How Computers Work:

Logic in Action

How Computers Work: Logic in Action

Rex Page
Ruben Gamboa

University of Oklahoma

Norman, OK.

Printed in the United States

01 10 09 08 07 06 05 04 03 02

First edition: 19 January 2011

Short contents

Contents

List of Figures

List of Asides

Part I

Introduction to Computers and Logic

One

Computer Systems: Simple Principles Lead to Complex Behavior

1.1 Hardware and Software

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 laptops 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

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 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 1.1: Models of Computation

is entirely different from, yet consistent with and equivalent to, the view of software as a list of instructions or as 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 hundreds of thousands of pixels, 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 speakers in an iPhone may be able to produce only a single tone at a time. But the software controlling it can produce 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.

1.2 Structure of a Program

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

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 to 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 1.1) 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 the familiar arithmetic functions (that is "operations") and the ability to define new functions 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 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 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.

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

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 1.2: Operations, Operands, Functions, and Arguments

But what about the input to the function `emily`? The value of a mathematical function is completely determined by its arguments. That's what it means to be a function! So if `emily` has no arguments, 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 make its selection by considering the previous round of the game, so the argument 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). In those models, programs store values in "variables" and can use those variables later. 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 an 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.

1.3 Deep Blue

Our model of computation is simple, but it has as much power to specify computations as any other one. 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 them. 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 required to deliver results, 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

$$r5 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

This is not an inductive definition. In fact, this "function" always produces the same value. It has no parameters, so not a function in the common sense of the word. But what about a function that computes the sum of the first N reciprocals? This calls for an inductive definition. As a step in that direction, observe that the sum of the first N reciprocals is $\frac{1}{N}$ plus the sum of the first $N - 1$ reciprocals. More formally, the sum, denoted by $r(n)$, satisfies the following equation.

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

The equation circular because it refers on both sides of the equation to a value delivered by the function r. However, the parameter in the circular part (that is, in the formula $r(n - 1)$ on the right-hand side of the equation) is closer to zero than the parameter on the left-hand side of the equation. And, in the specific case when the parameter on the left is zero, the equation is not circular.

This makes the equation a suitable definition of the function r, one that can be used to compute its value for any parameter that is a non-negative, whole number. One way to see this is to observe that you can compute $r(5)$ by computing $r(4)$ and adding $\frac{1}{5}$. Furthermore, you can compute $r(4)$ by computing $r(3)$ and adding $\frac{1}{4}$. Continuing this train of thought finally brings you to computing $r(0)$, and the equation specifies this value as zero. It does not involve the circular part of the equation. So, you can plug in the value 0 for $r(0)$, use that to compute $r(1)$ and so on up the line to $r(5)$. In the end you conclude that $r(5) = \frac{137}{60} \approx 2.3$.

This computational strategy determines the value $r(n)$, regardless of what number the parameter n stands for. Because the equation determines all the values of the function r, the equation can serve as a definition of the function. In this way the equation determines not only the values, $r(n)$ of the function, but also all of its other properties, such as the fact that the formula $r(n)$ must in all cases stand for a ratio of non-negative, whole numbers.

Surely it is surprising that all properties of a function can be derived from a definition that covers only a few special cases (in this example, two cases: $n = 0$ and $n > 0$), even when some of those cases involve circular references. But, that is the way it is. Every computable function has a definition of this form, and that is the way we will use this mathematical fact to 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 in our solar system 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 rather than all the way to the end of the game. In practice, *Deep Blue* is a complicated function, with a large definition. But in principle, it is a mathematical function that can be defined by specifying some of its properties, just as we did with the function, r, that adds up reciprocals. We will use definitions of this kind throughout the book. It will be our way of describing computations. That is, our way of programming computers.

EXERCISES

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

Ex. 2 — Define a function to compute $factor(k, n)$, which is true if k is a factor of n and false otherwise. Assume that the formula $mod(k, n)$ returns the remainder when dividing $k \div n$. For example, $mod(17, 5) = 2$, $mod(9, 2) = 1$, and $mod(12, 4) = 0$. Use the mod function in your definition of $factor$.

Ex. 3 — Define a function to compute $lf(n)$, the largest factor of n other than n itself. For example, $l(30) = 15$ and $l(15) = 5$. Assume that the formula $lft(n, k)$ returns the largest factor of n that doesn't exceed k (the largest factor up to k). For example, $lft(30, 7) = 6$. In your definition, use a formula like $lft(n, k)$, with appropriate choices for n and k.

Ex. 4 — Define a function to compute $p(n)$, which is true if n is a prime number and false otherwise. Refer to the function lf from the previous exercise in your definition.

Ex. 5 — Define a function to compute $rp(n)$, the sum of the reciprocals of all the prime numbers that are less than or equal to n. Your definition will be similar to that of the function r (page 8). That definition has two formulas, one for the case when n is zero, and the other for non-zero values of n. The new function will have a third case. It will refer to the function p from the previous exercise to avoid incorporating the reciprocals of numbers that are not primes.

Impress your friends with useless trivia: The number $r(n)$ grows without bound as n becomes large. The growth rate is about the same as that of $log(n)$, which isn't very fast. The number $rp(n)$ grows even more slowly, of course, but still grows without bound. However, the sum of the squares of the reciprocals is bounded. Leonhard Euler proved these facts over 200 years ago, during a kind of “Cambrian explosion” of mathematics initiated a hundred years earlier by Isaac Newton and Gottfried Leibniz with their invention of the infinitesimal calculus.

Two

Boolean Formulas and Equations

2.1 Reasoning with 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,

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 determines which formulas are well formed (that is, grammatically correct) and which are not. For example, the formula "$x + 3 \times (y + z)$" conforms to the grammar, so it is well formed (grammatically correct), but the non-formula "$x + 3 \times (y+) \times z$" is not.

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 2.1: 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}
$x \times y = y \times x$	{× commutative}
$x + (y + z) = (x + y) + z$	{+ associative}
$x \times (y \times z) = (x \times y) \times z$	{× associative}
$x \times (y + z) = (x \times y) + (x \times z)$	{distributive law}

Figure 2.1: Equations of Numeric Algebra

rather than numeric algebra, but our rule of inference, which is substitution of equals for equals, applies equally well in both Boolean and 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 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. We will derived the equation $(-1) \times (-1) = 1$ from other equations that you are familiar with and will surely accept without question.

The equations in Figure 2.1 (page 12) express some standard rules of numeric computation. In those equations, the letters stand in place of numbers. They can also stand in place of other formulas expressed in terms of common numeric operations (addition, multiplication, etc). So, the variable x stands for a grammatically correct formula, which could be a simple formula like "2", a more complicated formula like "$3 \times (y + 1)$, or some other formula, be more or less complex than these.

We refer to letters used in this way as "variables" even though, within a particular equation, they stand for a fixed number or other, particular formula. The formula associated with a variable, though unspecified, does not vary.

If we accept the equations of Figure 2.1 (page 12) 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 13) 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 12) to justify the transformation from the formula in the previous step. We are so accus-

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

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

tomed 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 13) 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), and 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 of the proof. 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 12). 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 rational consequence, accept the equation $(-1) \times (-1) = 1$.

Using this same kind of reasoning, we will derive new Boolean equations from a few, basic ones 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.

$x \vee False = x$	{$\vee$ 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}
$x \to y = (\neg x) \vee y$	{implication}
$\neg(x \vee y) = (\neg x) \wedge (\neg y)$	{$\vee$ DeMorgan}
$x \vee x = x$	{$\vee$ idempotent}
$x \to x = True$	{self-implication}
$\neg(\neg x) = x$	{double negation}

Figure 2.3: Basic Boolean equations (axioms)

EXERCISES

Ex. 6 — Use the equations of Figure 2.1 (page 12), together with the additional equation (1+1)=2, to derive the equation $(x + x) = (2 \times x)$.

Ex. 7 — Derive the following equation using using the equations of Figure 2.1.

$$((-1) \times x) + x = 0 \quad \{\times \text{ negation}\}$$

Ex. 8 — Derive the equation $((x+(((-1)\times(x+y))+z))+y) = z$ using using the equations of Figure 2.1. You may also use the {$\times$ negation} equation from the previous exercise.

2.2 BOOLEAN EQUATIONS

Let's start with the Boolean equations in Figure 2.3 (page 14). 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 ($\times$). Boolean formulas employ logic operations: logical-and ($\wedge$), logical-or ($\vee$), logical-negation ($\neg$), and implication ($\to$).

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 (page 14), 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

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 value for the first variable, there are choices for 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 ($\vee$) is the truth table for the formula $x \vee y$. That formula has two variables, so the truth table has four entries.

Aside 2.2: Truth Tables

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.

$$\begin{array}{lll} & False \vee False & \\ = & False & \{\vee \text{ identity}\} \ldots \text{taking } x \text{ in the axiom to stand for False} \end{array}$$

$$\begin{array}{lll} & False \vee True & \\ = & True & \{\vee \text{ null}\} \ldots \text{taking } x \text{ in the axiom to stand for False} \end{array}$$

…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 {$\vee$ identity} axiom in your proof of the third equation in the theorem and the {$\vee$ null} axiom in your proof of the fourth equation, right? We knew you could do it.

Derivations are usually more complicated, of course. For example, the following {$\vee$ 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 ({$\vee$ complement})**.** $(\neg x) \vee x = True$

Proof.

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

Q.E.D.

The {$\vee$ 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.

$$\begin{array}{lll} & \neg True & \\ = & \neg(False \to False) & \{\text{self-implication}\} \\ = & \neg((\neg False) \vee False) & \{\text{implication}\} \text{ (taking both } x \text{ and } y \text{ in the axiom to stand for } False\text{)} \\ = & \neg(\neg False) & \{\vee \text{ identity}\} \text{ (taking } x \text{ in the axiom to stand for } \neg False\text{)} \\ = & False & \{\text{double negation}\} \text{ (taking } x \text{ in the axiom to stand for False)} \end{array}$$

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

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 a 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 ($\to$) is a cornerstone of logic in real-world problems, but many people misunderstand its meaning (that is, its truth table).

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 and accurately.

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 2.3: Truth Tables and Feasibility

Citing proven theorems to prove new ones is similar to an idea known as "abstraction" that is a mainstay 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.

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 2.4: Abstraction

The {→ 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 14), but also equations from the {¬ 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 ({→ truth table})**.**
- $False \rightarrow False = True$
- $False \rightarrow True = True$
- $True \rightarrow False = False$
- $True \rightarrow True = True$

Proof.

$\quad False \rightarrow False$
$= \ (\neg False) \vee False$ {implication} ... taking both x and y in the axiom to stand for $False$
$= \ \neg False$ {$\vee$ identity}
$= \ True$ {$\neg$ truth table}

... 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 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 "$False \rightarrow y$" 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 $False$ 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 14) 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})**.** $x \wedge False = False$

Proof.

$\quad x \wedge False$
$= \ x \wedge (\neg True)$ {$\neg$ truth table}
$= \ (\neg(\neg x)) \wedge (\neg True)$ {double negation}
$= \ \neg((\neg x) \vee True)$ {$\vee$ DeMorgan} ... taking x in the axiom to stand for $(\neg x)$ and y in the axiom to stand for $True$
$= \ \neg True$ {$\vee$ null}
$= \ False$ {$\neg$ truth table}

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 give you a few to work out on your own, and go on to other topics. We're not abandoning proofs, though. Just taking a little break. It's going to be one proof after another, all the way down the line.

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 to 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.

EXERCISES

Ex. 9 — Use the equations of Figure 2.3 (page 14) and the ∧-absorption theorem (page 20), to derive the ∨-absorption equation: $(x \wedge y) \vee y = y$.

Ex. 10 — Derive the equation $(((\neg x) \wedge y) \vee (x \wedge (\neg y))) = ((x \vee y) \wedge (\neg (x \wedge y)))$ from the equations of Figure 2.3.

Ex. 11 — Derive the equation $(((\neg x) \wedge y) \vee (x \wedge (\neg y))) = (\neg ((x \rightarrow y) \wedge (y \rightarrow x)))$ from the equations of Figure 2.3.

Ex. 12 — Use the equations of Figure 2.3 (page 14) and the theorems of this section to derive the truth-table for the following formula $(x \vee ((\neg y) \wedge (\neg z)))$. *Note*: Since there are three variables in the formula, the truth table will have eight entries, and you will need to prove eight equations. Each equation will have on the left-hand side a different combination of $True/False$ values for the variables x, y, and z, and on the right-hand side will have the appropriate $True/False$ value for the formula.

2.3 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 lead to 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

v	{atomic}	
$(a \circ b)$	{bin-op}	*all grammatically correct formulas*
$(\neg a)$	{negation}	*must match one of these templates*
(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

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 a 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, 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 22), 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.

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 $((x \vee (\neg y)) \wedge (x \to z))$. 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* $((x \vee (\neg y)) \wedge (x \to z))$
a	$(x \vee (\neg y))$
$\circ$	$\wedge$
b	$(x \to z)$

$(x \vee False) = x$	{$\vee$ 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}
$(x \to y) = ((\neg x) \vee y)$	{implication}
$(\neg(x \vee y)) = ((\neg x) \wedge (\neg y))$	{$\vee$ DeMorgan}
$(x \vee x) = x$	{$\vee$ idempotent}
$(x \to x) = True$	{self-implication}
$(\neg(\neg x)) = x$	{double negation}
$((x)) = (x)$	{redundant grouping}
$(v) = v$	{atomic release}

requirements on symbols

- x, y, and z 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

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 $(x \to z)$ 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 15).

In the process of deriving the truth tables, we used the Boolean equations of Figure 2.3 (see page 14). 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 23) provides all of the information needed to determine the $True/False$ 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.

Exercises

Ex. 13 — Use the rules of grammar for Boolean formulas (Figure 2.4, page 22) 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. 14 — Derive the truth tables (see page 15) of the formulas from the previous exercise.

Ex. 15 — Use the axioms of Boolean algebra (Figure 2.5, page 23) to prove the following equations. After proving an equation, you may cite it in subsequent proofs.

$(x \to False) = (\neg x)$	{$\neg$ as $\to$}
$(\neg(x \wedge y)) = ((\neg x) \vee (\neg y))$	{$\wedge$ DeMorgan}
$(x \vee (\neg x)) = True$	{$\vee$ complement}
$(x \wedge (\neg x)) = False$	{$\wedge$ complement}
$(\neg True) = False$	{$\neg True$}
$(\neg False) = True$	{$\neg False$}
$(True \to x) = x$	{$\to$ identity}
$(x \wedge True) = x$	{$\wedge$ identity}
$(x \wedge y) = (y \wedge x)$	{$\wedge$ commutative}
$(x \wedge (y \wedge z)) = ((x \wedge y) \wedge z)$	{$\wedge$ associative}
$(x \wedge (y \vee z)) = ((x \wedge y) \vee (x \wedge z))$	{$\wedge$ distributive}
$(x \wedge x) = x$	{$\wedge$ idempotent}
$(x \to y) = ((\neg y) \to (\neg x))$	{contrapositive}
$(x \to (y \to z)) = ((x \wedge y) \to z)$	{Currying}
$((x \wedge y) \vee y) = y$	{$\vee$ absorption}
$((x \to y) \wedge (x \to z)) = (x \to (y \wedge z))$	{$\wedge$ implication}
$((x \to y) \wedge (x \to (\neg y))) = (\neg x)$	{absurdity}

2.4 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 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. 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 draw them 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.

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 23), the formula $(x \rightarrow y)$ has the same meaning as the formula $((\neg x) \vee y)$. 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 fabricate 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 2.5: No Gate for Implication

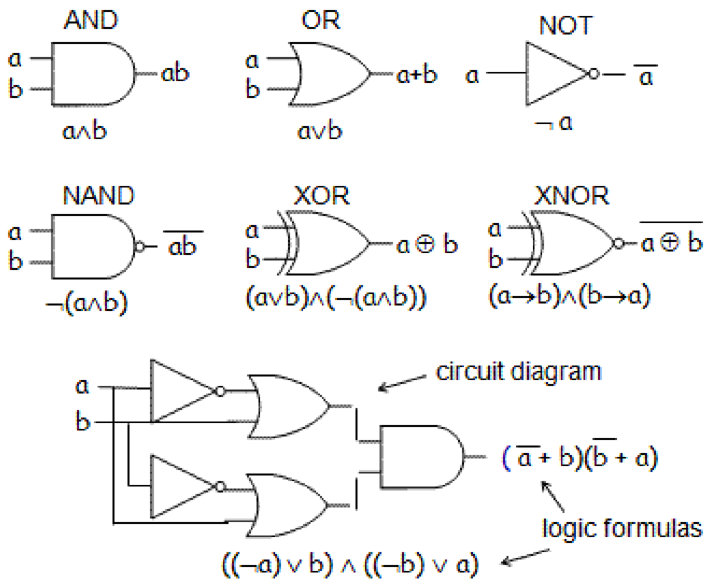


Figure 2.6: Digital Circuits = Logic Formulas

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 27) summarizes the relationships between the symbols for logic gates 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 used 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, 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, page 23) 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 {¬ as →} equation (see page 25) 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 (∧, ∨, and ¬) 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 {∧ idempotent} theorem (see page 25).

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

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 23).

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

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 29) displays the digital circuits that correspond to the equations verifying that nand is the only gate we really need.

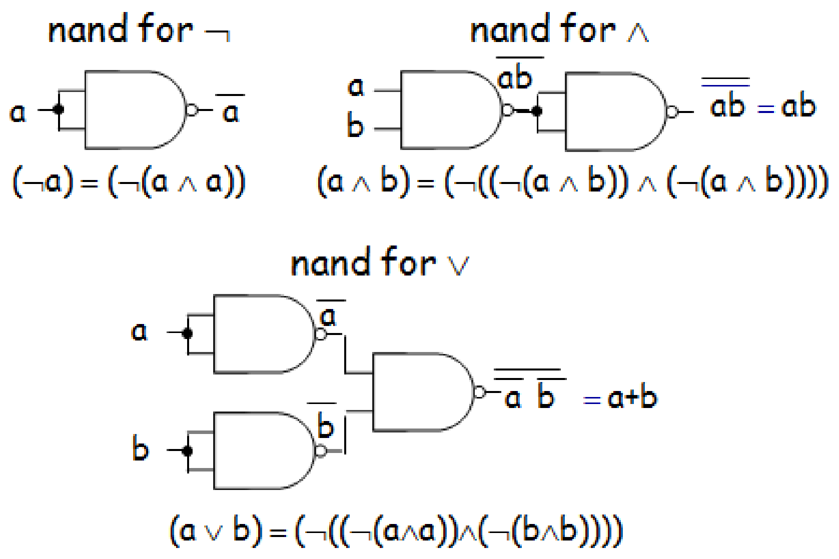

nand for ¬
a
$\overline{a}$
(¬a) = (¬(a ∧ a))
nand for ∧
a
b
$\overline{ab}$
$\overline{\overline{ab}}$ = ab
(a ∧ b) = (¬((¬(a ∧ b)) ∧ (¬(a ∧ b))))
nand for ∨
a
b
$\overline{a}$
$\overline{b}$
$\overline{\overline{a}\,\overline{b}}$ = a+b
(a ∨ b) = (¬((¬(a∧a))∧(¬(b∧b))))

Figure 2.7: Nand is All You Need

Exercises

Ex. 16 — 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 (Figure 2.5, page 23)).

Ex. 17 — For each of the following logic formulas, draw an equivalent circuit diagram. Since there are no "implication operators", they will need to be materialized in the form of the circuit diagram from the previous exercise.

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

Ex. 18 — 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. 19 — Draw circuit diagrams with behavior of the and-gate, or-gate, and negation-gate, but in these diagrams use implication circuits only—no other combinations of logic gates. Hint. You can specify that an input line on a gate in a circuit diagram carries a specific constant signal (0 or 1), rather than a variable.

2.5 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 18). 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 $((h_1 \wedge h_2 \wedge h_3) \to c)$, 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 25). 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 23) 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 by O'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 31). One of the rules, {$\to$ 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.

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

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

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

Figure 2.8: Rules of Inference for Reasoning by Deduction

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 $a \to b$ 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 $False$, 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 25), we can derive the other {$\wedge$ elimination} from Figure 2.8 (page 31).

The rule would be stated as follows.

$$\begin{array}{l}\text{Prove } a \wedge b\\ \hline \text{Infer } b\end{array}\quad \{\wedge\text{ elimination-2}\}$$

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

$$\begin{array}{ll}
\text{Prove } a \wedge b & \\ \cline{1-1}
 & \{\wedge \text{ commutative}\} \\
\text{Prove } b \wedge a & \\ \cline{1-1}
 & \{\wedge \text{ elimination}\} \\
\text{Infer } b &
\end{array}$$

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 {∧ 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.

$$\begin{array}{ll}
\text{Assume } ((a \to b) \wedge (b \to c)) & \\ \cline{1-1}
 & \{\wedge \text{ elimination}\} \\
\text{Infer } (a \to b) & \\
\text{Assume } a & \\ \cline{1-1}
 & \{\text{modus ponens}\} \\
\text{Infer } b & \\
\text{Assume } ((a \to b) \wedge (b \to c)) & \\ \cline{1-1}
 & \{\wedge \text{ elimination-2}\} \\
\text{Infer } (b \to c) & \\ \cline{1-1}
 & \{\text{modus ponens}\} \\
\text{Infer } c & \\ \cline{1-1}
 & \{\to \text{ introduction}\} \\
\text{Infer } (a \to c) & \\ \cline{1-1}
 & \{\to \text{ introduction}\} \\
\text{Infer } ((a \to b) \wedge (b \to c)) \to (a \to c) &
\end{array}$$

A tautology is a Boolean formula that has the value $True$ regardless of the values of its variables. Put another way, a tautology is a Boolean formula that is equal to the formula $True$.

Every theorem proved by deduction confirms a tautology. The tautological formula corresponding to a theorem is the implication with the logical-and of the hypotheses as the operand on the left of the implication and the conclusion of the theorem on the right. Since a tautology is a formula equal to $True$, every theorem leads to a corresponding Boolean equation.

The {→ chain} theorem that we just proved by deduction confirms that the formula derived from the theorem in this way 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 {→ chain} equation based on the axioms of Boolean algebra (Figure 2.5, page 23).

EXERCISES

Ex. 20 — Use the rules of inference for deduction (page 31) to derive the formula c from the formula $(a \wedge ((a \to b) \wedge (b \to c)))$.

Ex. 21 — Use the axioms of Boolean algebra (page 23) to derive the equation $((a \wedge ((a \to b) \wedge (b \to c))) \to c) = True$.

Ex. 22 — Explain the connection between the previous two exercises.

Three

Software Testing and Prefix Notation

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 8).

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

You could start with a few simple checks.

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

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.

$$\begin{aligned} r(n) &= (1 + \tfrac{1}{2} + \tfrac{1}{3} + \cdots + \tfrac{1}{(n-1)}) + \tfrac{1}{n} \\ &= r(n-1) + \tfrac{1}{n} \end{aligned}$$

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.

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

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

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

Proof Pad and DrACuLa ("Dracula" for short) are software development environments you can use to see logic in action. To install Proof pad, follow the instructions at `http://proofpad.org`. To install Dracula, follow the instructions at `http://www.ccs.neu.edu/home/cce/acl2/`. If that link fails, look for "dracula racket" or "Proof Pad" with your web search engine. They are well-known systems and should be easy to find. They are free, too.

These ACL2 development environments provide ways to develop, test, and run programs written in the ACL2 dialect of a programming language called Lisp. They also provide ways to use a mechanized logic that is part of the ACL2 system, which will 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 3.1: ACL2 development environments

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 the software should satisfy, and tell the computer to generate random data and make sure the software doesn't violate the equations.

The Proof Pad and Dracula environments have an automated testing facility 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 (+ x y).

