Perspectives on Computer Hardware and Software: Logic in Action

Rex Page Ruben Gamboa

September 17, 2010

Contents

Part I

Introduction to Computers and Logic

Chapter 1

Computer Systems: Complex Behavior from Simple Principles

Computer systems, both hardware and software, are some of the most complicated artifacts that humans have ever created. But at their very core, computer systems are straightforward applications of the same logical principles that philosophers have been developing for more than two thousand years, and this book will show you how.

The hardware component of a computer system is made up of the visible pieces of equipment that you are familiar with, such as monitors, keyboards, printers, web cameras, and USB storage devices. It also includes the components inside the computer, such as chips, cables, hard drives, DVD drives, and boards. The properties of hardware devices are largely fixed when the system is constructed. For example, a hard drive can store 2 gigabytes, or a cable may be able to transmit 25 different signals simultaneously—but no more.

The hardware that makes up a computer system is not much different than the electronics inside a television set or DVD player. But computer hardware can do something that other consumer electronic devices cannot; it can respond to information encoded in a special way, what is referred to as the software component of the system. We usually call the software a "program."

You may have heard about different computer chips and their instruction sets, such as the Intel chips that power the laptop we are using to write this book. Software for these chips consists of a list of instructions, and that is the way that most general-purpose computer systems in use today represent software. But there is nothing magical about this way of thinking about software; in fact, we believe that thinking about software in this way obscures its important features.

For example, during the 1980s and 1990s, special computers were created that used mathematical functions as the basis for the software. And other computer systems have completely different representations for software. The Internet service Second Life, for example, can be programmed using Scratch for Second Life (S4SL), and programs in S4SL look like drawings, not instructions. An even more graphical view of software is provided by LabView, which is used by scientists and engineers all over the world to control laboratory equipment. If you are at a university, there is a good chance that LabView is being used at one of the laboratories near you! And software in LabView looks like a large engineering drawing.

In this book, we present software as a collection of mathematical equations. This view is entirely different from, yet consistent with and equivalent to, the view of software as a list of instructions or a specialized drawings. The most important advantage of equations is that they make computer software accessible to anyone who understands high school algebra.

It is the software that gives a computer system much of its power and flexibility. An iPhone, for example, has a screen that can display 441,600 different pixels arranged in a 960x460 grid, but it is the software that determines whether the iPhone displays an album cover or a weather update. Software makes the hardware more useful by extending its range of behavior. For instance, the an iPhone audio device may be able to produce only a single tone at a time. But the software can instruct it to play a sequence of tones, one after another, that sound like Beethoven's Fifth Symphony.

Think of the hardware as the parts in a computer that you can see, and the software as information that tells the computer what to do. But the distinction between hardware and software is not as clear cut as this suggests. Many hardware components actually encode software directly and control other pieces of the system. In fact, hardware today is designed and built using many techniques first developed to build large software projects. And a major theme of this book is that both hardware and software are realizations of formal logic.

The distinction between hardware and software is well and good, but it leaves many questions to the imagination, such as:

Logicians and mathematicians have been studying models of computation since before computers were invented. This was done, in part, to answer a deep mathematical question: What parts of mathematics can, in principle, be fully automated? In particular, is it possible to build a machine that can discover all mathematical truths?

As a result, many different models of computation were developed, including Turing machines, Lambda calculus, partial recursive functions, unrestricted grammars, Post production rules, random-access machines, and many others. Historically, computer science theory has treated Turing machines as the canonical foundation for computation, the random-access model more accurately describes modern computers. The equational model of computation that we use in this book falls into the to the Lambda calculus bailiwick.

What is truly remarkable is that all of these different models of computation are equivalent. That is, any computation that can be described in one of the models can also be described in any of the other models. An extension of this observation conjectures that all realizable models of computation are equivalent. This conjecture is known as the Church-Turing Thesis, and it lies at the heart of computer science theory.

Another remarkable fact is that some problems cannot be solved by a program written in any of these computational models. Alan Turing, who many consider to be the first theoretical computer scientist, was the first to discover such uncomputable problems, shortly after the logician Kurt Gödel showed that no formal system of logic could prove all mathematical truths. Turing's and Gödel's discoveries of incomputability and incompleteness show that it is not possible to build a machine that can discover all mathematical truths, not even in principle.

Aside 1: Models of Computation

1. How can software affect hardware? Example: Instruct an audio device to emit a sound.

2. How can software detect the hardware's status? Example: Determine whether a switch is pressed or not.

3. What is the range of instructions that software should be able give hardware? Examples: Add two numbers. Select between two formulas, depending on the results of other computations. Replace one formula by another.

We'll start with the question about the range of instructions. Each model of computation (see Aside 1) provides an answer to this question. There are as many answers as there are models of computation, and logicians have been very creative when it comes to constructing such models! Luckily, these models are completely equivalent to each other, so we can choose the answer that is most convenient to us. And the most convenient answer is that a program consists of basic mathematical primitives, such as the the basic operations of arithmetic (addition, multiplication, etc) and the ability to define new operations based on previously defined operations.

Once we take basic arithmetic functions (that is "operation") and the ability to define new ones as the basic model of computation, we can talk about what it is possible for software to do. Software can affect hardware by the values that a function delivers. For example, a program—that is, a function—can tell an iPhone what to display in its screen by delivering a 960x460 matrix of pixels. Each entry in the matrix can be a number that represents a color, for example 16,711,680 for red or 65,280 for green. Similarly, the hardware can inform the software of the status of a component by triggering a function defined in the software and supplying the status in the parameters of the function. For example, the parameters of a particular function could be the coordinates of the pixel selected by touching the screen. Other gestures, such as tapping or scrolling, would trigger different operations.

Let's consider a simple example. We will build a simple computer device that plays rock-paper-scissors. The machine has three buttons, allowing the user to select rock, paper, or scissors. The device also has a simple display unit. After the user makes a selection, the display unit shows the computer's choice and lets the user know who won that round. Our program will make the selection and determine the winner. To keep things fair, we will write

We will use the terms "operation" and "function" interchangeably. Some models of computation use these terms to mean different things, but in the model we will use makes it convenient to think of them as the same thing. So, when we say "function" we mean "operation" and vice versa. And what we're talking about when we use either term is a transformation that delivers results when supplied with input. We refer to the results delivered by a function (or operation) as its "value", and we refer to the input supplied to the function as the parameters of the function. Sometimes we use the term "argument" instead of "parameter", but we mean the same thing in either case. When talking about the input to an "operation", we often use the term "operand", but we could also use either of the other terms for input, "parameter" or "argument", since they all mean the same thing in our model.
To summarize, a function (aka, operation) is supplied with parameters (aka, arguments, operands) and delivers a value. A formula like $f(x, y)$ denotes the value delivered by the function f when supplied with parameters x and y. For example, if f were the arithmetic operation of addition, then $f(x, y)$ would stand for the value $x + y$, $f(2, 2)$ would denote 4, and $f(3, 7)$ would stand for 10.

Aside 2: Operations, Functions, Operands, Parameters, and Arguments

this program as two separate functions, so the machine cannot "see" the human player's choice.

The first function, called `emily`[1], makes the computer's choice. This function will deliver either "rock" or "paper" or "scissors" as its value. We could use 0, 1, and 2 as shorthand for these values, but computers are versatile enough with any type of information, not just numbers, so we are going to stick with the longer names, to make it easier for us to keep track of what things mean.

But what about the input to the function `emily`? The value of a mathematical function is completely determined by its inputs. That's what it means to be a function! So if `emily` has no inputs, it must always return the same value, and that would be a very boring game. We've already decided that it would be unfair for `emily` to see the player's choice, but what about the last round? It is fair for the machine to makes its selection by

[1]This function is named after a child of one of the authors. The "real" Emily plays rock-paper-scissors game just like the program developed in this chapter.

considering the previous round of the game, so the input can be the user's *previous* choice—or a special token, like "N/A" for the game's first round. The function `emily` can now be described as follows:

$$emily(u) = \begin{cases} \text{“rock”} & \text{if } u = \text{“scissors”} \\ \text{“paper”} & \text{if } u = \text{“rock”} \\ \text{“scissors”} & \text{otherwise} \end{cases}$$

The second function, which can be called `score`, decides who was the winner of the round. Its input corresponds to the choices made by the computer and the user, respectively. Its output consists of a pair or values. The first value determines the winner, and the second "remembers" the user's choice.

$$score(c,u) = \begin{cases} (\text{“none”}, u) & \text{if } c = u \\ (\text{“computer”}, u) & \text{if } (c,u) = (\text{“rock”}, \text{“scissors”}) \\ (\text{“user”}, u) & \text{if } (c,u) = (\text{“rock”}, \text{“paper”}) \\ (\text{“computer”}, u) & \text{if } (c,u) = (\text{“paper”}, \text{“rock”}) \\ (\text{“user”}, u) & \text{if } (c,u) = (\text{“paper”}, \text{“scissors”}) \\ (\text{“computer”}, u) & \text{if } (c,u) = (\text{“scissors”}, \text{“paper”}) \\ (\text{“user”}, u) & \text{if } (c,u) = (\text{“scissors”}, \text{“rock”}) \end{cases}$$

What is the point of the second value of `score`? As you can see, it is always equal to u, i.e., the user's selection. The intent is that this second returned value will be passed as the input to the next call of `emily`. This is how the program can remember the user's previous choice.

We could certainly have dropped this second return value, and simply stipulated that the hardware is responsible for remembering the user's last selection. However, we chose to write the program in this way for two reasons. First, it gives us more flexibility. The hardware is simply responsible for storing the value returned by `score` and sending it to `emily` in the next round. We can make the program a more sophisticated player of rock-paper-scissors by changing the software, possibly changing the value that is passed from `score` to `emily` in the process. For example, a more sophisticated program may want to keep track of the number of times that the user has selected each of the choices. Since it's the software that decides what to remember from each round, this choice can be changed very easily. That is the flexibility and power of software.

The second reason for writing the program in this way is that it gives us an opportunity to make an important point about our computational model. Since programs in our model consist of collections of equations defining mathematical functions, they cannot "remember" anything. This will come as a surprise to programmers who are used to other computational models (such as C++ or Java). Those models to store values in "variables" and use those variables later in the program. We cannot do that in our purely functional model. If we did, we would lose the ability to understand our programs in terms of classical logic and would have to move into and unfamiliar and much more complicated domain.

Fortunately, our model allows the *hardware* to keep track of certain values, in much the same way that the hardware can tell which button has been pressed. Our model of computation simply allows the hardware to have some "hidden" buttons that can store information and later pass it along to a function in the form of a parameter.

This model is simple, but has as much power to specify computations as any other model. It is reasonable to believe, based on what they can do, that computers must be much more complicated than that. But in fact, that's all there is to. Power from simplicity, a bargain if there ever was one.

Consider *Deep Blue*, the computer that beat world champion Gary Kasparov at chess on May 11, 1997. This program can be written as a function whose input is an 8x8 matrix of numbers that represent the position of the pieces on the board after Kasparov's last move. The function's output is either the position of the board after its move, or a special white-flag token used to "resign" the game.

In principle, a chess-playing function can be written as follows. Given the input board, determine all possible legal moves. If there are no legal moves, resign. If there is a legal move that results in checkmate, do that move. Otherwise, for each legal move, consider each of the possible moves by the opponent. Each one of those results in a new board, which can then be examined by our chess-playing function!

It may seem surprising that a function can be defined this way, but circular definitions of functions are common in mathematics. We usually refer to them as "inductive", but "circular" is just as good. The trick is that the circularity, that is the place in the definition that refers to the function being defined, represents a reduced level of computation. Some parts of the definition will not be circular, and any circular reference will involve parameters that are closer to a non-circular portion of the definition than the parameters supplied to that portion of the definition.

Functions that have circular (that is, inductive) definitions are sometimes called “recursive” functions. We try to avoid that term because it is often associated with specialized ways to carry out the computation that the function describes, but we may slip up from time to time and use the term “recursive”.

For example, the “function” that adds the first five reciprocals of the natural numbers can be written as

$$reciprocals\_5 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

This is obviously not an inductive definition. For that matter, this “function” is really a constant, not a function in the common sense of the word. But what about a function that computes the sum of the first N reciprocals? That is where we need to use an inductive definition. The key idea is that this sum is simply $\frac{1}{N}$ plus the sum of the first $N-1$ reciprocals. More formally, it is defined by the following equation

$$reciprocals(n) = \begin{cases} 0 & \text{if } n = 0 \\ reciprocals(n-1) + \frac{1}{n} & \text{otherwise} \end{cases}$$

As you can see, the definition of reciprocals is circular, but the parameter in the circular part is closer to zero than the parameter that arrived in the first place. And the definition is not circular when the parameter is zero. So, you can use the definition to compute the value of $reciprocals(5)$ by computing $reciprocals(4)$, which is computed by first computing $43ciprocals(3)$, and so on down the line. Eventually, you come to $reciprocals(0)$, for which the definition supplies the value one in non-circular fashion. Then, you can work your way back up the line and arrive that conclusion that $reciprocals(5) = \frac{137}{60} \approx 2.3$.

Surely it is surprising that all properties of a function can be derived from a definition that covers only a few special cases, and with circularity in the mix to boot. But that is the way it is. Every computable function has a definition of this form, and that is the way we will define functions throughout this book.

To get back to the chess-playing function, the problem with our naive approach is that there are too many moves to consider. While the function can, *in principle*, select the best move to play next, *in practice*, the computation would take too much time. The sun would be long dead before the computer could decide on its first move. But in fact, *Deep Blue* did play like this, only it used a massively parallel computer so that it could consider many moves at the same time, and in most cases it considered only six to eight moves in the future. In practice, *Deep Blue* is a complicated function, with a large definition. But in principle, it is just a mathematical function, just like the programs we will study in this book.

Ex. 1 — What happens if you try to compute $reciprocals(-1)$? How about $reciprocals(\frac{1}{2})$?

Ex. 2 — Define a function called $factors(k, n)$ that returns true if and only if k is a factor of n. You may assume that the function $mod(k, n)$, which returns the remainder of k/n, has already been written.

Ex. 3 — Define a function called $largest\_factor(n)$ that returns the largest factor of n, other than n itself. For example, $largest\_factor(30) = 15$ and $largest\_factor(15) = 5$. *Hint:* You may find it necessary to start with the function $largest\_factor\_upto(n, k)$ that finds the largest factor of n, but no larger than k.

Ex. 4 — Modify the definition of the function $reciprocals(n)$ so that it adds the reciprocals only of the primes less than or equal to n. You can use the function $largest\_factor(n)$ in your solution. Impress your friends with useless trivia: While the original function $reciprocals(n)$ grows without bound as n becomes large, the modified version that only adds primes is bounded.

Chapter 2

Boolean Formulas and Equations

Symbolic logic, like other parts of mathematics, starts from a small collection of axioms and employs rules of inference to find additional propositions consistent with those axioms. This chapter will define a grammar of logic formulas, specify a few equations stating that certain formulas carry the same meaning as others, and derive new equations using substitution of equals for equals as the rule of inference.

You will probably find this familiar from your experience with numeric algebra, but we will attend carefully to details, and this formality may extend beyond what you are accustomed to. What it buys us is mechanization. That is, our logic formulas and our reasoning about them will amount to mechanized computation, and this will make it possible to computers to check that our reasoning follows all the rules, without exception. This gives us a higher level of confidence in our conclusions than would otherwise be possible.

We will be doing all of this in the domain of symbolic logic, which includes operations like "logical or" and "logical negation", rather than arithmetic operations, such as addition and multiplication. Logical operations put our formulas in the domain of Boolean algebra, rather than numeric algebra, but our rule of inference, which is substitution of equals for equals, applies equally well in both the Boolean and the numeric domains. To illustrate the level of formality that we are shooting for, let's see how it works with a problem in numeric algebra.

You are surely familiar with the equation $(-1) \times (-1) = 1$, but you may

Hold on to your seat! This section illustrates an essential method used throughout the book. It introduces the notion of "formality" in mathematical argumentation. This is "formality" in the sense of being "based on formulas." The formulas involved have a prescribed grammar similar to the one for numeric formulas that you have used for many years. This grammar is how you know that the formula "$x + 3 \times (y + z)$" is well formed and the non-formula "$x + 3 \times (y+) \times z$" is not well formed.
Things may seem overly simple at the very beginning. Then, suddenly, you may find yourself thrashing around in deep water. Take a deep breath, and slowly work through the material. It provides a basis for everything to follow. It calls for careful study and frequent review when things start to go off track. Fold the corner of this page down. You will probably need to come back to it later.

Aside 3: Hold on to Your Seat

$x + 0 = x$	{+ identity}
$(-x) + x = 0$	{+ complement}
$x \times 1 = x$	{× identity}
$x \times 0 = 0$	{× null}
$x + y = y + x$	{+ commutative}
$x + (y + z) = (x + y) + z$	{+ associative}
$x \times (y + z) = (x \times y) + (x \times z)$	{distributive law}

Figure 2.1: Equations of Numeric Algebra

not know that it is a consequence of some basic facts about arithmetic calculation. That is, the fact that multiplying two negative numbers produces a positive one is not independent of other facts about numbers. Instead, it is an inference one can draw from an acceptance of other familiar equations.

