from top to bottom, where $\{r_1, r_2, \ldots, r_n\} = \{1, 2, \ldots, n\}$ (but not necessarily in order). Write down a fully quantified logical expression that expresses the condition that ...

3.168 ... the given pancakes are sorted.

3.169 ... the given pancakes can be sorted with exactly one flip (see Figure 3.38).

3.170 ... the given pancakes can be sorted with exactly two flips. (*Hint: writing a program to verify that your indices aren't off by one is a very good idea!*)

Let P be a set of people, and let T be a set of times. Let friends(x, y) *be a predicate denoting that $x \in P$ and $y \in P$ are friends. Let* bought(x, t) *be a predicate denoting that $x \in P$ bought an iPad at time $t \in T$.*

3.171 Formalize this statement in predicate logic: "Everyone who bought an iPad has a friend who bought one previously."

3.172 Is the claim from Exercise 3.171 true (in the real world)? Justify your answer.

[14] W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979.

In programming, an assertion *is a logical statement that announces ("asserts") a condition φ that the programmer believes to be true. For example, a programmer who is about to access the* 202nd *element of an array A might assert that* length$(A) \geq 202$ *before accessing this element. When an executing program in languages like* C *and* Java *reaches an* `assert` *statement, the program aborts if the condition in the statement isn't true.*

For the following, give a nonempty *input array A* that would cause the stated assertion from Figure 3.39 to fail (*that is, for the asserted condition to be false*).

3.173 `foo`

3.174 `bar`

3.175 `baz`

```
foo(A[1...n]):
   last = 0
   for index = 1 ... n-1:
      if A[index] > A[index+1]:
         last = index
   assert(last >= 1 and last <= n-1)
   swap A[last], A[last+1]
```

```
bar(A[1...n]):
   total = A[1]
   i = 1
   for i = 2 ... n-1:
      if A[i+1] > A[i]:
         total = total + A[i]
         assert(total > A[1])
   return total
```

```
baz(A[1...n]):
   for start = 1 ... n-1:
      min = start
      for i = start+1 ... n:
         assert(start == 1
                  or A[i] > A[start-1])
         if A[min] > A[i]:
            min = i
         swap A[start], A[min]
```

Figure 3.39: Some functions using `assert` statements.

> **Taking it further:** Using assertions can be an extremely valuable way of documenting and debugging programs, particularly because liberally including assertions will allow the revelation of unexpected data values much earlier in the execution of a program. And these languages have a global toggle that allows the testing of assertions to be turned off, so once the programmer is satisfied that the program is working properly, she doesn't have to worry about any running-time overhead for these checks.

While the quantifiers $\forall$ and $\exists$ are by far the most common, there are some other quantifiers that are sometimes used. For each of the following quantifiers, write an expression that is logically equivalent to the given statement that uses only the quantifiers $\forall$ and $\exists$; standard propositional logic notation $(\wedge, \neg, \vee, \Rightarrow)$; standard equality/inequality notation $(=, \geq, \leq, <, >)$; and the predicate P in the question.

3.176 Write an equivalent expression to $\exists!\, x \in \mathbb{Z} : P(x)$ ("there exists a unique $x \in \mathbb{Z}$ such that $P(x)$"), which is true when there is one and only one value of x in the set $\mathbb{Z}$ such that $P(x)$ is true.

3.177 Write an equivalent expression to $\exists_\infty\, x \in \mathbb{Z} : P(x)$ ("there exist infinitely many $x \in \mathbb{Z}$ such that $P(x)$"), which is true when there are infinitely many different values of $x \in \mathbb{Z}$ such that $P(x)$ is true.

Here are two formulations of Goldbach's conjecture (see Example 3.47):

$$\forall n \in \mathbb{Z} : \big[\; n > 2 \wedge 2 \mid n \Rightarrow \big(\exists p \in \mathbb{Z} : \exists q \in \mathbb{Z} : [\text{isPrime}(p) \wedge \text{isPrime}(q) \wedge n = p+q]\big)\;\big]$$
$$\forall n \in \mathbb{Z} : \exists p \in \mathbb{Z} : \exists q \in \mathbb{Z} : \big[\; n \leq 2 \vee 2 \nmid n \vee [\text{isPrime}(p) \wedge \text{isPrime}(q) \wedge n = p+q]\;\big].$$

3.178 Prove that these two formulations of Goldbach's conjecture are logically equivalent.

3.179 Rewrite Goldbach's conjecture without using *isPrime*—that is, using only quantifiers, the $\mid$ predicate, and standard arithmetic $(+, \cdot, \geq$, etc.).