That seems simple enough, but it gets gradually more unfamiliar as the formulas get more complicated. For example, a formula for the sum of x and thrice y, $(x + 3 * y)$, comes out as (+ x ($*$ 3 y)) 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 properties of the reciprocal sum function, r, discussed previously would take the following form.

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

A “natural number” is an whole number (that is, an integer) that is zero or bigger. Calling them “natural numbers” is a bit of a conceit. It took people a quarter million years or so to invent zero, and that’s just the first “natural” number.

Aside 3.2: Natural Numbers

The test itself comes next. The test specified in this property definition compares (r n) with zero when n is zero, but with (+ (r (- n 1) (/ 1 n)), when n isn’t zero. As you can see, the comparison operator, “=”, comes first (it’s prefix notation). Its first operand is the (r n) formula, and its second operand is one of two alternative formulas selected by an “if” operation.

The “if” operator takes three operands. It interprets the first operand as a Boolean value (true or false). If that operand is true, the “if” operator delivers the value specified in its second operand. Otherwise, it delivers the value specified in its third operand.

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. 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.

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

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

Let’s look at another example. Suppose the function “mod” delivers the remainder when dividing one number by another. (Think long division: divisor, dividend, quotient, remainder . . . third-grade stuff.) 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 does only one test at a time, and there is no random data involved, but is handy for running some sanity checks to make sure a function delivers the right answers for a few special cases. Here are a few more sanity checks, these with divisions by three.

```
(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 a more general property of the function. Since you know that the remainder doesn't change when you subtract the divisor from the dividend, you use that as a basis for some automated testing.

```
(defproperty mod-test
  (divisor-minus-1 :value (random-natural)
   dividend        :value (random-natural))
  (let* ((divisor (+ divisor-minus-1 1))) ; avoid zero divisor
    (= (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.

To see the testing facility in action, define the "mod-test" property in a .lisp file, and use Proof Pad or 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.

Another property of the "mod" function 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 uses the doublecheck testing facility to make sure the mod function delivers values in this range.

```
(defproperty mod-upper-limit-test
  (divisor-minus-1 :value (random-natural)
   dividend        :value (random-natural))
  (let* ((divisor (+ divisor-minus-1 1))) ; avoid zero divisor
    (< (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 fact about remainders in division is that they are non-negative integers (that is, natural numbers). We can use the logical-and operator ("and" is ACL2 for $\wedge$) to combine the upper-limit test with the natural-number test. in one property definition. The value of the formula "(natp x)" is true if x is a natural number and false if it isn't.

Axiom {*natp*}

$$(\forall x.((\text{natp } x) = (x \text{ is a natural number})))$$

Maybe you think “floor” is an odd name for a function that produces the quotient, but wait . . . it gets worse. Names a lot stranger than that will pop up soon enough. The floor function delivers the quotient. Get used to it. There’s no crying in baseball.

Aside 3.3: Think “floor” is an odd name?

```
(defproperty mod-range-test
  (divisor-minus-1 :value (random-natural)
   dividend        :value (random-natural))
  (let* ((divisor (+ divisor-minus-1 1))) ; avoid zero divisor
    (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. Similarly, we make sure the dividend isn’t zero by adding one.

```
(defproperty quotient-upper-limit-test
  (divisor-minus-2   :value (random-natural)
   dividend-minus-1  :value (random-natural))
  (let* ((divisor  (+ divisor-minus-2 2))   ; divisor  > 1
         (dividend (+ dividend-minus-1 1))) ; dividend > 0
    (= (+ (* 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. We hope by now you are starting to get comfortable with prefix notation. The exercises will give you a chance to practice.

```
(defproperty division-test
  (divisor-minus-1 :value (random-natural)
   dividend        :value (random-natural))
  (let* ((divisor (+ divisor-minus-1 1))) ; avoid zero divisor
    (= (+ (* divisor (floor dividend divisor))
          (mod dividend divisor))
       dividend)))
```

Exercises

Ex. 23 — Define a test of the floor operator that checks to make sure its value is a natural number when its operands are natural numbers, and the divisor (second operand) is not zero. Use Proof Pad or Dracula to run the test.

Ex. 24 — The "max" operator chooses the larger of two numbers: (max 2 7) is 7, (max 9 3) is 9. Define a doublecheck property that tests to make sure (max x y) is greater than or equal to both x and y. Use Proof Pad or Dracula to run tests of the property.

Ex. 25 — Define a doublecheck property to test the distributive law of arithmetic (Figure 2.1, page 12). Use Proof Pad or Dracula to run your test.

Ex. 26 — Modular arithmetic (sometimes called "clock arithmetic") deals with integers in a fixed range, $0 \ldots (m-1)$, where m is an integer greater than zero known as the "modulus". If x is an integer, the formula (x mod m) stands for the remainder in the division of x by m.
Modular arithmetic is consistent with ordinary arithmetic. That is, the sum of two numbers, mod m, is the same as the sum, mod m, of the corresponding numbers in the "mod m" range. The same is true of subtraction and multiplication.

$$((x+y) \text{ mod } m) = (((x \text{ mod } m) + (y \text{ mod } m)) \text{ mod } m)$$
$$((x-y) \text{ mod } m) = (((x \text{ mod } m) - (y \text{ mod } m)) \text{ mod } m)$$
$$((x \times y) \text{ mod } m) = (((x \text{ mod } m) \times (y \text{ mod } m)) \text{ mod } m)$$

The ACL2 operator "mod" converts numbers to the modular range for a given modulus. That is, (mod x m) delivers the remainder in the division of x by m. Define doublecheck properties to test the above properties of clock arithmetic. Use Proof Pad or Dracula to run your tests.

Four

Mathematical Induction

4.1 Predicates and Lists as Mathematical Objects

A sequence is an ordered list of elements. In fact, for our purposes, the terms "list" and "sequence" are synonyms, and we will more often use the term "list". Many things that computers do come down to keeping track of lists. That makes lists an important class of mathematical objects, so we will need a formal notation, including an algebra of formulas, to discuss lists with the level of mathematical precision required in specifications of computer hardware and software.

Formally, we will write lists as sequences of their elements, separated by spaces, with square brackets marking the beginning and end of the list. For example, "[8 3 7]" denotes the list with first element 8, second element 3, and third element 7, and "[9 8 3 7]" denotes a list with the same elements, plus an additional element "9" at the beginning. We use the symbol "nil" for the empty list (that is, the list with no elements). We use square brackets rather than round ones in formulas specifying lists, to avoid confusion with formulas that invoke operators. For example, "[4 7 9]" denotes a three-element list, while "(+ 7 9)" is a numeric formula representing the value "16". However, ACL2 does not employ this square-bracket notation. When it displays the list "[4 7 9]", it uses round brackets: "(4 7 9)".

That is, our "pencil and paper" formulas for lists employ square brackets to distinguish them from computational formulas, but when ACL2 displays lists, it does not make this distinction. It uses round brackets both for lists and for computational formulas.

The algebra of lists includes some basic operators. One of them, the list construction operator, "cons", inserts a new element at the beginning of a list. 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)))".

Let P be a predicate. The formula $\forall x.P(x)$ is false when there is at least one index x in the universe of discourse for which $P(x)$ is false. Otherwise, the $\forall$ quantification is true. If the universe of discourse is empty, there aren't any indexes at all, let alone one for which the predicate is false. Therefore, $\forall x.P(x)$ would have to be true. In other words, a $\forall$ quantification with an empty universe of discourse is true, by default.

Using a similar rationale, a $\exists$ quantification with an empty universe of discourse is false because it can only be true if there is at least one proposition in the predicate that has the value true.

Aside 4.1: Quantifier with Empty Universe

The operator "consp" checks for non-empty lists. That is, 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*}
$(\forall xs.(\text{(consp } xs) = (\exists y.\ (\exists ys.\ (xs = \text{(cons } y\ ys))))))$

Wait a minute! Where do those funny-looking symbols come from. What do upside-down "A" and backwards "E" mean? Let's start with the backwards "E". The operator "$\exists$" converts a collection of true/false formulas into a single true/false formula.

We will use the term "proposition" to mean "true/false formula". So, "$\exists$" converts a collection of propositions into a single proposition. The collection of propositions is indexed (or "parameterized") by a variable that ranges across a "universe of discourse." For each element in the universe of discourse, there is a corresponding proposition in the collection.

Any collection of propositions indexed by the elements of a universe of discourse is called, when the collection is taken as a whole, a "predicate". For example let $P(xs, y, ys)$ be shorthand for the equation xs = (cons y ys). Given a particular list xs together with a particular entity y, we can view the equation $P(xs, y, ys)$ as a collection of propositions indexed by the variable ys, whose universe of discourse is a collection of lists. In this collection of propositions, the one corresponding to the list ys is the equation that $P(xs, y, ys)$ stands for. If that equation holds, the value of $P(xs, y, ys)$ is true. Otherwise, it's false.

For example, if xs denotes the list "[1 2 3]" and y denotes the object "1", then $P(xs, y, ys)$ stands for $P($[1 2 3], 1, $ys)$ which is an equation involving the variable ys. There is one such equation for each possible list ys. Taken all together those equations comprise a predicate.

The formula $(\exists ys.P($(1 2 3), 1, $ys))$ denotes true if there is some list ys for which the equation (1 2 3) = (cons 1 ys) is valid. If there were no such list, the formula $(\exists ys.P($(1 2 3), 1, $ys))$ would denote false. In this case, $(\exists ys.P($(1 2 3), 1, $ys))$ denotes true because when ys is the list "[2 3]", the formula $P($[1 2 3], 1, $ys)$ stands for the equation [1 2 3] $=$ (cons 1 [2 3]), which is true.

If, on the other hand, xs were the list "[1 2 3]" and y were the number "2", there would be no list ys that would make the equation [1 2 3] = (cons 2 ys) valid because the list on the left-hand side of the equation starts with 1 the the one on the right-hand side starts with 2. So, the formula $(\exists ys.P($[1 2 3], 2, $ys))$ is false.

More generally, if we take xs to stand a particular list and y to stand for a particular object, then $P(xs, y, ys)$ is an equation that is either true or false. That makes $P(xs, y, ys)$ a different proposition for each possible value of the variable ys, and $(\exists ys.P(xs, y, ys))$, which is shorthand for $(\exists ys.$ (xs = (cons y ys))), is true if there is a list ys that makes the equation xs = (cons y ys) valid and false if there is no such list ys.

We refer to the variable ys in the formula $(\exists ys.P($[1 2 3], 2, $ys))$ as the "bound variable" corresponding to the $\exists$ operator in the formula. Likewise, ys is the bound variable corresponding to the $\exists$ operator in the formula $(\exists ys.$ (xs = (cons y ys))).

Every $\exists$ operator that occurs in a formula must be followed immediately by a variable, then a period. The variable between $\exists$ operator and the period is known as the "bound variable" associated with the $\exists$ operator. The $\exists$ operator "quantifies" the formula after the period with respect to the universe of discourse of the variable before the period.

Now, let's take a step back. We can view the formula $(\exists ys.$ (xs = (cons y ys))) as a collection of propositions, one for each object y from the universe of discourse for y. The formula $(\exists ys.P(xs, y, ys))$ is a shorthand for that collection of propositions. Since any collection of propositions is a predicate, we can view $(\exists ys.P(xs, y, ys))$ as a predicate indexed by the objects that y can stand for (that is, the objects in the universe of discourse for y).

We can convert the predicate $(\exists ys.P(xs, y, ys))$ into a true/false value (that is, convert it to a proposition) by applying the $\exists$ operator again, but this time with y as the bound variable: $(\exists y.(\exists ys.P(xs, y, ys)))$. Again, this formula is a different proposition for each possible list that xs can stand for. This collection of propositions is a predicate, and we can apply a quantifier to the predicate to convert it to a proposition.

The only quantifier we've discussed is the $\exists$ operator. We could use it again, but we don't want to. We want to use a new quantifier, the one signified the the upside-down "A": the $\forall$ operator. If Q is a predicate and x is a variable standing for an element of the universe of discourse of Q, then the formula $(\forall x.Q(x))$ stands for true if there are no elements x in the universe of discourse for which $Q(x)$ is false. Otherwise, that is if there is a value x such that $Q(x)$ is false, then $(\forall x.Q(x))$ is false.

When we apply $\forall$ quantification to the formula $(\exists y.(\exists ys.P(xs, y, ys)))$, we get the formula $(\forall xs.(\exists y.(\exists ys.P(xs, y, ys))))$. Of course, that formula is shorthand for $(\forall xs.(\exists y.$ $(\exists ys.$ (xs = (cons y ys))))), which is the formula defining the {*cons*} axiom. The meaning of the {*cons*} axiom, therefore, is that the formula (cons xs) always has the same value as the formula $(\exists y.$ $(\exists ys.$ (xs = (cons y ys)))), regardless of what list xs denotes.

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 variables appearing in subscripts are natural numbers.)

Axiom {*cons*} (informal version)

$$[x_1 x_2 \ldots x_{n+1}] = (\text{cons } x_1 [x_2 \ldots x_{n+1}])$$

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 ({*fst*} and {*rst*}), 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	{*fst*}
(rest (cons x xs)) = xs	{*rst*}
(first nil) = nil	{*fst0*}
(rest nil) = nil	{*rst0*}

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

The axioms {*fst0*} and {*rst0*} cover a situation that could be regarded as illegitimate. What should the first element of an empty list be, anyway? There is no first element. The same goes for the formula (rest nil). How does it make sense to drop an element from a list that has none? Nevertheless, the axioms provide values for the formulas (first nil) and (rest nil), which legitimizes the formulas. The other axioms provide no interpretation for those formulas, so the extra axioms, {*fst0*} and {*rst0*}, cannot damage the validity of our mathematical reasoning by introducing inconsistencies. No harm, no foul.

We will use equations like the ones in these axioms in the same way we used the logic equations in Figure 2.2 (page 13) and the arithmetic equations of Figure 2.1 (page 12). 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 {*fst*} to replace "(first (cons x xs))" by the simpler formula "x". Vice versa, we can also cite equation {*fst*} 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 {*rst*} to justify replacing the formula "(rest (cons x xs))" by "xs" and vice versa, regardless of what formulas the symbols "x" and "xs" 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}, {fst}, and {rst} axioms. 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)
         (+ (len (rest xs)) 1)
         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)) = (+ 1 (len xs))	{*len1*}

We expect the "len" operator to deliver a natural number, regardless of what its argument is. For the record, we state this property 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 (page 38). It delivers true if its argument is a natural number and false otherwise.

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

We can derive this property of len from its axioms, but instead of plodding through that derivation at this point, we are going to proceed to some more interesting issues. A related fact is that the formula (consp xs) always has the same value as the formula (> (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*=len>0}
$\forall xs.$((consp xs) $=$ (> (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 other operators, 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 subscripted variables. The number of subscripts in the 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.

(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.

What is the single-quote mark doing in the formula ’(1 2 3 4)? This is how ACL2 avoids confusing lists with computational formulas. Our pencil-and-paper notation uses square brackets to make this distinction, but ACL2 uses the single-quote. 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 an ACL2 formula, rather than in a paper-and-pencil 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 formula would make no sense.

Aside 4.2: Single-quote Shorthand for Lists

```
(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) = ys. 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 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)))
```

The append-test property 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.

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.

Aside 4.3: "equal" vs "="

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 the append operation in different situations. One of them specifies the meaning when the first argument is the empty list, the other when the list has one or more elements (that is, when the list is constructed by the cons operator).

Axioms: {*append*}

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

These equations about the append operation are 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 some 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). That makes L a predicate whose universe of discourse is the natural numbers.

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.

L(0) ≡ (= (len (append nil ys))
(+ (len nil) (len ys)))
L(1) ≡ (= (len (append [x_1] ys))
(+ (len [x_1]) (len ys)))
L(2) ≡ (= (len (append [x_1 x_2] ys)
(+ (len [x_1 x_2]) (len ys)))
L(3) ≡ (= (len (append [x_1 x_2 x_3] ys))
(+ (len [x_1 x_2 x_3]) (len ys)))
L(4) ≡ (= (len (append [x_1 x_2 x_3 x_4] ys))
(+ (len [x_1 x_2 x_3 x_4]) (len ys)))

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).

$$\begin{array}{rll}
 & \text{(len (append nil } ys\text{))} & \\
= & \text{(len } ys\text{)} & \{\textit{app0}\}\text{ (page 47)} \\
= & \text{(+ (len } ys\text{) 0)} & \{+\text{ identity}\}\text{ (page 12)} \\
= & \text{(+ 0 (len } ys\text{))} & \{+\text{ commutative}\}\text{ (page 12)} \\
= & \text{(+ (len nil) (len } ys\text{))} & \{\textit{len0}\}\text{ (page 45)}
\end{array}$$

That was easy. How about L(1)?

$$\begin{array}{rll}
 & \text{(len (append [}x_1\text{] } ys\text{))} & \\
= & \text{(len (append (cons } x_1 \text{ nil) } ys\text{)} & \{\textit{cons}\}\text{ (44)} \\
= & \text{(len (cons } x_1 \text{ (append nil } ys\text{)))} & \{\textit{app1}\} \\
= & \text{(+ 1 (len (append nil } ys\text{)))} & \{\textit{len1}\} \\
= & \text{(+ 1 (+ (len nil) (len } ys\text{)))} & \{\text{L(0)}\} \\
= & \text{(+ (+ 1 (len nil)) (len } ys\text{))} & \{+\text{ associative}\}\text{ (page 12)} \\
= & \text{(+ (len (cons } x_1 \text{ nil)) (len } ys\text{))} & \{\textit{len1}\} \\
= & \text{(+ (len [}x_1\text{] (len } ys\text{))} & \{\textit{cons}\}
\end{array}$$

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

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

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.

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

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

Figure 4.1: Mathematical Induction–a rule of inference

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. That is, we think we know that the formula ($\forall n$.L(n)) is true. However, to complete proof of that formula, 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 proposition, then mathematical induction is an applicable inference rule in a proof that is ($\forall n$.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 inverse is also true: mathematical induction cannot help if the universe of discourse is not the natural numbers.

The rule goes as follows: one can infer the truth of ($\forall n$.P(n)) from proofs of two other propositions. Those two propositions are P(0) and ($\forall n$.(P(n)→P($n+1$))). It's a very good deal if you think about it. A direct proof of ($\forall n$.P(n)) would require a proof of proposition 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 P(n) is true in the proof of P($n+1$) is because the goal is to prove that the formula P(n)→P($n+1$) has the value "true". We know from the truth table of the implication operator (page 18) that the implication P(n)→P($n+1$) is true when P(n) is false, regardless of the value of P($n+1$). So, we only need to verify that the formula is true when P(n) is true. The implication will be true in this case only if P($n+1$) is true. So, we need to prove that P($n+1$) under the assumption that P(n) is true.

That is, in the proof of P($n+1$), you can cite P(n) to justify any step in the proof. P(n) gives you a leg up in the proof of 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 (page 47).

We have already proved L(0). All that is left is to prove ($\forall n$.(L(n)→L($n+1$))). That is, we have to derive L($n+1$) from 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).

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

This completes the proof by mathematical induction of the additive law of concatenation.

Theorem {*additive law of concatenation*}
$\forall n$.((len (append [$x_1\ x_2 \ldots x_n$] ys)) = (+ (len [$x_1\ x_2 \ldots x_n$]) (len ys)))

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 derive 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 axioms (page 47). Those axioms state properties of the append operation in two separate 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 of the append axioms 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 axioms (page 47). The keys to an inductive definition of an operator are listed in Figure 4.2 (page 51). 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 meth-

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	(1) There is at least one non-inductive equation (that is, an equation in which the operator being defined occurs only on the left-hand side). (2) All invocations of the operator on the right-hand side of inductive equations 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 inductive equation.

Figure 4.2: Key to Inductive Definition: The Three C's

ods cannot be derived using classicl 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 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*} (*Note*: n stands for a natural number)

(nthcdr 0 xs) = xs	{*sfx0*}
(nthcdr (+ n 1) xs) = (nthcdr n (rest xs))	{*sfx1*}

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 case number n. There will be one case for each natural number.

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

If S(n) is 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 ... x_n] is empty, which is normally denoted by the symbol "nil". So, S(0) stands for the following equation.

```
S(0) ≡ (equal   (nthcdr (len nil) (append nil ys))
                ys)
```

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.

```
     (nthcdr (len nil) (append nil ys))
  =  (nthcdr (len nil) ys)                   {app0} (page 47)
  =  (nthcdr 0 ys)                           {len0} (page 45)
  =  ys                                      {sfx0}
```

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

```
S(n + 1) ≡ (equal   (nthcdr   (len [x1 x2 ... x(n+1)])
                              (append [x1 x2 ... x(n+1)] ys))
                    ys)
```

```
     (nthcdr   (len [x1 x2 ... x(n+1)])
               (append [x1 x2 ... x(n+1)] ys))
  =  (nthcdr   (len (cons x1 [x2 ... x(n+1)])))        {cons} (page 43)
               (append (cons x1 [x2 ... x(n+1)]) ys))  {cons}
  =  (nthcdr   (+ 1 (len [x2 ... x(n+1)]))             {len1} (page 45)
               (cons x1 (append [x2 ... x(n+1)]) ys))  {app1} (page 47)
  =  (nthcdr   (+ (len [x2 ... x(n+1)]) 1)             {+ commutative} (page 12)
               (cons x1 (append [x2 ... x(n+1)]) ys))
  =  (nthcdr   (len [x2 ... x(n+1)])                   {sfx1}
               (append [x2 ... x(n+1)] ys))
  =  ys                                                {S(n)}
```

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 ... x_n] in the definition of S(n) to stand for any list with n elements. The elements in the list [x_2 ... 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 ... x_n] denotes a list with max($n - m + 1$, 0) elements {*nlst*}

At this point, we know that (append xs ys) delivers a list that has the right elements at the end. How about the beginning? We expect the concatenation to start with the elements

Axioms {*prefix*}	
(prefix 0 xs) = nil	{*pfx0 a*}
(prefix n nil) = nil	{*pfx0 b*}
(prefix (+ n 1) (cons x xs)) = (cons x (prefix n xs))	{*pfx1*}

of the list xs, so if we extract the first n elements of (append xs ys), where n is (len xs), we would expect to get a list identical to xs. To express this expectation formally, we need a function that, given a number n and a list xs, delivers the first n elements of xs. Let's call that function "prefix" and think about properties it would have to satisfy.

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.

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.

P(n) $\equiv$ (equal (prefix (len [x_1 x_2 $\ldots x_n$])
                                (append [x_1 x_2 $\ldots x_n$] ys))
                [x_1 x_2 $\ldots x_n$])

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) $\equiv$ (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*} (page 45)
=	nil	{*pfx0*}

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

P($n + 1$) $\equiv$ (equal (prefix (len [x_1 x_2 $\ldots x_{n+1}$])
                                (append [x_1 x_2 $\ldots x_{n+1}$] ys))
                [x_1 x_2 $\ldots x_{n+1}$])

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

= {*cons*} (page 43)
(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 45) {*app1*} (page 47)
(prefix (+ 1 (len [$x_2 \ldots x_{n+1}$]))
    (cons x_1 (append [$x_2 \ldots x_{n+1}$] ys)))

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

= {*fst*} (page 44) {*arithmetic*} {*rst*} (page 44)
(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 43)

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. There 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 the third list at the end.

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

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

$$\text{A}(n) \equiv \quad \text{(equal} \quad \begin{array}{l}\text{(append } [x_1\ x_2 \ldots x_n] \text{ (append } ys\ zs))\\ \text{(append (append } [x_1\ x_2 \ldots x_n]\ ys)\ zs)\end{array}$$

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.

EXERCISES

Ex. 27 — Prove the {*app-assoc*} theorem (page 54).

Ex. 28 — Assume the following axioms {*expt0*} and {*expt1*}.

Axioms {*expt*}

(expt x 0) = 1	{*expt0*}
(expt x (+ n 1)) = ($*$ x (expt x n))	{*expt1*}

x must be a number
n must be a natural number
($*$ x y) = $x \times y$

Prove the following theorem {*expt*}, assuming n is a natural number and x is a number.

Theorem {*expt*}
(expt x n) = x^n

Ex. 29 — Suppose the function rep is defined as follows.