The equations in Figure 2.1 (page 14) express some standard rules of numeric computation. In those equations, the letters stand in place of numbers, or in place of other formulas expressed in terms of common numeric operations (addition, multiplication, etc). We refer to letters used in this way as "variables" even though, within a particular equation, they stand for a fixed number or for a particular formula. Their values, though unspecific, do not vary.

If we accept the equations of Figure 2.1 (pare 14) as axioms, we can apply one of them to transform the formula $(-1) \times (-1)$ to a new formula that stands for the same number. Then, we can apply another axiom to transform that formula to a new one, and so on. We apply the axiomatic equations in such a way that, at some point, we arrive at the formula "1". At every stage, we know that the new formula stands for the same value as the old one, so in the end we know that $(-1) \times (-1) = 1$.

Figure 2.2 (page 16) displays this sort of equation-by-equation derivation of the formula "1" from the formula "$(-1) \times (-1)$". To understand Figure 2.2, you must remember that each variable can denote any grammatically correct formula. For example, in the {+ identity} equation, $x+0 = x$, the variable x could stand for a number, such as 3, or it could stand for a more complicated formula, such as $(1+3)$. It could even stand for a formula with variables in it, such as $(a + (b \times c))$ or $(((-1) \times (x + 3)) + (x + y))$.

Another crucial point is that each step cites exactly one equation from Figure 2.1 (page 14) to justify the transformation from the formula in the previous step. We are so accustomed to calculating numeric formulas that we often combine many basic steps into one. When we reason formally, we must not do this. We must justify each step citing an equation from a list of known equations. In our proof of $(-1) \times (-1) = 1$, we will justify steps by citing equations from Figure 2.1 and from no other source. We will not skip steps. Think of that as you go through the proof, line by line.

The first step in the proof (Figure 2.2, page 16) uses a version of the {+ identity} equation in which the variable x stands for the formula $((-1) \times (-1))$. The second step reads the {+ complement} equation backwards (equations go both ways), but in a form where the variable x stands for the number 1. And so on. The transformations, step by step, finally confirm that the two formulas $(-1) \times (-1)$ and 1 stand for the same number.

Pay particular attention to the last three lines. Most people tend to jump from the formula $0 + 1$ to the formula 1 in one step. This requires knowing the equation $0+1 = 1$. However, that equation is not among those listed in Figure 2.1 (page 14). To do the proof without citing any equations other than those in Figure 2.1, we need two steps, and those are the last two steps in the proof.

One of the things we hope you will glean from this derivation is that the equation $(-1)\times(-1) = 1$ does not depend on vague, philosophical assertions like "two negatives make a positive." Instead, the equation $(-1)\times(-1) = 1$ is a consequence of some basic arithmetic equations. If you accept the basic equations and the idea of substituting equals for equals, you must, as a

$$
\begin{array}{lll}
 & (-1) \times (-1) & \\
= & ((-1) \times (-1)) + 0 & \{+ \text{ identity}\} \\
= & ((-1) \times (-1)) + ((-1) + 1) & \{+ \text{ complement}\} \\
= & (((-1) \times (-1)) + (-1)) + 1 & \{+ \text{ associative}\} \\
= & (((-1) \times (-1)) + ((-1) \times 1)) + 1 & \{\times \text{ identity}\} \\
= & ((-1) \times ((-1) + 1)) + 1 & \{\text{distributive law}\} \\
= & ((-1) \times 0) + 1 & \{+ \text{ complement}\} \\
= & 0 + 1 & \{\times \text{ null}\} \\
= & 1 + 0 & \{+ \text{ commutative}\} \\
= & 1 & \{+ \text{ identity}\}
\end{array}
$$

Figure 2.2: Why $(-1) \times (-1) = 1$

rational consequence, accept the equation $(-1) \times (-1) = 1$.

Using this same kind of reasoning, we will derive equations between formulas in logic from a few, simple equations postulated as axioms. We will also learn that digital circuits are physical representations of logic formulas, and we will be able to parlay this basic idea to derive behavioral properties of computer components.

Likewise, because a computer program is, literally, a logic formula, we will be able to derive properties of those programs directly from the programs, themselves. This makes it possible for us to be entirely certain about some of the behavioral characteristics of software, and of computer hardware, too, since a hardware component is also a logic formula. Our certainty stems from the mechanistic formalism that we insist on from the beginning, which can be checked to the last detail with automated computation.

2.1 Boolean equations

Let's start with the Boolean equations in Figure 2.3 (page 17). If they look strange to you, try not to worry. Everything new seems strange for a while. Try to view them as ordinary, algebraic equations, but with a different collection of operators. A formula in numeric algebra contains operations like addition (+) and multiplication (×). Boolean formulas employ logic operations: logical-and (∧), logical-or (∨), logical-negation (¬), and implication (→).

$x \vee False = x$	$\{\vee$ identity$\}$
$x \vee True = True$	$\{\vee$ null$\}$
$x \vee y = y \vee x$	$\{\vee$ commutative$\}$
$x \vee (y \vee z) = (x \vee y) \vee z$	$\{\vee$ associative$\}$
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$\{\vee$ distributive$\}$
$x \rightarrow y = (\neg x) \vee y$	$\{$implication$\}$
$\neg(x \vee y) = (\neg x) \wedge (\neg y)$	$\{\vee$ DeMorgan$\}$
$x \vee x = x$	$\{\vee$ idempotent$\}$
$x \rightarrow x = True$	$\{$self-implication$\}$
$\neg(\neg x) = x$	$\{$double negation$\}$

Figure 2.3: Basic Boolean equations (axioms)

Furthermore, Boolean formulas stand for logic values ($True$, $False$), rather than for numbers (. . . -1, 0, 1, 2 . . .). So, in Boolean algebra there are just two basic values ($True$, $False$), not an infinite collection of numbers. That does not limit the potential domain of discourse, however. By aggregating the basic $True/False$ elements in sequences, we will find that we can deal with numbers and with the full range of things that numbers can represent.

When we derive new equations from equations we already know, we refer to the derived equations as "theorems" to distinguish them from axioms. We call the derivation a "proof" of the theorem.

The first equation in the following theorem $\{\vee$ truth table$\}$ is a special case of the $\{\vee$ identity$\}$ axiom in Figure 2.3 (see page 17), and the proof of that equation consists of that observation. The proof of the second equation is equally short, but cites a different equation in the axioms of Figure 2.3 to justify the transformation from $False \vee True$ to $True$. For practice, try to prove the other two equations in the $\{\vee$ truth table$\}$ theorem by citing axioms in a similar way.

Theorem 1 ($\{\vee$ truth table$\}$)**.**

- $False \vee False = False$
- $False \vee True = True$
- $True \vee False = True$

- $True \vee True = True$

Proof.

$$\begin{array}{ll} & False \vee False \\ = & False \end{array}$$
{∨ identity} …taking x in the axiom to stand for False

$$\begin{array}{ll} & False \vee True \\ = & True \end{array}$$
{∨ null} …taking x in the axiom to stand for False

…for practice, prove the other two equations yourself …

Q.E.D.

We are serious about that. Did you prove the other two equations? No? Well …go back and do it, then. Without participation, there is no learning.

Finished now? Good for you. You cited the {∨ identity} axiom in your proof of the third equation in the theorem and the {∨ null} axiom in your proof of the fourth equation, right? We knew you could do it.

A "truth table" for a formula is a list of the values the formula represents, with one entry in the list for every possible combination of the values of its variables. If there is only one variable in the formula, there will be two entries in its truth table, one for the case when the variable has the value $True$ and one for the case when the variable has the value $False$. If there are two variables in the formula, there will be four entries in the truth table because for each choice of the first variable, there are choices of the other. Three variables lead to eight entries. The number of entries doubles with each new variable.
A truth table for a logical operator is the truth table for the formula that has variables in place of the operands. For example, the truth table for the logical-or operator (∨) is the truth table for the formula $x \vee y$. That formula has two variables, so the truth table has four entries.

Aside 4: Truth Tables

Derivations are usually more complicated, of course. For example, the following {∨ complement} theorem is not a special case of any of the axioms, but has a two-step proof, citing implication and self-implication.

Theorem 2 ({$\lor$ complement})**.** $(\neg x) \lor x = True$

Proof.

$$\begin{array}{lll} & (\neg x) \lor x & \\ = & x \to x & \{\text{implication}\} \\ = & True & \{\text{self-implication}\} \end{array}$$

Q.E.D.

The {$\lor$ complement} theorem is often referred to as the "law of the excluded middle" because it states that any logical statement, together with its negation, comprises all of the possibilities. A logical statement is either true of false. There is no middle ground.

All of the logical operators have truth tables, and we can derive the equations in those truth tables from the axioms. The following theorem provides the truth table for the negation ($(\neg x)$) operator. The proof of the first equation in the {$\neg$ truth table} theorem has a four step proof. To beef up your comprehension of the ideas, construct your own proof of the second equation in the theorem.

Theorem 3 ({$\neg$ truth table})**.**

- $\neg True = False$
- $\neg False = True$

Proof.

	$\neg True$	
$=$	$\neg(False \to False)$	{self-implication}
$=$	$\neg((\neg False) \lor False)$	{implication} (taking both x and y in the axiom to stand for $False$)
$=$	$\neg(\neg False)$	{$\lor$ identity} (taking x in the axiom to stand for $\neg False$)
$=$	$False$	{double negation} (taking x in the axiom to stand for False)

$$\begin{array}{ll} & \neg False \\ = & \ldots \text{you fill in the details here} \ldots \\ = & True \end{array}$$

Q.E.D.

An important facet of these proofs is that they are entirely syntactic. That is, they apply axioms by matching the grammar of a formula f in the proof with the formula g in an equation from the axioms. This matching associates the variables in g with certain sub-formulas in the formula f. Then, the formula h on the other side of the equation, with the same association between its variables and sub-formulas of f, becomes the new, derived formula. We know that the derived formula stands for the same value as the original formula because the axiom asserts this relationship, and we are assuming that axioms are right.

Let's do another truth-table theorem, partly to practice reasoning with equations, but also to discuss a common point of confusion about logic. The implication operator ($\rightarrow$) is a cornerstone logic in real-world problems, but many people misunderstand its meaning—that is, its truth table.

The {$\rightarrow$ truth table} theorem provides the truth table for the implication operator. An important aspect of the proof is that it cites not only axioms from Figure 2.3 (see page 17), but also equations from the {$\neg$ truth table} theorem. This is the way mathematics goes. Once we have derived a new equation from the axioms, we can cite the new equation to derive still more equations.

Theorem 4 ({$\rightarrow$ truth table})**.**

- $False \rightarrow False = True$
- $False \rightarrow True = True$
- $True \rightarrow False = False$
- $True \rightarrow True = True$

Proof.

$$\begin{array}{lll}
 & False \rightarrow False & \\
= & (\neg False) \vee False & \{\text{implication}\} \ldots \text{taking both } x \text{ and } y \text{ in the} \\
 & & \text{axiom to stand for } False \\
= & \neg False & \{\vee \text{ identity}\} \\
= & True & \{\neg \text{ truth table}\}
\end{array}$$

...for practice, prove the other equations yourself...

Q.E.D.

In day to day life outside the sphere of symbolic logic, the interpretation of the logical implication $x \rightarrow y$ is that we can conclude that y is true if we know that x is true. However, the implication says nothing about y when

Another way to prove that two formulas stand for the same value is to build truth tables for both formulas. A truth table lists all possible combinations of values for the variables in a formula, and displays the value that the formula denotes for each of those combinations. (Theorem {$\vee$ truth table} provides the truth table for the logical-or operation, and theorem {$\neg$ truth table} provides the truth table for the logical-negation operation.) Two truth tables that list identical values of the corresponding formulas for all combinations of values for the variables demonstrate that the formulas always stand for the same value. This proof method works well for formulas with only a few variables. In that case, there are only a few combinations of values for the variables, and the comparison can be completed quickly.

On the other hand, if there are many variables in the formulas, things get out of hand. With two variables, as in the truth table for logical-or, there are four combinations of values (two choices for each variable, True or False, so two times two combinations in all). With three variables, there are eight (2^3) combinations, which makes the truth-table method tedious, but not infeasible. After that, it gets rapidly out of hand. Ten variables produce 1,024 (2^{10}) combinations of values. That makes it difficult for people, but no real problem for a computer. Even twenty variables (a little more than a million combinations) also can be checked quickly by computers.

However, the formula specifying a computing component, hardware or software, has hundreds of variables. Our goal is to be able to reason about computing components, and there is no hope of doing that when the formulas have hundreds of variables. The number of combinations of values for the variables in a formula with, say, 100 variables is 2^{100}, and that number is so large that no computer could finish checking for equality before the sun runs out of fuel.

So, it is definitely not feasible to know the full meaning of computing components by analyzing the truth tables of the formulas that comprise their designs. Reasoning based on grammatical form makes it feasible to deal with realistic computing components because the reasoning process can be split into parts small enough to manage, and those parts can be reintegrated, based on their grammatical relationships, to produce a full analysis. It will take some diligence to reach this goal, and we will need a few more tools, but the proof methods of this chapter will get us started in the right direction.

Aside 5: Truth Tables and Feasibility

Citing proven theorems to prove new ones is similar an idea known as "abstraction" that is employed in engineering design. At the point where we cite an old theorem to prove a new one, we could, instead, copy the proof of the old theorem into new proof. However, that would make the proof longer, harder to understand, and more likely to contain errors.

Computer programs are built from components that are, themselves, other computer programs. As the project proceeds, more and more components become available, and they are used to build more complex ones. Sometimes, a component has almost the right form to be used in a new program, but not quite. Maybe the existing component doubles a number where the new program would need to triple it. Most software engineers are tempted to make a copy the old component and change the $2 \times x$ formula to $3 \times x$ at the point where a doubling should be a tripling. *This practice the single most common cause of errors in computer software.*

What the engineer should do is to make a new component in which the 2 is replaced by a variable, say m. This is known as creating an "abstraction" of the component ("abstract" as opposed to "specific" or "concrete"). The new component can be used for both doubling and tripling, simply by specifying 2 for m in one case and 3 for m in the other. That way, if the component turns out later to have an error in it, the error can be fixed in one place instead of two (or maybe ten or a hundred places, depending on how many engineers have made copies of the original component to change the 2 to 7 or 9 or whatever.

This notion of abstraction is one of the most important methods in all of engineering design. Citing old theorems to prove new ones, instead of copying their proofs into the new proof with appropriate choices for the variables, is part of that tradition.

Aside 6: Abstraction

x is not true. In particular, it does not say that y is not true whenever x is not true. A quick look at theorem $\{\rightarrow$ truth table$\}$ shows that the formula $\mathit{False} \rightarrow y$ has the value True when y is True and also when y is False. In other words, the truth of the formula $x \rightarrow y$ in the case where the hypothesis, x, of the implication is False has provides no information about the conclusion, y.

A common mistake in everyday life is to assume that if the implication $x \rightarrow y$ is true, then the implication $(\neg x) \rightarrow (\neg y)$ is also true. Sometimes this leads to bad results, even in everyday life. In symbolic logic, it is worse than that. Such a conclusion puts an inconsistency into the mathematical system, and that renders the system useless.

Over half of the basic Boolean equations in Figure 2.3 (see page 17) have names associated with the logical-or ($\vee$) operation. One of them, the $\{\vee$ DeMorgan$\}$ equation establishes a connection between logical-or and logical-and. It converts the negation of a logical-or to the logical-and of two negations: $\neg(x \vee y) = (\neg x) \wedge (\neg y)$. We can use this connection to prove a collection of equations for logical-and that are similar to the basic ones for logical-or. An example is the null law for logical-and.

Theorem 5 ($\{\wedge$ null$\}$). $x \wedge \mathit{False} = \mathit{False}$

Proof.

$$\begin{array}{lll}
 & x \wedge \mathit{False} & \\
= & x \wedge (\neg \mathit{True}) & \{\neg \text{ truth table}\} \\
= & (\neg(\neg x)) \wedge (\neg \mathit{True}) & \{\text{double negation}\} \\
= & \neg((\neg x) \vee \mathit{True}) & \{\vee \text{ DeMorgan}\} \ldots \text{taking } x \text{ in the axiom to stand for } (\neg x) \text{ and } y \text{ in the axiom to stand for } \mathit{True} \\
= & \neg \mathit{True} & \{\vee \text{ null}\} \\
= & \mathit{False} & \{\neg \text{ truth table}\}
\end{array}$$

Q.E.D.

This regime of theorem after theorem, proof after proof, is a little tiresome, isn't it? Nevertheless, let's push through one more. Then we'll go on to other topics, and give you a few to work out on your own, later.