3.180 Even the $\mid$ predicate implicitly involves a quantifier: $p \mid q$ is equivalent to $\exists k \in \mathbb{Z} : p \cdot k = q$. Rewrite Goldbach's conjecture without using the $\mid$ predicate either—that is, use only quantifiers and standard arithmetic symbols $(+, \cdot, \geq$, etc.).

3.181 (*programming required*) As we discussed, the truth value of Goldbach's conjecture is currently unknown. As of April 2012, the conjecture has been verified for all even integers from 4 up to 4×10^{18}, through a massive distributed computation effort led by Tomás Oliveira e Silva. Write a program to test Goldbach's conjecture, in a programming language of your choice, for even integers up to 10,000.

Most real-world English utterances are ambiguous*—that is, there are multiple possible interpretations of the given sentence. A particularly common type of ambiguity involves* order of quantification*. For each of the following English sentences, find as many different logical readings based on order of quantification as you can. Write down those interpretations using pseudological notation, and also write a sentence that expresses each meaning unambiguously.*

3.182 A computer crashes every day.

3.183 Every prime number except 2 is divisible by an odd integer greater than 1.

3.184 Every student takes a class every term.

3.185 Every submitted program failed on a case submitted by a student.

3.186 You should have found two different logical interpretations in Exercise 3.183. One of these interpretations is a theorem, and one of them is not. Decide which is which, and prove your answers.

Let S be an arbitrary nonempty set and let P be an arbitrary binary predicate. Decide whether the following statements are always true (for any P and S), or whether they can be false. Prove your answers.

3.187 $\big[\exists y \in S : \forall x \in S : P(x,y)\big] \Rightarrow \big[\forall x \in S : \exists y \in S : P(x,y)\big]$

3.188 $\big[\forall x \in S : \exists y \in S : P(x,y)\big] \Rightarrow \big[\exists y \in S : \forall x \in S : P(x,y)\big]$

Consider any unary predicate $P(x)$ over a nonempty set S. It turns out that both of the following propositions are theorems of propositional logic. Prove them both.

3.189 $\forall x \in S : \Big[P(x) \Rightarrow \big(\exists y \in S : P(y)\big)\Big]$

3.190 $\exists x \in S : \Big[P(x) \Rightarrow \big(\forall y \in S : P(y)\big)\Big]$

The following blocks of code use nested loops to compute some fact about a predicate P. For each, write a fully quantified statement of predicate logic whose truth value matches the value returned by the given code. (Assume that S is a finite universe.)

3.191

```
for x in S:
   for y in S:
      flag = False
      if P(x) or P(y):
         flag = True
   if flag:
      return True
return False
```

3.192

```
for x in S:
   flag = False
   for y in S:
      if not P(x,y):
         flag = True
   if flag:
      return True
return False
```

3.193

```
for x in S:
   flag = True
   for y in S:
      if not P(x,y):
         flag = False
   if flag:
      return True
return False
```

3.194

```
for x in S:
   flag = False
   for y in S:
      if not P(x,y):
         flag = True
   if not flag:
      return False
return True
```

3.195

```
for x in S:
   for y in S:
      if P(x,y):
         return False
return True
```

3.196

```
flag = False
for x in S:
   for y in S:
      if P(x,y):
         flag = True
return flag
```

3.197 As we've discussed, there is no algorithm that can decide whether a given fully quantified proposition φ is a theorem of predicate logic. But there are several specific types of fully quantified propositions for which we *can* decide whether a given statement is a theorem. Here you'll show that, when quantification is only over a *finite* set, it is possible to give an algorithm to determine whether φ is a theorem. Suppose that you are given a fully quantified proposition φ, where the domain for every quantifier is a finite set—say $S = \{0,1\}$. Describe an algorithm that is guaranteed to figure out whether φ is a theorem.