```
(defun rep (n x)
  (if (zp n)
      nil                         ; {rep0}
      (cons x (rep (- n 1) x)))) ; {rep1}
```

Prove the following theorem {*rep-len*}. In the theorem, n can be any natural number and x can be any entity.

Theorem {*rep-len*}
(len (rep n x)) = n

Ex. 30 — Assume the following axioms {*mem0*} and {*mem1*}.

Axioms {*member-equal*}

(member-equal y nil) = nil	{*mem0*}
(member-equal y (cons x xs)) = (equal y x) $\vee$ (member-equal y xs))	{*mem1*}

Prove the following theorem {*rep-mem*}

Theorem {*rep-mem*}
(member-equal y (rep n x)) $\rightarrow$ (member-equal y (cons x nil))

Ex. 31 — Prove the following theorem {*drop-all0*}. In the theorem, xs can be any list.

Theorem {*drop-all0*}
(len (nthcdr (len xs) xs)) = 0

Hint: For each natural number n, let $C(n)$ stand for the following equation.

$$C(n) \equiv ((\text{len (nthcdr (len}[x_1\ x_2 \ldots x_n])\ [x_1\ x_2 \ldots x_n])) = 0)$$

Use mathematical induction to prove that the proposition $(\forall n. C(n))$ is true. Since the list $[x_1\ x_2 \ldots x_n]$ could be any list, the truth of $(\forall n. C(n))$ means that the equation (len (nthcdr (len xs) xs)) = 0 is true.

Ex. 32 — Assume that Theorem {*drop-all0*} (page 56) has been proven. Prove the following theorem {*drop-all*}. In the theorem n can be any natural number.

Theorem {*drop-all*}
(len (nthcdr (+ (len xs) n) xs)) = 0

Hint: Prove, by induction on n, the following equations, {*sfx-additive*} and {*sfx-nil*}. You can then cite those equations in your proof of {*drop-all*}.

(nthcdr (+ m n) xs) = (nthcdr m (nthcdr n xs)) {*sfx-additive*}
(nthcdr n nil) = nil {*sfx-nil*}

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 Proof Pad and 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 (page 50). Our statement of the theorem asserted that a proposition L(n) is true, regardless of which natural number n stands for: ($\forall n$.L(n)). L(n) is a shorthand for the following formula:

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

We could have used the doublecheck facility of Proof Pad or 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 (52). 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, use Proof Pad or Dracula to ask ACL2 to prove the above theorem. ACL2 will succeed in this case.

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 ($\forall n$.S(n)), where S(n) is a shorthand for the following formula.

S(n) ≡ (equal (nthcdr (len [x_1 x_2 … x_n])
(append [x_1 x_2 … x_n] ys))
ys)

The following definition specifies this theorem in ACL2 notation.

A theorem that takes the form of an implication, $x \rightarrow y$, says that the conclusion, y, will be true with the hypothesis, x, is true, but says nothing about the status of the conclusion when the hypothesis is false. The ACL2 equivalent of the Boolean formula $x \rightarrow y$ is (implies x y). For example, one can conclude that $u - 1 < v - 1$ if one knows that $u < v$. In ACL2 this fact would be stated as an implication.

```
(defthm simple-example
  (implies (< u v)
           (< (- u 1) (- v 1))))
```

Aside 4.4: Implication: a way to constrain conclusion

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

ACL2 can prove this theorem, but, as with the {*app-pfx*} theorem (page 54), the proof requires knowing some equations from numeric algebra, so it is necessary to import those theorems from the "arithmetic-3/top" book. The following include-book command makes that theory accessible to the mechanized logic.

(include-book "arithmetic-3/top" :dir :system)

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.

Exercises

Ex. 33 — Use Proof Pad or Dracula to run the {*app-pfx*} theorem (page 54) through the ACL2 mechanized logic. Don't forget to import the equations of numeric algebra contained in the "arithmetic-3/top" book (page 58).

Ex. 34 — Define the {*append associativity*} theorem in ACL2 notation, and use Proof Pad or Dracula to run it through the ACL2 mechanized logic. If you state the theorem correctly, ACL2 will succeed in proving it.

Ex. 35 — Define the {*rep-len*} theorem (page 55) in ACL2 notation, and use Proof Pad or Dracula to run it through the ACL2 mechanized logic.
Note: You will need to state the theorem as an implication (page 58) constraining n, the first argument of rep, to be a natural number. If you state the theorem correctly, ACL2 will succeed in proving it.

Ex. 36 — Define the {*drop-all0*} theorem (page 56) in ACL2 notation, and use Proof Pad or Dracula to run it through the ACL2 mechanized logic. ACL2 will succeed if you import the equations of numeric algebra contained in the "arithmetic-3/top" book (page 58).

Ex. 37 — Define the {*drop-all*} theorem (page 56) in ACL2 notation, and use Proof Pad or Dracula to run it through the ACL2 mechanized logic. The theorem must be stated as an implication constraining n to be a natural number, and ACL2 will need to have access to the equations of numeric algebra, as in the previous exercise.

Part II

Computer Arithmetic

Five

Binary Numerals

5.1 Numbers and Numerals

Numbers are mathematical objects with certain properties, and they come with operators, such as addition and multiplication, that produce new numbers from numeric operands. Because numbers are mathematical objects, they are ephemeral. You can't really get your hands on them. They are figments of the imagination.

However, numbers are useful and to deal with them, we need to write them down some way—decimal numerals, for example. The numeral 144 stands for the number of eggs in a dozen cartons of eggs. The numeral 1215 stands for the number of years between the twenty-seventh year of the reign of Caesar Augustus and the signing of the Magna Carta.

However, things numerals like 144 and 1215 are numerals. They are not numbers, but instead are symbols that stand for numbers, and they are not the only symbols we use for that purpose. The symbols CXLIV and MCCXV stand for the same two numbers. So do the symbols 90_{16} and $4BF_{16}$. The other symbols are Roman numerals and hexadecimal numerals. The symbols 144 and 1215 are decimal numerals, which is the representation most people turn to when they do arithmetic.

The decimal representation is, in fact, so embedded in our experience and practice that we often confuse the symbol with the number. In fact, the dictionary lists "number" and "numeral" as synonyms. Usually, there is no harm in considering them to be the same thing, but we are going to use numerals to do arithmetic in a mechanized way, and we will be careful to separate numbers as mathematical objects from the symbols we use to represent them. We will refer to the mathematical object as a "number" and to the symbol representing it as a "numeral".

Let's think about how we interpret a decimal numeral as a number. Take the numeral 1215, for example. Each digit in the numeral has a different interpretation. The first digit is the number of thousands in the number that 1215 stands for. The second tells us the number of hundreds, then the tens, and finally the units. The following formula is a way to express this interpretation.

$$1 \times 10^3 + 2 \times 10^2 + 1 \times 10^1 + 5 \times 10^0$$

This formula computes a number from the individual digits in the numeral using standard arithmetic operations (addition, multiplication, and exponentiation). It shows us what the individual digits in the numeral stand for, and gives us a leg up up on figuring out other kinds of numerals. The digits in the hexadecimal numeral have a similar

Perhaps you noticed a subtle confusion in the formulas we use to explain the meaning of numerals. At first, we claim that 1215 is merely a symbol standing for a mathematical object. And, we claim that the digit 2 is merely a symbol standing for the number of items in a pair, along with similar claims for the digits 1 and 5. Then, we use those symbols in the formula $1 \times 10^3 + 2 \times 10^2 + 1 \times 10^1 + 5 \times 10^0$ as if they were numbers.

There is some slight of hand going on here. We are trapped by our terminology. Numbers as mathematical objects are figments of our imagination, but when we write formulas, we have to choose some symbols to represent them. So, in the formula $1\times10^3+2\times10^2+1\times10^1+5\times10^0$, we use the symbols 1, 10, 3, 2, 5, and 0 as if they were numbers. But, in the numeral 1215, the symbols 1, 2, and 5 are not numbers. They are symbols standing for numbers.

It's even worse with the hexadecimal numeral $4BF_{16}$ and the formula $4 \times 16^2 + 11 \times 16^1 + 15 \times 16^0$. In the formula we have rewritten the symbol "B" as the decimal numeral 11 and the symbol "F" as the decimal numeral 15. And, we've had the temerity to pretend that symbols in the formula are numbers when they are really decimal numerals, just as we did in the formula that was supposed to explain the meaning of the decimal numeral 1215.

Furthermore, we've really mixed things up in the numeral $4BF_{16}$ because the "4BF" part is in hexadecimal notation and the "16" part is a decimal numeral indicating that we are to interpret the digits in base sixteen rather than the conventional base ten.

Try to get your head around this slight of hand. We're more-or-less stuck with it. Figments of our imagination have to be materialized, somehow, if we are going to talk about them.

Aside 5.1: Digits as Numbers

meaning, but with a different basis. Decimal numerals are based on powers of ten, but hexadecimal numerals are based on powers of sixteen.

The system of decimal numerals calls of ten different symbols to represent digits, and we use the symbols 0, 1, 2, . . . 9 for this purpose. The hexadecimal system calls for sixteen different symbols, and we use 0, 1, 2, . . . 9, A, B, C, D, E, F to represent them. The digits stand for the customary numbers (0 for zero, 2 for two, and so on). The letters stand for the numbers beyond nine.

There are no conventional squiggles for digits beyond 9. The letters A, B, . . . F were chosen for this purpose arbitrarily. So, in hexadecimal notation, "A" stands for ten, "B" for eleven, and so on up to "F" for fifteen. That leads to the following formula to express the meaning of the hexadecimal numeral $4BF_{16}$. (Remember, B stands for 11, F for 15.)

$$4 \times 16^2 + 11 \times 16^1 + 15 \times 16^0$$

Formulas like $1\times10^3+2\times10^2+1\times10^1+5\times10^0$ convert numerals to numbers. No doubt you could use this example to construct the appropriate formula for any given numeral.

What! The function delivers the digits backwards! Why is that?

Of course, the function could have delivered the digits in the customary order, but we have a reason for encoding the numeral backwards in the sequence that we are using to represent it. The reason will become apparent soon, but for now, just get used to it. We write numerals like “1215” in the usual way, but the dgts function delivers them in the form of a sequence “[5 1 2 1]” in which the digits appear in the reverse order.

Besides being backwards, the elements in the sequence are numbers, not symbolic digits. That is another part of our slight of hand. We are using numbers to represent the digit symbols. We could convert them to pure symbols, but at this point, we are better off leaving them in the form of numbers because we will want to use them as numbers later.

By the way, the sequence notation “[5 1 2 1]” is the symbol we use to describe the sequence, but the sequence itself is another kind of mathematical object. That is, it’s another figment of our imagination. The “real” object is ephemeral in the same sense as numbers.

Aside 5.2: Numerals as Sequences . . . Backwards

Base 10 numerals for sure, and hex numerals, too, and probably any other base.

We’ll say more about to converting numerals to numbers later, but what about going the other direction, converting numbers to numerals? Suppose someone gives you an operator that is supposed to mechanize the conversion of a number to a decimal numeral. The name of the operator is “dgts”, so the formula “(dgts 1215)” would deliver the sequence “[5 1 2 1]”. That is, dgts delivers a list of the decimal digits. The digits are in reverse order in the list, but they are the right digits.

Before you make heavy use of the operator, you will want to check it out. You might try it on a few numbers, for a start.

```
(check-expect (dgts 1215) '(5 1 2 1))
(check-expect (dgts 1964) '(4 6 9 1))
(check-expect (dgts 12345) '(5 4 3 2 1))
(check-expect (dgts 0) '())
```

Wait a minute! Why does (dgts 0) deliver the empty sequence instead of “[0]”? That’s another little trick. Besides delivering the digits in reverse order, leading zeros are always omitted. We could write the numeral “1964” with as many leading zeros as we like. The numerals “01964” and “000001964” also stand for the number 1964. Those numerals would be “[4 6 9 1 0]” and “[4 6 9 1 0 0 0 0 0]” in the reverse order that the dgts operator delivers.

However, dgts never includes any leading zeros in the numerals it delivers. It leaves them all off, even for the number zero. That’s why (dgts 0) is nil. Of course (dgts 012345) would be the same as (dgts 12345), too, because “012345” and “12345” stand for the same number. Both of those formulas would deliver the sequence “[5 4 3 2 1]”. Remember: dgts operates on numbers, not numerals.

To get this all straight, transport yourself back the the third grade. In those days, when you learned long division, there were four parts to the problem, and they all had names.

divisor	number you divide by
dividend	number you divide by the divisor
quotient	what you get when you do the division
remainder	what's left over to make up the difference

Aside 5.3: Think Third-Grade Division

ACL2 will not admit a function into its logic world unless it can prove that the function always delivers a value in a finite number of computational steps. This is because functions that don't terminate complicate the reasoning process. Proving that the dgts function, which converts numbers to numerals, terminates for all inputs requires an understanding of modular arithmetic. Fortunately, someone has put together a package of theorems on that topic. They reside in the "arithmetic-3/floor-mod/floor-mod" book (page 69). ACL2 will admit the definition of dgts when those theorems have been included in its logic world.

Aside 5.4: Termination, ACL2 Admit, and Floor/Mod Equations

The computer interprets the decimal numeral in the formula (dgts 012345) as a number and converts it to a mathematical object. That is, a number. What does that number look like? None of your business. That's the computer's business. It has it's own way of dealing with numbers. Later, we'll study the way most computers do this, but for now we will assume that the computer has some way of turning numerals into whatever form it uses to deal with numbers.

Now that we've run a few sanity checks, we want to get down to some serious testing. That means big batches of automated tests on random data. Coming up with automated tests calls for a little more thought. Let's start small. How about the last digit in a decimal numeral? What mathematical formula would deliver the last digit in a decimal numeral, given an arbitrary, positive integer n?

The last digit in a decimal numeral is the remainder when you divide the number by ten. The formula that converts a numeral to a number makes that clear.

$$1 \times 10^3 + 2 \times 10^2 + 1 \times 10^1 + 5 \times 10^0$$

Each of the terms in the formula is a product of a power of ten with another number. A power of ten is, of course, a multiple of ten, so none of the terms contribute to the remainder when dividing by ten. None of them, that is, except the last one. It does not have a factor of ten in it because the ten is raised to the power zero, and anything to the power zero is one, which is not a multiple of ten. So, to get the last digit in the numeral, all we need to do is to compute the remainder in the division of the number by ten.

The remainder is what the "mod" function delivers (page 37). The following formula uses mod to make sure the last digit in the numeral that the dgts operator delivers for the number n is correct.

```
(= (first (dgts n)) (mod n 10))
```

Since dgts delivers the digits backwards, "(first (dgts n))", the first digit in the sequence, is the last digit in the numeral. The formulas checks to make sure that digit is "(mod n 10)", the remainder when dividing n by 10.

We can use the doublecheck facility of Proof Pad or Dracula to run this test on a batch of random numbers. We need to be careful not to allow zero to pop up in the testing because (dgts 0) is nil, so there is no first digit to check. Besides, we've already completed the testing of (dgts 0) in our sanity checks. We can avoid retesting zero by adding one to a random natural number. That produces a random, non-zero, positive integer.

```
(defproperty dgts-last-digit-tst
  (n-minus-1 :value (random-natural))
  (let* ((n (+ n-minus-1 1))) ; avoid zero (no digits in numeral)
    (= (first (dgts n))
       (mod n 10))))
```

That takes care of testing the last digit (that is, the units digit), but what about the others? We can do something about those by observing that the quotient, when n is divided by 10, is a number with the same digits as n, except that the last digit is missing. Remember, we're doing third-grade arithmetic here. The quotient is the main result of the division. No fraction, no decimal point, no remainder. Just the whole-number quotient. Since we've already tested to make sure the last digit is correct, we don't need to worry about that. We only need to worry about the other digits.

One way to test the other digits is to apply dgts to the quotient. The intrinsic function "floor" (page 39) produces the quotient (discarding the remainder).

The following formula implements the test we have in mind. It checks to make sure the digits other than the units digit in the sequence dgts delivers for n are the same as the digits in the sequence dgts delivers for the quotient in $(n \div 10)$.

```
(equal (rest (dgts n))       ; all digits except the units digit
       (dgts (floor n 10)))) ; digits of the quotient
```

As with the test of the units digit, we can run a batch of tests based on our rest-of-the-digits observation by defining a doublecheck property.

```
(defproperty dgts-other-digits-tst
  (n-minus-1 :value (random-natural))
  (let* ((n (+ n-minus-1 1))) ; avoid zero (empty numeral)
    (equal (rest (dgts n))
           (dgts (floor n 10)))))
```

The function "zp" is used to test for zero in the domain of natural numbers. In the ACL2 logic, the formula (zp n) has the value true if n is zero, or if n is not a natural number. That makes zp the operator normally chosen to select the non-inductive case in inductive definitions of functions.

You might think a comparison with zero using the "=" operator would work equally well, but it doesn't. For example, the formula (= n 0) would have the value false if n were 3/2. However, if fractions are outside the intended domain of a function being defined, then something needs to be done to make the computation terminate when a fraction pops up, perhaps unintentionally, as argument in a formula referring to the function. Usually, the most desirable choice is to select a non-inductive case in the definition of the function, and zp makes that easy to do.

The function "posp" (page 53) is also useful in inductive definitions that with arguments that are expected to be natural numbers. The formula (posp n) is true if n is a non-zero natural number.

Aside 5.5: Natural Number Tests: Zero (zp) and Non-Zero (posp)

It would be nice to run these tests, but "dgts" is not an intrinsic operator. We have to provide a definition for it. To do that we use the "defun" command, which is similar to defproperty, but without any value specifications. The definition will be inductive, of course, using the same ideas as the tests, and will follow the three C's of Figure 4.2 (page 51), which we repeat here, in a form tailored to the dgts function.

Complete	Two cases: the number is zero or it isn't. So, two formulas.
Consistent	The cases do not overlap—no chance for inconsistency.
Computational	In the inductive case (when n is not zero), the argument of "dgts" is divided by ten, which makes it closer to zero (the non-inductive case).

The definition takes the following form.

```
(defun dgts (n)
  (if (zp n)
      nil                          ; {dgts0}
      (cons (mod n 10)             ; {dgts1}
            (dgts (floor n 10)))))
```

This definition uses the operator, "zp" (page 68). The formula (zp n) delivers true if n is the natural number zero (or, if it isn't a natural number at all), and false otherwise.

If you put the definition of dgts at the beginning of a program and import the "testing" and "doublecheck" facilities, you can enter the tests and run them using Proof Pad or Dracula. You can also enter formulas in the command panel to compute decimal numerals for any natural numbers you choose.

EXERCISES

Ex. 38 — Let y stand for the number of years between the signing of the Magna Carta and the signing of the United States Declaration of Independence. Figure out what the decimal numeral for y is and prove that you got it right using the definition of dgts (page 68), but without citing any of the theorems from this section.

Ex. 39 — Prove {*mod-div*}: (mod (* a x) (* a b)) = (* a (mod x b))
Hint: You won't need induction, but the following facts will help. Suppose x is the dividend, d the divisor, and q the quotient in a a problem in third-grade division (page 66). Then, r = (mod x d) is the remainder (page 37). The standard method of verifying a correct solution is to check $(qd + r) = x$. Furthermore, it is always the case that $0 \leq r < d$. In other words (mod x d) is the number r in the range $0 \leq r < d$ such that $qd + r = x$.

Ex. 40 — Define the {*mod-div*} theorem in ACL2 notation, and use ACL2 to verify that it is a theorem. Since the theorem does not hold for all numbers a, b, and x, you will need to include certain hypotheses in the implication that you ask ACL2 to prove. If you state it correctly and import theorems about modular arithmetic contained in the floor-mod book, ACL2 will succeed.

```
(include-book "arithmetic-3/floor-mod/floor-mod" :dir :system)
```

5.2 NUMBERS FROM NUMERALS

The "dgts" function (page 68) provides a way to produce a decimal numeral, given a number. How about going in the other direction? Given a decimal numeral, produce the corresponding number. You already know that the formulas look like these:

$$1 \times 10^3 + 2 \times 10^2 + 1 \times 10^1 + 5 \times 10^0 \text{ — for 1215}$$
$$1 \times 10^2 + 4 \times 10^1 + 4 \times 10^0 \text{ — for 144}$$

But, let's think of properties instead of complicated formulas. What properties would a function converting decimal numerals to numbers have? Since the dgts function produces numerals represented as sequences of decimal digits (numbers in the range 0 to 9), and in the reverse of the usual order (that is, with the units digit first, then the tens digit, then hundreds, and so on), let's assume that numerals will take that form.

Let's call the function that converts decimal numerals to numbers "nmb10". We know that (nmb10 nil) must be zero because (dgts 0) delivers zero, and we are trying to convert numerals produced by dgts back the the numbers they came from. How about a one-digit numeral $[x_0]$. The equation in that case would be (nmb10 $[x_0]$) = x_0.

If there are two or more digits, the numeral would take the form $[x_0\ x_1 \ldots x_{n+1}]$. Then, the equation would be

$$\text{(nmb10 } [x_0\ x_1\ x_2 \ldots x_{n+1}]\text{)} = (x_0 + x_1 \times 10^1 + x_2 \times 10^2 + \ldots x_{n+1} \times 10^{n+1})$$

Since all of the terms in the sum include a factor of ten, except the first term, we can factor ten out of all of the terms beyond the first. Factoring the formula in this way produces a new equation.

$$\text{(nmb10 } [x_0\ x_1\ x_2 \ldots x_{n+1}]\text{)} = (x_0 + 10 \times (x_1 \times 10^0 + x_2 \times 10^1 + \ldots x_n \times 10^n))$$

But, the sequence [$x_1\ x_2 \ldots x_{n+1}$] is also a decimal numeral. And, the value is stands for is $(x_1 \times 10^0 + x_2 \times 10^1 + \ldots x_n \times 10^n)$, which is the value nmb10 should deliver, given the numeral [$x_1\ x_2 \ldots x_{n+1}$]. That is, the following equation holds.

$$\text{(nmb10 [}x_1\ x_2 \ldots x_{n+1}\text{])} = (x_1 + x_2 \times 10^1 + \ldots x_{n+1} \times 10^n)$$

Observe that this formula for (nmb10 $(x_1 x_2 \ldots x_{n+1})$) is identical to the factor multiplied by ten in the previous equation. Therefore, we can rewrite that equation as follows.

$$\text{(nmb10 [}x_0\ x_1\ x_2 \ldots x_{n+1}\text{])} = (x_0 + 10\times \text{ (nmb10 [}x_1\ x_2 \ldots x_{n+1}\text{]))}$$

Now, we have an inductive equation that delivers the right value for numerals with two or more digits. Furthermore, the same formula works for one-digit numerals because (nmb10 nil) is zero.

$$\text{(nmb10 [}x_0\text{])} = (x_0 + 10\times \text{ (nmb10 nil))}$$

We now have one equation that covers all numerals with one or more digits. Furthermore, we know by the {cons} axiom (page 43):

$$[x_0\ x_1 \ldots x_n] = \text{(cons } x_0\ [x_1 \ldots x_n])$$

Therefore, by the axioms relating the functions first, rest, and cons (page 44), we arrive at the following equations.

$$x_0 = \text{(first [}x_0\ x_1 \ldots x_n\text{])}$$

$$[x_1 \ldots x_n] = \text{(rest [}x_0\ x_1 \ldots x_n\text{])}$$

This, along with using xs as a shorthand for [$x_0\ x_1\ x_2 \ldots x_n$] allows us to rewrite the equation for non-empty numerals in prefix notation as follows.

$$\text{(nmb10 } xs\text{)} = \text{(+ (first } xs\text{) (}*\text{ 10 (nmb10 (rest } xs\text{))))}$$

Now we have the basis to apply the rule of the three C's (page 51) to define nmb10.