Some equations simplify the target formula when used in one direction, but make it more complicated when used in the other direction. For example, applying the null law for logical-or, $\{\vee$ null$\}$, from left to right simplifies a logical-or formula to True. The formula on the right in the $\{\wedge$ null$\}$ equation discards both the logical-or operation and the variable

x. When you use this equation in the other direction, you can make the formula as complicated as you like because the variable x stands for any formula you want to make up (as long as it's grammatically correct). It can have hundreds of variables and thousands operations. This may seem perverse, but if that's what it takes to complete the proof, so be it.

The null law for logical-and, {∧ null}, is also asymmetric. It goes from complicated to simple in one direction and from simple to complicated in the other. A particularly interesting and important asymmetric equation is the absorption law for logical-and. It has two variables and two operations on one side, but only one variable and no operations on the other.

Theorem 6 ({∧ absorption})**.** $(x \vee y) \wedge y = y$

Proof.

$$\begin{array}{lll} & (x \vee y) \wedge y & \\ = & (x \vee y) \wedge (y \vee False) & \{\vee \text{ identity}\} \\ = & (y \vee x) \wedge (y \vee False) & \{\vee \text{ commutative}\} \\ = & y \vee (x \wedge False) & \{\vee \text{ distrubutive}\} \\ = & y \vee False & \{\wedge \text{ null}\} \\ = & y & \{\vee \text{ identity}\} \end{array}$$

Q.E.D.

We hope the gauntlet of theorems and proofs we have run you through (or, more likely, asked you to plow through, to the point of exhaustion) helps you understand how to derive a new equation from equations you already know. The technique requires matching the formula to one side of a known equation, then replacing it by the corresponding formula on the other side.

The "matching" process is a crucial step. It involves replacing the variables in the known equation by constituents of the formula you are trying to match. This is based in the mechanics of a formal grammar. It is surprisingly easy have a lapse of concentration and make a mistake.

Fortunately, it is easy for computers to verify correct matchings and report erroneous ones. A computer system that does this is called a "mechanized logic." After you have enough practice to have a firm grasp on the process, we will begin to use a mechanized logic to make sure our reasoning is correct.

. . . put exercises here? at end of chapter? . . .

2.2 Boolean formulas

We have been doing proofs based on the grammatical elements of formulas, but instead of taking the time to put together a precise definition of that grammar, we have been relying on your experience with formulas of numeric algebra, such as $2(x + y) + 3z$. Surely it would be better to have a precise definition, so we can determine whether or not a formula is grammatically correct. We are going to define a grammar by starting with the most basic elements, and working up from there to more complicated ones.

The simplest Boolean formulas are the basic constants, $True$ and $False$, and variables. We normally use ordinary, lower-case letters, such as x and y for variables. Sometimes variables are letters with subscripts, such as x_3, y_i, or z_n. This gives us sufficient variety for all of the situations we will encounter, but we will assume there is an infinite pool of names for variables. If we run out of Roman letters, we can use Greek letters. If we run out of those, we can start making up recognizable squiggles like Dr Seuss did in some of his stories.

So, if you write $True$, or $False$, or a lower-case letter, you have composed a grammatically correct Boolean formula. This is the first rule of Boolean grammar. Formulas conforming to this rule have no substructure, so we call them "atomic" formulas.

To make more complicated formulas, we use Boolean operators. We refer to the operators that require two operands as the "binary operators" ($\wedge$, $\vee$, and $\rightarrow$). These operators lead to the second rule of Boolean grammar: If a and b are grammatically correct Boolean formulas, and $\circ$ is a binary operator (that is, $\circ$ is one of the symbols $\wedge$, $\vee$, or $\rightarrow$), then $(a \circ b)$ is also a grammatically correct Boolean formula.

For example, the first rule confirms that x and $True$ are grammatically correct Boolean formulas. Since $\wedge$ is a binary operator, $(x \wedge True)$ is a grammatically correct Boolean formula by the second rule of grammar. Furthermore, since $\rightarrow$ is a binary operator, and y is a grammatically correct Boolean formula (by the first rule), $((x \wedge True) \rightarrow y)$ must be a grammatically correct Boolean formula (by the second rule). As you can see, the first two rules of Boolean grammar provide an infinite variety of grammatically correct Boolean formulas.

The third rule of Boolean grammar shows how to incorporate the negation operator in formulas. The rule is that if a is a grammatically correct Boolean formula, then so is $(\neg a)$.

These three rules cover the full range of grammatically correct Boolean formulas, but there is a fine point to discuss about parentheses. Parentheses are important because they make it easy to define the grammar and to explain the meaning of grammatically correct formulas.

The formulas covered by the three rules are fully parenthesized, including a top level of parentheses enclosing the entire formula. Top level parentheses are often omitted in informal presentations, and we have been omitting them on a regular basis.

For example, we have been writing formulas like "$x \vee y$", even though they do not have the top-level parentheses required in the grammar of binary operations. To conform to the grammar, we would have to write the formula with its top-level parentheses: "$(x \vee y)$". Because we have often omitted top-level parentheses, requiring them probably comes as a surprise. But, allowing non-atomic formulas without top-level parentheses requires additional rules of grammar, and the extra complexity is not compensated by added value.

Here is less surprising example of incorrect grammar: $x \wedge y \vee z$. This formula is missing two levels of parentheses. Even worse, there are two options for the inner parentheses. Does $x \wedge y \vee z$ mean $((x \wedge y) \vee z)$ or $(x \wedge (y \vee z))$? There is a way to deal with formulas that omit some of the parentheses, but to avoid confusion, we are not going to allow such formulas. The same problem occurs with formulas in numeric algebra. We know that $x \times y + z$ means $((x \times y) + z)$ and not $(x \times (y + z))$ because we know the convention that gives multiplicative operations a higher precedence than additive operations. But that gets some getting used to, and since Boolean formulas may be new to you, we want to minimize the possibility of misinterpretation.

We will sometimes be informal enough to omit the top level of parentheses around the whole formula, but we will not omit any interior parentheses. We will, however, allow redundant parentheses in grammatically correct formulas. For example, we the formula $(x \vee ((x \wedge y)))$ is grammatically correct and has the same meaning as the formula $(x \vee (x \wedge y))$. The first formula has redundant parentheses, but the second one doesn't.

Allowing redundant parentheses requires a fourth rule of grammar: If a is a grammatically correct Boolean formula, then so is (a).

With the four rules of Figure 2.4 (see page 27), we can determine whether or not any given sequence of symbols is a grammatically correct Boolean formula. The definition of the grammar is circular, but in a useful way that shows how to build more complicated formulas from simpler ones.

Grammatically correct Boolean formulas

v	{atomic}
$(a \circ b)$	{bin-op}
$(\neg a)$	{negation}
(a)	{group}

Requirements on symbols

- v is a variable or $True$ or $False$
  (a variable is a letter or a letter with a subscript)
- a and b are grammatically correct Boolean formulas
- $\circ$ is a binary operator

Figure 2.4: Rules of Grammar for Boolean Formulas

To verify that a formula is grammatically correct, find the rule of grammar that matches it, then verify that each part of the formula that matches with a letter in the rule of grammar is also grammatically correct. Atomic formulas have no substructure, so they require no further verification. A line of verification terminates when it arrives at an atomic formula.

For example, consider the formula $((x \vee (\neg y)) \wedge (x \rightarrow z))$. It matches with the {bin-op} rule. The symbols in the rule match with the elements of the formula in the following way.

symbol from {bin-op} rule	*matching element in* $((x \vee (\neg y)) \wedge (x \rightarrow z))$
a	$(x \vee (\neg y))$
$\circ$	$\wedge$
b	$(x \rightarrow z)$

The only other symbols in the rule are the top-level parentheses, and these match identically with the outer parentheses in the target formula. Therefore, the target formula is grammatically correct if the formulas $(x \vee (\neg y))$ and $(x \rightarrow z)$ are grammatically correct. We use the same approach to verify the grammatical correctness of those formulas.

The first one, $(x \vee (\neg y))$, again matches with the {bin-op} rule, but this time the matchings of the elements in the target formula with symbols in the rule are as follows:

symbol from {bin-op} rule	*matching element in* $(x \vee (\neg y))$
a	x
$\circ$	$\vee$
b	$(\neg y)$

This reduces the verification of the grammatical correctness of $(x \vee (\neg y))$ to the verification of the two formulas x and $(\neg y)$. Since x matches with the {atomic} rule, it must be grammatically correct. The $(\neg y)$ element matches with the {negation} rule, with y from the formula matching a in the rule. So, $(\neg y)$ is grammatically correct if y is, and y is grammatically correct because it matches with the {atomic} rule. That completes the verification of the $(x \vee (\neg y))$ formula.

The second element of the original formula, $(x \to z)$, is easier to verify. It matches the {bin-op} rule with x corresponding to a in the rule, y corresponding to b, and $\to$ corresponding to $\circ$ in the rule. Since x and z match the {atomic} rule, they are grammatically correct. This completes the verification of the grammatical correctness of the formula $((x \vee (\neg y)) \wedge (x \to z))$.

Let's look at another example: $(x \vee (\wedge y)$. This formula matches with the {bin-op} rule, with x corresponding to a in the rule, $\vee$ corresponding to $\circ$, and $(\wedge y)$ corresponding to b. So, the formula is grammatically correct if x and $(\wedge y)$ are. However, there is no rule that matches $(\wedge y)$. The only place the symbol $\wedge$ could match a rule in the table is in the {bin-op} rule. In the {bin-op} rule, there must be a grammatically correct formula between the opening parenthesis and the operator. Since there is no such element present between the opening parenthesis and the $\wedge$ operator in the target formula, it cannot be grammatically correct.

That covers the grammar of Boolean formulas. What about meaning? Every grammatically correct Boolean formula denotes, in the end, either the constant $True$ or the constant $False$. To find out which, you only need to know which value ($True$ or $False$) each of the variables in the formula stands for. Each of the binary operators, given specific operands ($True$ or $False$), delivers a specific result ($True$ or $False$). We worked out what the delivered values would be for some of the operators when we proved truth-table theorems for them (see page 17).

$(a \vee \mathit{False}) = a$	$\{\vee$ identity$\}$
$(x \vee \mathit{True}) = \mathit{True}$	$\{\vee$ null$\}$
$(a \vee b) = (b \vee a)$	$\{\vee$ commutative$\}$
$(a \vee (b \vee c)) = ((a \vee b) \vee c)$	$\{\vee$ associative$\}$
$(a \vee (b \wedge c)) = ((a \vee b) \wedge (a \vee c))$	$\{\vee$ distributive$\}$
$(a \to b) = ((\neg a) \vee b)$	{implication}
$(\neg(a \vee b)) = ((\neg a) \wedge (\neg b))$	$\{\vee$ DeMorgan$\}$
$(a \vee a) = a$	$\{\vee$ idempotent$\}$
$(a \to a) = \mathit{True}$	{self-implication}
$(\neg(\neg a)) = a$	{double negation}
$((a)) = (a)$	{redundant grouping}
$(v) = v$	{atomic release}

Requirements on symbols

- a, b, and c are grammatically correct Boolean formulas
- v is a variable or True or False
  (a variable is a letter or a letter with a subscript)

Figure 2.5: Axioms of Boolean Algebra

In the process of deriving the truth tables, we used the Boolean equations of Figure 2.3 (see page 17). We can use this same method to derive the meaning (True or False) of any grammatically correct formula that contains no variables. However, to deal with parentheses in a completely mechanized way, we need to add two equations to those of Figure 2.3. Figure 2.5 (see page 29) provides all of the information needed to determine the $\mathit{True}/\mathit{False}$ value of any grammatically correct formula that contains no variables. In fact it has even more general applicability. It provides all the information needed to verify not only whether a given formula has the same meaning as the formula True or the formula False, but also to verify whether or not any two given, grammatically correct, formulas have the same meaning.

Ex. 5 — Use the rules of grammar for Boolean formulas (Figure 2.4, see page 27) to determine which of the following formulas are grammatically correct.

$((x \wedge y) \vee y)$
$((x \to y) \wedge (x \to (\neg y)))$
$((False \to (\neg y))\neg(x \vee True))$

Ex. 6 — Derive the truth tables (see page 18) of the grammatically correct formulas from the previous exercise.

Ex. 7 — Use the axioms of Boolean algebra (Figure 2.5, see page 29) to prove that the following equations are valid. After you have proved the validity of an equation, you may cite it in subsequent proofs of other equations.

$(a \to False) = (\neg a)$	$\{\neg$ as $\to\}$
$(\neg(a \wedge b)) = ((\neg a) \vee (\neg b))$	$\{\wedge$ DeMorgan$\}$
$(a \vee (\neg a)) = True$	$\{\vee$ complement$\}$
$(a \wedge (\neg a)) = False$	$\{\wedge$ complement$\}$
$(\neg True) = False$	$\{\neg True\}$
$(\neg False) = True$	$\{\neg False\}$
$(True \to a) = a$	$\{\to$ identity$\}$
$(a \wedge True) = a$	$\{\wedge$ identity$\}$
$(a \wedge b) = (b \wedge a)$	$\{\wedge$ commutative$\}$
$(a \wedge (b \wedge c)) = ((a \wedge b) \wedge c)$	$\{\wedge$ associative$\}$
$(a \wedge (b \vee c)) = ((a \wedge b) \vee (a \wedge c))$	$\{\wedge$ distributive$\}$
$(a \wedge a) = a$	$\{\wedge$ idempotent$\}$
$(a \to b) = ((\neg b) \to (\neg a))$	$\{$contrapositive$\}$
$(a \to (b \to c)) = ((a \wedge b) \to c))$	$\{$Currying$\}$
$((a \wedge b) \vee b) = b$	$\{\vee$ absorption$\}$
$((a \to b) \wedge (a \to (\neg b)))$	$\{$absurdity$\}$

2.3 Digital Circuits

Logic formulas provide a mathematical notation for concepts in symbolic logic. These same concepts can be materialized as electronic devices. The basic operators of logic represented in the form of electronic devices are called "logic gates". There are logic gates for the logical-and, the logical-or, negation, and several other operators that we have not yet discussed.

A logic gate takes input signals that correspond to the operands of logical operators and delivers output signals that correspond to the values

One of the operators we have discussed at length, implication ($\rightarrow$), is not among the operators normally represented in the form of logic gates. This does not restrict the kinds of operations that can be performed by digital logic because, as we know from the {implication} axiom of Boolean algebra (Figure 2.5, page 29), the formula $(a \rightarrow b)$ has the same meaning as the formula $((\neg a) \vee b)$. So, anything we can do with the implication operator, we can also do with by combining negation and logical-or in a particular way. Therefore, since we have logic gates for logical-or and negation, there is no loss in the range of behavior of logic gates, compared to that of the basic logical operators.

The lack of a conventional logic gate is ironic for at least three reasons. One is that George Boole himself, the inventor of Boolean algebra, called implication the queen of logical operators. Another is that the implication operator is one of only a few binary operators that are sufficient for representing any formula in logic. That is, given any grammatically correct logic formula, there is a formula using only implication operators (no logical-and, logical-or, or negation operators) that produces the same result as the original formula, given the same values for the variables in the formula.

But, the most dramatic reason for the irony in having implication turn up missing from the conventional collection of basic logic gates is that recent discoveries make it possible that the implication operator will in the future be the only gate used in large-scale digital circuits. This is because implication can be materialized as an electronic component in a form that allows three-dimensional stacking of circuits in a way that has never before been feasible. This makes it possible to fabicate digital circuits with vastly greater computational capacity than present-day circuits. If this intrigues you, view the online video of R. Stanley Williams on "memristor chips" (`http://www.youtube.com/watch?v=bKGhvKyjgLY`).

Aside 7: No Gate for Implication

delivered by logical operations. A logic gate with two inputs is a physical representation of a binary operator. The negation operator is represented by a logic gate with one input.

There are only two kinds of operands for logical operators: $True$ and $False$. And, they can deliver only those two kinds of values. Similarly, a logic gate can interpret only two kinds of input signals and deliver only two kinds of output signals (the same two kinds, of course). Those signals could be called $True$ and $False$, but conventionally they are written as 1 (for $True$) and 0 (for $False$). Of course, logic gates are electronic devices, so 0 and 1 are just labels for the signals. Any two different symbols could be used to represent them in writing. The choice of 1 and 0 is more-or-less arbitrary.