3.6 Chapter at a Glance

Propositional Logic

negation		$\neg p$	"not p"
disjunction	(inclusive: "p, q, or both")	$p \vee q$	"p or q"
conjunction		$p \wedge q$	"p and q"
implication		$p \Rightarrow q$	"if p, then q" or "p implies q"
equivalence		$p \Leftrightarrow q$	"p if and only if q"
exclusive or	("p or q, but not both")	$p \oplus q$	"p xor q"

Figure 3.40: Logical connectives.

A *proposition* is the kind of thing that is either true or false. An *atomic proposition* (or *Boolean variable*) is a conceptually indivisible proposition. A *compound proposition* (or *Boolean formula*) is one built up using a *logical connective* and one or more simpler propositions. The most common logical connectives are the ones shown in Figure 3.40. A proposition that contains the atomic propositions $p_1, \ldots, p_k$ is sometimes called a *Boolean formula over* $p_1, \ldots, p_k$ or a *Boolean expression over* $p_1, \ldots, p_k$.

The *truth value* of a proposition is its truth or falsity. (The truth value of a Boolean formula over $p_1, \ldots, p_k$ is determined only by the truth values of each of $p_1, \ldots, p_k$.) Each logical connective is defined by how the truth value of the compound proposition formed using that connective relates to the truth values of the constituent propositions. A *truth table* defines a connective by listing, for each possible assignment of truth values for the constituent propositions, the truth value of the entire compound proposition. See Figure 3.41. Observe that the proposition $p \Rightarrow q$ is true if, whenever p is true, q is too. So the only situation in which $p \Rightarrow q$ is false is when p is true and q is false. False implies anything! Anything implies true!

p	$\neg p$
T	F
F	T

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

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

p	q	$p \oplus q$	$p \Leftrightarrow q$
T	T	F	T
T	F	T	F
F	T	T	F
F	F	F	T

Figure 3.41: Truth tables for the basic logical connectives.

Consider a Boolean formula over variables $p_1, \ldots, p_k$. A *truth assignment* is a setting to true or false for each variable. (So a truth assignment corresponds to a row of the truth table for the proposition.) A truth assignment *satisfies* the proposition if, when the values from the truth assignment are plugged in, the proposition is true. A Boolean formula is a *tautology* if *every* truth assignment satisfies it; it's *satisfiable* if *some* truth assignment satisfies it; and it's *unsatisfiable* or a *contradiction* if no truth assignment does. Two Boolean propositions are *logically equivalent* if they're satisfied by exactly the same truth assignments (that is, they have identical truth tables).

Consider an implication $p \Rightarrow q$. The *antecedent* or *hypothesis* of the implication is p; the *consequent* or *conclusion* of the implication is q. The *converse* of the implication $p \Rightarrow q$ is the implication $q \Rightarrow p$. The *contrapositive* is the implication $\neg q \Rightarrow \neg p$. Any implication is logically equivalent to its contrapositive. But an implication is *not* logically equivalent to its converse!

A *literal* is a Boolean variable or the negation of a Boolean variable. A proposition is in *conjunctive normal form* (*CNF*) if it is the conjunction (and) of a collection of clauses, where a clause is a disjunction (or) of a collection of literals. A proposition is in *disjunctive normal form* (*DNF*) if it is the disjunction of a collection of clauses, where a clause is a conjunction of a collection of literals. Every proposition is logically equivalent to a proposition that is in CNF, and to another that is in DNF.

Predicate Logic

A *predicate* is a statement containing some number of variables that has a truth value once values are plugged in for those variables. (Alternatively, a *predicate* is a Boolean-valued function.) Once particular values for these variables are plugged in, the resulting expression is a proposition. A proposition can also be formed from a predicate through *quantifiers:*

- The *universal quantifier* $\forall$ ("for all"): the proposition $\forall x \in U : P(x)$ is true if, for every $x \in U$, we have that $P(x)$ is true.
- The *existential quantifier* $\exists$ ("there exists"): the proposition $\exists x \in U : P(x)$ is true if, for at least one $x \in U$, we have that $P(x)$ is true.

The set U is called the *universe* or *domain of discourse*. When the universe is clear from context, it may be omitted from the notation.

In the expression $[\forall x : \underline{\quad}]$ or $[\exists x : \underline{\quad}]$, the *scope* or *body* of the quantifier is the underlined blank, and the variable x is *bound* by the quantifier. A *free* or *unbound* variable is one that is not bound by any quantifier. A *fully quantified* expression is one with no free variables.