```
(defun nmb10 (xs)
  (if (consp xs)
      (+ (first xs)               ; {n10.1}
         (* 10 (nmb10 (rest xs))))
      0))                         ; {n10.0}
```

We have derived this definition carefully, from things we know about numbers, but let's try to use logic to be sure we got it right. We want to prove that (nmb10 [$x_0\ x_1 \ldots x_n$]) delivers the same number as the formula $(x_0 + x_1 \times 10^1 + x_2 \times 10^2 + \ldots x_n \times 10^n)$.

Theorem {*Horner 10*}
(nmb10 [$x_0\ x_1 \ldots x_n$]) = $x_0 + x_1 \times 10^1 + x_2 \times 10^2 + \ldots x_n \times 10^n$

Theorem {*Horner 10*} states that the function nmb10 computes a sum of multiples of successive powers of ten. The multipliers (known as "coefficients") of the powers of ten are the digits in a decimal numeral. We call it the theorem "*Horner 10*" because the scheme it uses to carry out the computation is known as "Horner's rule".

Proving Theorem {*Horner 10*} amounts to verifying that $(\forall n.P(n))$ is true, where the predicate P is defined as follows. Of course the natural numbers are the universe of discourse for P.

$P(n) \equiv$ ((nmb10 $[x_0\ x_1 \ldots x_n]$) = $(x_0 + x_1 \times 10^1 + x_2 \times 10^2 + \ldots x_n \times 10^n)$)

Proceeding by mathematical induction, we first prove that $P(0)$ is true.

	$P(0) \equiv$ ((nmb10 $[x_0]$) = x_0)	
	(nmb10 $[x_0]$)	
$=$	(nmb10 (cons x_0 nil))	{*cons*} (page 44)
$=$	(+ (first (cons x_0 nil)) ($*$ 10 (nmb10 (rest (cons x_0 nil)))))	{*n10.1*} (page 70)
$=$	(+ x_0 ($*$ 10 (nmb10 (rest (cons x_0 nil)))))	{*first*} (page 44)
$=$	(+ x_0 ($*$ 10 (nmb10 nil)))	{*rest*} (page 44)
$=$	(+ x_0 ($*$ 10 0))	{*n10.0*} (page 70)
$=$	x_0	{2^{nd}*-grade arithmetic*}

Now, we prove that $P(n+1)$ is true, and because we cite mathematical induction, we can assume that $P(n)$ is true. In the last step, we do some standard calculation in numeric algebra (in prefix notation) using the distributive law. We cite "algebra" to justify that step in the proof. Generally, we will take advantage of the ordinary laws of algebra when we need them in numeric formulas.

	$P(n+1) \equiv$ ((nmb10 $[x_0\ x_1 \ldots x_{n+1}]$) = $(x_0 + x_1 \times 10^1 + \ldots x_{n+1} \times 10^{n+1})$)	
	(nmb10 $[x_0\ x_1 \ldots x_{n+1}]$)	
$=$	(nmb10 (cons x_0 $[x_1 \ldots x_{n+1}]$))	{*cons*}
$=$	(+ (first (cons x_0 $[x_1 \ldots x_{n+1}]$)) ($*$ 10 (nmb10 (rest (cons x_0 $[x_1 \ldots x_{n+1}]$)))))	{*n10.1*}
$=$	(+ x_0 ($*$ 10 (nmb10 (rest (cons x_0 $[x_1 \ldots x_{n+1}]$)))))	{*first*}
$=$	(+ x_0 ($*$ 10 (nmb10 $[x_1 \ldots x_{n+1}]$)))	{*rest*}
$=$	(+ x_0 ($*$ 10 $(x_1 + x_2 \times 10^1 + \ldots x_{n+1} \times 10^n)$))	{$P(n)$}
$=$	$(x_0 + x_1 \times 10^1 + x_2 \times 10^2 + \ldots x_{n+1} \times 10^{n+1})$	{*alg*}

Since we proved that $P(0)$ is true and that $P(n) \to P(n+1)$ is true if $P(n)$, regardless of what natural number n stands for, we conclude by mathematical induction that $P(n)$ is true for all natural numbers n. In other words, the formula (nmb10 $[x_0\ x_1 \ldots x_n]$) delivers the correct number whenever $[x_0\ x_1 \ldots x_n]$ is a decimal numeral represented as a list, starting with the units digit and proceeding, power of ten by power of ten, up to the high-order digit.

We derived this theorem from the axioms defining the nmb10 function. Besides the axioms, our reasoning cited equations we proved before (or knew from numeric algebra), along with the deductive inference rule "mathematical induction." We would also like to know that the axioms defining the dgts function produces the correct numeral. Since we know that nmb10 delivers the correct number, given a decimal numeral, we can confirm the correctness of (dgts n) by proving the following theorem.

Theorem {*dgts-ok*}
(nmb10 (dgts n)) = n

In other words, we want to prove that $(\forall n.D(n))$ is true, where D(n) stands the following proposition.

$$D(n) \equiv \text{((nmb10 (dgts } n\text{))} = n\text{)}$$

Since the universe of discourse for the predicate D is the natural number, we may as well try to use mathematical induction to prove that $(\forall n.D(n))$ is true. That requires a proof of $D(0)$ and a proof of $(\forall n.(D(n) \rightarrow D(n+1)))$.

	$D(0) \equiv$ ((nmb10 (dgts 0)) = 0)	
	(nmb10 (dgts 0))	
$=$	(nmb10 nil)	{*dgts0*} (page 68)
$=$	0	{*n10.0*} (page 70)

	$D(n+1) \equiv$ ((nmb10 (dgts (+ n 1))) = (+ n 1))	
	(nmb10 (dgts (+ n 1)))	
$=$	(nmb10 (cons (mod (+ n 1) 10) (dgts (floor (+ n 1) 10)))	{*dgts1*} (page 68)
$=$	(+ (mod (+ n 1) 10) ($*$ 10 (nmb10 (dgts (floor (+ n 1) 10)))))	{*n10.1*} (page 70)
$=$	(+ (mod (+ n 1) 10) ($*$ 10 (floor (+ n 1) 10)))	{D(floor (+ n 1) 10)}
$=$	(+ n 1)	{3rd-grade division} (page 66)

If you were paying very close attention, you may have noticed that we our proof of $D(n) \rightarrow D(n+1)$ was not according to Hoyle. To prove this implication, we need to prove that $D(n+1)$ is true whenever $D(n)$ is true. That means we can cite $D(n)$ to justify any step in our proof of $D(n+1)$. However, instead of citing $D(n)$, we cited D(floor (+ n 1) 10). That's a different proposition, but it happens that (floor (+ n 1) 10) is strictly smaller than $(n+1)$.

In this proof, we are relying on a more general inference rule than ordinary mathematical induction. The more general rule, which is called "strong induction" is equivalent to ordinary mathematical induction. That is, one can verify that if the rule of ordinary mathematical induction is a valid rule of inference, then so is strong induction, and vice versa. The proof is not difficult, but it's a distraction, so we are just going to assume that strong induction works. You can summon a rationale for strong along the lines of the rationale we had for ordinary induction (page 49).

The rationale goes like this. Imagine that you are proving the propositions $P(0)$, $P(1)$, $P(2)$, ... and so on, one by one, in sequence. When you get to the point where you want to prove $P(n+1)$, you will have already proven all of the propositions with smaller indices—that is $P(0)$, $P(1)$, $P(2)$, ... $P(n)$. So, in the proof of $P(n+1)$, you would be able to cite any of the previous propositions, not just $P(n)$. When you cite $P(n)$, but not propositions with smaller indices, in the proof of $P(n+1)$, you are doing a proof by ordinary mathematical induction. When you cite one or more propositions with indices smaller than n, you are doing a proof by strong induction.

Since ordinary mathematical induction and strong induction are equivalent rules of inference, we are going to refer to both of them the same way. We'll just call it mathematical induction. But, the formal statement of the strong induction rule is different from the formal statement of ordinary induction (Figure 4.1, page 49). The strong induction rule is state formally in Figure 5.1 (page 73).

$$\frac{\text{Prove } (\forall m < n.P(m)) \to P(n)}{\text{Infer } (\forall n.P(n))}$$

Figure 5.1: Mathematical Induction–strong induction version

The strong induction rule looks more different from the one for ordinary induction than it really is. The ordinary induction rule required two proofs before making the conclusion: (1) Prove $P(0)$ and (2) Prove $\forall n$.(P(n)→P($n + 1$)). The strong induction rule only calls for one proof: Prove $(\forall m < n.P(m)) \to P(n)$. However, when n is zero, the formula to be proved reads as follows: $(\forall m < 0.P(m)) \to P(0)$. The formula is strange because m is supposed to be a natural number, and the $\forall$ quantification runs over all values of m that are strictly less than zero. Since there are no natural numbers less than zero, the universe of discourse for this quantification is empty. That makes the formula $(\forall m < 0.P(m))$ true because a $\forall$ quantification with an empty universe of discourse is true, by default (page 42).

So, proving $(\forall m < 0.P(m)) \to P(0)$ is the same as proving $True \to P(0)$, which requires proving that $P(0)$ is true, just as in ordinary induction. In other words, in strong induction there are still two proofs to do, one for P(0), and one for $(\forall m < n.P(m)) \to P(n)$ when n is not zero. The strong induction inference rule looks like it requires only one proof, but that is an illusion caused by the fact that a $\forall$ quantification with an empty universe is automatically true.

Aside 5.6: Strong Induction Requires Two Proofs or One?

From now on, when we cite the mathematical induction rule of inference, we will mean this new, strong induction rule. After all, it encompasses the ordinary induction rule as a special case, anyway, since it only cites $P(n)$ is the proof of $P(n + 1)$, and not any proposition in the predicate P with a smaller index. So, we may as well cite strong induction, even when we're only relying on the ordinary induction rule.

Exercises

Ex. 41 — Let d stand for the number of furlongs in the Boston Marathon, not counting the last 165 yards. Prove that (nmb10 (dgts d)) = d using the definitions of dgts (page 68) and nmb10 (page 70), but without citing any of the theorems from this section.

Ex. 42 — Use Proof Pad or Dracula to create a .lisp file containing the definitions of dgts and nmb10. In the same file, define a property that tests the formula (= (nmb10 (dgts n)) n) for random natural numbers n. Of course, all of the tests should succeed because we proved that the formula always delivers true. If a test fails, something is wrong with one or more of the definitions in the .lisp file.

Ex. 43 — Add the definition of another property to the .lisp file from the previous exercise. The new property will test the formula (equal (dgts (nmb10 xs)) xs) for random, non-empty decimal numerals xs. *Hint*. (random-list-of (random-between 0 9)) generates random decimal numerals. *Note*. This test can fail. If it does, check out the data that causes the failure.

Ex. 44 — We proved that the function nmb10 inverts the function dgts. That is, (nmb10 (dgts n)) is the same as n for any natural number n (page 71). However, it is not quite true that dgts inverts nmb10. Why not? Give an example of a decimal numeral xs for which (dgts (nmb10 xs)) is different from xs.

Ex. 45 — Describe a class of decimal numerals such that that (dgts (nmb10 xs)) is the same as xs when xs is a numeral from that class.

Ex. 46 — Use mathematical induction to prove that (dgts (nmb10 xs)) is the same as xs when xs is a numeral from the class you described in the previous exercise.

Ex. 47 — Let $\lfloor log(n) \rfloor$ stand for the smallest integer for which $10^{\lfloor log(n) \rfloor} \geq n$, where n is a non-zero natural number. That is, the following inequalities hold.

$$10^{\lfloor log(n) \rfloor} \leq n < 10^{\lfloor log(n) \rfloor + 1}$$

Prove the following relationship between the functions len (page 45) and dgts (page 68).

(len (dgts n)) = $\lfloor log(n) \rfloor + 1$

5.3 Binary Numerals

Digital circuits, since they are a materialization of mathematical logic, have components that can represent two different values. We call them zero and one, and we choose those names primarily because circuits that deal with numbers do so in terms of binary numerals. Decimal numerals require ten different symbols (0, 1, 2, . . . 9) to represent digits.

Binary numerals only need two (0 and 1) for their binary digits, which are usually called "bits", and that makes them well-suited for representation in the form of digital circuits. We will be discussing the design and analysis of circuits to do arithmetic in terms of binary numerals, so we will need to know how to construct and interpret them.

Decimal numerals represent numbers as sums of multiples of powers of ten. Binary numerals are similar, but use two as a basis instead of ten. So, the binary numeral with bits $x_n x_{n-1} \ldots x_2 x_1 x_0$ stands for the number $(x_0 + x_1 \times 2^1 + x_2 \times 2^2 + \cdots + x_{n-1} \times 2^{n-1} x_n \times 2^n)$. The only differences between this formula and the one that interprets decimal numerals is that it has powers of two where the decimal formula had powers of ten, and multipliers are bits (0, 1) instead of digits (0, 1, 2, . . . 9).

Therefore, we can convert the functions for decimal numerals to binary by simply changing the base from ten to two. The following definitions of the functions "bits" and "nmb" to construct and interpret binary numerals come directly from that observation. Like the corresponding functions for decimal numerals (dgts, 68, and nmb10, 70), bits and nmb use the function zp (page 68) to choose between the base case ($n = 0$) and the inductive case ($n > 0$). Also, ACL2 needs access to the theorems in the floor-mod book (page 66) to admit the function "bits" to its logic, just as in the definition of dgts (68).

```
(defun bits (n)
  (if (zp n)
      nil                         ; {bits0}
      (cons (mod n 2)             ; {bits1}
            (bits (floor n 2)))))
```

```
(defun nmb (xs)
  (if (consp xs)
      (+ (first xs) (* 2 (nmb (rest xs)))) ; {nmb1}
      0))                                  ; {nmb0}
```

The theorem for correctness of the function nmb that interprets binary numeral as numbers and the theorem about the function nmb being the inverse of the function bits are also true in the new context, and the proofs are similar to the ones on decimal numerals presented earlier in this chapter. Constructing those proofs is a good exercise. It will help you understand decimal numerals better, and clarify your understanding of binary numerals.

EXERCISES

Ex. 48 — Adapt the proof of {*Horner 10*} (page 70) to prove {*Horner 2*}:

$$(\text{nmb } [x_0\ x_1\ x_2 \ldots\ x_n]) = (x_0 + x_1 \times 2^1 + x_2 \times 2^2 + \ldots x_n \times 2^n)$$

Ex. 49 — Prove theorem {*bits-ok*}: ((nmb (bits n)) = n), assuming n is a natural number. That is, the function nmb (page 75) inverts the function bits (page 74).

Ex. 50 — We created the definitions of bits and nmb by copying those of dgts and nmb10, and changing the twos to tens. The practice of defining new entities by copying old ones and making a few changes is the single most common cause of errors in computer software (page 18). Instead, we should have done is to define new functions with an additional parameter for the base to deal with positional numerals with any base.

```
(defun numeral-from-number (b n)
  (if (zp n)
      nil
      (cons (mod n b)
            (numeral-from-number b (floor n b)))))
```

```
(defun number-from-numeral (b xs)
  (if (consp xs)
      (+ (first xs)
         (* b (number-from-numeral b (rest xs))))
      0))
```

Make new definitions of the functions dgts, nmb10, bits, and nmb that rely on the abstraction of the base in the functions numeral-from-number and number-from-numeral.

Ex. 51 — In a manner similar to that suggested in the previous exercise, define functions to convert between numbers and hexadecimal (base 16) numerals.

Ex. 52 — The hexadecimal numerals in the previous exercise use numbers between zero and fifteen to represent hexadecimal digits. Define new functions that use, instead, strings of characters from the set 0, 1, 2, . . . , 9, A, B, C, D, E, F to represent hexadecimal numerals. While you're at it, represent the numerals as string in the usual ordering from high-order hexadecimal digit to low-order, instead of the reverse order we have been using. You may refer to the functions defined below that convert between hex strings and numerals. Strings in ACL2 are delimited by double-quote characters.

```
(defun char-to-digit (chr)
  (let* ((dgt9  (char-code #\9))
         (code (char-code chr)))
    (if (> code dgt9)
        (+ 10 (- code (char-code #\A)))
        (- code (char-code #\0)))))
(defun chars-to-digits (chrs)
  (if (consp chrs)
      (cons (char-to-digit (first chrs))
            (chars-to-digits (rest chrs)))
      nil))
(defun digits-to-chars (dgts)
  (if (consp dgts)
      (cons (digit-to-char (first dgts))
            (digits-to-chars (rest dgts)))
      nil))
(defun string-to-numeral (str)
  ;e.g. (string-to-numeral "41C5") = '(5 11 1 4)
  (let* ((chrs (coerce (string-upcase str) 'list)))
    (chars-to-digits (reverse chrs))))
(defun numeral-to-string (nml)
  ;e,g. (numeral-to-string '(5 11 1 4)) = "41C5"
  (let* ((dgts (reverse nml)))
    (coerce (digits-to-chars dgts) 'string)))
```

The definition of pad also makes use of the "let*" formula, which makes it possible to give names to values to be used within the scope of the "let*" formula. A "let*" formula starts with the the keyword "let*", then a list of name-value pairings, and finally a formula for the value to be delivered by the "let*". Each pairing is list of two elements: first a name, then a formula specifying a value to be associated with the name. In this example, there is only one such pairing, but there can be any number of pairings. The most common use of "let*" is to give names to values needed more than once in a computation, but sometimes it's used just to give mnemonic names to significant elements in a computational formula.

Aside 5.7: Naming Local Values with Let*

Ex. 53 — Prove theorem {*len-pad*}: (len (pad $n\ x\ xs$)) = n, where "pad" is defined as follows. *Note*: The definition of pad refers to the functions rep (page 55) and prefix (page 53). It also uses a "let*" formula (page 77), which provides a way to associate names with values within a scope bounded by parentheses.

```
(defun pad (n x xs)
  (let* ((padding (- n (len xs))))
    (if (natp padding)
        (append xs (rep padding x)) ; {pad+}
        (prefix n xs))))            ; {pad-}
```

5.4 Numerals for Negative Numbers

So far, all the numerals we've seen have represented positive numbers. Now, we need to attend to the issue of negative numbers. There are many ways to do that, but a format known as "twos complement" stands out because computer hardware designers have adopted it as the standard way of dealing with negative numbers.

The twos complement approach limits numbers to a range that runs from a specific negative limit to a specific positive limit. Then, for negative numbers, it uses binary numerals that would normally stand for numbers beyond the limit on the positive side. The idea relies on properties of modular arithmetic (page 40).

The modulus chosen for a twos-complement representation is always a power of two. Call it 2^w. It happens that binary numerals for numbers in the range 0, 1, ... $(2^{w-1} - 1)$ have no more than $(w - 1)$ bits (Theorem {*len-bits*$\leq$}, page 80). So, the If the hardware allocates w bits for recording binary numerals, numbers in the range 0, 1, ... $(2^{w-1} - 1)$ will only need $w - 1$ of those bits. They will always have a leading zero in the high-order bit. Bit-patterns with a one in the high-order bit can be used for negative numbers. That's where the twos complement trick comes into play. Let's see how it works.

The number of bits that the hardware allocates for recording binary numerals is known as the "word size" of the computer. A computer with 32-bit words has circuits to deal with numbers in the range $-2^{31}, \cdots - 1, 0, 1, 2, \ldots 2^{31} - 1$. In the positive part of the range, it represents numbers as ordinary binary numerals, but with enough leading zeros to fill the 32-bit word.

For a number $(-n)$ in the negative part of the range, the twos-complement system uses the binary numeral for ($2^{32}-n$) to represent $(-n)$. Since $(-n)$ is in the negative range, $-2^{31} \leq -n < 0$. Therefore, $2^{31} \leq 2^{32}-n < 2^{32}$. That means the twos-complement, binary numeral that stands for the negative number $(-n)$ has exactly 32 bits (Theorem {*len-bits*}, page 80), the high-order bit being a 1 (Theorem {*hi-1*}, page 79).

Modular arithmetic makes the numerals chosen for negative numbers act like the negative numbers they stand for. For negative numbers $(-n)$ in the range $-2^{31} \leq -n < 0$, the value of ($(-n)$ mod 2^{32}) is ($2^{32}-n$). Therefore, since addition and subtraction produce consistent results in modular arithmetic (page 40), it follows that $((m+(-n))$ mod 2^{32} = (((m) mod 2^{32}) + ($(-n)$ mod 2^{32})) mod 2^{32} = (((m mod 2^{32}) + ($2^{32}-n$))) mod 2^{32}.

That is, adding the numbers represented by twos-complement numerals, including numbers in the negative range, is just like adding ordinary numbers, but using modular arithmetic. Subtraction is handled by negating a number (that is, computing the twos-complement representation of its negative), then performing addition modulo 2^{32}.

The same thing works for any other word size. With word size w, the twos complement system handles addition and subtraction, modulo 2^w, for numbers in the range $-2^{w-1}, \cdots -1, 0, 1, 2, \ldots 2^{w-1}-1$. Circuits for performing addition (and subtraction, since it is the same operation in a twos-complement system) take advantage of these mathematical facts. We will make use of this to design circuits to perform addition.

In summary, the twos-complement representation for computers with word size w applies to numbers in the range $-2^{w-1}, \cdots -1, 0, 1, 2, \ldots 2^{w-1}-1$. We will refer to this set of integers as $I(w)$.

$$I(w) = \{-2^{w-1}, \cdots -1, 0, 1, 2, \ldots 2^{w-1}-1\}$$

We will discuss the design of a digital circuit that performs addition on numbers in $I(w)$. That circuit will be capable computing the sum $x+y$ if x, y, and $x+y$ are numbers in the set $I(w)$. That circuit will use twos-complement representation for numbers in $I(w)$.

The *twos-complement representation* for a negative number $(-n)$ in $I(w)$ is the binary numeral for the number ($2^w - n$). Twos-complement numerals for numbers in the negative range have exactly w bits, with a one-bit in the high-order slot. That is because (len (bits (- (expt 2w) (n)))) = w, since $2^{w-1} \leq 2^w - n < 2^w$ (Theorem {*len-bits*}, page 80).

The twos-complement representation for numbers in the positive range of $I(w)$ take the form of ordinary binary numerals, except that in hardware they are padded with enough leading zeros to fill out a w-bit word, where w is the word size of the computer. This is necessary because all of the bits in a circuit, at any point in time, stand for either zero or one. There is no other possible state. So, the high-order bits in a w-bit word containing the twos-complement representation of a positive number are padded out with enough leading zeros to make w bits in all, but without changing the number that the numeral stands for.