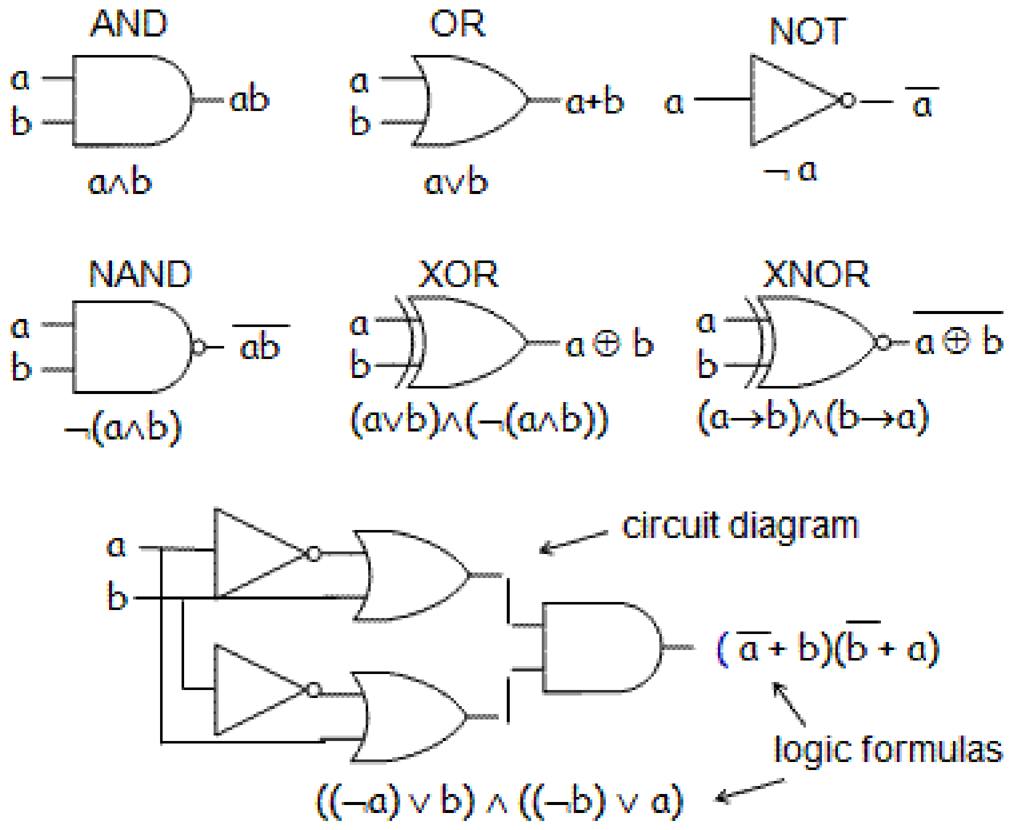
There are many ways to handle the electronics. We are going to leave the physics to the electrical engineers. If you want to, you can imagine representing the signal 1 as a small electrical current on a wire and the signal 0 as the lack of electrical current. That is one way to do it. But, we are going to focus on the logic and trust that the electronic hardware can faithfully represent the two kinds of signals we need.

Engineers who design circuits represent their designs as wiring diagrams in which "wires" (represented by lines) carry signals between gates. They use distinctive shapes to represent the different logic gates in circuit diagrams, and they use an algebraic notation similar to, but not the same as, the notation logicians normally use when they want to represent their circuits as formulas.

In the algebraic notation commonly used by circuit designers, the logical-and is represented by the juxtaposition of the names being used for the signals (in the same way that juxtaposition of variables is used to denote multiplication in numeric algebra formulas). Logical-or is represented by a plus sign (+), and negation is represented by a bar over the variable being negated (for example, $\bar{a}$).

Figure 2.6 (see page 33) summarizes the relationships between the symbols for logic gates used in circuit diagrams (that is, wiring diagrams that route signals between a collection of logic gates), the algebraic notation used by circuit designers, and the logic formulas we have use up to now.

The important fact to remember is this: All three notations represent the same concepts in logic. Circuit diagrams, logic formulas, and the algebraic notation used by circuit designers are three different notations for exactly the same mathematical objects. In this sense, digital circuits, and,



TODO: I did not have Visio with me. Improvised with PowerPoint. Figure will need to be redrawn.

Figure 2.6: Digital Circuits = Logic Formulas

therefore, computers, are materializations of logic formulas. Computers truly are "logic in action".

The logical operators that we have been using are logical-and, logical-or, implication, and negation. These are sufficient to write formulas that have any possible input/output relationship between the variables in the formula and the value it delivers.

The {implication} axiom (Figure 2.5, see page 29) expresses implication in terms of logical-and, logical-or, and negation, which means we lose no expressive power by discarding implication from the set of logical operations.

Surprisingly, the reverse is also true. That is, for any given input/output relationship that can be expressed in a formula using logical-and, logical-or, and negation, there is a logic formula using implication as the only operator that has the same input/output relationship. The $\{\neg \text{ as } \rightarrow\}$ equation (see page 30) provides at start in this direction by showing how to express negation in terms of the implication operator. Logical-and and logical-or are a little trickier. You will will get a chance to look into that in the exercises at the end of this section.

Furthermore, implication is not the only one-operator basis for the entire system of logic. Another one is the negation of logical-and, which is called "nand".

The nand gate is one of the standard logic gates, and the fact that all of the other logical operators can be expressed in terms of nand alone makes the nand gate especially important. It happens that the nand gate is the basis for designing most large-scale digital circuits, such as computer chips. Part of the reason for this is that the physics of putting gates on chips is simplified when all of the gates are the same.

Let's see how nand can be a basis for the whole system. We will express the operators in the basis we have at this point ($\wedge$, $\vee$, and $\neg$) in terms of nand, starting with negation. Negation has only one input signals, and nand has two. Feeding the same signal into both inputs of a nand gate produces the behavior of the negation operator.

It is easy to verify this from what we already know about logical operators because there is a one-step proof of the following equation. The proof cites the $\{\wedge$ idempotent$\}$ theorem (see page 30).

$$(\neg a) = (\neg(a \wedge a)) \quad \{\neg \text{ as nand}\}$$

The {¬ as nand} equation expresses negation as a nand operation (the negation of a logical-and). That takes care of negation. What about logical-and?

That's easy, too, because we only need to negate the output from a nand gate, and we already know we can use a nand gate to do that negation. So, we can construct a circuit with the same behavior as the logical-and by feeding the output signal from one nand gate into both inputs of a second nand gate.

Algebraically, this circuit corresponds to the following equation. It takes a two-step proof to verify the equation. The first step converts the outside nand to negation using the {¬ as nand} equation, and the second step cites the {double negation} axiom from Figure 2.5 (page 29).

$$(a \wedge b) = (\neg((\neg(a \wedge b)) \wedge (\neg(a \wedge b)))) \quad \{\wedge \text{ as nand}\}$$

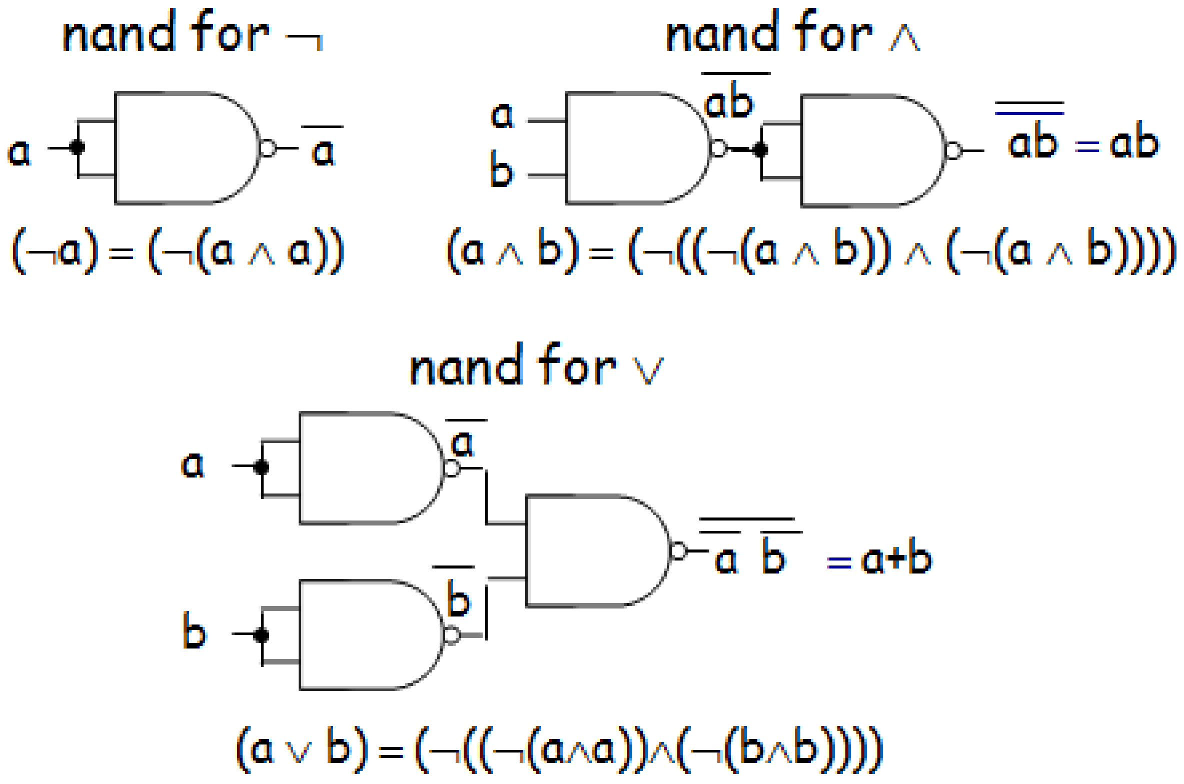
That takes care of logical-and, which brings us to logical-or. Expressing logical-or in terms of nand is a little trickier than the other two. We got negation using one nand gate and logical-and using two nand gates. We will need three nand gates for logical-or, and to verify the equation we will need a multi-step proof involving the {¬ as nand} equation, DeMorgan's laws, and double negation.

$$(a \vee b) = (\neg((\neg(a \wedge a))) \wedge ((\neg(b \wedge b))))) \quad \{\vee \text{ as nand}\}$$

We will rely on you to carry out the proofs of the {¬ as ∧}, {∧ as nand}, and {∨ as nand} equations. Figure 2.7 (page 36) displays the digital circuits that correspond to the equations verifying that nand is the only gate we really need.

Ex. 8 — Using a negation-gate and an or-gate, draw a circuit diagram with the input/output behavior of the implication operator. We refer to this circuit diagram as an "implication circuit". Hint: Follow the example of the {implication} axiom (**??**). One of the inputs will need to be a constant rather then a variable.

Ex. 9 — For each of the following logic formulas, draw an equivalent circuit diagram. Implication operators will need to be represented in the form of the circuit diagram from the previous exercise.


nand for $\neg$
a
$\overline{a}$
$(\neg a) = (\neg(a \wedge a))$
nand for $\wedge$
a
b
$\overline{ab}$
$\overline{\overline{ab}} = ab$
$(a \wedge b) = (\neg((\neg(a \wedge b)) \wedge (\neg(a \wedge b))))$
nand for $\vee$
a
b
$\overline{a}$
$\overline{b}$
$\overline{\overline{a}\,\overline{b}} = a+b$
$(a \vee b) = (\neg((\neg(a \wedge a)) \wedge (\neg(b \wedge b))))$

TODO: I did not have Visio with me. Improvised with PowerPoint. Figure will need to be redrawn.

Figure 2.7: Nand is All You Need

$$((a \vee (b \wedge (\neg a))) \vee (\neg (a \vee b)))$$
$$(((\neg a) \wedge (\neg b)) \wedge (b \wedge (\neg c)))$$
$$(a \to (b \to c))$$
$$((a \wedge b) \to c)$$

Ex. 10 — Rewrite each of the formulas in the previous exercise in the algebraic notation used by electrical engineers: juxtaposition for $\wedge$, $+$ for $\vee$, and $\bar{a}$ for $(\neg a)$.

Ex. 11 — Draw circuit diagrams with behavior of the and-gate, or-gate, and negation-gate, but in these diagrams use implication circuits only—no other kinds of logic gates.

TODO: do we still want to add half-adder and/or full-adder circuits as examples? problem: they have two outputs, so need to talk about tapping outputs from subformulas to show correspondence to algebraic form

2.4 Deduction

We have been reasoning with equations, which means we are reasoning in two directions at the same time, since equations go both ways. Deductive reasoning is one-directional. It derives a conclusion from hypotheses using one-directional rules of inference. A proof shows that the conclusion is true whenever the hypotheses are true, but provides no information about the conclusion when the truth of one or more of the hypotheses is unknown.

Another way to say this is that a deductive proof of the formula c (a conclusion) from the formula h (the hypothesis) guarantees that the formula $(h \to c)$ is true. Whenever we want to use deduction to prove that a formula with implication as its top-level operator, $(h \to c)$, is true, this is the way we will do it. We will construct a deductive proof of the conclusion c, assuming that the hypothesis h is true.

It is important to realize that proving an implication formula does not require a proof of the hypothesis, only a derivation of the conclusion from the hypothesis. This is because an implication formula is always true if its hypothesis is false (by the $\{\to$ truth table$\}$ theorem, page 20). So, we can be sure that the implication formula is true as long as we know that the only combination of values that makes the implication $(h \to c)$ false, that is $h = True$ and $c = False$, cannot arise.

There might, of course, be any number of hypotheses, combined with the $\wedge$ operator. If for example there are three hypotheses, h_1, h_2, and h_3, then to use deduction to prove $((h_1 \wedge h_2 \wedge h_3) \rightarrow c)$, we need to derive, through the rules of deductive reasoning, the formula c from the formula $(h_1 \wedge h_2 \wedge h_3)$.

By the way, we have been a little cagey with the formula $(h_1 \wedge h_2 \wedge h_3)$. It is not fully parenthesized. Does it mean $((h_1 \wedge h_2) \wedge h_3)$ or $(h_1 \wedge (h_2 \wedge h_3))$? The answer is, it doesn't matter because of the {$\wedge$ associative} theorem (page 30). Both formulas have the same meaning.

There might even be no hypotheses at all, in which case a proof of the conclusion c would guarantee the truth of the formula c. That is, the proof would guarantee the equation $c = True$.

We are not going to discuss deductive reasoning in great detail because most of the things we will prove will come from reasoning with equations. A formal treatment of deductive reasoning makes it possible to prove all of the axioms of Boolean algebra (Figure 2.5, page 29) from a small set of inference rules, each of which is easily seen to be consistent with the use of logic in everyday life.

So, in a sense deductive reasoning gets closer to the fundamentals of logic than the axioms of Boolean algebra, but we are more interested in getting at how computers work than in building a formal system of logic from the ground up. If you are interested in fundamentals, an accessible treatment can be found in a text byO'Donnell, Hall, and Page (*Discrete Mathematics Using a Computer*, Springer, 2006). See the sections on "natural deduction" (an invention of Gerhard Gentzen).

To give you a light introduction to how it works, consider the rules of inference for deductive reasoning in Figure 2.8 (page 39). One of the rules, {$\rightarrow$ introduction}, is a formal statement of the discussion at the beginning of this section about proofs of implication formulas.

The {$\vee$ elimination} rule supports case-by-case proofs. That is, if you can make a list of cases that covers all of the possibilities, you can prove that a formula is true if you derive it from each of the cases separately.

The {modus ponens} rule, probably the most famous one because of the well known "Socrates was a man" application, says that if you know that the formula a is true and that the formula $a \rightarrow b$ is true, you can conclude that b is true.

The {reductio ad absurdum} rule supports "proof by contradiction". It says that if you can derive $False$ from $(\neg a)$, then $(\neg a)$ must be $False$,

$$\begin{array}{l}\text{Prove } a\\ \text{Prove } a \to b\\ \hline \text{Infer } b\end{array}\quad\{\text{modus ponens}\}$$

$$\begin{array}{l}\text{Prove } a \vee b\\ \text{Prove } a \to c\\ \text{Prove } b \to c\\ \hline \text{Infer } c\end{array}\quad\{\vee\text{ elimination}\}$$

$$\begin{array}{l}\text{Prove } a\\ \text{Prove } b\\ \hline \text{Infer } a \wedge b\end{array}\quad\{\wedge\text{ introduction}\}$$

$$\begin{array}{l}\text{Prove } (\neg a) \to False\\ \hline \text{Infer } a\end{array}\quad\{\text{reductio ad absurdum}\}$$

$$\begin{array}{l}\text{Assume } a\\ \text{Prove } b\\ \hline \text{Infer } a \to b\end{array}\quad\{\to\text{ introduction}\}$$

$$\begin{array}{l}\text{Prove } a \wedge b\\ \hline \text{Infer } a\end{array}\quad\{\wedge\text{ elimination}\}$$

Figure 2.8: Rules of Inference for Reasoning by Deduction

which means that a must be $True$.

The $\{\wedge$ elimination$\}$ rule says that if you know $a \wedge b$, you can conclude that a must be $True$. The same goes for b of course, but that is another rule. We have not included it in the table because our goal is to give you the general idea, not to put together a complete set of rules. Following the approach we have stated here, we would need to add five more rules (the other version of $\{\wedge$ elimination$\}$ being one of them) to have a complete set that would allow us to derive all of the equations of Boolean algebra.