A *theorem* of predicate logic is a fully quantified expression that is true for all possible meanings of the predicates in it. Two expressions are *logically equivalent* if they are true under precisely the same set of meanings for their predicates. (Alternatively, two expressions φ and ψ are *logically equivalent* if $\varphi \Leftrightarrow \psi$ is a theorem.) Two useful theorems of predicate logic are De Morgan's laws: $\neg\forall x \in S : P(x) \Leftrightarrow \exists x \in S : \neg P(x)$ and $\neg\exists x \in S : P(x) \Leftrightarrow \forall x \in S : \neg P(x)$.

There is no general algorithm that can test whether any given expression is a theorem. If we wish to prove that an implication $\varphi \Rightarrow \psi$ is an theorem, we can do so with a *proof by assuming the antecedent:* to prove that the implication $\varphi \Rightarrow \psi$ is always true, we will rule out the one scenario in which it wouldn't be; specifically, we *assume* that φ is true, and then *prove* that ψ must be true too, under this assumption.

A *vacuously quantified* statement is one in which the domain of discourse is the empty set. The vacuous universal quantification $\forall x \in \varnothing : P(x)$ is a theorem; the vacuous existential quantification $\exists x \in \varnothing : P(x)$ is always false.

Quantifiers are *nested* if one quantifier is inside the scope of another quantifier. Nested quantifiers work in precisely the same way as single quantifiers, applied in sequence. A proposition involving nested quantifier like $\forall x \in S : \exists y \in T : R(x,y)$ is true if, for every choice of x, there is some choice of y (which can depend on the choice of x) for which $R(x,y)$ is true. Order of quantification matters in general; the expressions $\forall x : \exists y : R(x,y)$ and $\exists y : \forall x : R(x,y)$ are *not* logically equivalent.

Key Terms and Results

Key Terms

Propositional Logic

- proposition
- truth value
- atomic and compound propositions
- logical connectives:
  - negation ($\neg$)
  - conjunction ($\wedge$)
  - disjunction ($\vee$)
  - implication ($\Rightarrow$)
  - exclusive or ($\oplus$)
  - if and only if ($\Leftrightarrow$)
- truth assignments and truth tables
- tautology
- satisfiability / unsatisfiability
- logical equivalence
- antecedent and consequent
- converse, contrapositive, and inverse
- conjunctive normal form (CNF)
- disjunctive normal form (DNF)

Predicate Logic

- predicate
- quantifiers:
  - universal quantifier ($\forall$)
  - existential quantifier ($\exists$)
- free and bound variables
- fully quantified expression
- theorems of predicate logic
- logical equivalence in predicate logic
- proof by assuming the antecedent
- vacuous quantification
- nested quantifiers

Key Results

Propositional Logic

1. We can build a truth table for any proposition by repeatedly applying the definitions of each of the logical connectives, as shown in Figure 3.4.
2. Two propositions φ and ψ are logically equivalent if and only if $\varphi \Leftrightarrow \psi$ is a tautology.
3. An implication $p \Rightarrow q$ is logically equivalent to its contrapositive $\neg q \Rightarrow \neg p$, but not to its converse $q \Rightarrow p$.
4. There are many important propositional tautologies and logical equivalences, some of which are shown in Figures 3.10 and 3.12.
5. We can show that propositions are logically equivalent by showing that every row of their truth tables are the same.
6. Every proposition is logically equivalent to one that is in disjunctive normal form (DNF) and to one that is in conjunctive normal form (CNF).

Predicate Logic

1. We can build a proposition from a predicate $P(x)$ by plugging in a particular value for x, or by quantifying over x as in $\forall x : P(x)$ or $\exists x : P(x)$.
2. Unlike with propositional logic, there is no algorithm that is guaranteed to determine whether a given fully quantified predicate-logic expression is a theorem.
3. There are many important predicate-logic theorems, some of which are shown in Figure 3.23.
4. The statements $\neg \forall x : P(x)$ and $\exists x : \neg P(x)$ are logically equivalent. So are $\neg \exists x : P(x)$ and $\forall x : \neg P(x)$.
5. We can think of nested quantifiers as a sequence of single quantifiers, or as "games with a demon."