```
(defun twos (w n) ; w (word size): natural number
  (if (< n 0)     ; n: integer, -2^(w-1) <= n < 2^(w-1)
      (bits (+ (expt 2 w) n)) ; {2s-}
      (pad w 0 (bits n))))    ; {2s+}
```

$n \in I(3)$	$2^3 + n$	(twos 3 n)	*binary numeral*
-4	4	[0 0 1]	100
-3	5	[1 0 1]	101
-2	6	[0 1 1]	110
-1	7	[1 1 1]	111
0		[0 0 0]	000
1		[1 0 0]	001
2		[0 1 0]	010
3		[1 1 0]	011

word size $w = 3$

$$I(w) = I(3) = \{-2^{3-1}, -3, -2, -1, 0, 1, 2, 2^{3-1} - 1\}$$

Figure 5.2: 2s-Complement Numerals for 3-bit Words

There will always be some padding for numerals representing numbers in the positive range because such a number n lies within the limits $0 \leq n < 2^{w-1}$. The twos-complement numeral for a value, n, in that range is (bits n), and $0 \leq$(len (bits n)) $< w$ (Theorem {*len-bits*$\leq$}, page 80). Figure Figure 5.2 displays twos-complement representations for the numbers in the set $I(3)$. Of course, the table in the figure is only for purposes of illustration. The word-size 3 is ridiculously small. No computer would have words that short.

Now, here is an interesting little trick. Suppose n is a natural number in the range $1 \leq n \leq 2^{31}$. By definition (page 78), the twos-complement numeral for the negation of n, $(-n)$, is the binary numeral for $2^{32} - n$. That method produces the twos-complement numeral from the number. But, there is also a way to compute this numeral for $(-n)$ directly from the numeral for n by inverting its bits (that is, changing all the zeros to ones and all the ones to zeros) and then computing the binary numeral for the number that is one more than the number that the numeral with inverted bits stands for.

It works like this. Let $x_{w-1}x_{w-2}\ldots x_2x_1x_0$ be the w-bit, binary numeral for n, padded with leading zeros if necessary to fill out the full w bits. In other words [x_0 x_1 x_2 $\ldots$ x_{w-2} x_{w-1}] = (twos n). Then, the twos-complement numeral for $(-n)$ can be computed by inverting the bits in (twos n), interpreting the inverted-bit numeral as a number, increasing that number by one, then converting the new number to a binary numeral. That may seem a bit indirect, but when a circuit to do arithmetic on numerals is available, inverting the bits and incrementing by one is an attractive way to proceed. Figure 5.3 (page 80) explains why this trick works.

EXERCISES

Axiom {*list*}

$$(\forall x.((\text{list } x) = (\text{cons } x \text{ nil})))$$

Ex. 54 — Prove {*hi-1*}: (fin (bits n)) = 1
Assume that the function "fin" satisfies the following equations.

```
(defun fin (xs)
  (if (consp (rest xs))
```

$1 \le n \le 2^{31}$	range of numbers to negate
(len (bits n)) $\le w$	{*len-bits*$\le$} (page 80)
xs = [$x_0\ x_1\ x_2 \ldots x_{w-2}\ x_{w-1}$])	xs=(pad w 0 (bits n)), padded binary numeral
(nmb xs) = n	{*leading-0s*} (page 80)
ys = [$y_0\ y_1\ y_2 \ldots y_{w-2}\ y_{w-1}$]	inverted bits $y_i = 1 - x_i$ (0s for 1s, 1s for 0s)
(bits (+ 1 (nmb ys)))	2s-complement-trick numeral of $(-n)$

$$\begin{array}{rllll}
 & 1 & + & (\text{nmb } ys) & \\
= & 1 & + & y_0 2^0 + y_1 2^1 + \cdots + y_{w-1}2^{w-1} & \{\textit{Horner 2}\} \\
= & 1 & + & (1-x_0)2^0 + (1-x_1)2^1 + \cdots + +(1-x_{w-1})2^{w-1} & \forall i.(y_i = 1 - x_i) \\
= & 1 & + & (2^0 + 2^1 + \cdots + 2^{w-1}) & \{\textit{algebra}\} \\
 & & - & (x_0 2^0 + x_1 2^1 + \cdots + x_{w-1}2^{w-1}) & \\
= & 1 & + & (2^w - 1) - (x_0 2^0 + x_1 2^1 + \cdots + x_{w-1}2^{w-1}) & \{\textit{geom. progression}\} \\
= & 2^w & - & (x_0 2^0 + x_1 2^1 + \ldots x_{w-1}2^{w-1}) & \{\textit{arithmetic}\} \\
= & 2^w & - & (\text{nmb } xs) & \{\textit{Horner 2}\} \\
= & 2^w & - & n & (\text{nmb } xs) = n
\end{array}$$

	(bits (+ 1 (nmb ys)))	2s-complement-trick numeral
=	(bits ($2^w - n$))	1+(nmb ys) = $2^w - n$
=	(bits ($2^w + (-n)$))	{*algebra*}
=	(bits (+ (expt 2 w) $(-n)$))	convert $(2^w + (-n))$ to ACL2 notation
=	(twos w $(-n)$)	definition of function “twos” (page 78)

Figure 5.3: Twos-Complement Negation Trick

```
      (fin (rest xs)) ; {fin2}
      (first xs)))    ; {fin1}
```

Ex. 55 — Express theorem {*hi-1*} of the previous exercise in ACL2 notation and run it through the ACL2 mechanized logic. ACL2 will succeed if you state the theorem correctly as an implication, using the function “posp” (page 53) to constrain n to a non-zero, natural number. Of course the definition of “bits” (page 74) will need to be admitted to the logic (page 66) before ACL2 can attempt the proof of this theorem.

Ex. 56 — Prove {*len-bits*}: $\forall n.((2^{n-1} \le m < 2^n) \to$ ((len (bits m)) $= n$))

Ex. 57 — Prove {*len-bits*$\le$}: $\forall n.((0 \le m < 2^n) \to$ ((len (bits m)) $\le n$))

Ex. 58 — Prove {*log-bits*}: $\forall n > 0.$((len (bits n)) $= \lfloor log_2(n) \rfloor + 1$)
Note: $\lfloor x \rfloor$ = biggest integer not exceeding x

Ex. 59 — Prove {*len-2s*}: $\forall w.((n \in I(w)) \to$ ((len (twos w n)) $= w$))

Ex. 60 — Prove {*leading-0*}: $\forall n.$((nmb (append (bits n) (list 0))) = (nmb (bits n)))

Ex. 61 — Prove {*leading-0s*}: $\forall z.(\forall n.$((nmb (append (bits n) (rep z 0))) = (nmb (bits n))))
Assume that theorem {*leading-0*} in the previous exercise has been proven and that z stands for a natural number.

Ex. 62 — Express theorem {*leading-0s*} of the previous exercise in ACL2 notation and run it through the ACL2 mechanized logic. ACL2 will succeed if you state the theorem correctly as an implication, referring to "natp" to constrain n and z to be natural numbers. Of course the definitions of "bits" and "nmb" will need to be admitted to the logic (page 74) before ACL2 can attempt the proof of this theorem.

Ex. 63 — Prove {*pfx-mod*}: $\forall w.(\forall n.$((nmb (prefix w (bits n))) = (mod n 2^w)))
Hint: Induct on w. Split the inductive case into two parts, one part where $n = 0$ and one where $n > 0$, and apply {$\vee$ elimination} (page 31) to complete the proof of the inductive case. In the $n > 0$ part, the {*mod-div*} theorem (page 69) will be helpful.

Ex. 64 — Prove {*minus-sign*}: $\forall n \in I(w).((n < 0) \to$ (fin(twos w n)) = 1)

Ex. 65 — Prove {*plus-sign*}: $\forall n \in I(w).((n \geq 0) \to$ (fin(twos w n)) = 0)

Six

Adders

6.1 Addition by Numeral

When people do paper-and-pencil arithmetic, they do it using numerals. For example, to add two numbers, a person writes down the decimal numerals for the numbers, one under the other with the digits lined up so that the units digit of one number is directly under the units digit of the other, and similarly for the tens digits, hundreds digits, and so on. Then, the units digits are added together, and the low-order digit of that sum is written in the units-digit column.

If the sum of the units digit is ten or more, a "carry" is indicated in the tens-digit column. The sum of the tens digits is computed and the carry added to the sum if there was a carry. As before, the low-order digit of the sum of the tens digits (and carry) goes in the numeral for the sum, but this time in tens-digit column. A carry is marked, if necessary, in the hundreds digit column. This process moves across the digits of the summands until all the digits are accounted for (Figure 6.1).

To perform addition in this way, a person needs to know the table for one-digit sums (0+0=0, 0+1=1, . . . 2+2=4 . . . 9+8=17, 9+9=18). There are a hundred entries in the table (10×10, one for each possible pair of decimal digits).

Decimal numerals are not the only kind, and addition can be carried out similarly, regardless of numeral base. The table of one-digit sums is, of course, much smaller for binary numerals than for decimal numerals. It has only four entries ($0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 0, 1 + 1 = 10_2$). Except for the new, smaller table for one-digit sums, the process is the same (Figure 6.2).

The small size of the one-digit-sum table simplifies the both pencil-and-paper process and the difficulty of designing digital circuits for performing addition on binary numerals, compared to what it would be for decimal numerals.

```
  11 1    carries
   ----
   9542   summand A
 +  638   summand B
   ----
  10180   sum A+B
```

Figure 6.1: Adding Decimal Numerals

```
 11 111 1    carries
  --------
  01011101   summand A
+ 11010101   summand B
  --------
 100110010   sum A+B
```

Figure 6.2: Adding Binary Numerals

$x+y$	c	s
$0+0$	0	0
$0+1$	0	1
$1+0$	0	1
$1+1$	1	0

Figure 6.3: One-Bit Addition Table

EXERCISES

Ex. 66 — Describe by example the process of multiplying a pair of decimal numerals.

Ex. 67 — Describe by example the process of multiplying a pair of binary numerals.

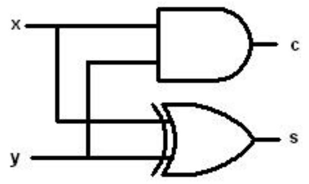
6.2 ADDING ONE-BIT BINARY NUMERALS

The addition table for one-bit, binary numerals has only four entries, as shown in Figure 6.3. In the figure, the sum is shown in all four cases as two separate bits, a carry bit c and a sum bit s.

A close look at the table shows that the carry bit matches the table of values of the digital gate for logical-and (Figure 2.6). That is, the carry bit is 1 only if both inputs are 1s. Otherwise, the carry bit is 0. So, constructing a digital circuit to compute the carry bit in the addition of two one-bit, binary numerals is simply a matter of feeding the signals for the one-bit numerals into a logical-and gate.

Another close look reveals that the sum bit matches the table of values of the digital gate for exclusive-or (Figure 2.6). That is, the sum bit is 0 if the two inputs are the same and 1 if they are different. So, constructing a digital circuit to compute the sum bit amounts to feeding the signals for the one-bit numerals into an exclusive-or gate. Combining these ideas for carry-bit and sum-bit circuits leads to a two-input, two-output circuit known as a half-adder (Figure 6.4).

Since we reason about digital circuits using the methods of Boolean algebra, we need an algebraic representation of the circuit-diagram for the half-adder circuit. Remember, digital circuits are only one of four equivalent representations of Boolean formulas that we have studied: circuit diagrams, well-formed formulas in the notation of mathematical



$x+y$	c	s
$0+0$	0	0
$0+1$	0	1
$1+0$	0	1
$1+1$	1	0

Figure 6.4: Half-Adder Circuit

```
(defun and-gate (x y) (if (and (= x 1) (= y 1)) 1 0))
(defun or-gate (x y) (or (= x 1) (= y 1)))
(defun xor-gate (x y) (if (and (= x 1) (= y 1)) 0 (or-gate x y)))
(defun half-adder (x y) (list (xor-gate x y) (and-gate x y)))
```

Figure 6.5: ACL2 Model of Half-Adder Circuit

logic (for example, $x \wedge y$), Boolean formulas in engineering notation (justaposition for $\wedge$, $+$ for $\vee$, and over-bar for $\neg$), and ACL2 notation.

The ACL2 form allows us to take advantage of the mechanization of some aspects of the reasoning process. So, we specify the half-adder circuit in ACL2 terms (Figure 6.5) and refer to this specification as an ACL2 model of the half-adder circuit. It delivers the two output signals as a list of two elements, the first element being the sum bit and the second, the carry bit.

In the end, we would like to have a circuit that adds binary numerals, and we saw in an example (Figure 6.2) that this would require us to deal with three input bits for each bit in the numerals: the corresponding bits in the two summands (ones-bit, twos-bit, fours-bit, ...) and the carry bit from the previous column.

The half-adder circuit is not up to this task, since it has only two input signals. However, we can put together a "full-adder circuit" by combining two half-adders with an or-gate, as shown in the circuit of Figure 6.6. Since the full-adder circuit has three inputs, each of which is either 0 or 1, there are exactly eight possible input configurations. We could build a comprehensive verification of the ACL2 model of the full-adder circuit of Figure 6.6 by writing eight check-expect tests matching the full-adder table.

```
(check-expect (full-adder 0 0 0) (list 0 0))
(check-expect (full-adder 0 0 1) (list 1 0))
  ...
(check-expect (full-adder 1 1 0) (list 0 1))
(check-expect (full-adder 1 1 1) (list 1 1))
```

Another approach is to observe that the input combinations match the list of binary numerals for the numbers 0 through 7, padded with leading zeros to make them all have exactly three bits. The function "twos" (page 78) produces these numerals.

However, that approach is tedious, and it's easy to make a mistake. Another approach is to observe that the value delivered by (full-adder c_{in} x y) is a binary numeral for the number $c_{in} + x + y$, padded with a leading zero if necessary to make it have exactly two bits. Since the function "twos" (page 78) produces padded numerals of that form, we expect that (twos 2 (+ c_{in} x y)) = (full-adder (list c_{in} x y)). In other words, making sure this equation holds for each combination of values for c_{in}, x, and y is the same as running the above check-expect tests.

Furthermore, the set of combinations of values for c_{in}, x, and y is the same as the set of numerals for the numbers 0 through 7 padded with leading zeros to make them all have exactly three bits. That is, the bit-combinations in the numerals (twos 3 0), (twos 3 1), (twos 3 2), . . . (twos 3 7) comprise the full set of combination of values for c_{in}, x, and y. Therefore, each of the equations we want to check has the form (chk-combo n), for some n between 0 and 7, where the function "chk-combo" is defined in Figure 6.7. That is, the ACL2 model of full-adder in Figure 6.6 is correct if all of the formulas following formulas are true: (chk-combo 0), (chk-combo 1), (chk-combo 2), . . . (chk-combo 7). We conclude that the constant[1] "*chk-full-adder*" defined in Figure 6.7 is true if the full-adder model is correct and false if it isn't.

We now have a fully mechanized verification of the ACL2 model of the full-adder circuit. That makes it safe to use this model as a component in the design of a circuit to carry out addition on binary numerals with more than one bit.

6.3 ADDING TWO-BIT BINARY NUMERALS

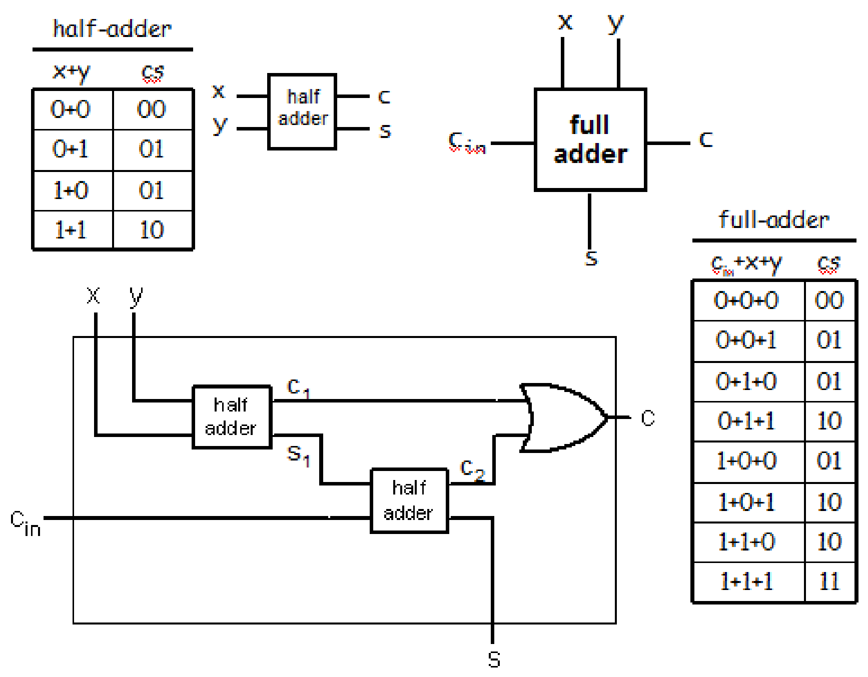
Adding binary numerals with two bits is simply a matter of connecting two full-adder circuits in a manner that directs the carry from adding the low-order bits to the carry-input of the second full-adder circuit. Since there is never a carry into the low-order position, we just supply zero as the carry-in for the full-adder circuit that deals with the low-order bit.

This two-bit addition circuit takes two, two-bit binary numerals (x_0 x_1) and (y_0 y_1) as inputs, and delivers a two-bit numeral along with a carry-bit, as shown in Figure 6.8 (88). We refer to the two-bit numeral without the carry bit as the "sum bits" of the output from the two-bit adder. There is also a carry bit. Ignoring the carry bit amounts to doing modular arithmetic, mod 4 (2^2), as is confirmed by theorem {*pfx-mod*} (page 81).

A mechanized verification of the correctness of the model could be constructed in a manner similar to the one for one-bit addition (Figure 6.7,page 88). However, it would have 32 cases to check because there are five bits of input in all (a carry-in and two bits in each numeral, 2^5 combinations in all). That makes it tedious to construct, and it's easy to make a mistake. A better way is to think of a formula in logic that expresses the properties we expect the model to have.

One approach is to observe the the numbers represented by the input numerals, along with the input carry, must add to the number represented by the output numeral (that is the sum bits), with the output carry bit appended to the high-order bit position at the end. Of course, the theorem will be stated as an implication, so that we can constrain the inputs to be bits (zeros and ones), and not some other kind of data.

[1]Constants in ACL2 are defined like functions, but with the keyword "defconst" instead of "defun". Names of constants are required to start and end with asterisks, and constants do not have parameters.



half-adder

x+y	cs
0+0	00
0+1	01
1+0	01
1+1	10

full-adder

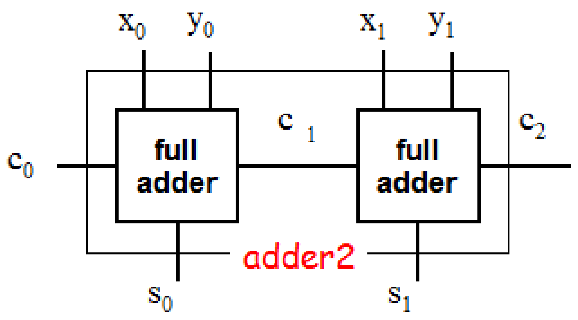
C_{in}+x+y	cs
0+0+0	00
0+0+1	01
0+1+0	01
0+1+1	10
1+0+0	01
1+0+1	10
1+1+0	10
1+1+1	11

```
(defun and-gate (x y) (if (and (= x 1) (= y 1)) 1 0))
(defun or-gate (x y) (if (or (= x 1) (= y 1)) 1 0))
(defun xor-gate (x y) (if (and (= x 1) (= y 1)) 0 (or-gate x y)))
(defun half-adder (x y) (list (xor-gate x y) (and-gate x y)))
(defun full-adder (c-in x y)
  (let* ((h1 (half-adder x y))
         (s1 (first h1))
         (c1 (second h1))
         (h2 (half-adder s1 c-in))
         (s  (first h2))
         (c2 (second h2))
         (c  (or-gate c1 c2)))
    (list s c)))
```

Figure 6.6: Full-Adder Circuit and ACL2 Model

```
(defun chk-combo (n)
  (let* ((bit-combo (twos 3 n))
         (c-in (first bit-combo))
         (x    (second bit-combo))
         (y    (third bit-combo)))
    (equal (twos 2 (+ c-in x y))
           (full-adder c-in x y))))
(deconst *chk-full-adder* ; true iff full-adder model is correct
  (and (chk-combo 0) (chk-combo 1) (chk-combo 2) (chk-combo 3)
       (chk-combo 4) (chk-combo 5) (chk-combo 6) (chk-combo 7)))
```

Figure 6.7: Mechanized Verification of Full-Adder Model



```
(defun adder2 (c0 x y)
  (let* ((x0 (first x)) (x1 (second x))
         (y0 (first y)) (y1 (second y))
         (a0 (full-adder c0 x0 y0))
         (s0 (first a0)) (c1 (second a0))
         (a1 (full-adder c1 x1 y1))
         (s1 (first a1)) (c2 (second a1)))
    (list (list s0 s1) c2)))
```

Figure 6.8: Two-Bit Adder and ACL2 Model

To make it easy to state the constraints, we define a Boolean function that delivers the value true if its input is a bit and false otherwise. The, stating the hypothesis of the implication is just a matter of forming the logical-and of tests that require the inputs to be bits. The conclusion is the equation between the sum of the numeric interpretation of the input numerals (along with the input carry) and the numeric interpretation of the output numeral (with the output carry at the end). Fortunately, we already have a trusted function "nmb" (page 75) that interprets binary numerals as numbers, so we can use that function in the statement of theorem {*adder-ok*} specifying the correctness of our model of the two-bit adder circuit.