Fortunately, citing equations from Boolean algebra to justify steps is consistent with proof by deduction. That is, a formula in a deductive proof may be replaced by an equivalent formula justified by an equation known from Boolean algebra. So, if we accept the $\{\wedge$ commutative$\}$ equation (see page 30), we can derive the other $\{\wedge$ elimination$\}$ from Figure 2.8 (page 39).

The rule would be stated as follows.

$$\frac{\text{Prove } a \wedge b}{\text{Infer } b} \quad \{\wedge \text{ elimination-2}\}$$

The derivation of the new rule, $\{\wedge$ elimination-2$\}$, proceeds by deductive reasoning as follows.

$$\begin{array}{ll}
\underline{\text{Prove } a \wedge b} & \{\wedge \text{ commutative}\} \\
\underline{\text{Prove } b \wedge a} & \{\wedge \text{ elimination}\} \\
\text{Infer } b &
\end{array}$$

Just as with Boolean equations, once a new rule is proven, it can be cited to justify steps in proofs. So, at this point we could use $\{\wedge$ elimination-2$\}$ in a proof by deductive reasoning.

To firm up the idea at least a little, we will do one more proof using deductive reasoning. The theorem is known as the implication chain rule.

$$((a \to b) \wedge (b \to c)) \to (a \to c) \quad \{\to \text{chain}\}$$

The proof proceeds as follows.

$$\begin{array}{ll}
\underline{\text{Assume } ((a \to b) \wedge (b \to c))} & \{\wedge \text{ elimination}\} \\
\text{Infer } (a \to b) & \\
\underline{\text{Assume } a} & \{\text{modus ponens}\} \\
\text{Infer } b & \\
\underline{\text{Assume } ((a \to b) \wedge (b \to c))} & \{\wedge \text{ elimination-2}\} \\
\underline{\text{Infer } (b \to c)} & \{\text{modus ponens}\} \\
\underline{\text{Infer } c} & \{\text{modus ponens}\} \\
\underline{\text{Infer } (a \to c)} & \{\text{modus ponens}\} \\
\text{Infer } ((a \to b) \wedge (b \to c)) \to (a \to c) &
\end{array}$$

This theorem is a tautology. That is, the $\{\to \text{chain}\}$ theorem confirms that a particular formula is equivalent to $True$. In the form of an equation, this theorem would be stated as follows.

$$(((a \to b) \wedge (b \to c)) \to (a \to c)) = True \quad \{\to \text{chain}\}$$

If you want to get a little more practice in reasoning with equations, construct a prooof of the $\{\to \text{chain}\}$ equation based on the axioms of Boolean algebra (Figure 2.5, page 29).

TODO: Not sure whether we need a section on one-directional reasoning or not ... seems like we do, but a full-fledged Gentzen-style treatment gets tedious ... how do we keep it lively, but still provide the necessary apparatus? ... Maybe better to introduce inference rules when we introduce induction? ... how many rules do we need? Would modus ponens and or-elim (plus induction) be enough? How about reductio-ad-absurdum? law of excluded middle? Just the rules we will be using in doing inductive proofs about software and circuits

TODO: after writing this, I'm not sure it does any good. I'm especially not sure the proof notation I've used is clearly explained. I went through all the lectures, homeworks, and exams in the existing applied logic course, and did not find any theorems proved by deductive reasoning that were not more easily handled by stating them as implications and proving them as equations of the form $(a \to b) = True$. *I guess we could leave this stuff in, but give it short shrift, and refer back to it if necessary. We will use deduction when we come to induction, which is a deductive inference rule, but I'm not sure we need to make a big deal out of it*

TODO: Check to make sure predicates and quantifiers are covered somewhere. Show how to convert between forall and exists.

Chapter 3

Functions and Prefix Notation

3.1 Testing Software

Suppose you have purchased a piece of software from someone, and you want to take it for a test drive to see if it works. Let's say it's the function that delivers sum of the first n reciprocals, given a number n. (You've seen this function before: page **??**.)

$$reciprocals(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

You could start with a few simple checks.

$$reciprocals(1) = 1 \qquad reciprocals(2) = \frac{3}{2} \qquad reciprocals(2) = \frac{11}{6}$$

That gets old fast. It would be nice to do a whole bunch of tests, not just a few specific ones. One way to describe a lot of tests at once is to find a way to specify a relationship among values of the function that you know will always be true. For example, you know that $reciprocal(n)$ is just like $reciprocal(n-1)$, except that it has one more term in the sum, namely the reciprocal of n.

$$reciprocal(n) = reciprocals(n-1) + 1/n, if n > 1$$

This relationship provides a different test for each possible value of n. You could tell the computer to run thousands of tests using random values for n and have it report an error any of the tests fail. If the computer encounters errors, you will of course want your money back.

Since you do not expect the function to work unless the input is a non-negative integer, you need to tell the computer to choose random numbers of that kind. In fact, the test as it stands doesn't work when n is zero, so we need to fix that problem, either by avoiding zero in the testing process or by modifying to test to deal with zero.

$$reciprocal(n) = \begin{cases} 0 & \text{if } x = 0 \\ reciprocals(n-1) + 1/n & \text{otherwise} \end{cases}$$

After this modification, the computer can run thousands of tests automatically, just by choosing random, non-negative, integer values for n, computing $reciprocals(n)$ and $reciprocals(n-1)+1/n$, and checking to make sure they are the same. If zero comes up as the random input, the computer compares $reciprocals(n)$ to zero instead of $reciprocals(n-1)+1/n$.

It's nice to have an automated testing facility if you're buying software, and you want to make sure it works. It's also nice if you're building the software yourself. Then, you can deliver extensively tested software to your customers, and incur less risk of them asking for their money back.

We will be using a software development environment that facilitates automated testing. To use it, we define tests in the form of equations that that formulas using the software should satisfy, and tell the computer to generate random data and make sure the equations aren't violated.

The environment we will be using for this purpose is known as Dracula, and it's automated testing facility is called "doublecheck". The tool employs a notation that requires formulas to be written in "prefix" form. From our experience with arithmetic formulas, we are more accustomed to "infix" form, where operators come between their operands, as in the formula $(x+y)$. In prefix form, this formula would be $(+xy)$.

That seems simple enough, but it gets gradually more unfamiliar as the formulas get more complicated. For example, the formula sum of x and 3 times y, $(x + 3 * y)$, comes out as $(+x(*3y))$ in prefix form. It takes a while to get used to prefix notation, but not long. Some people end up liking it better than infix.

In prefix notation, the combined property of the *reciprocals* function specified above would take the following form.

Dr ACuLa, or "Dracula" for short, is the software development environment we will be using in to illustrate logic in action. Follow the instructions at the following URL to download and install it on your computer system: `http://www.ccs.neu.edu/home/cce/acl2/`. If that link fails, use look for "Dracula DrRacket" with your web search engine. It's a well-known system and should be easy to find. It's free, too.
Dracula provides a way to develop, test, reason about, and run programs written in the ACL2 dialect of a programming language called Lisp. It also provides a way to use a mechanized logic that is part of the ACL2 system to assist us in reasoning about our programs. At present we are discussing the testing process. Later, we will discuss developing programs, reasoning about them, and running them.

Aside 8: DrACuLa – a Software Development Environment

```
(defproperty reciprocals-test
  (n :value (random-natural))
  (= (reciprocals n)
      (if (= n 0)
          0
          (+ (reciprocals (- n 1)) (/ 1 n)))))
```

As you can see, all of the formulas are in prefix notation, even the operator that sets up the definition: defproperty. It comes first, then comes the information it needs to carry out the tests. The first part of the data describes the random values to be used. In this case, values for n will be random natural numbers (that is, integers that are zero or bigger).

A specification of the test comes next. This one compares (reciprocals n) with 0, if n is zero, or with (+ (reciprocals (- n 1) (/ 1 n)), if n isn't zero. As you can see, the comparison operator, "=", comes first (it's prefix notation). Its first operand is the (reciprocals n) formula, and its second operand is one of two alternative formulas selected by an "if" operation. The "if" operator takes three operands and selects between the formulas supplied as its last two operands. It selects the second operand if the first operand has the value true and selects the third operand if the first operand has the value false.

Given this property definition, the doublecheck facility will perform many tests using random data and report successes and failures. You can ask for as many tests as you like. The default is fifty tests, so if you don't say how many you want, you get fifty tests. The following property definition is the same as before, except that it calls for a hundred tests.

```
(include-book "doublecheck" :dir :teachpacks)

(defproperty reciprocals-test  :repeat 100
  (n :value (random-natural))
  (= (reciprocals n)
       (if (= n 0)
           0
           (+ (reciprocals (- n 1)) (/ 1 n)))))
```

The "include-book" command at the beginning tells the computer what testing facility to use and where to find it. The tests will not take place without this directive.

Let's look at another example. Suppose the function "mod" delivers the remainder when dividing one number by another. Again, imagine that you have downloaded the function from your software supplier, and you are test driving it to decide whether to pay for the download or discard it. You might do some simple checks, such as using mod to compute the remainder when dividing by two. You expect this to be zero for even numbers and one for odd numbers.

```
(include-book "testing" :dir :teachpacks)

(check-expect (mod 12 2) 0)
(check-expect (mod 27 2) 1)
```

Here, we have used the "testing" system, which is a facility provided by our computing system to help with simple tests. It doesn't generate random data and run lots of tests, like the doublecheck facility, but is handy for building packages of sanity checks to make sure a function delivers the right answers for a few special cases.

You might also include some divisions by 3 in your package of tests.

```
(check-expect (mod 14 3) 2)
(check-expect (mod  7 3) 1)
(check-expect (mod 18 3) 0)
```

Probably, though, you will want to put "mod" through its paces on a large number of tests using doublecheck. For that, you will need to come up with a relationship that expresses an important property of the function. Since you know that the remainder doesn't change when you subtract the divisor from the dividend, at least when the divisor isn't zero and the dividend is at least as big as the divisor, you use that as a basis for some automated testing.

```
(defproperty mod-test
  (divisor   :value (+ 1 (random-natural))
   dividend  :value (+ divisor (random-natural)))
  (= (mod dividend divisor)
     (mod (- dividend divisor) divisor)))
```

Generating random data is an art. In this example, we have made sure the divisor isn't zero by adding one to a natural number. Since negative numbers aren't "natural" numbers, we know that adding one to a natural number ensures that the sum is non-zero. Using a similar trick, we make sure the dividend is at least as large as the divisor be choosing a random natural number and adding it too the divisor. That way, all of the random inputs will meet the constraints that our tests are based on.

To see the testing facility in action, define the "mod-test" property in a .lisp file, and use Dracula to run the tests. To put a .lisp file into operation with Dracula, just start the file running as you would any other program. When the Dracula window is ready, press the "Run" button in the tool bar on the upper right.

We might want to do some additional testing of the mod function. Another of its properties that we know from our experience with division as an arithmetic operation is that the remainder is always smaller than the divisor. That is, (mod dividend divisor) ¡ divisor. The following property definition makes it possible to use the doublecheck testing facility to check to make sure the mod function delivers values in this range.

```
(defproperty mod-upper-limit-test
  (divisor   :value (+ 1 (random-natural))
   dividend  :value (+ divisor (random-natural)))
  (< (mod dividend divisor) divisor))
```

In this test, the property is not expressed as an equation, as in the previous case, but as an inequality based on the less-than operator ("¡").

As always, the formula puts the operation in the prefix position, in front of its operands. For practice, add this property to the .lisp file with the other tests and run it.

Another well-known fact about remainders in division is that they are non-negative integers (that is, natural numbers). We can use the logical-and operator ("and") combine the upper-limit test with the natural-number test ("natp") in one property definition.

```
(defproperty mod-range-test
  (divisor   :value (+ 1 (random-natural))
   dividend  :value (+ divisor (random-natural)))
  (and (natp (mod dividend divisor))
       (< (mod dividend divisor) divisor))
```

As you know, there are two parts to the result of dividing one number by another: the quotient and the remainder. The mod operator delivers the remainder, and an operator called "floor" delivers the quotient. From our experience with division, we known that the quotient is always strictly smaller than the dividend when the divisor is bigger than one and the dividend is a non-zero, natural number. The following test checks for that property. The random-value generator for the divisor makes sure the divisor exceeds one by adding two to a random natural number. A similar trick (which we also used in previous tests) ensures that the divisor is not zero.

```
(defproperty quotient-upper-limit-test
  (divisor   :value (+ 2 (random-natural))
   dividend  :value (+ 1 (random-natural)))
  (= (+ (* divisor (floor dividend divisor))
        (mod dividend divisor))
     dividend))
```

Checking the result of a division is a matter of multiplying the quotient by the divisor and adding the remainder. If this fails to reproduce the dividend, something has gone wrong in the division process. The following property tests this relationship between the mod and floor operators. It needs to use the multiplication operator, which is denoted by an asterisk.

```
(defproperty division-test
  (divisor   :value (+ 1 (random-natural))
   dividend  :value (+ divisor (random-natural)))
  (= (+ (* divisor (floor dividend divisor))
        (mod dividend divisor))
     dividend))
```

We hope by now you are starting to get comfortable with prefix notation. The exercises will give you a chance to practice.

Ex. 12 — Define a test of the floor operator the checks to make sure its value is a natural number when its operands are natural numbers, and the divisor (second operand) is not zero.

Ex. 13 — The “max” operator chooses the larger of two numbers: (max 2 7) is 7, (max 9 3) is 9. Define a property that tests to make sure (max x y) is greater than or equal to x and greater than or equal to y.

Chapter 4

Mathematical Induction

Induction, basic proofs, . . .

4.1 Lists as Mathematical Objects

A sequence is an ordered list of elements. In fact, we will more often use the term shorter term "list" for this kind of mathematical object. Our notation for lists displays the elements of the list, enclosed in parentheses, separated by spaces. For example, the formula "(8 3 7)" denotes the list with first element 8, second element 3, and third element 7. The formula "(9 8 3 7)" denotes a list with the same elements, but with an additional element "9" at the beginning. We use the symbol "nil" for the empty list (that is, the list with no elements).

We will start with three basic operators for lists. One of them, the construction operator, "cons", inserts a new element at the beginning of a sequence. Formulas using "cons", like all formulas in the mathematical notation we have been using to discuss software concepts, are written in prefix form. So, the formula "(cons x xs)" denotes the list with the same elements as the list xs, but with an additional element x inserted at the beginning. If x stands for the number "9", and xs stands for the list "(8 3 7)", then "(cons x xs)" constructs the list "(9 8 3 7)".

Any list can be constructed by starting from the empty list and using the construction operator to insert the elements of the list, one by one. For example, the formula "(cons 8 (cons 3 (cons 7 nil)))" is another notation for the list "(8 3 7)". In fact, using the operator "cons" is the only way to construct non-empty lists. The empty list "nil" is given. All other lists (that is, all non-empty lists) are constructed using the "cons" operator. The formula "(8 3 7)" is shorthand for "(cons 8 (cons 3 (cons 7 nil)))".

The operator that checks for non-empty lists is "consp". The formula "(consp xs)" delivers true if xs is a non-empty list and false otherwise. The {*cons*} axiom of list construction is a formal statement of the fact that all non-empty lists are constructed with the "cons" operator.

Axiom {*cons*} (consp xs) $\leftrightarrow$ ($\exists y. \exists ys.$ (xs = (cons y ys)))

We will often cite the {*cons*} axiom to write a formula like (cons x xs) in place of any list we know is not empty. When we do this, we will take care to choose the symbols x and xs to avoid conflicts with other symbols that appear in the context of the discussion. Furthermore, we will often cite a less formal version of the {*cons*} axiom when we know we are dealing with a non-empty list. For example, the list (x_1 x_2 $\ldots x_{n+1}$) cannot be empty because it has $n + 1$ elements, and $n + 1$ is at least one when n is a natural number. (We will always assume that subscripts are natural numbers.)

Informal Axiom {*cons*} (x_1 x_2 $\ldots x_{n+1}$) = (cons x_1 (x_2 $\ldots x_{n+1}$))

The construction operator, "cons", cannot be the whole story, of course. To compute with lists, we need to be able to construct them, but we also need to be able to take them apart. There are two basic operators for taking lists apart: "first" and "rest". We express the relationship between these operators and the construction operator in the form of equations ({*first*} and {*rest*}), along with the informal version of the {*cons*} axiom.

Axioms: {*cons*}, {*first*}, and {*rest*}

(x_1 x_2 $\ldots x_{n+1}$) = (cons x_1 (x_2 $\ldots x_{n+1}$))	{*cons*}
(first (cons x xs)) = x	{*first*}
(rest (cons x xs)) = xs	{*rest*}

The {*first*} axiom is a formal statement of the fact that the operator "first" delivers the first element from non-empty list. The {*rest*} axiom states that the operator "rest" delivers a list like its argument, but without the first element. Note that the list in the {*first*} and {*rest*} axioms have at least one element because they are constructed by the cons operator.