```
(defthm adder2-thm
  (let* ((a (adder2 c0 (list x0 x1) (list y0 y1)))
         (s (first a)) (c (second a)))
        (= (nmb (append s (list c)))
           (+ (nmb(list c0)) (nmb (list x0 x1)) (nmb (list y0 y1))))))
```

EXERCISES

Ex. 68 — Define a doublecheck property that tests for correctness of the two-bit adder model (Figure 6.8, page 88). Run the test using Proof Pad or Dracula. Note: The data generator (random-between 0 1) delivers a 0 or a 1 at random.

Ex. 69 — By default, Proof Pad or Dracula does fifty repeats when it runs a doublecheck test. How many tests with random data do you think would be needed to be reasonably confident all 32 different cases for the two-bit adder have been tested?

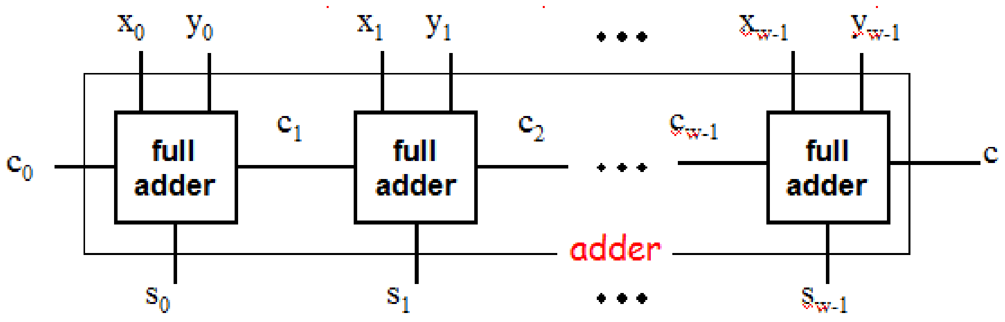
6.4 ADDING W-BIT BINARY NUMERALS

By now, you can predict what a circuit for adding three-bit binary numerals would look like. Just put another full adder in the circuit, and feed the carry from adding the two low-order bits into the carry-in of the new full adder in the circuit, along with the high-order bits in the two numerals.

Producing the circuit diagram for any number of bits is just a matter of wiring the appropriate number of full adders together in the appropriate way. Figure 6.9 (page 90) presents a schematic for doing this. The circuit is known as a "ripple-carry adder" because of the way the carry-bit propagates across the line of one-bit circuits.

The ACL2 model in Figure 6.9 (page 90) relies on inductive definition. It feeds the carry-in and the low-order bits from the two numerals into a full adder. (The low-order bit in a numeral is the "ones bit," which is the first element of the list that we use to represent the numeral.) The sum bit from that full adder is the low-order bit of the numeral representing the sum of the numbers that the input numerals represent. The remaining bits in that sum are those delivered by the adder operating on the other bits in the input numerals (that is, all the bits in the input numerals except the low-order bits).

Because the model defines a function in ACL2, you can run the function to see that it works in specific cases. To add two binary numerals, supply lists of 0s and 1s representing those numerals in an invocation of the function "adder", and specify zero as the input



```
(defun adder (c0 x y)
  (if (consp x)
      (let* ((x0 (first x)) (y0 (first y))
             (a0 (full-adder c0 x0 y0))
             (s0 (first a0)) (c1 (second a0))
             (a  (adder c1 (rest x) (rest y)))
             (ss (first a)) (c (second a)))
        (list (cons s0 ss) c))
      (list nil c0)))
```

Figure 6.9: Ripple-Carry Adder and ACL2 Model

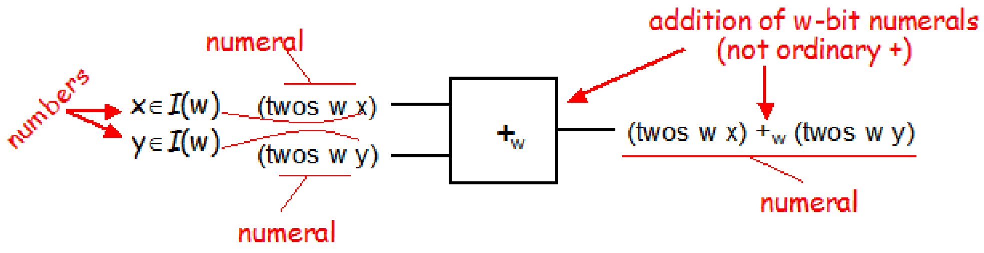
What if the input numerals have a different number of bits? Suppose one of them has eight bits, and the other has ten. The model is not designed to deal with that case. We could change the design to accommodate input numerals of differing lengths, but since we are modeling a circuit in which the input numerals must both be of a specific length, the model does not need to account for that possibility. The issue of numerals of different lengths is irrelevant in the context of the circuit.

Aside 6.1: Adder Circuit and Numerals of Different Lengths

carry-bit. The output will be the numeral for the sum, mod 2^w, of the numbers that two input numerals represent(where w is the number of bits in the two numerals). The output will also include a carry bit, which will be zero if the sum of the two numbers is less than 2^w. It will be a one-bit if sum is 2^w or greater.

Exercises

Ex. 70 — Define in ACL2 a function "add-bin" that adds any two binary numerals, even if the numerals contain a different number of bits. That is, the value (add-bin c x y) should be a binary numeral for the number (+ c (nmb x) (nmb y)), as long as x and y are binary



```
(defthm adder-thm
  (let* ((a (adder c0 x y))
         (s (first a))
         (c (second a)))
    (= (nmb (append s (list c)))
       (+ (nmb(list c0)) (nmb x) (nmb y)))))
```

Figure 6.10: Adding 2s-Complement Numerals

numerals and c is 0 or 1, regardless of (len x) or (len y). Design and run some sanity checks (check-expect) on your function.

Ex. 71 — Define in ACL2 a theorem that expresses the correctness of the function "add-bin" from the previous exercise.

6.5 Adding Numerals for Negative Numbers

The ripple-carry adder circuit (Figure 6.9, page 90) provides a way to do addition entirely through numerals. The inputs are numerals, and the output is a numeral. Figure 6.10 (page 91) depicts the idea graphically, throwing away the internal details and ignoring the input carry bit and the output carry bit.

If the input carry is zero and the input numerals are interpreted as 2s-complement numerals for numbers in the set $I(w) = \{-2^{w-1}, \cdots - 1, 0, 1, 2, \ldots 2^{w-1} - 1\}$, then the output numeral (ignoring the carry) is the 2s-complement numeral for the sum of the input numerals. The theorem stated in Figure 6.10 (page 91) formalizes this fact.

It is similar to the correctness theorem for the model of the two-bit adder (page 89), but the hypothesis of the implication must constrain inputs to be binary numerals (that is, lists of bits). An object is a list of bits if it is either constructed by the cons operation in which the first operand is a bit and the second operand is a list of bits or if it is the empty list. We define a Boolean function "bit-listp" to encapsulate the idea of a list of bits.

Theorem {*adder-ok*} (page 91) states the correctness property that we expect the adder circuit to satisfy. We now turn our attention to proving that theorem. Our proof will, of course, be about the model of the adder circuit (Figure 6.9, page 90) not the circuit itself.

proof of adder theorem goes here ...

We assume that our ACL2 models accurately encode the circuits they represent. In a full formalization, we would need a way to convert models into instructions for fabricating the circuits. We expect that you could, given a basket of logic gates, wires, and enough time, use the diagram of the adder circuit to build one for any specified word size, and we think you can convince yourself that the model matches the diagram.

Formalizing the construction of instructions a machine could follow to fabricate a circuit, given an ACL2 model, can be done using the methods we have employed to formalize other operations. But, in this treatment of the subject we have left that step to the imagination.

Aside 6.2: Models and Circuit Fabrication

EXERCISES

Ex. 72 — Diagram a circuit that employs the 2s-complement negation trick (Figure 5.3, page 80) to deliver the numeral for the negation of a number in the range $\{-2^{w-1}+1, \cdots -1, 0, 1, 2, \ldots 2^{w-1}-1\}$ (that is, a number in the set $I(w)$ other than the most negative number in that set). You may assume that you have an adder circuit for word-size w. Use the gate-like symbol in Figure 6.10 (page 91) to depict that circuit in your diagram.

Ex. 73 — Define an ACL2 model for the negation circuit diagrammed in the previous exercise. Your model may refer to the ACL2 model of a ripple carry adder in Figure 6.9 (page 90).

Ex. 74 — Define and run a doublecheck property that tests the correctness of the ACL2 model of the negation circuit from the previous exercise.

Ex. 75 — Diagram a circuit that subtracts 2s-complement numerals. You may assume that you have an adder circuit of word-size w and a negation circuit (as in the preceding exercise). Use the gate-like symbol in Figure 6.10 (page 91) to depict that circuit in your diagram. Make up a similar symbol to use for the negation circuit.

Ex. 76 — Define an ACL2 model of the 2s-complement subtraction circuit diagrammed in the previous exercise. Your model may refer to the model of a negation circuit from a previous exercise.

Ex. 77 — Define and run a doublecheck property that tests the correctness of the ACL2 model of the subtraction circuit from the previous exercise.

Ex. 78 — Design a comparison circuit that takes a pair of 2s-complement numerals as inputs and delivers the number 1 if the first number is smaller than the second and the number 0 otherwise. You may assume that you have an subtraction circuit of word-size w as described in an earlier exercise. Make up a symbol to depict that circuit in your diagram. *Hint*: Apply theorems {*minus-sign*} and {*plus-sign*} (page 81).

Ex. 79 — Define an ACL2 model of the comparison circuit for 2s-complement numerals diagrammed in the previous exercise. Your model may refer to the model of a subtraction circuit from a previous exercise.

Ex. 80 — Define and run a doublecheck property that tests the correctness of the comparison circuit for 2s-complement numerals from the previous exercise.

Seven

Multipliers and Arbitrary-Precision Arithmetic

Chapter 6 discussed circuits for adding binary numerals of a fixed word length. The ACL2 model (Figure 6.9, page 90) assumed that both numerals had exactly w bits, w being the word size. The circuit diagram (same figure) reflected this assumption by showing $2w$ input wires (w lines for each numeral) and w output wires for the bits of the numeral representing the sum.

The diagram has an additional input wire for the carry bit coming into the adder (normally a zero bit, unless the circuit is being used for some sort of multi-word arithmetic) and an additional output wire for the output carry bit. The output carry bit is normally ignored in single-word arithmetic when one summand is positive and the other is negative, but used to detect overflow[1] when they have the same sign.

The ACL2 model received the input carry as its first argument and the two input numerals as lists of length w. It delivered the output as a list of two elements, the first element being a list of w sum bits, and the second element being the carry out. The model, like the circuit diagram, did not allow for input numerals of differing lengths.

That is the nature of physical circuits. They have a fixed number of wires and gates. Software has no such constraint. A software model for adding binary numerals can accept numerals of any length, and the two numerals need not have the same length. The numeral representing the sum would then have as many bits as might be required to represent the sum of the numbers represented by the two input numerals.

An adder expressed in software that is able to deal with numerals of any length is sometimes called a "bignum" adder. It performs arbitrary-precision arithmetic rather than arithmetic on fixed word sizes. The twos-complement scheme used with arithmetic on fixed-size words does not work with arbitrary-precision arithmetic, so negative numbers have to be accounted for in some other way.

In this chapter, we will discuss arbitrary-precision arithmetic for non-negative integers, but not for negative ones. We think adding the complexity of negative numbers detracts from the clarity of the presentation without adding much in the way of intellectual depth. It also provides a fertile ground for small projects that can be used for practice with the concepts, so we leave it to that realm.

7.1 Bignum Adder

To begin, let's see what it takes to convert our ACL2 model for the ripple-carry adder to software that performs addition on binary numerals with an arbitrary number of bits,

[1]Overflow occurs when the sum of the two input numerals fails to fall into the range of integers representable in w-bit, twos-complement arithmetic.

representing, of course, numbers of unlimited size.

One way to start the discussion is to work out a way to increment a binary numeral by one. That is, given a binary numeral x representing the number n, we seek a way to compute the numeral for the number $n + 1$.

(add-1 x) = numeral for (+ (nmb x) 1)

Following our usual practice when we are trying to define an operator, we assume that someone has already defined it, and all we have to do is to write some equations that it would have to satisfy if it worked. If we are able to come up with equations that are consistent, comprehensive, and computational (Figure 4.2, page 51) we will have at least defined some operator. But not just any opertor. We will have defined the only operator that satisfies those equations).

A particularly simple situation occurs when the numeral to be incremented has no bits in it. The interpretation we settled on in Chapter 5 is that the empty numeral stands for the number zero (definition of nmb, page 75). So, incrementing the empty numeral should produce a numeral for the number 1.

(add-1 nil) = (list 1) {add1nil}

Another simple situation occurs when the low-order bit in the numeral to be incremented is a zero. In that case, the numeral for the incremented number is just like the numeral to be incremented, except that its low-order bit is a one rather than a zero.

(add-1 (cons 0 x)) = (cons 1 x) {add10}

At this point, we have equations to cover all numerals that have either no bits at all or whose low-order bit is zero. If we can work out an equation for numerals with a low-order bit of one, our equations will be comprehensive.

To do this, let's think about the low-order bit of the incremented numeral. Since adding a one-bit to a one-bit produces a sum-bit of zero and a carry-bit of one (Figure 6.4, page 85), we conclude that the low-order bit of the incremented numeral is zero.

But, what about the carry-bit? What do we do with that? It will need to be added to the higher-order bits of the input numeral. But, that is just a matter of incrementing the higher-order bits by one. The higher-order bits are, themselves, a numeral, and because of our standard assumption that someone has already defined the function we need, we can just use it to increment that numeral. That is, we can write an inductive equation that the add-1 operator must satisfy.

(add-1 (cons 1 x)) = (cons 0 (add-1 x)) {add10}

Now we have three equations, and they are consistent (no overlapping cases) and comprehensive (all cases covered). The equations are also computational because the input numeral in the inductive invocation of add-1 in the only inductive equation (equation {add10}) is a shorter numeral than the one on the left-hand-side of that equation. Therefore, the equations define the add-1 operator. All we need to do now is to translate them to proper ACL2 notation from the informal sketches we wrote down while we were thinking about the problem.

```
; (add-1 x) = numeral for (+ 1 (numb x)))
; (add-1 nil) = (list 1)                           {add1nil}
; (add-1 (cons 0 x)) = (cons 1 x)                    {add10}
; (add-1 (cons 1 x)) = (cons 0 (add-1 x))            {add11}
(defun add-1 (x)
  (if (and (consp x) (= (first x) 1))
      (cons 0 (add-1 (rest x))) ; add11
      (cons 1 (rest x))))       ; add10
```

In doing so, we observe that the {add1nil} equation and the {add10} equation can be expressed as one equation because the ACL2 formula (cons 1 (rest x)), which is the proper translation for the right-hand-side of {add10} equation, also works for the right-hand-side of {add1nil} equation because (cons 1 (rest nil)) has the same value as (list 1). This observation reduces the definition from three equations to two. That solves the increment-by-1 problem.

The ripple-carry adder propagated the carry from each bit position to the next higher-order bit position. Each bit position involved three input bits (a carry and one bit from each addend). Our bignum adder will need to do something similar. That is, each bit in the sum will depend on the corresponding bits in the addends and the carry from the previous, lower-order bit.

We already have the apparatus for this: the full-adder model (Figure 6.6). We can use that function, as it was defined on page 87, to add two corresponding bits, x_n and y_n from the addend numerals, incorporating the carry, c_n, from the lower-order bit position. The full-adder function delivers the sum bit, s_n, for the current bit position and the carry bit, c_{n+1}, for the next bit position.

$$[s_n\ c_{n+1}] = \text{(full-adder } c_n\ x_n\ y_n\text{)}$$

That analysis provides the basis for one of the equations for the bignum adder. That equation covers the situation when there are two, corresponding bits from the addend numerals to deal with. In the particular case where the corresponding bits involved are the low-order bits in the numerals, that equation would sketch out as follows:

(add c_0 $[x_0\ x_1\ x_2 \ldots]$ $[y_0\ y_1\ y_2 \ldots]$) = $[s_0\ s_1\ s_2 \ldots]$ {addxy}
where
$[s_0\ c_1]$ = (full-adder $c_0\ x_0\ y_0$)
$[s_1\ s_2 \ldots]$ = (add c_1 $[x_1\ x_2 \ldots]$ $[y_1\ y_2 \ldots]$)

This equation covers all summands whose numerals have at least one bit. So, all we need to do to make our collection of equations comprehensive is to have equations for the cases when one or the other summand is an empty numeral.

If either summand represents zero, which it would if it were an empty numeral, the sum would be the other summand, with the carry added to it. We already have a function that produces this result when the carry is one. When the carry is zero, the result is the same as the summand, so the add-1 function (page 96) provides the basis for adding a carry bit.

```
; (add-c c x) = numeral for (+ c (nmb x)))
(defun add-c (c x)
  (if (= c 1)
      (add-1 x)  ; addc1
      x))        ; addc0
```

That gives us a way to write the equations when either of the summands is empty.

(add c x nil) = (add-c c x) {add10}
(add c nil y) = (add-c c y) {add01}

Now, with all the cases covered, we can translate our sketched equations to proper ACL2 notation.

```
; arbitrary-precision adder (bignum adder)
; (add c0 x y) = numeral for (+ c0 (nmb x) (nmb y))
(defun add (c0 x y)
  (if (not (consp x))
      (add-c c0 y)                                 ; add0y
      (if (not (consp y))
          (add-c c0 x)                             ; addx0
          (let* ((x0 (first x)) ; x is not nil
                 (y0 (first y)) ; y is not nil
                 (a  (full-adder c0 x0 y0))
                 (s0 (first a))
                 (c1 (second a)))
            (cons s0 (add c1 (rest x) (rest y))))))) ; addxy

```

Here are a few examples of the bignum add function in operation:

(add 0 [0 1] [0 1]) = [0 0 1]	2 + 2 = 4
(add 0 [0 1 1 1] [1 0 1]) = [1 1 0 0 1]	14 + 5 = 19
(add 0 [1 0 0 1 1] [0 1 1]) = [1 1 1 1 1]	25 + 6 = 31

The property the bignum add function is designed to satisfy delivers a binary numeral for the sum of the numbers represented by the two input numerals and the input carry. The examples show that the property holds for three particular pairs of addends. Of course we would like to know that the property holds for all input numerals.

The following theorem is a formal statement of the property.

```
(defthm bignum-add-thm
  (= (nmb(add c x y))
     (+ (nmb(list c)) (nmb x) (nmb y))))
```

The ACL2 system succeeds in proving this theorem by induction, following a similar strategy to one we used in in Section 6.5 (page 91) to prove a theorem about the ripple-carry adder for fixed-size words. The upshot is that we now know, to a mathematical certainty, that the bignum add function delivers the sum of the two input numerals.

Exercises

Ex. 81 — Do a pencil-and-paper proof by induction of the following theorem, which says that if the high-order bit in both input numerals is a one, then the high-order bit of the numeral representing the sum that is delivered by the bignum add function is also one. The theorem refers to the fin function (page 79).

```
(defthm bignum-add-hi-order-bit-thm
  (implies (and (= (fin x) 1) (fin y 1))
           (= (fin (add c0 x y)) 1)))
```

7.2 Shift-and-Add Multiplier

The grade-school method of multiplying big numbers proceeds one digit at a time, from right (low-order digit of the decimal numeral) to left (higher order digits). The first step is to multiply the entire multiplicand by the low-order digit of the multiplier. Then comes next-to-last digit of the multiplier (the tens digit). In this case the product is written below the one from the first step, but shifted left one position. After all the products are computed for all the digits in the multiplier, they are totaled, taking care to keep the digits lined up according to the shift to the left that occurred at each stage.

Grade-school students learn this procedure without knowing the algebra behind it. However, we want to use a collection of equations to specify the product of numerals, so it will be helpful to know the equations behind the procedure. Our analysis will use the following nomenclature:

x	number represented by numeral of multiplier
x_0	low-order digit of numeral of multiplier
y	number represented by numeral of multiplicand

The procedure for multiplying numerals relies on the following equation, which grade-school students use to check the correctness of their long-division problems:

$x = (\lfloor x \div d \rfloor \cdot d) + (x \bmod d)$ {long-division}

where $\lfloor x \div d \rfloor$ stands for the greatest integer that is $x \div d$ or less
and $(x \bmod d)$ is the remainder in dividing the integers x and d (modular arithmetic)

We derive the following equation by taking $d = 10$ and multiplying both sides of the above equation y.

$xy = (\lfloor x \div 10 \rfloor \cdot 10 \cdot y) + ((x \bmod 10) \cdot y)$ {long division by 10}

Observe that $(x \bmod 10)$ is the low-order digit of the decimal numeral for x, which is what x_0 stands for in our nomenclature. So, $(x \bmod 10) \cdot y$ is $x_0 \cdot y$, (that is, the low-order digit of the numeral for x, times y). Also, the numeral for the number $\lfloor x \div 10 \rfloor$ is the same as the numeral for x, but without its low-order digit Therefore, $\lfloor x \div 10 \rfloor \cdot y$ is the number that comes from multiplying y by the numeral for x without its low-order digit.

In other words, the {long-division} equation provides the basis for computing xy. First, multiply y by the low-order digit of the numeral for x. Then, multiply y by the number represented by the other digits of the numeral for x. Finally, shift that product one digit to the left, and add the number from the first step in the procedure. It simplifies the procedure a bit to observe that because the product will be shifted before the addition takes place, the low-order digit of the sum will be the low-order digit of the numeral from the first part of the procedure, when we multiplied y by the low-order digit of the numeral for x.

The "shift" portion amounts to multiplying by 10: $\lfloor x \div 10 \rfloor \cdot y \cdot 10$ The number from the first step in the procedure is $(x \bmod 10) \cdot y$. Therefore, the procedure computes $xy = \lfloor x \div 10 \rfloor \cdot y \cdot 10 + (x \bmod 10) \cdot y$

A digital circuit for multiplication will use binary numerals rather than decimal numerals. Except for this detail, the procedure is the same as with grade-school multiplication of decimal numerals. In this case, we specialize the {long division} equation to the case when $d = 2$.

$$xy = \lfloor x \div 2 \rfloor \cdot y \cdot 2 + (x \bmod 2) \cdot y \qquad \text{\{long division by 2\}}$$

In this case, $(x \bmod 2)$ is the low-order bit in the binary numeral for x. And, $\lfloor x \div 2 \rfloor$ is the number represented by the binary numeral for x without its low-order bit. That is, if [x_0 x_1 x_2 ...] were the numeral for x, written as a sequence of bits, starting with the low-order bit, then x_0 would be $(x \bmod 2)$ and [x_1 x_2 ...] would be the numeral for $\lfloor x \div 2 \rfloor$.

The grade-school procedure, applied to binary numerals, goes as follows: First, multiply x_0 times y. The low-order bit of this product will be the low-order bit of the binary numeral of xy. Call it m_0. In grade school, this is called "bringing down" the digit. The way is clear because the other part of the product will be shifted.

Then, multiply the other bits of the binary numeral of x times the multiplicand, y. Add the binary numeral for this product to the binary numeral of x without its low-order bit. Let's give names to the bits of this numeral: [m_1 m_2 ...] Then insert the low-order bit from the first product (namely, m_0) to form the numeral for xy = [m_0 m_1 m_2 ...].

There is another aspect of using binary numerals rather than decimal numerals that simplifies the procedure. It is this:
1. If $x_0 = 0$, $x_0 \cdot y = 0$, so the empty numeral serves as the numeral for $x_0 \cdot y$.
2. If $x_0 = 1$, $x_0 \cdot y = y$, so the numeral for y serves as the numeral for $x_0 \cdot y$.

Therefore, when x_0 is zero, the m_0 that we "bring down" and insert into [m_1 m_2 ...] to form the numeral of the product of x and y is, in this case, a zero. On the other hand, when x_0 is a one, the m_0 that we bring down is the low-order bit of the numeral y.

We are looking for some equations that define a binary multiplication operator. That is, an operator that delivers the binary numeral for the numbers represented by its operands, which are also binary numerals. As usual, we assume that such and operator has already been defined, and we look for a set of consistent, comprehensive, and computational equations that it would have to satisfy.

Now, let's focus on the multiplication operator for binary numerals. Call it "mul." The invocation (mul x y), given binary numerals x and y, would deliver the numeral for the product of numbers that x and y represent. (We have shifted our nomenclature to reflect our new focus: x and y now stand for binary numerals instead of numbers.) If the numeral x is empty, it denotes the number zero, so the product will be zero, and the empty numeral will serve as the result. That gives us one equation:

(mul nil y) = nil {mul0y}

If the multiplier x = [x_0 x_1 x_2 ...] is not empty, then we are going to need to compute the product of the other high-order bits of x and y: m = (mul [x_1 x_2 ...] y).

The low-order bit of x (call it x_0, as we did earlier), must be either zero or one. If x_0 is zero, we bring down a zero and insert it into the product, m, of [x_1 x_2 ...] and y. That gives us another equation:

(mul [0 x_1 x_2 ...] y) = (cons 0 m) {mul0xy}

If x_0 is one, the computation is a bit more complicated. To examine it in detail, let u, v, and w be the numbers that the numerals x, y, and m stand for. That is u = (nmb x), v =

(nmb y), and w = (nmb m). We are trying to compute the product of the numerals x and y That is, we are trying to compute the numeral for $u \cdot v$.

Since x_0, the low-order bit of x, is one, u is an odd number. Therefore, $u = 1 + 2\lfloor u \div 2 \rfloor$, Also, $w = \lfloor u \div 2 \rfloor \cdot v$ because (nmb [x_1 x_2 ...]) = $\lfloor u \div 2 \rfloor$.

Therefore,

$$\begin{aligned} u \cdot v &= (1 + 2\lfloor u \div 2 \rfloor) \cdot v && u = 1 + 2\lfloor u \div 2 \rfloor \\ &= v + 2\lfloor u \div 2 \rfloor \cdot v && \text{\{distributive law\}} \\ &= (\text{mod } v\ 2) + 2\lfloor v \div 2 \rfloor + 2\lfloor u \div 2 \rfloor \cdot v && \text{\{long division of } v \text{ by 2\}} \\ &= (\text{mod } v\ 2) + 2(\lfloor v \div 2 \rfloor + \lfloor u \div 2 \rfloor \cdot v) && \text{\{distributive law\}} \\ &= (\text{mod } v\ 2) + 2(\lfloor v \div 2 \rfloor + w) && w = \lfloor u \div 2 \rfloor \cdot v \end{aligned}$$

We observe that (mod v 2) is the low-order bit of y, and we recall that (rest y) is the numeral for $\lfloor v \div 2 \rfloor$ and that m is the numeral for w. Therefore, we can get the numeral for $u \cdot v$ by inserting low-order bit of y, which is (mod v 2), at the beginning of the sum of the numerals (rest y) and m. This amounts to bringing down the low-order bit of y and insert it into the sum of the other bits of y and m.

That gives us another equation:

(mul [1 x_1 x_2 ...] y) = (cons (first y) (add 0 (rest y) m)) {mul1xy}

Finally, if y is empty, it denotes zero, so the empty numeral serves as the product. That gives us one more equation:

(mul x nil) = nil {mulx0}

These equations are comprehensive because y is either empty, in which case equation {mx0} applies, or y is not empty. If y is not empty, then either x is empty, in which case equation {m0y} applies, or x is not empty. If x is not empty, its low-order bit is either zero, in which case equation {m0xy} applies, or it is one, in which case equation {m1xy} applies.

The equations are consistent because in the one case where they might overlap, namely in case both x and y are empty, they deliver the same result, namely the empty numeral.

They are computational because in the inductive equations, {m0xy} and {m1xy}, the first operand in the invocation of the operator, mul, on the right-hand side is the numeral [x_1 x_2 ...], which has fewer bits than the numeral on the left-hand side, which is [x_0 x_1 x_2 ...]. So, the inductive invocation is closer to a non-inductive case than formula on the left-hand side of the equation.

Putting this all together in ACL2 notation leads to the following definitions.