We will use equations like the ones in these axioms in the same way we used the logic equations in Figure 2.2 (see page 16) and the arithmetic equations of Figure 2.1 (see page 14). That is, whenever we see a formula like "(first (cons x xs))", no matter what formulas x and xs stand for, we will be able to cite equation {*first*} to replace "(first (cons x xs))" by the simpler formula "x". Vice versa, we can also cite equation {*first*} to replace any formula "x" by the more complicated formula "(first (cons x xs))". Furthermore, the formula "xs" in the replacement can be any formula we care to make up, as long as it is grammatically correct.

Similarly, we can cite the equation {*rest*} to justify replacing the formula "(rest (cons x xs))" by "xs" and vice versa, regardless of what formulas the symbols "x" and "x" stand for. In other words, these are ordinary algebraic equations. The only new factors are (1) the kind of mathematical object they denote (lists, instead of numbers or True/False propositions), and (2) the syntactic quirk of prefix notation (instead of the more familiar infix notation).

All properties of lists, as mathematical objects, derive from the {cons} axiom and equations {first} and {rest}. For example, suppose there is an operator called "len" that delivers the number of elements in a list. We can use check-expect to test len in some specific cases.

```
(check-expect (len (cons 8 (cons 3 (cons 7 nil)))) 3)
(check-expect nil 0)
```

We can use the doublecheck facility to automate tests. We expect that the number of elements in a non-empty list is one more than the number of elements remaining in the list after the first one is dropped using the "rest" operator. The following property tests this expectation.

```
(defproperty len-test
  (xs :value (random-list-of (random-natural)))
  (= (len xs)
     (if (consp xs)
         (+ 1 (len (rest xs)))
         0)))
```

This property holds under all circumstances. We can express the idea in the form of equations that serve as axioms for the len operator.

Axioms: {*len*} (len nil) = 0 {*len0*}
(len (cons x xs)) = (+ (len xs) 1) {*len1*}

We expect the "len" operator to deliver a natural number, regardless of what its argument is, and we can derive this property of len from its axioms. Instead of plodding through this derivation at this point, we are going to proceed to some more interesting issues. For the record, we state it as a theorem. Later, you will have a chance to derive this theorem from the {*len*} axioms. The theorem refers to the natp operator, which you have seen before (see page **??**). It delivers true if its argument is a natural number and false otherwise.

Theorem {*len-nat*} $\forall xs.$(natp (len xs))

A related fact is that the formula (consp xs) is logically equivalent to the formula (> (len xs) 0). In the notation from Chapter 2: (consp xs)$\leftrightarrow$(> (len xs) 0). This theorem, too, can be derived from the{*len*} axioms, but we will take a pass on proving the theorem, for the moment, and state it without proof.

Theorem {*consp*$\leftrightarrow$len>0} $\forall xs.$(consp xs)$\leftrightarrow$(> (len xs) 0)

4.2 Mathematical Induction

The cons, first, and rest operators form the basis for computing with lists, but there are lots of others, too. For example, consider an operator "append" that concatenates two lists. We describe this operator using an informal schematic for lists that labels the elements of the list as variables with natural numbers as subscripts. The number of integers in the subscript sequence implicitly reveals the number of elements in the list.

In the following list schematics, the "x" list has m elements, the "y" list has n elements, and the concatenated list has $m + n$ elements.

$$(\text{append } (x_1\ x_2 \ldots x_m)\ (y_1\ y_2 \ldots y_n)) = (x_1\ x_2 \ldots x_m\ y_1\ y_2 \ldots y_n)$$

Some simple tests might bolster our understanding of the operator.

```
(check-expect (append '(1 2 3 4) '(5 6 7))
              '(1 2 3 4 5 6 7))
(check-expect (append '(1 2 3 4 5) nil)
              '(1 2 3 4 5))
```

We can use doublecheck for more extensive testing. If we concatenate the empty list nil with a list ys, we expect to get ys as a result: (append nil ys) = nil. If we concatenate a non-empty list xs with a list ys, we expect the first element of the result to be the same as the first element of xs. Furthermore, we expect the rest of the elements to be the elements of the list we would get if we concatenated a list made up of the other elements of

What is the single-quote mark doing in the formula '(1 2 3 4)? It is there to avoid confusing lists overlaps with computational formulas. By default, the ACL2 system interprets a formula like (f *x* *y* *z*) as an invocation of the operator "f" with operands *x*, *y*, and *z*. ACL2 interprets the first symbol it encounters after a left parenthesis as the name of an operator, and it interprets the other formulas, up to the matching right parenthesis, as operands. So, ACL2 interprets the "1" in the formula (1 2 3 4) as the name of an operator. Because there is no operator with the name "1", the interpretation fails.
If we want to specify the list "(1 2 3 4)" in a formula, we can, of course, use the cons operator to construct it: (cons 1 (cons 2 (cons 3 (cons 4 nil)))). But, that's too bulky for regular use. The single-quote trick provides a shorthand: '(1 2 3 4) has the same meaning as the bulky version. The single-quote mark suppresses the default interpretation of the first symbol after the left-parenthesis and delivers the list whose elements are in the parentheses. Without the single-quote mark, the "1" in "(1 2 3 4)" would be interpreted as an operator, and because there is no operator named "1", the formula would make no sense.

Aside 9: Single-quote Shorthand for Lists

xs, that is (rest *xs*), with *ys*. The following property definition expresses this idea formally.

```
(defproperty append-test
  (xs :value (random-list-of (random-natural))
   ys :value (random-list-of (random-natural)))
  (equal (append xs ys)
         (if (consp xs)
             (cons (first xs)
                   (append (rest xs) ys))
             ys)))
```

This might not be the first test you would think of, but if the test failed to pass, you would for sure know something was wrong with the append operator. In fact the property is so plainly correct, we are going to state it in the form of equations that we accept as axioms.

Like the {*len*} theorem, there are two {*append*} equations, and they specify the meaning of an append operation in different situations. One of

Why does the property say "(equal (append xs ys) …)" instead of "(= (append xs ys) …)"? the "=" operator is restricted to numbers. The "equal" operator can check for equality between other kinds of objects. You can always use "equal", but you can only use "=" when both operands are numbers. Why bother with "=", when its use is so limited? We might say it makes the formula look more like an equation, but that's not really much of an excuse, since we have already had to conform to prefix notation instead of the more familiar infix notation. So, feel free to use the "equal" operator all the time if you want to. We will be using "=" when we can and hope it doesn't put too much of an extra burden on you.

Aside 10: "equal" vs "="

them specifies the meaning when the first argument in the invocation is a list with at least one element (that is, when (consp xs) is true), the other when it has no elements (that is, when it is nil).

Axioms: {*append*}
(append nil ys) = ys {*app0*}
(append (cons x xs) ys) = (cons x (append xs ys)) {*app1*}

These equations about the append operation simple enough, but it turns out that lots of other properties of the append operation can be derived from them. For example, we can prove that the length of the concatenation of two lists is the sum of the lengths of the lists. We call this theorem the "additive law of concatenation". Let's see how a proof of this law could be carried out.

First, let's break it down into a sequence of special cases. We will use L(n) as shorthand for the proposition that (len (append (x_1 x_2 … x_n) ys)) is the sum of (len (x_1 x_2 … x_n)) and (len ys):

L(n) ≡ (= (len (append (x_1 x_2 … x_n) ys))
            (+ (len (x_1 x_2 … x_n)) (len ys)))

For the first few values of n, L(n) would stand for the following equations.

$$
\begin{array}{lll}
\text{L}(0) & \equiv & (= \ (\text{len (append nil } ys)) \\
 & & \quad (+ \ (\text{len nil}) \ (\text{len } ys))) \\
\text{L}(1) & \equiv & (= \ (\text{len (append } (x_1) \ ys)) \\
 & & \quad (+ \ (\text{len } (x_1)) \ (\text{len } ys))) \\
\text{L}(2) & \equiv & (= \ (\text{len (append } (x_1 \ x_2) \ ys) \\
 & & \quad (+ \ (\text{len } (x_1 \ x_2)) \ (\text{len } ys))) \\
\text{L}(3) & \equiv & (= \ (\text{len (append } (x_1 \ x_2 \ x_3) \ ys)) \\
 & & \quad (+ \ (\text{len } (x_1 \ x_2 \ x_3)) \ (\text{len } ys))) \\
\text{L}(4) & \equiv & (= \ (\text{len (append } (x_1 \ x_2 \ x_3 \ x_4) \ ys)) \\
 & & \quad (+ \ (\text{len } (x_1 \ x_2 \ x_3 \ x_4)) \ (\text{len } ys)))
\end{array}
$$

We can derive L(0) from the {*append*} and {*len*} axioms as follows, starting from the first operand in the equation that L(0) stands for (the left-hand side, if the equation were written in the conventional way rather than prefix form), and ending with the second operand (right-hand side).

$$
\begin{array}{lll}
 & (\text{len (append nil } ys)) & \\
= & (\text{len } ys) & \{\textit{app0}\} \\
= & (+ \ (\text{len } ys) \ 0) & \{+ \text{ identity}\} \text{ (page 14)} \\
= & (+ \ 0 \ (\text{len } ys)) & \{+ \text{ commutative}\} \text{ (page 14)} \\
= & (+ \ (\text{len nil}) \ (\text{len } ys)) & \{\textit{len0}\}
\end{array}
$$

That was easy. How about L(1)?

$$
\begin{array}{lll}
 & (\text{len (append } (x_1) \ ys)) & \\
= & (\text{len (append (cons } x_1 \text{ nil) } ys) & \{\textit{cons}\} \\
= & (\text{len (cons } x_1 \text{ (append nil } ys))) & \{\textit{app1}\} \\
= & (+ \ 1 \ (\text{len (append nil } ys))) & \{\textit{len1}\} \\
= & (+ \ 1 \ (+ \ (\text{len nil}) \ (\text{len } ys))) & \{\text{L}(0)\} \\
= & (+ \ (+ \ 1 \ (\text{len nil})) \ (\text{len } ys)) & \{+ \text{ associative}\} \text{ (page 14)} \\
= & (+ \ (\text{len (cons } x_1 \text{ nil)) } (\text{len } ys)) & \{\textit{len1}\} \\
= & (+ \ (\text{len } (x_1) \ (\text{len } ys)) & \{\textit{cons}\}
\end{array}
$$

That was a little harder. Will proving L(2) be still harder? Let's try it.

$$\begin{array}{lll}
& (\text{len (append } (x_1\ x_2)\ ys)) & \\
= & (\text{len (append (cons } x_1\ (x_2))\ ys)) & \{\textit{cons}\} \\
= & (\text{len (cons } x_1 \text{ (append } (x_2)\ ys))) & \{\textit{app1}\} \\
= & (+\ 1 \text{ (len (append } (x_2)\ ys))) & \{\textit{len1}\} \\
= & (+\ 1\ (+\ (\text{len } (x_2))\ (\text{len } ys))) & \{\text{L}(1)\} \\
= & (+\ (+\ 1\ (\text{len } (x_2)))\ (\text{len } ys)) & \{+ \text{ associative}\} \\
= & (+\ (\text{len (cons } x_1\ (x_2)))\ (\text{len } ys)) & \{\textit{len1}\} \\
= & (+\ (\text{len } (x_1\ x_2))\ (\text{len } ys)) & \{\textit{cons}\}
\end{array}$$

Fortunately, proving L(2) was no harder than proving L(1). In fact the two proofs cite exactly the same equations all the way through, except in one place. Where the proof of L(1) cited the equation L(0), the proof of L(2) cited the equation L(1). Maybe the proof of L(3) will work the same way.

$$\begin{array}{lll}
& (\text{len (append } (x_1\ x_2\ x_3)\ ys)) & \\
= & (\text{len (append (cons } x_1\ (x_2\ x_3))\ ys)) & \{\textit{cons}\} \\
= & (\text{len (cons } x_1 \text{ (append } (x_2\ x_3)\ ys))) & \{\textit{app1}\} \\
= & (+\ 1 \text{ (len (append } (x_2\ x_3)\ ys))) & \{\textit{len1}\} \\
= & (+\ 1\ (+\ (\text{len } (x_2\ x_3))\ (\text{len } ys))) & \{\text{L}(2)\} \\
= & (+\ (+\ 1\ (\text{len } (x_2\ x_3)))\ (\text{len } ys)) & \{+ \text{ associative}\} \\
= & (+\ (\text{len (cons } x_1\ (x_2\ x_3)))\ (\text{len } ys)) & \{\textit{len1}\} \\
= & (+\ (\text{len } (x_1\ x_2\ x_3))\ (\text{len } ys)) & \{\textit{cons}\}
\end{array}$$

By now, it's easy to see how to derive L(4) from L(3), then L(5) from L(4), and so on. If you had the time and patience, you could surely prove L(100), L(1000), or even L(1000000) by deriving the next one from the one you just finished proving, following the established pattern. We could even write a program to print out the proof of L(n), given any natural number n.

Since we know how to prove L(n) for any natural number n, it seems fair to say that we know all those equations are true. However, to complete proof of the formula ($\forall n$.L(n)), we need a rule of inference that allows us to make conclusions from patterns like those we observed in proving L(1), L(2), and so on. That rule of inference is known as "mathematical induction".

Mathematical induction provides a way to prove that formulas like ($\forall n$.P(n)) are true when P is a predicate whose universe of discourse is the natural numbers. If for each natural number n, P(n) stands for a

$$\begin{array}{l} \text{Prove P}(0) \\ \text{Prove } (\forall n.(\mathrm{P}(n) \rightarrow \mathrm{P}(n+1))) \\ \hline \text{Infer } (\forall n.\mathrm{P}(n)) \end{array}$$

Figure 4.1: Mathematical Induction–a rule of inference

proposition, then mathematical induction is an applicable inference rule in a proof that is $(\forall n.\mathrm{P}(n))$ true. That is not to say that such a proof can be constructed. It's just that mathematical induction might provide some help in the process.

The rule goes as follows: one can infer the truth of $(\forall n.\mathrm{P}(n))$ from proofs of two other propositions. Those two propositions are P(0) and $(\forall n.(\mathrm{P}(n) \rightarrow \mathrm{P}(n+1)))$. It's a very good deal if you think about it. A direct proof of $(\forall n.\mathrm{P}(n))$ would require a proof of proposition $\mathrm{P}(n)$ for each value of n (0, 1, 2, ...). But, in a proof by induction, the only proposition that needs to be proved on its own is P(0). In the proof any of the other propositions, you are allowed to cite the previous one in the sequence as a justification for any step in the proof.

The reason you can assume that $\mathrm{P}(n)$ is true in the proof of $\mathrm{P}(n+1)$ is because the goal is to prove that the formula $\mathrm{P}(n) \rightarrow \mathrm{P}(n+1)$ has the value "true". We know from the truth table of the implication operator (see page 20) that the formula $\mathrm{P}(n) \rightarrow \mathrm{P}(n+1)$ is true when $\mathrm{P}(n)$ is false. So, we only need to verify that the formula is true when $\mathrm{P}(n)$ is true. The formula will be true in this case when $\mathrm{P}(n+1)$ is true. So, all we need to prove is that $\mathrm{P}(n+1)$ under the assumption that $\mathrm{P}(n)$ is true.

That is, in the proof of $\mathrm{P}(n+1)$, you can cite $\mathrm{P}(n)$ to justify any step in the proof. $\mathrm{P}(n)$ gives you a leg up in the proof of $\mathrm{P}(n+1)$ and is known as the "induction hypothesis".

Now, let's apply mathematical induction to prove the additive law of concatenation. Here, the predicate that we will apply the method to is L:

$$\begin{array}{l} \mathrm{L}(n) \equiv (= (\text{len } (\text{append } (x_1\ x_2\ \ldots x_n)\ ys)) \\ \qquad\qquad\quad (+\ (\text{len } (x_1\ x_2\ \ldots x_n))\ (\text{len } ys))) \end{array}$$

We have already proved L(0). All that is left is to prove $(\forall n.(\mathrm{L}(n) \rightarrow \mathrm{L}(n+1)))$. That is, we have to derive $\mathrm{L}(n+1)$ from $\mathrm{L}(n)$ for an arbitrary natural number n. Fortunately, we know how to do this. Just copy the derivation of, say L(3) from L(2), but start with an append formula in which the first operand is a list with $n+1$ elements, and cite L(n) where we would have cited L(3).

$$\begin{array}{cll}
 & (\text{len } (\text{append } (x_1\ x_2 \ldots x_{n+1})\ ys)) & \\
= & (\text{len } (\text{append } (\text{cons } x_1\ (x_2 \ldots x_{n+1}))\ ys)) & \{\textit{cons}\} \\
= & (\text{len } (\text{cons } x_1\ (\text{append } (x_2 \ldots x_{n+1})\ ys))) & \{\textit{app1}\} \\
= & (+\ 1\ (\text{len } (\text{append } (x_2 \ldots x_{n+1})\ ys))) & \{\textit{len1}\} \\
= & (+\ 1\ (+\ (\text{len } (x_2 \ldots x_{n+1}))\ (\text{len } ys))) & \{\text{L}(n)\} \\
= & (+\ (+\ 1\ (\text{len } (x_2 \ldots x_{n+1})))\ (\text{len } ys)) & \{+\text{ associative}\} \\
= & (+\ (\text{len } (\text{cons } x_1\ (x_2 \ldots x_{n+1})))\ (\text{len } ys)) & \{\textit{len1}\} \\
= & (+\ (\text{len } (x_1\ x_2 \ldots x_{n+1}))\ (\text{len } ys)) & \{\textit{cons}\}
\end{array}$$

An important point to notice in this proof is that we could not cite the {*cons*} equation to replace $(x_2 \ldots x_{n+1})$ with (cons x_2 $(x_3 \ldots x_{n+1})$). The reason we could not do this is that we are trying to derived L($n+1$) from L(n) without making any assumptions about n other than the fact that it is a natural number. Since zero is a natural number, the list $(x_2 \ldots x_{n+1})$ could be empty, and the cons operation cannot deliver an empty list as its value.