```
; (mxy x y) = numeral for (nmb x)*(nmb y)
; ASSUME: y is not nil
(defun mxy (x y) ; x, y: binary numerals
  (if (consp x)
      (let* ((m  (mxy (rest x) y)))
        (if (= (first x) 1)
            (cons (first y) (add 0 (rest y) m)) ; mul1xy
            (cons 0 m)))                        ; mul0xy
      nil))                                     ; mul0y

; (mul x y) = numeral for (* (nmb x) (nmb y))
(defun mul (x y)
  (if (consp y)
      (mxy x y) ; y is not nil, invoke mxy
      nil))                                    ; mulx0
```

From these definitions, ACL2 successfully finds an inductive proof of the following theorem.

```
(defthm bignum-mul-thm
  (= (nmb (mul x y)) (* (nmb x) (nmb y))))
```

EXERCISES

Ex. 82 — Use an induction on the number of bits in the multiplier to prove the bignum multiplier theorem (bignum-mul-thm, above).

Part III

Algorithms

Eight

Multiplexors and Demultiplexors

8.1 Multiplexor

Suppose you want to take two lists and shuffle them into one. You're looking for a perfect shuffle, an element from one list, then one from the other list, back to the first list, and so on. This is sometimes called "multiplexing."

The term comes from the realm of signal transmission. There are many more signals than channels to send them on. One way to share a channel between two signals is to send a small part of one signal, then part of the other, then part of the first one again, and so on. It could be any number signals sharing the channel, not just two, but the same kind of round-robin approach would work for any number of signals. Multiplexing.

We call the operator "mux". It conforms to the pattern suggested by the following equation.

(mux [x_1 x_2 x_3 ...] [y_1 y_2 y_3 ...]) = [x_1 y_1 x_2 y_2 x_3 y_3 ...] {*mux*}

As usual we want to define the mux operator in terms of a collection of comprehensive, consistent, and computational equations that it would have to satisfy if it worked properly (Figure 4.2, page 51). If both lists are non-empty, then the first element of the multiplexed list is the first element of the first list, and the second element of the multiplexed list is the first element of the other list. So, the following formula would get the first two elements of the multiplexed list right.

(mux (cons x xs) (cons y ys)) = (cons x (cons y ... *rest of formula goes here* ...))

Fortunately, there is no great mystery concerning the missing part of the formula. Multiplexing what's left of the two input lists will get all the elements in the right place for the perfect shuffle that the mux operator is supposed to deliver.

That gives us an equation that the mux operator would have to satisfy if it worked properly:

(mux (cons x xs) (cons y ys)) = (cons x (cons y (mux xs ys))) {*mux11*}

The {mux11} equation covers the case when both lists are non-empty. It's an inductive equation, so we need to be careful to make sure that the invocation of the mux operator on the right-hand-side represents a smaller computation the invocation on the left. If not, the equation will fail to be computational, and we won't be able to use it to define the operator. We observe that the operands in the invocation on the right are shorter lists than the operands on the left. Only element shorter, but that's enough. There is less data to multiplex, and we would expect that to require less computation.

Therefore, the equation {*mux11*} can be used as a defining axiom. It applies whenever both lists are non-empty. If both lists are empty, there is nothing to multiplex, so mux would deliver the empty list in that case, but what should it deliver if one list is empty, but the other isn't?

There are several reasonable choices, and each of them produces a different operotr. One choice is to incorporate the elements in the non-empty list, just as they are, into the multiplexed list that the mux operator delivers.. That would make mux satisfy the following equations.

(mux nil ys) = ys {*mux01*}
(mux xs nil) = xs {*mux10*}

The three equations, {*mux01*}, {*mux10*}, and {*mux11*}, are comprehensive (either both operands are non-empty or at least one of them is) and computational (inductive invocations involve less computation). They are consistent because in the only overlapping case which occurs when both lists are empty, both equations ({mux01} and {mux10}) specify the same result (namely, the empty list). We can, therefore, take the equations as axioms defining the mux operator.

Converting the axioms to ACL2 notation leads to the following definition.

```
(defun mux (xs ys)
  (if (not (consp xs))
      ys                                        ; mux0
      (if (not (consp ys))
          xs                                    ; mux10
          (cons (first xs)
                (cons (first ys)
                      (mux (rest xs) (rest ys)))))) ; mux11
```

As always, the axioms that define an operator determine not only the properties they specify but also determine all of the other properties of the operator. What properties would we expect the mux operator to have? Surely, the number of elements in the multiplexed list would be the sum of the lengths of it operands. The following ACL2 definition states this property formally, and ACL2 succeeds in finding a proof by induction.

```
(defthm mux-length-thm
  (= (len (mux xs ys))
     (+ (len xs) (len ys))))
```

For practice, let's construct a pencil-and-paper proof of the theorem. There are many ways to proceed. One is to induct on the length of the first operand. We are trying to prove that the following equation holds for all natural numbers, n.

(len(mux [x_1 x_2 $\ldots$ x_n] ys)) = n + (len ys) {$L(n)$}

The proof of the base case, L(0) (that is, xs = nil), is short.

Base case: L(0) ≡ ((len(mux nil ys)) = 0 + (len ys))

	(len(mux nil ys))	
$=$	(len ys)	{*mux01*}
$=$	0 + (len ys)	{*algebra*}

The multiplexor operator can be defined with two equations instead of three by swapping the operands in the inductive equation. When the first operand is non-empty, then mux must satisfy the following equation.

(mux (cons x xs) ys) = (cons x (mux ys xs))) {mux1y}

The inductive invocation "(mux ys xs)" on the right-hand-side of the equation {mux1y} delivers a perfect shuffle of the lists, starting with the ys list. So, the formula on the right hand side is a list that starts with x, then alternates between elements of ys and xs, which is exactly what mux should deliver. This leads to a two-equation definition of the mux operator. The axiom {mux01} from the three-equation definition (page 106) covers the case when the first operand is empty.

These two equations form a definition of the mux operator that delivers the same results as the three-equation definition. However, the two-equation definition makes reasoning a bit more complicated. It must take into account the role-switch between the operands that the inductive equation employs. The ACL2 system does not find this reasoning trick on its own. It needs a some guidance. To make ACL2 admit the two-equation definition, the program must declare an induction scheme, as in the following defun.

```
(defun mux (xs ys)       ; declare induction scheme
   (declare (xargs :measure (+ (len xs) (len ys))))
   (if (consp xs)
       (cons (first xs) (mux ys (rest xs))) ; mux1y
       ys))                                 ; mux01
```

Aside 8.1: Defining mux with Two Equations

Inductive case: L(0) $\equiv$ (len(mux [x_1 x_2 ... x_{n+1}] ys)) = $(n+1)$ + (len ys)

We split the inductive case, $L(n+1)$, into two parts. The second operand of mux is either nil or it's not. We derive the conclusion from both possibilities and cite the {$\vee$ elimination} inference rule (page 31) to conclude that $L(n+1)$ holds.

The proof of the part when ys is nil is like the proof when xs is nil, except that it cites {*mux10*} instead of {*mux01*}, The proof of the part when the second operand is non-empty, and therefore has the form (cons y ys), is more complicated. Let's look at it in detail.

	(len(mux [x_1 x_2 ... x_{n+1}] (cons y ys)))	
$=$	(len(mux (cons x_1 [x_2 ... x_{n+1}] (cons y ys))))	{*cons*} (page 43)
$=$	(len(cons x (cons y (mux [x_2 ... x_{n+1}] (cons ys)))))	{*algebra*}
$=$	1 + (1 + (len(mux [x_2 ... x_{n+1}] (cons ys))))	{*len1*} twice (page 45)
$=$	1 + (1 + (n + (len(cons ys))))	{*L(n)*} (ind hyp)
$=$	(n + 1) + (1 + (len(cons ys)))	{*algebra*}
$=$	(n + 1) + (len(cons y ys))	{*len1*}

This completes the proof of the inductive case, so we conclude that the equation L(n) is true for all natural numbers n. Therefore, the mux-length theorem is true.

The next section discusses an operator that goes in the other direction. It "demultiplexes" a list into two lists, inverting the perfect shuffle. We will prove that the demultiplexor preserves total length and values, like the multiplexor. We will also prove that the two operators invert each other.

EXERCISES

Ex. 83 — Our proof of the inductive case, L($n+1$), of the mux-length theorem (page 107) glossed over the part when the second operand is empty. Complete that part of the proof. That is, prove the following equation.

(len(mux [x_1 x_2 $\ldots$ x_{n+1}] nil)) = $(n+1)$ + (len nil)

Ex. 84 — Prove that the mux operator neither adds nor loses values from its operands. That is, a value that occurs in either xs or ys also occurs in (mux xs ys) and, vice versa, a value that occurs in (mux xs ys) also occurs in either xs or ys. We will call this the "mux-val" theorem:

((occurs-in v xs) $\vee$ (occurs-in v ys)) $\leftrightarrow$ (occurs-in v (mux xs ys)) {*mux-val-thm*}

The "$\leftrightarrow$" operator in the formula is Boolean equivalence. If p and q are Boolean values (True or False), then $p \leftrightarrow q = ((p \rightarrow q) \wedge (q \rightarrow p))$.

The "occurs-in" predicate is defined as follows.

(occurs-in v xs) = (consp xs) $\wedge$ ((v = (first xs)) $\vee$ (occurs-in v (rest xs))) {*occurs-in*}

Hint: The inductive case of your proof (that is, the case when xs is non-empty) can cite the {$\vee$ elimination} inference rule, as in the proof of the mux-length theorem (page 106). The proof of the inductive case will have two parts. In one part, the value v will be equal to the first element of xs: v = (first xs). In the other part, v will occur in (rest xs). That is, (occurs-in v (rest xs)) will be true. Prove both parts, separately. Since they have the same conclusion, {$\vee$ elimination} confirms that the conclusion is true.

8.2 DEMULTIPLEXOR

A demultiplexor transforms a list of signals that alternate between x-values and y-values into two lists, with the x-values in one list and y-values in the other.

(dmx [x_1 y_1 x_2 y_2 x_3 y_3 $\ldots$]) = [[x_1 x_2 x_3 $\ldots$] [y_1 y_2 y_3 $\ldots$]] {*dmx*}

If dmx works properly, then it should package a list that alternates between y-values and x-values (starting with y-values) into two lists, the first one containing the y-values, the second, the x-values.

(dmx [y_1 x_2 y_2 x_3 y_3 $\ldots$]) = [[y_1 y_2 y_3 $\ldots$] [x_2 x_3 $\ldots$]]

Therefore, dmx must satisfy the following equation.

(dmx (cons x yxs)) = [(cons x xs) ys] {*dmx1*}
where [ys xs] = (dmx yxs)

The equation {*dmx1*} applies whenever the operand is non-empty. When the operand is empty, dmx must deliver a package of two empty lists:

(dmx nil) = [nil nil] {*dmx0*}

Together, the equations {*dmx0*} and {*dmx1*} are comprehensive, consistent, and computational, so we take them as defining axioms for dmx.

The following definition in ACL2 provides a formal version of the axioms.

In Exercise 84, the mux-val theorem, and the axioms for the occurs-in predicate have been stated in the form we use for pencil-and-paper proofs. These proofs are rigorous, but not formal in the sense of the mechanized proofs of ACL2.

Below is an ACL2 formalization of these ideas, which the ACL2 system succeeds in admitting. The "iff" operator is Boolean equivalence: (iff p q) means $(p \leftrightarrow q)$, that is, $((p \to q) \wedge (q \to p))$.

```
(defun occurs-in (x xs)
  (if (consp xs)
      (or (equal x (first xs))
          (occurs-in x (rest xs)))
      nil))
(defthm mux-val-thm
  (iff (occurs-in v (mux xs ys)) ; iff means <->
       (or (occurs-in v xs)
           (occurs-in v ys))))
```

Aside 8.2: Formal Version of Mux-Val Theorem

```
(defun dmx (xys)
  (if (consp xys)
      (let* ((x (first xys))
             (ysxs (dmx (rest xys)))
             (ys (first ysxs))
             (xs (second ysxs)))
        (list (cons x xs) ys))      ; dmx1
      (list xys xys)))              ; dmx0
```

Like the the multiplexor operator, the demultiplexor operator, preserves total length and neither adds nor loses values the list supplied as its operand. ACL2 succeeds in verifying these facts without assistance, and the pencil-and-paper proofs are similar to the corresponding theorems for the multiplexor.

The two operators also satisfy some round-trip properties that provide a lot of confidence that they do what we expect them to do. Demultiplexing a list of x-y values into the list of x-values and the list of y-values, then multiplexing those two lists, reproduces the original list of x-y values.

```
(defthm mux-inverts-dmx-thm
  (equal (mux (first  (dmx xys))
              (second (dmx xys)))
         xys))
```

The ACL2 version of equation {*dmx0*} is a bit tricky. Whereas, the pencil-and-paper version of the equation says that dmx delivers the value [nil nil], the ACL2 version says it delivers the value (list xys xys), where xys is the operand. The "if" operation in the definition selects this formula when (consp xys) is false.

Normally, (consp xys) being false would mean that xys is nil. However, it doesn't necessarily carry that meaning. It just means that xys is not an object constructed by the cons operator. It could be, for example, a number, such as 27 or 12 or zero. We are expecting xys, as an operand of dmx, to be a list. If xys is a list, then the only list it could be when (consp xys) is false is the empty list. So, (list xys xys) is the same as (list nil nil) under normal circumstances. That is, when xys is a list and (consp xys) is false.

In the formal, ACL2 definition, we us the formula (list xys xys) instead of (list nil nil) to account for abnormal circumstances, in case the operand xys is not a list. That makes it possible to derive theorems that hold under more general circumstances, thereby simplifying the mechanized reasoning that ACL2 carries out.

It works the other way around, too, if the operands of the mux operator are lists of the same length.

```
(defthm dmx-inverts-mux-thm
  (implies (and (true-listp xs) (true-listp ys)
                (= (len xs) (len ys)))
           (equal (dmx (mux xs ys))
                  (list xs ys))))
```

These inversion properties provide the kinds of guarantees that people using the operators might want to know, for sure, to have confidence that the operators are correctly defined.

EXERCISES

Ex. 85 — Prove that the dmx operator preserves total length. That is, prove the theorem stated formally in ACL2 as follows.

```
(defthm dmx-length-thm
  (= (len xys)
     (+ (len (first (dmx xys)))
        (len (second (dmx xys))))))
```

Ex. 86 — State formally, in ACL2, the "dmx-val" theorem analogous to the mux-val theorem (page 109).

Ex. 87 — The dmx-val theorem (Exercise 86) says that dmx neither adds nor drops values from its operand. Do a paper-and-pencil proof of the dmx-val theorem.

Ex. 88 — Do a pencil-and-paper proof of the mux-inverts-dmx theorem (page 109).

Ex. 89 — Do a pencil-and-paper proof of the dmx-inverts-mux theorem (page 110).

Nine

Sorting

Many computing applications require arranging a list of records in an order determined by an identifying key. Alphabetical order by name, for example, or chronological order by date. Records that have been taken in a random or otherwise unpredictable order need to be rearranged, and software to perform such rearrangements employ sorting operators do so.

The problem of sorting records into a desired order by a particular key is one of the most studied areas in computing. Solutions abound. The savings that fast sorting methods provide, compared to slower methods, tend to increase with the number of records in the lists to be sorted. For example, if a fast operator consumes half the computational resources of a slower one when sorting several hundred records, it typically would consume a few percent of the resources of the slower method when sorting several thousand records and well under one per cent when there are tens of thousands of records. Many applications deal with record archives of this size, so the time required for sorting operations can be reduced from hours to seconds by choosing a faster method.

Different sorting operators are defined by different collections of equations. So, different equations do not necessarily change the results delivered, but can have a dramatic effect on the time required to deliver them.

This chapter will discuss two sorting operators that differ greatly in the amount of time they take to carry out a rearrangement, but not at all in terms of results delivered. Both operators deliver the same rearrangement of any given list. That makes them equivalent, in terms of mathematical equality, but vastly different computationally, and the chapter will analyze both the computational differences and mathematical equivalences.

The choice of defining equations for an operator amount to a choice of algorithm, and the computational resources required by the algorithm can be derived from the equations in the same manner as other properties of the operator.

Our earlier discussions of software have focussed on expectations concerning the correctness of the results delivered by operators. A concern with computational resources is something new. Now we discuss engineering choices that affect the usefulness of software as the amount of data increases. Engineering requires not only producing correct results, but also dealing with scale in practical ways.

9.1 Insertion Sort

To maintain our attention on the essentials of arranging records in order by a key, we will assume that the entire content of a record resides in its key. Furthermore, we will use numbers for keys and discuss operators that rearrange lists of numbers into increasing order. For example, if the operand of the sorting operator were the list [5 9 4 6 5 2], the operator would deliver the list [2 4 5 5 6 9], which contains the same numbers, but arranged so that the smallest one comes first, and so on up the line to the largest at the end.

Suppose someone has defined an operator that, given a number and a list of numbers that has already been arranged into increasing order, delivers the list with the given number inserted into a place that preserves order. If we call the operator "insert", then the formula (insert 8 [2 4 5 5 6 9]) would deliver [2 4 5 5 6 8 9].

What are some equations that we would expect the insert operator to satisfy? If the list were empty, then the operator would deliver a list whose only element would be the number to be inserted in the list.

(insert x nil) = (cons x nil) {*ins0*}

If the number to be inserted does not exceed the first number in the list, the operator could simply insert the number at the beginning of the list.

(insert x (cons x_1 xs)) = (cons x (cons x_1 xs)) if $x \leq x_1$ {*ins1*}

If the number to be inserted exceeds the first number in the list, we don't know where it will go in the list, but we do know it won't come first. The first number will stay first in this case. If we trust the operator to insert it in the right place, we can let it do the insertion at the proper place in among the numbers after the first one, then insert the first number from the original list into the result.

(insert x (cons x_1 xs)) = (cons x_1 (insert x xs)) if $x > x_1$ {*ins2*}

These equations, {*ins0*}, {*ins1*}, and {*ins2*}, are comprehensive, consistent, and computational, so we can take them as defining axioms for the insertion operator. The corresponding definition in ACL2 could be expressed as follows. It consolidates {*ins0*} and {*ins1*} into one equation, which is possible because the right-hand-sides of both equations simply use the cons operator to deliver a list whose first element is the first operand of the insert operator and whose other elements form the second operand.

```
(defun insert (x xs) ; assume x1 <= x2 <= x3 ...
  (if (and (consp xs) (> x (first xs)))
      (cons (first xs) (insert x (rest xs))) ; ins2
      (cons x xs)))                          ; ins1
```

Now suppose someone has defined a sorting operator called "isort". Empty lists and one-element lists already have their elements in order, by default. Therefore, the formula (isort nil) would have to deliver nil, and the formula (isort (cons x nil)) would have to deliver (cons x nil).

(isort nil) = nil {*isrt0*}

(isort (cons x nil)) = (cons x nil) {*isrt1*}

If the list to be rearrange has two or more elements, it has the form (cons x_1 (cons x_2 xs)). If the isort operator works properly, the formula (isort (cons x_2 xs)) would be a list made up of the number x_2 and all the numbers in the list xs taken together and arranged in increasing order. In this situation, the insert operator can be used to put the number x_1 in the right place in that list, producing a list made up of the numbers in the original list, rearranged into increasing order.

(isort(cons x_1 (cons x_2 xs))) = (insert x_1 (isort(cons x_2 xs))) {*isrt2*}

The equations {*isrt0*}, {*isrt1*}, and {*isrt2*} are comprehensive, consistent, and computational, so we can take them as defining axioms for the isort operator. Again, they can be consolidated into two equations because the sorted list is identical to the input list when it has only one element, or none. The definition in ACL2 could be expressed as follows.

```
(defun isort (xs)
  (if (consp (rest xs)) ; xs has 2 or more elements?
      (insert (first xs) (isort (rest xs))) ; isrt2
      xs))              ; (len xs) <= 1     ; isrt1
```

9.2 Insertion Sort: Correctness Properties

We expect the insertion-sort operator to preserve the number of elements in its operand, and to neither add nor drop values from the list. Theorems stating these properties would be similar to the corresponding theorems for the multiplex and demultiplex operators discussed in Chapter 8. The theorem on conservation of values could use the occurs-in predicate (page 108) for determining whether or not a value occurs in a list.

```
(defthm isort-len-thm
  (= (len (isort xs)) (len xs)))

(defthm isort-val-thm
  (iff (occurs-in e xs)
       (occurs-in e (isort xs))))
```

We also expect the numbers in the list that the isort operator delivers to be in increasing order. To state that property, we need a predicate to distinguish between lists containing numbers in increasing order and lists that have numbers out of order. Of course, a list with only one element or none is already in order. A list with two or more elements is in order if its first element doesn't exceed its second and if all the elements after the first element are in order.

If we call this predicate "up", it would be True if its operand is either empty or has only one element, or if it satisfies the following inductive equation.

(up(cons x_1 (cons $x2$ xs))) = ($x_1 \leq x_2$) $\wedge$ (up(cons x_2 xs)) {*up2*}

Those observations lead to the following ACL2 definition of the "up" predicate.

```
(defun up (xs) ; (up[x1 x2 x3 ...]) = x1 <= x2 <= x3 ...
  (or (not (consp (rest xs)))          ; (len xs) <= 1
      (and (<= (first xs) (second xs)) ; x1 <= x2
           (up (rest xs)))))           ; x2 <= x3 <= x4 ...
```

Our expectations about ordering in the list that the isort operator delivers can be expressed formally in ACL2 in terms of the up predicate.