In the next section, we will prove some properties of append that confirm its correctness with respect to a specification in terms of other operators. These properties, and in fact all properties of the append operator, can be derived from the append theorem. That theorem states properties of the append operation in two cases: (1) when the first operand is the empty list (the {*app0*} equation), and (2) when the first operand is a non-empty list (the {*app1*} equation). When the first operand is the empty list, the result must be the second operand, no matter what it is. When the first operand is not empty, it must have a first element. That element must also be the first element of the result. The other elements of the result are the ones you would get if you appended the rest of the first operand with the second operand.

Both of these properties are so straightforward and easy to believe that we would probably be willing to accept them as axioms, with no proof at all. It might come as a surprise that all of the other properties of the append operation can be derived from the two simple properties {*app0*} and {*app1*}. That is the power of mathematical induction. The two equations

Foundational	There is at least one non-inductive equation (that is, an equation that only refers to the operator being defined on one side of the equation).
Complete	All possible combinations of operands are covered by at least one equation in the definition.
Consistent	Combinations of operands covered by by two or more equations imply the same value for the operation.
Computational	In inductive equations (that is, equations that refer to the operator being defined on both sides of the equation), all invocations of the operator on the right-hand side of the equation have operands that are closer to the operands on the left-hand side of a non-inductive equation than the operands on the left-hand side of the equation.

Figure 4.2: Characteristics of Inductive Definitions

of the append theorem amount to an inductive definition of the append operator.

An inductive definition is circular in the sense that some of the equations in the definition refer to the operator on both sides of the equation. Most of the time, we think circular definitions are not useful, so it may seem surprising that they can be useful in mathematics. Some aren't, but some of them are, and you will gradually learn how to recognize and create useful, circular (that is, inductive) definitions.

It turns out that all functions that can be defined in software have inductive definitions in the style of the equations of the append theorem. The keys to an inductive definition of an operator are listed in Figure 4.2 (see page 61). All of the software we will discuss will take the form of a collection of inductive definitions of operators. That makes it possible to use mathematical induction as the fundamental tool in verifying properties of that software to a logical certainty.

This is not the only way to write software. In fact, most software is not written in terms of inductive definitions. But, properties of the software written using conventional methods cannot be derived using conventional logic. So, in terms of understanding what computers do and how they do it, inductive definitions provide solid footing. That is why we base our presentation on software written in terms of inductive definitions rather than by conventional methods.

4.3 Contatenation, Prefixes, and Suffixes

If you concatenate two lists, xs and ys, you would expect to be able to retrieve the elements of ys by dropping some of the elements of the concatenated lists. How many elements would you need to drop? That depends on the number of elements in xs. If there are n elements in xs, and you drop n elements from (append xs ys), you expect the result to be identical to the list ys. We can state that expectation by using an intrinsic operation in ACL2 with the arcane name “nthcdr”. The nthcdr operation takes two arguments: a natural number and a list. The formula (nthcdr n xs) delivers a list like xs, but without its first n elements. If xs has fewer than n elements, then the formula delivers the empty list.

The following equations state some simple properties of the nthcdr operation that we take as axioms.

Axioms {*nthcdr*}	
(nthcdr 0 xs) = xs	{*sfx0*}
(nthcdr n nil) = nil	{*sfx-nil*}
(nthcdr (+ n 1) (cons x xs)) = (nthcdr n xs)	{*sfx1*}

TODO: Rex: sfx1 isn't true, right? I'm not sure we want to introduce it as an axiom, if later we'll have to explain it isn't really true. Ruben: left as is for now, with (natp n) implicit, but inserted a comment about the type of n

Given this background, we state the expected relationship between the append and nthcdr operators in terms of a sequence of special cases. We will use S(n) as a shorthand for special case number n. There will be one case for each natural number.

$$\begin{array}{llll} \mathrm{S}(n) \equiv (\text{equal} & (\text{nthcdr} & (\text{len } (x_1\ x_2 \ldots x_n)) \\ & & (\text{append } (x_1\ x_2 \ldots x_n)\ ys)) \\ & ys) & \end{array}$$

If S(n) to be true, regardless of what natural number n stands for, then the formula ($\forall n$.S(n)) is true. Since the universe of discourse of the predicate S is the natural number, mathematical induction may be useful in verifying that formula. All we need to do is to prove that (1) the formula S(0) is true and (2) the formula S($n+1$) is true under the assumption that S(n) is true, regardless of what natural number n stands for. Let's do that.

First, we prove S(0). When n is zero, the list ($x_1\ x_2\ \ldots x_n$) is empty, which is normally denoted by the symbol "nil". So, S(0) stands for the following equation.

$$\begin{array}{ll} \mathrm{S}(0) \equiv (\text{equal} & (\text{nthcdr (len nil) (append nil } ys)) \\ & ys) \end{array}$$

Following our usual practice when proving an equation, we start with the formula on one side and use previously known equations to gradually transform that formula to the one on the other side of the equation.

$$\begin{array}{lll} & (\text{nthcdr (len nil) (append nil } ys)) & \\ = & (\text{nthcdr (len nil) } ys) & \{\mathit{app0}\}\text{ (see page 56)} \\ = & (\text{nthcdr 0 } ys) & \{\mathit{len0}\}\text{ (see page 53)} \\ = & ys & \{\mathit{sfx0}\} \end{array}$$

That takes care of S(0). Next, we prove S($n+1$), assuming that S(n) is true.

$$\begin{array}{llll} \mathrm{S}(n+1) \equiv (\text{equal} & (\text{nthcdr} & (\text{len } (x_1\ x_2\ \ldots x_{n+1})) \\ & & (\text{append } (x_1\ x_2\ \ldots x_{n+1})\ ys)) \\ & ys) & \end{array}$$

$$\begin{array}{llll} & (\text{nthcdr} & (\text{len } (x_1\ x_2\ \ldots x_{n+1})) & \\ & & (\text{append } (x_1\ x_2\ \ldots x_{n+1})\ ys)) & \\ = & (\text{nthcdr} & (\text{len (cons } x_1\ (x_2\ \ldots x_{n+1}))) & \{\mathit{cons}\}\text{ (page 52)} \\ & & (\text{append (cons } x_1\ (x_2\ x_2\ \ldots x_{n+1}))\ ys)) & \{\mathit{cons}\} \\ = & (\text{nthcdr} & (+\ (\text{len } (x_2\ \ldots x_{n+1}))\ 1) & \{\mathit{len1}\}\text{ (page 53)} \\ & & (\text{cons } x_1\ (\text{append } (x_2\ \ldots x_{n+1}))\ ys)) & \{\mathit{app1}\}\text{ (page 56)} \\ = & (\text{nthcdr} & (\text{len } (x_2\ \ldots x_{n+1})) & \{\mathit{sfx1}\} \\ & & (\text{append } (x_2\ \ldots x_{n+1})\ ys)) & \\ = & ys & & \{\mathrm{S}(n)\} \end{array}$$

The last step in the proof is justified by citing S(n). This is a little tricky because the formula that S(n) stands for is not exactly the same as the formula in the next-to-last step of the proof. We interpret the formula ($x_1\ x_2 \ldots x_n$) in the definition of S(n) to stand for any list with n elements. The elements in the list ($x_2 \ldots x_{n+1}$) are numbered 2 through $n+1$, which means there must be exactly n of them.

With this interpretation, the formula in the next-to-last step matches the formula in the definition of S(n), which makes it legitimate to cite S(n) to justify the transformation to *ys* in the last step of the proof. We will use this interpretation frequently in proofs. We refer to it as the "numbered-list interpretation", or {*nlst*} for short.

($x_m \ldots x_n$) denotes a list with max($n - m + 1$, 0) elements {*nlst*}

TODO: Rex: We may want to skip this paragraph for now. We can introduce this notation later, when needed. Ruben: right, done

Of course, if n is zero, or if *xs* is empty, (prefix n *xs*) must be the empty list. If n is non-zero natural number and *xs* is not empty, then the first element of (prefix n *xs*) must be the first element of *xs*, the the other elements must be the first $n - 1$ elements of (rest *xs*). The following equations, which we take as axioms, put these expectations in formal terms. The formula (posp n) refers to the intrinsic ACL2 operator "posp". It is true if n is a non-zero natural number (that is, a strictly positive integer) and false otherwise.

Axioms {*prefix*}

(prefix 0 xs) = nil	{*pfx0*}
(prefix n nil) = nil	{*pfx-nil*}
(prefix (+ n 1) (cons x xs)) = (prefix n xs)	{*pfx1*}

We can derive the prefix property of the append function from the equations for the prefix and append operations. The proof will cite mathematical induction. As before, we will use a shorthand for special case number n.

TODO: Rex: Should we replace xs with (x1 x2 ... xn). Ruben: right, done

P(n) ≡ (equal (prefix (len ($x_1\ x_2 \ldots x_n$))

```
                (append ($x_1$ $x_2$ ... $x_n$) $ys$))
        ($x_1$ $x_2$ ... $x_n$))
```

We will prove that P(0) is true, and also that P($n+1$) is true whenever P(n) is true. Then, we will cite mathematical induction to conclude that P(n) is true, regardless of which natural number n stands for.

```
P($n$) ≡ (equal (prefix (len nil)
                        (append nil $ys$))
                nil)
```

As in the proof of the append suffix theorem, we start with the formula on one side of the equation and use known equations to gradually transform that formula to the one on the other side of the equation.

```
    (prefix (len nil) (append nil $ys$))
=   (prefix 0 (append nil $ys$))          {len0} (see page 53)
=   nil                                   {pfx0}
```

That takes care of P(0). Next, we prove P($n+1$), assuming that P(n) is true.

```
P($n+1$) ≡ (equal (prefix (len ($x_1$ $x_2$ ... $x_{n+1}$))
                          (append ($x_1$ $x_2$ ... $x_{n+1}$) $ys$))
                  ($x_1$ $x_2$ ... $x_{n+1}$))
```

TODO: Rex: This following indentation isn't perfect, but it's close. I haven't figured out how to remove the vertical space before the tabbing, though I can probably hack it....

(prefix (len (x_1 x_2 $\ldots x_{n+1}$))
    (append (x_1 x_2 $\ldots x_{n+1}$) ys))

= {*cons*} (page 52)
(prefix (len (cons x_1 (x_2 $\ldots x_{n+1}$)))
    (append (cons x_1 (x_2 x_2 $\ldots x_{n+1}$)) ys))

= {*len1*} (page 53)
{*app1*} (page 56)
(prefix (+ (len (x_2 $\ldots x_{n+1}$)) 1)
    (cons x_1 (append (x_2 $\ldots x_{n+1}$) ys)))

= {*pfx1*}
(cons (first (cons x_1 (x_2 $\ldots x_{n+1}$)))
    (prefix (- (+ (len (x_2 $\ldots x_{n+1}$)) 1) 1)
        (rest (cons x_1 (append (x_2 $\ldots x_{n+1}$) ys)))))

= {*first*} (page 52)
{*arithmetic*}
{*rest*} (page 52)
(cons x_1
    (prefix (len (x_2 $\ldots x_{n+1}$))
        (append (x_2 $\ldots x_{n+1}$) ys)))

= {P(n)}
(cons x_1
    (x_2 $\ldots x_{n+1}$))

= (x_1 x_2 $\ldots x_{n+1}$) {*cons*} (page 52)

At this point we know three important facts about the append function:

- additive length theorem: (len (append xs ys)) = (+ (len xs) (len ys))
- append-prefix theorem: (prefix (len xs) (append xs ys)) = xs
- append-suffix theorem: (nthcdr (len xs) (append xs ys)) = ys

Together, these theorems provide a deep level of understanding of the append operation. They give us confidence that it correctly concatenates lists. We refer to these theorems as "correctness properties" for the append operation. They are, of course, an infinite variety of other facts about the append operation. Their relative importance depends on how we are using the operation.

A property that is sometimes important to know is that concatenation is "associative". That is, if there are three lists to be concatenated, you you could concatenate the first list with the concatenation of the last two. Or, you could concatenate the first two, then append that with the third.

Theorem {*app-assoc*}
(append xs (append ys zs)) = (append (append xs ys) zs)

Addition and multiplication of numbers are associative in an analogous way (but subtraction and division aren't associative). Another way to say this is that the formula $(\forall n.\mathrm{A}(n))$ is true, where the predicate A is defined as follows.

$$\begin{array}{lll} \mathrm{A}(n) \equiv & (\text{equal} & (\text{append } (x_1\ x_2 \ldots x_n) \text{ (append } ys\ zs)) \\ & & (\text{append (append } (x_1\ x_2 \ldots x_n)\ ys)\ zs) \end{array}$$

Putting it this way makes the theorem amenable to a proof by mathematical induction. We leave that as a something you can use to practice your proof skills.

TODO: Ruben: The ExerciseList tag doesn't put in any vertical space, but should, I think.

Ex. 14 — Carry out a paper-and-pencil proof by mathematical induction of the {*app-assoc*} theorem.

TODO: next section will introduce defthmd and proofs using the ACL2 mechanized logic by replaying all of the theorems of this section in ACL2 notation

4.4 Mechanized Logic

The proofs we have been doing depend on matching grammatical elements in formulas against templates in axioms and theorems. The formulas are then transformed to equivalent ones with different grammatical structures. Gradually, we move from a starting formula to a concluding one to verify an equation for a new theorem.

It is easy to make mistakes in this detailed, syntax-matching process, but computers carry it out flawlessly. This relieves us from an obligation to focus with monk-like devotion on the required grammatical analysis. We can leave it to the computer count on having it done right.

There are several mechanized logic systems that people use to assist with proofs of the kind we have been doing. One of them is ACL2 (A Computational Logic for Applicative Common Lisp). Theorems for the ACL2 proof engine are stated in the same form as properties for the DoubleCheck testing facility in Dracula. ACL2 has a built-in strategy for finding inductive proofs, and for some theorems it succeeds in fully automating proofs. It also permits people to guide it through proofs while it pushes through all of the grammatical details.

To illustrate how this works, we will go the theorems discussed earlier in this chapter, one by one. The notation for stating theorems in ACL2 form will be familiar, but not identical to the one we have been using for our paper-and-pencil proofs. For one thing, it employs prefix notation throughout, and we have been using a mixture of prefix and infix.

Our first proof by mathematical induction verified the additive law of concatenation (see page 56). Our statement of the theorem asserted that a proposition L(n) is true, regardless of which natural number n stands for: $(\forall n.\mathrm{L}(n))$. L($n$) is a shorthand for the following formula:

$$\begin{aligned}\mathrm{L}(n) \equiv (= &(\text{len (append } (x_1\ x_2 \ldots x_n)\ ys))\\ &(+\ (\text{len } (x_1\ x_2 \ldots x_n))\ (\text{len } ys)))\end{aligned}$$

We could have used the DoubleCheck facility of Dracula to run tests on this property.

```
(defproperty additive-law-of-concatenation-tst
    (xs :value (random-list-of (random-natural))
     ys :value (random-list-of (random-natural)))
  (= (len (append xs ys))
     (+ (len xs) (len ys))))
```

Of course, the DoubleCheck specification of this property cannot employ the informal notation of the numbered list interpretation (see page 64). Instead, the property simply uses a symbol xs to stand for the list. The property does not state the length of xs, but we know its length will be some natural number, n, so the property, as stated, has the same meaning as the formula that L(n) stands for.

The statement of the additive law as a theorem in the form required by ACL2 cannot use the informal notation, either. In fact, the theorem takes a form that is like the property specification, except for the :value portion and the keyword "defproperty".

Theorem statements in ACL2 start with the "defthmd" keyword. After that comes a name for the theorem and the Boolean formula that expresses the meaning of the theorem, as illustrated in the following definition.

```
(defthmd additive-law-of-concatenation-thm
  (= (len (append xs ys))
     (+ (len xs) (len ys))))
```

The mechanized logic of ACL2 fully automates the proof of this theorem. It uses a built-in, heuristic procedure to find an induction scheme and pushes the proof through on its own. To see ACL2 in action, enter the above theorem in the program pane of the Dracula window (the upper left pane), press the start button in the ACL2 pane (the right half of the ACL2 window). When the Admit button lights up, press it.

After a short time, the theorem will turn green, indicating a successful proof. Details of the proof appear in the ACL2 pane. Later we will learn how to interpret some of these details, but for now, we are just looking for success (green coloring of the theorem in the program pane) or failure (red coloring).

Probably you can follow the above example to convert all theorems from this chapter into ACL2 theorem statements. Just to make sure, we will look at another one, then leave the rest for practice exercises.

The append-suffix theorem, which states that when the first argument in an append formula is a list of length n, then you can reconstruct the second argument by dropping n elements from the front of the concatenation. We stated this this theorem in the form $(\forall n.\mathrm{S}(n))$, where $\mathrm{S}(n)$ is a shorthand for the following formula.

$$\begin{array}{lll} \mathrm{S}(n) \equiv (\text{equal} & (\text{nthcdr} & (\text{len } (x_1\ x_2\ \ldots x_n)) \\ & & (\text{append } (x_1\ x_2\ \ldots x_n)\ ys)) \\ & ys) & \end{array}$$

The following definition specifies this theorem in ACL2 notation.

```
(defthmd append-suffix-thm
  (equal (nthcdr (len xs) (append xs ys))
         ys))
```

ACL2 can prove this theorem, but the proof requires knowing something about the algebra of numbers, such as the associative law of addition. Fortunately, someone has worked out a basic theory of numeric algebra in ACL2 terms, and we can take advantage of that by importing it into the working environment. To do this, we use a command called “include-book”. The name of the book with the theory we need is “arithmetic/top”, and it resides in the “system” directory of ACL2.

```
(include-book "arithmetic/top" :dir :system)
```

The include-book command makes that theory accessible to the mechanized logic. When you put the command above the append-suffix theorem in the program pane and press the start button and then the Admit All button, ACL2 succeeds in the proof without further assistance. For practice, try it yourself.

Ex. 15 — Define the {*append-prefix*} theorem in ACL2 notation, and use Dracula to run it through the mechanized logic. If you state it correctly, the proof should succeed.

Ex. 16 — Define the {*append associativity*} theorem in ACL2 notation, and use Dracula to run it through the mechanized logic. If you state it correctly, the proof should succeed.

Part II

Computer Arithmetic

Chapter 5

Binary Notation

binary notation, basic sequence operations (so we can talk about lists of bits), ...

Chapter 6

Adders

ripple-carry adders, twos complement, subtraction, . . .

Chapter 7

Multipliers

shift and add multipliers, ...

Part III

Inductive Data Structures

Chapter 8

Multiplexors and Demultiplexors

Chapter 9

Sorting Sequences

Chapter 10

Hashtables

Chapter 11

Trees

Chapter 12

Recurrence Relations

Part IV

Computation in Practice

Chapter 13

Parallel Execution with MapReduce

Vertical and Horizontal Scaling

Some important computer applications are so large that they would take unacceptably long to execute on typical computers. For example, your personal computer is more than powerful enough to balance your checkbook. But what about a banking application that tracks credit card usage in real time to detect instances of fraud? The sheer number of credit card transactions make this application far too time-consuming for any personal computer to handle.

Since these applications are important uses of computers, computer scientists have come up with different ways of coping with problems at large scale. The traditional way is called *vertical scaling*, and it consists of running the application on a single, very powerful machine.

This is, of course, the easiest possible solution, since the program is unchanged. But what happens as the problems get bigger? For example, the number of credit card transactions increases over time. Can a single computer, even a very large one, keep up? And if it can't, do you have to upgrade by buying an even larger, more powerful (and more expensive) computer?

Horizontal scaling offers an alternative. Instead of running the application in a single, large computer, with horizontal scaling the application is split into smaller chunks, and each chunk is run on a separate computer.

Ideally, the computers used are similar in almost all respects to personal computers, so that the cost of an additional machine is minimal. This makes it more economical to scale the hardware platform as the problems become larger. Not surprisingly, horizontal scaling has become the de facto scalability solution for modern websites, including Google, eBay, Amazon, and many others.

The problem with horizontal scaling, however, is that it is not always easy to split your application into smaller chunks. In fact, this is particularly hard when the program is written using traditional programming languages, such as C++ or Java. In those languages, the program is described as a sequence of steps that the computer must take. But that becomes more difficult to do, as the number of computers increases, because the programmer must also take into consideration the interaction between the computers.

One of the advantages of the functional style of programming that we have presented in this book is that there are no interactions between different pieces of the code. Instead, everything is defined by mathematical functions, and it does not matter where or when the functions are executed, since the result is a matter of mathematical definition.

The engineers at Google were faced with one of the largest scaling problems in the world, namely searching the entire web. To match the scale of this problem, they decided to use the horizontal scaling approach. Even though they use C++ internally, they invented and adopted a style of programming called MapReduce, which is based on a functional programming paradigm, very similar to the programming style presented in this book. Since Google's introduction, MapReduce has been adopted in many other settings. For instance, the Apache Foundation implemented Hadoop, an open-source implementation of MapReduce that you can freely download to your computer. Hadoop is widely used in industry and academia. Although we will not try to describe the full details of Google's MapReduce or Apache's Hadoop implementation, we will show you how the MapReduce framework simplifies the development of programs that can scale horizontally by focusing on just two operations.

Introduction to MapReduce

The MapReduce paradigm is applicable to problems that process a dataset that can be described as a sequence of key/value pairs. Clearly, not all problems can be characterized in this manner, but Google engineers recognized that many practical problems do fit in this category. Here are some examples.

- **Counting Words in a Document.** The dataset, of course, is the collection of words in the document. It can be organized as a list of word/count pairs, where the counts are initially set to one. Note that each word may have more than one word/count pair initially. The result of the operation would be a list of word/count pairs where each word appears only once.

- **Finding Words that Link to a Webpage.** The purpose of this operation is to find the words that are used most commonly to link to a particular page. For example, your name may be the most common phrase used to link into your Facebook page. Google uses this information to select which pages to display for a particular search. This problem, too, can be implemented using MapReduce. The dataset is the (very large) collection of links on the Internet. This can be represented as a list of word/URL pairs, where the same word may appear multiple times. The solution will be a list of URL/word pairs, where each URL will appear once, associated only with the word that is used the most to link to it.

- **Finding Extreme Values.** For example, consider an application that finds the record high and low temperature for each state. The initial dataset consists of a list of city/temperature data. There would be one record for each city and each day, for as long as records are kept. In fact, there may be more than one record for a given data per day, e.g., one record each hour. Similarly, the output consists of state/high or state/low records.

What all these applications have in common is that it is possible to process each individual input record without having to simultaneously examine all other records. For example, you can process the November 9, 2010 temperature entry for Norman, OK without considering the May 3, 1932 entry for Laramie, WY. This enables horizontal scaling, because the different entries can be processed in different machines.

However, these applications also show the need for combining the entries for a specific key at a later time. For example, to find the low temperature record for Laramie, WY, it is necessary to consider all entries for Laramie, WY at some point.

The MapReduce paradigm makes it easy to implement programs such as the above. Processing is divided into two parts. The first part is the `map` function, which receives each of the initial key/value pairs, one at a time, processes the pair, and produces an arbitrary number of intermediate key/value pairs. These intermediate pairs use keys that may or may be completely different than the input keys. In the word counting example, the intermediate keys may be the same as the input keys, namely the words that are being counted. On the other hand, when looking for words that are used to link to URLs, the input keys are the words, but the intermediate keys are the URLs. The MapReduce framework leaves this entirely up to you, and that is one of the reasons why MapReduce is so widely applicable.

The second step is the `reduce` function, which combines all the entries for each intermediate key and produces zero or more final key/value pairs. As before, the final key/value pairs may use the same keys as the intermediate or initial key/value pairs—or they may use entirely different keys.

For example, consider the problem of counting words in a document. We can assume that the document has already been read, and that it has been broken up into key/value pairs where each key is a word and the value is always one. For instance, the Gettysburg address would be represented with the list

- (four . 1)
- (score . 1)
- (and . 1)
- (seven . 1)
- (years . 1)
- . . .
- (from . 1)
- (the . 1)
- (earth . 1)

Here we use the notation “(key . value)” to denote a single key/value pair.

Recall that the map function takes in an initial key and value, and it should return zero or more intermediate key/value pairs. For the wordcount program, map should return a single key/value pair, equal to the initial key and value.

$$map(k, v) = [(k.v)]$$

The reduce function accepts an intermediate key and a list of the values returned by map for that key. It should return a list of final key/value pairs. In the case of wordcount, reduce returns only one key/value pair, namely the key and the sum of the counts in the list.

$$reduce(k, vs) = [(k.sumlist(vs))]$$

The function sumlist, which adds all the elements of its input, is not defined, but you should be able to fill in the details.

The magic of MapReduce is now apparent. You only need to define the map and reduce functions as above. The MapReduce framework takes care of running the program in a single computer, or in a cluster of hundreds of computers, depending on the size of the problem.

What, exactly, does the MapReduce framework do? First, it takes the initial key/value pairs and splits them across many different machines. On each machine, it calls the map function on each of the key/value pairs that is assigned to that machine. As it does this, it combines the intermediate key/value pairs returned by each call to map into a single list. The lists from all of the machines are then combined.

On first thought, it may appear that all of the intermediate lists need to be combined. But actually, this is not the case. The reason is that the reduce function needs to see an intermediate key and all the values associated with that key. But different intermediate keys are independent, so they can be processed by different machines. What this means is that it is necessary to collect all of the intermediate values for each of the intermediate keys. This can help to distribute the calls to reduce across the entire cluster of machines running MapReduce.

Once all the values for a given intermediate key are collected, the MapReduce framework can call the reduce function on that intermediate key. The result is a list of final key/value pairs, and MapReduce collects all these results and returns them as the final answer.

As you can see, MapReduce is doing all of the work related to distributing the program across multiple machines. That is exactly the way it was designed, and it explains its growing popularity. You can develop a MapReduce program on your local computer, modifying it until it behaves exactly as you want it to on a small set of data. Then, you can submit the program to a large MapReduce cluster and run it on the entire data set.

Data Mining with MapReduce

Now that we have seen the basics of MapReduce, it is time to consider a larger example, something that illustrates how MapReduce is being used in practice. The application we will look at is a recommendation engine, a piece of code that is used to recommend new things to you based on other things you like. For example, when you visit a product page at Amazon.com, you will likely see a section called "Customers Who Bought This Item Also Bought" that recommends related items. Based on your past purchases and browsing habits, Amazon.com also builds a web page full of recommended items that is customized for you. How can Amazon.com do this?

The answer is easy to understand after we break the problem down into two components. First, Amazon.com needs to finds *customers like you.* In the case of a single product, this means other customers who have bought this product. In the more general sense, used to create custom recommendation lists for you, it means other customers who have bought many of the same items that you have purchased in the past. Once the group of customers like you is identified, the rest of the problem is easy. Each person in that group has made some purchases, so it is only necessary to find the most popular items in that group.

Finding the most popular items is essentially the same as counting words in a document. Amazon.com keeps a history of all the purchases that each of its customers have made. To process this list with MapReduce, think of it as consisting of entries of the form customer/item, meaning that the given customer bought that particular item. Similar to the wordcount program, the map operation produces intermediate entries of the type item/1, meaning the given item was bought (once). Such an entry should be generated for each purchased *made by a customer in the reference group.* That is, the map operation filters out the purchases made by customers who are not like you. The reduce operation is identical to wordcount's, but this time it counts the number of purchases for each item. The only remaining detail is to consider the results of the reduce operation and select the items that were purchased most often.

Unfortunately, that leaves the first problem, namely finding the group of customers who are most like you. This is the most critical aspect for generating useful recommendations. For instance, if you have only ever purchased gardening books from Amazon,com, you are likely to ignore a recommendation engine that alerts you to the latest novel in a long-running, young adult, vampire series. Worse, you may start thinking of the recommendations as unwarranted spam.

How can Amazon.com find customers just like you? Imagine, first of all, that you have rated all the purchases you have made, giving each item a grade between 0 (hated it) and 5 (loved it). To keep things simple, imagine that Amazon.com sells only two items. Then your ratings for these items can we expressed as a pair of numbers, say (2, 0). Now suppose that other customers have similarly rated the items. The customers who are most like you are precisely the ones whose ratings are close to your score or (2, 0). Effectively, your purchase history is represented by a point at coordinate (2, 0), and the customers who are most like you are have purchase histories that correspond to nearby points.

Of course, Amazon.com sells many more than two items! And neither you nor any of Amazon.com's other customers are likely to have rated even a fraction of them. But the principle stays the same. Instead of using pairs to represent your ratings, we need many more coordinates—as many as Amazon.com has products. But this is still just a point, albeit in a space with many dimensions. And the customers most like you will still be represented by points close to yours.

A remaining complication is that customers do not always explicitly rate the items they like, but this can be easily resolved by using implicit ratings. For example, if you buy an item, we can give it a rating of 4, unless you explicitly change it. And any product that you have never even looked at can be given a rating of 0.

So the problem is to find which points in this huge dimensional space are near your own. It is actually more useful to think of this a little differently. Instead of finding points near yours, think in terms of finding groups of points that are clustered together. One cluster, for example, may consist of avid gardeners, while another includes fans of young adult, vampiric fiction.

In general, finding clusters in a large dataset is a very complicated problem. But there are some useful approximations. For example, suppose that you expect to find 10 clusters. You could try to find them as follows:

1. Initially, guess at the location of each cluster. The guess can consist of the center point of each alleged cluster.

2. For each point in the dataset, decide which cluster center is nearest to the point. Split the points into clusters so that each point is in the cluster determined by the nearest center point.

3. Next, recalculate each cluster's center point by averaging all the points that were assigned to that cluster.

4. Repeat the previous two steps as many times as necessary to find the clusters.

The middle two steps can be implemented using MapReduce. The map operation can assign points to clusters, while the reduce function can computes the new center point of each cluster. Note that if we know the center points, it is a trivial matter to determine which cluster you belong to. And once we know that, we can determine, as we did before, which products are relevant to customers in your cluster.

We have to be a little careful with the definition of the map and reduce functions. In the earlier examples, the map function only had two arguments, a key and an associated value. But in this case, the map functions needs a purchase history (i.e., a point that represents an individual customer) and the current cluster center points. These center points should be the same for all purchase histories, so they should not be included in either the key or the value. Instead, we add an argument to the map function as follows.

$$map(hist, v, centers) = [(closest\_center(hist, centers).hist)]$$

As you can see, the result of a map operation is an intermediate key/value pair, where the key identifies a specific center point, and the value is the point representing the current history.

The MapReduce framework will collect all the intermediate values and call the reduce operation once for each key. So the reduce operation sees all the points in each cluster, and it can compute the new center point of

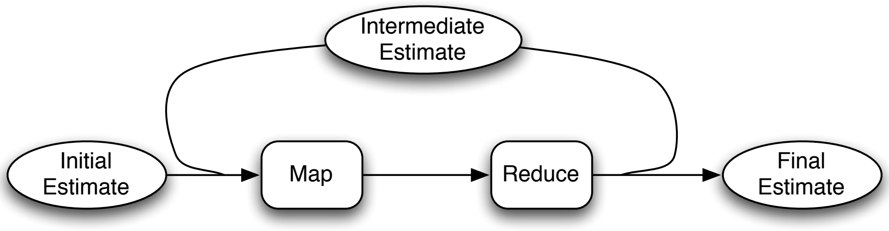


Figure 13.1: Iterative Map Reduce Operation

the cluster by averaging the points in the list. Although reduce does not need to see the original center points, we pass these to the reduce function to keep the symmetry with the map operation.

$$\begin{aligned}
reduce(&cluster,\\
&points, \quad = \quad [(cluster.average(points))]\\
&centers)\\
average(points) \quad &= \quad avg\_aux(points, (0,0), 0)\\
avg\_aux(points, sum, count) \quad &= \quad \begin{cases} sum/count, & \text{if } points = []\\ avg\_aux(rest(points), \\ \qquad sum + first(points), \\ \qquad count + 0), & \text{otherwise} \end{cases}
\end{aligned}$$

The auxiliary functions average and avg_aux are used to compute the new center point of the cluster. Notice that the addition and division operations are adding points, not just numbers.

The result of the reduce operation is the new list of center points. This should be in the same format as the initial guess. What this means is that the output of the reduce operation can be passed back as the initial guess for subsequent passes of the map step. This allows us to execute as many map and reduce operations as necessary to find the clusters, as illustrated in Figure 13.1.

Chapter 14

Sharding with Facebook

Chapter 15

Generating Art with Computers