```
(defthm isort-ord-thm
  (up (isort xs)))
```

ACL2 succeeds in proving the length-preservation (isort-len-thm, page 115), value-conservation (isort-val-thm, page 115), and ordering (isort-ord-thm, page 115) properties of the isort operator without assistance. Pencil and paper proofs can employ induction on the length of the list supplied as the operand of isort.

Later, we will analyze the computational behavior of the isort operator and will find that it is extremely slow for long lists. The next section will start us down a path that will end with the definition of a sorting operator that does much better, even on long lists.

EXERCISES

Ex. 90 — Do a pencil-and-paper proof that the isort operator preserves the length of its operand (isort-len-thm, page 115).

Ex. 91 — Do a pencil-and-paper proof that the isort operator neither adds nor drops numbers its operand (isort-val-thm, page 115).

Ex. 92 — Do a pencil-and-paper proof that the isort operator delivers a list arranged in increasing order (isort-ord-thm, page 115).

9.3 ORDER-PRESERVING MERGE

The multiplex operator (Section 8.1) combines two lists into one in a perfect shuffle. There are of course many ways to combine two lists, and the one we will discuss now combines the list in a way that preserves order. If the lists contain numbers arranged in increasing order, the merge operator, which we will call "mrg", will combine the numbers from both lists into one list in which the numbers are arranged in increasing order.

Two of the equations for the multiplex operator (mux, 106) specify the results when one of the lists is empty. The equations for the operator "mrg" will be the same as those for the mux operator in these cases.

(mrg nil ys) = ys {*mg0*}
(mrg xs nil) = xs {*mg1*}

It is when both lists are non-empty that the equations for mrg will differ from those of mux. When both lists are non-empty, then the merged list will start with the smaller of the values at the beginning of the operands. The remaining elements in the merged list come from merging what's left of the list whose first element is smaller with all of the elements in the other list. That divides the case when both lists are non-empty into two subcases, one when the first operand starts with the smaller number and the other when the second operand starts with the smaller number.

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys))) if $x \leq y$ {*mgx*}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys)) if $x > y$ {*mgy*}

The four equations, taken as a whole, are comprehensive because either one list or the other is empty, or both are not, in which case one of them must begin with the smaller element. They are consistent because, as with the mux operator, the only overlapping situation is when both lists are empty, in which case equation {*mg0*} delivers the same result as equation {*mg1*}. Two of the equations ({*mgx*} and {*mgy*}) are inductive, so we need to make sure they are computational. In both equations, there are fewer elements in the operands on the right-hand side than on the left-hand side. That is, the total number of elements to be merged in the inductive invocation on the right-hand side is less than the the total on the left. So, it takes less computation to carry out the merge on the right than to do the one on the left. That makes the equations computational, and we can take them as axioms for the mrg operator.

The formal definition in ACL2 is easily constructed from the pencil and paper form discussed above. However, ACL2 needs some help in finding an induction scheme to prove that the axioms provide a way for the operation to be completed under all circumstances. That is why the definition suggests inducting on the total number of elements in the two operands.

```
(defun mrg (xs ys)
  (declare (xargs :measure (+ (len xs) (len ys)))) ; induction scheme
  (if (and (consp xs) (consp ys))
      (let* ((x (first xs)) (y (first ys)))
        (if (<= x y)
            (cons x (mrg (rest xs) ys))   ; mgx
            (cons y (mrg xs (rest ys))))) ; mgy
      (if (not (consp ys))
          xs     ; ys is empty           ; mg0
          ys)))  ; xs is empty           ; mg1
```

The mrg operator preserves the total length of its operands, and it neither adds nor drops any of the numbers in those operands. The equations specifying these properties are like those of the corresponding properties of the mux operator, namely the mux-length-thm (106) and the mux-val-thm (108).

The mrg operator also preserves order. If the numbers in both operands are in increasing order, the numbers in the list it delivers are also in increasing order. A formal statement of this property can employ the same order predicate "up" (115) that was used to state a similar property of the isort operator. However, in the case of the mrg operator, the property is guaranteed only under the condition that both operands are in order. So, the property is stated as an implication.

```
(defthm mrg-ord-thm
  (implies (and (up xs) (up ys))
           (up (mrg xs ys))))
```

ACL2 can verify this property without assistance. It uses the same induction scheme that it used to prove that the mrg operator terminates. That is, it inducts on the total number of elements in the operands. A pencil-and-paper proof can follow the same strategy.

Exercises

Ex. 93 — Using the mux-length theorem (page 106) as a model, make a formal, ACL2 statement of the mrg-length theorem.

Ex. 94 — Do a pencil-and-paper proof of the mrg-length theorem (Exercise 93).

Ex. 95 — Using the mux-val theorem (page 108) as a model, make a formal, ACL2 statement of the mrg-val theorem.

Ex. 96 — Do a pencil-and-paper proof of the mrg-val theorem (Exercise 95).

Ex. 97 — Do a pencil-and-paper proof that the mrg operator preserves order (mrg-ord theorem, page 117).

9.4 Merge Sort

We can use the mrg operator (page 117), together with the demultiplexor (dmx, page **??**), to define a sorting operator that is fast enough for long lists. We use dmx to split the list into two parts, sort each part into increasing order, then use the mrg operator to combine the sorted parts back into one list while preserving order. This algorithm is known as merge-sort. Our ACL2 definition will use the name msort, similar to the name isort that we used for the insertion sort operator.

The sorting step, which involves to sorting operations (one for each part), is the inductive step. It refers to the operator being defined. The steps are computational because both of the lists delivered by dmx are shorter than the original list, so the inductive invocations represent smaller computations.

We do have to make sure that the lists dmx delivers are strictly shorter than its operand. We expect this to be true, since half of the elements go into one list, and the other half go into the other list. If the operand is an empty list, it cannot get any shorter, and dmx in that case delivers two empty lists, both of which are the same size as the operand. It also happens that if the list has only one element, one of the two lists that dmx delivers is the empty list, but the other one is the operand itself. Therefore, one part is not strictly shorter than the operand.

However, if the operand of the sorting operation has only one element, or none, the list is already in increasing order by default. So, the equations for the new sorting operator, msort, are the same as those for isort (114).

(msort nil) = nil {*msrt0*}
(msort (cons x nil)) = (cons x nil) {*msrt1*}

The other equation is the one that splits the list into two parts, sorts them (inductively), and merges the sorted lists.

(msort (cons x_1 (cons x_2 xs))) = (mrg (msort odds) (msort evns)) {*msrt2*}
where
[odds, evns] = (dmx (cons x_1 (cons x_2 xs)))

The ACL2 version of the msort operator is easily constructed from these equations. However, the ACL2 system needs some help in verifying the msort always terminates. It needs to know that the lists delivered by the dmx operator are shorter then its operand, as stated in the following theorem.

```
(defthm dmx-shortens-list-thm ; lemma to help ACL2 admit def of msort
  (implies (consp (rest xs))  ; can't shorten one-element or empty lists
           (mv-let (odds evns)
                   (dmx xs)
```

```
              (and (< (len odds) (len xs))
                   (< (len evns) (len xs))))))
```

ACL2 also needs to be told why the equations are computational (because the lengths of the lists in the inductive invocations are less than the length of the operand). Finally, ACL2 needs to know what theorem to cite to verify that the inductive operands are shorter. Withe annotations specifying these points of proof strategy, ACL2 admits the following definition of msort.

```
(defun msort (xs)
  (declare (xargs
            :measure (len xs)
            :hints (("Goal"
                    :use ((:instance dmx-shortens-list-thm))))))
  (if (consp (rest xs))   ; xs has 2 or more elements?
      (let* ((splt (dmx xs))
             (odds (first splt))
             (evns (second splt)))
        (mrg (msort odds) (msort evns)))    ; {msrt2}
      xs))                ; (len xs) <= 1     {msrt1}
```

The msort operator has the ordering, length-preserving, and value-preserving properties as the isort operator. We use the order predicate "up" (115) and the occurs-in predicate (108) to state these properties.

```
(defthm msort-order-thm-base-case
  (up (msort xs)))
(defthm msort-ord-thm
  (up (msort xs)))
(defthm msort-len-thm-base-case
  (implies (not (consp (rest xs)))
           (= (len (msort xs)) (len xs))))
(defthm msort-len-thm-inductive-case
  (= (len (msort (cons x xs)))
     (1+ (len (msort xs)))))
(defthm msort-len-thm
  (= (len (msort xs)) (len xs)))
(defthm msort-val-thm
  (iff (occurs-in e xs)
       (occurs-in e (msort xs))))
```

ACL2 succeeds in verifying two of the above properties of msort, It needs some help in verifying the value-preserving properties, but going through the litany does not provide much elucidation about how computers work, so we are going to skip it.

EXERCISES

Ex. 98 — Do a pencil-and-paper proof that, under certain conditions, the dmx operator delivers lists that are shorter than its operator (dmx-shortens-list-thm, page 118). defthm:dmx-shortens-list
Do a pencil-and-paper proof that the msort operator delivers a list that is in increasing order. (msort-ord-thm, page 119).
Do a pencil-and-paper proof that the msort operator preserves the length of its operand (msort-len-thm, page 119).
Do a pencil-and-paper proof that the msort operator preserves the values in its operand (msort-val-thm, page 119).

9.5 Analysis of Sorting Algorithms

In this section, we discuss an operational model for ACL2 that gives us a way to count the number of computational steps involved in computing the value of a formula. We use the equations defining the isort and msort operators to derive inductive equations for the number of computational steps required to rearrange lists into increasing order by isort versus msort formulations. Then, we derive from those equations, by mathematical induction, an bounds on the number of computational steps required by msort. It turns out that the number of steps for msort is proportional to the product of the number of elements in the list to be sorted and the logarithm of that number.

We also compute an approximation to the expected number of computational steps required by the isort operator to deliver its result. The number of steps for isort depends on the particular arrangement of of the numbers in the operand, so it's not always the same. But we can get an approximation of the average, and that turns out to be proportional to the square on the number of elements in the list to be sorted.

Ten

Search Trees

10.1 How to Find a Folder

Suppose you have a bunch of file folders (physical ones, those manilla-colored things, not folders on the computer system). Each folder contains some documents and is identified by a number on its filing tab. Every now and then someone gives you an identifying number and asks you to retrieve the corresponding folder. How much work is that going to be?

It depends on the number of file folders and the care you take in organizing them. If you have only a few folders, it doesn't matter much. You can just scatter them around on the desktop, and when you're asked to get one, look around until you find it. It might be in the first folder you pick up, or the second, and the worst that can happen is that you will have to look at all of the folders before you find the one you want. Since there aren't very many, it won't take long.

That's fine when there are only a few folders, but what if there were hundreds? Then, the frustration in finding the right folder might motivate you to rethink the arrangement. Scattering the folders around the desktop would be an unattractive way to keep track of them. You might go out and buy a four-drawer filing cabinet and arrange the folders in the cabinet in order by their identifying numbers. When somebody asks you for a particular folder, you quickly locate the folder by relying on the numbering. Much better, eh?

How much better? Say you have a thousand folders and you were using the old desktop method. Sometimes you would find it on the first go, sometimes the second, sometimes the thousandth. Because the folders are scattered on the desktop at random, no folder is more likely to be the one you're looking for than any other folder, so the average search time would be the average of the numbers 1, 2,. . . 1,000, which is about 500.

Now, how about the filing-cabinet method? The time it takes you here depends on how much you take advantage of the numbering, but it turns out that the typical search time will be much improved over the desktop method.

You could proceed by looking at the identifying number and comparing it to the number on the middle file in your four-drawer cabinet. If the number on the folder you're looking for is bigger than the first one in the middle file drawer, then you know to look in the bottom half of the file drawers. If it's smaller, you look in the top half. That is, you've narrowed down the search to two of the four file drawers.

Then, you could do the same thing again to narrow it down to one file drawer, and after that, look at a folder near the middle of the file drawer to figure out whether the one you're looking for is in the front half of the drawer or the back half. At each stage, you eliminate half of the folders. When you started, the folder you're looking for might have been any of the 1,000 folders. After the first step, you can limit your search to 500 folders. After the second step, 250 folders, and so on, halving the number of candidates at each stage until you finally narrow it down to one folder.

Of course, you might have found it somewhere along the way, but the worst that can happen is that you don't find it until you narrow it down to one folder. That takes ten steps, worst case, so the average case must be at least that good, which fifty times better than the 500-step average required in the desktop method.

Good as that sounds, it gets even better. If you have 10,000 folders, instead of just 100, the first go knocks it down to 5,000 folders, then 2,500, and so on down to a single folder in fourteen steps, max, for the filing cabinet method, which is hundreds of times faster than the 5,000-step average for the desktop method. If you have a million folders, the improvement is even more dramatic. With a million folders, the filing cabinet method has a twenty-five-thousand-fold advantage over the desktop method. It gets better and better, the more folders you have. Of course, you will have to buy more filing cabinets, but regardless of how many filing cabinets you have, you can halve the search space at every stage.

10.2 The Filing Cabinet Advantage

Let's work out a formula for the filing-cabinet advantage. If you have n folders, the most steps you have to make in your search is the number of times you have to successively halve n to get down to one. That number is the base-two logarithm of n. Actually, the logarithm is not an integer, it's the next integer up, so round up to the next integer. The notation $\lceil x \rceil$ means the smallest integer that is x or bigger. For example $\lceil 3.5 \rceil$ is 4.

Here are the formulas for the typical number of steps in the two methods: desktop versus filing cabinet, as functions of the number of folders.

D(n) = $\lceil n/2 \rceil$ *desktop method*
C(n) = $\lceil log_2(n) \rceil$ *filing cabinet method*

The ratio $D(n)/C(n)$ expresses the advantage of the filing-cabinet method over the desktop method. As the examples indicated, it gets better and better as n, the number of folders, gets larger. For a billion folders, there is more than a ten-million-fold advantage. Plus, the desktop would be long buried by that time, and the folders would be scattered all over the office building.

10.3 The New-File Problem

The filing-cabinet method performs an algorithm known as a "binary search". It's a simple idea, and probably one that almost anyone would think of, faced with the problem of organizing a large number of folders for retrieval purposes. It works amazingly well, as long as the number of folders doesn't change very often.

However, when a new folder needs to be added to the collection, it can be a problem. The problem usually doesn't come up on the first few new folders. For those, there will be room to squeeze them into the drawers where they belong. The problem comes up when the number of folders keeps growing. Eventually, there is no room left in the drawer where a new folder belongs, so some of the folders in that drawer have to be moved down to another drawer to make room. What's worse, the next drawer may also be full. The problem can cascade further and further along. With a whole office building filled with cabinets for billions of folders, putting in one new folder can call for a lot of heavy lifting.

In fact, with an office building filled with cabinets, it will be advantageous to keep the folders in as few cabinets as possible. If they were scattered out, and many cabinets were nearly empty, there could be a lot of walking involved to find the appropriate cabinet.

In other words, to keep down the walking, folders must be kept in the smallest number of drawers possible. This means that the drawers involved are almost always full, and whenever a new folder comes in, a bunch of folders have to be moved around to accommodate the new one.

How much moving around? As in the desktop method, it depends on where the new folder goes. If it goes at the end, no problem. Just put it at the end of the last non-empty drawer. But, if it goes at the beginning, all of the folders must be moved down one slot to accommodate the new one. Bummer!

On the average, just as with finding folders using the desktop method, about half of the folders must be moved to accommodate a new folder. Something similar happens if a folder needs to be discarded. Again, on the average, about half of the folders must be moved to keep the folders compressed into the smallest number of drawers.

10.4 The AVL Solution

In a computer, the filing-cabinet method is normally implemented using an array to store the folders. The identifying numbers on the folders are arranged in order in the array, and to find a particular identifying number, the software looks first at the middle element of the array, then at the middle element of the narrowed ranged resulting from the first comparison of identifying numbers, and so on. That is, a binary search is performed in the array.

When a new folder comes in, it must be inserted in the place required by its identifying number to keep the numbers in order. So, all of the elements beyond that point must be moved down to make room. This can be time-consuming if there are, say, billions of folders in the array.

The same thing happens in deleting a folder from the array. All the folders beyond the point of deletion must be moved up to eliminate the gap. In either case, the number of array elements to be moved is, on the average, about half of the number of folders in the array. That's not good, and it's not an easy problem to fix. In fact, it can't be fixed with array-based methods.

In the 1960s (ancient history in the computing business), two Russian mathematicians, Adelson-Velskii and Landis, figured out a way to store folders in a tree structure so that finding a folder, adding a new folder, and removing a folder all require only about $log(n)$ steps, where n is the number of folders stored in the tree. The structure these mathematicians invented, together with their insertion and deletion methods, is known as the AVL tree.

The simple part of the Adelson-Velskii/Landis solution is to store the folders in a tree that facilitates binary search. Except for leaf-nodes, each node in the tree stores a folder and has a left subtree and a right subtree. (Leaf nodes just mark the boundaries of the tree. They don't store folders, and they don't have subtrees.) All of the folders in the left subtree have identifying numbers that are smaller than that of the folder in the root node (that is, the one you start with), and all of the folders in the right subtree have identifying numbers that are larger. To find a folder, look at the root. If the folder stored at the root has the identifying number you're looking for, deliver that folder. If not, look at the left subtree if the sought-for identifying number is smaller than the one in the root, and look in the right subtree if it is larger.

If a folder with the identifying number you're looking for is in the tree, it will be found. Otherwise, the search will arrive at a leaf-node, indicating that the required folder is not in the tree.

That's the simple part. The hard part has to do with insertion and deletion. The number of steps in the search depends on where the folder occurs in the tree, but at worst, the number of steps cannot exceed the number of levels in the tree. Nodes in a tree have subtrees, and the subtrees have subtrees, and so on. Eventually, the desired folder is encountered, or a leaf is encountered, and the search cannot continue. The maximum number of steps in this search, for all possible search paths in the tree, is known as the *height* of the tree. AVL trees maintain a balance among subtree heights, at all levels, as nodes are inserted and deleted. By maintaining balance, the AVL tree preserves a high ratio between number of folders it contains and the height of the tree.

In fact, the height of an AVL tree is approximately the base-2 logarithm of the number of folders it contains. That means every search will terminate within $log_2(n)$ steps, where n is the number of folders stored in the tree.

If you want to get a feeling for just how ingenious the AVL solution is, try to find a way to insert folders into a tree that maintains order (left subtrees have folders with smaller identifying numbers, right subtrees larger) and balance (left subtrees have about the same number of nodes as right subtrees, at all levels). To match the effectiveness of AVL trees, your method will have to be able to insert or delete a folder in about $log(n)$ steps, where n is the number of folders stored in the tree. After a few hours work, you'll begin to appreciate the contribution that Adelson-Velskii and Landis made to the technology for maintaining large data bases.

10.5 Search Trees and Occurrence of Keys

It's a long road from here to the complete AVL solution. As usual, the road starts with formulas and equations that make the ideas amenable to mathematical reasoning. The first step is to define a formal representation of a tree.

To avoid unnecessary details, the definition does not specify the kind of data associated with nodes. They may contain any kind of data. We assume that each node is identified by a numeric key, which is the basis for locating the node. We are going to assume that keys are natural numbers, but any set with an ordering among its elements could serve as the domain of the keys.

Each node in the tree is either a "leaf node", represented by "nil", or an "interior node", represented by a list of four elements: key (a number), data (of any kind), left subtree, and right subtree. The "root node" is the one that is not preceded by any other node. In the diagram, it's the node at the top. In a formula, it's the entire formula. The key of the root node is the first element of the list that represents the tree, and the data associated with the root is the second element.

We will be using "binary trees", which are trees in which each node has no more than two subtrees. We only need two subtrees to be able to use binary search to find items in the tree.

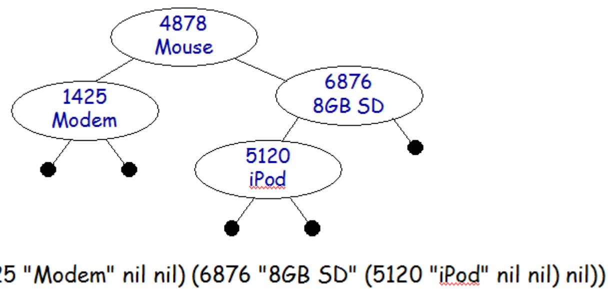

4878
Mouse
1425
Modem
6876
8GB SD
5120
iPod
(4878 "Mouse" (1425 "Modem" nil nil) (6876 "8GB SD" (5120 "iPod" nil nil) nil))

Figure 10.1: Search Tree Diagram and Corresponding Formula

Figure 10.1 shows a formula for a search tree in which the data are strings. The algebraic formula represents the tree in the diagram. We will rely extensively on diagrams to illustrate ideas expressed algebraically in terms of formulas. You will need to develop an ability to see the connection and convert freely between diagrams and formulas.

We use nil to represent the empty tree. For non-empty trees, we use four-element lists in which the key and data at the root are the first two elements and the left and right subtrees are the last two elements. A "subtree" is any part of a tree beginning with a node other than the root and continuing all the way from that node to the leaves. A key occurs in a search tree if it is the key at the root or if it occurs in either the left or right subtree.

The height of an empty tree is zero, by default. The height of a non-empty tree is one more than the larger of the heights of its left and right subtrees. The size of a search tree is the number of keys in the tree.

Figure 10.2 makes these ideas precise by providing formal definitions.

The only tree of height zero is the empty tree because the height of a non-empty tree is one more than the maximum of two other numbers, which makes it at least one. Furthermore, the height of a subtree is strictly less than the height of the tree. Theorems {*ht-emp*} and {*subtree height*} are formal statements of these facts.

Theorem {*ht-emp*} ((height s) = 0) = (emptyp s)

Theorem {*ht* $<$}. (subp r s) $\rightarrow$ ((height r) $<$ (height s))

Proof of {*ht* $<$}: induction on the height of the tree

Base case: (height s) = 0, so (emptyp s) = true {*ht-emp*}

	(subp r s) $\rightarrow \ldots$	
=	(and (treep r) (treep s) (not (emptyp s))) ...) $\rightarrow \ldots$	{*emptyp*}
=	(and true true (not true)) $\rightarrow \ldots$	{*assumption*}
=	(and true true nil) $\rightarrow \ldots$	{$\neg$ *truth table*}
=	nil $\rightarrow \ldots$	{$\wedge$ *truth table*}
=	true	{$\rightarrow$ *truth table*}

In the inductive case, (height s) $>$ 0. We conclude that (emptyp s) is false as a consequence of the {*ht-emp*} theorem. Therefore, the definition of (subp r s) reduces to (r = (lft s)) $\vee$ (subp r (lft s)) $\vee$ (r = (rgt s)) $\vee$ (subp r (rgt s)). So, we can prove the inductive case by deriving it from each of operands of this $\vee$-formula, independently ({$\vee$ *elimination*}). We refer to these four cases as "inductive case 1", inductive case 2", etc. The following proof

```
(defun key (s) (first s))          ; key at root
(defun dat (s) (second s))         ; data at root
(defun lft (s) (third s))          ; left subtree
(defun rgt (s) (fourth s))         ; right subtree
(defun emptyp (s) (not (consp s))) ; empty tree?
(defun height (s)                  ; tree height
  (if (emptyp s)
      0                                          ; ht0
      (+ 1 (max (height (lft s)) (height (rgt s)))))) ; ht1
(defun size (s)                    ; number of keys
  (if (emptyp s)
      0                                ; sz0
      (+ 1 (size (lft s)) (size (rgt s))))) ; sz1
(defun n-element-list (n xs)
  (if (zp n)
      (equal xs nil)
      (and (consp xs) (n-element-list (- n 1) (rest xs)))))
(defun treep (s)                  ; search tree?
  (or (emptyp s)
      (and (n-element-list 4 s) (natp (key s))
           (treep (lft s)) (treep (rgt s)))))
(defun subp (r s)                ; r subtree of s?
  (and (treep r) (treep s) (not (emptyp s))
       (or  (equal r (lft s)) (subp r (lft s))
            (equal r (rgt s)) (subp r (rgt s)))))
(defun keyp (k s)                ; key k occurs in s?
  (and (not (emptyp s))
       (or (= k (key s)) (keyp k (lft s)) (keyp k (rgt s)))))
```

Figure 10.2: Search Tree Functions and Predicates

deals with the inductive case 2: (subp r (lft s)). Proofs of the other inductive cases are similar.

Inductive case 2: (subp r (lft s)) is true

	(height s)	
$=$	1 + (max (height (lft s)) (height (rgt s)))	{*ht1*}
$\geq$	1 + (height (lft s))	{*algebra*}
$>$	1 + (height r)	{*induction hypothesis*}
$>$	(height r)	{*algebra*}

10.6 Ordered Search Trees

A search tree is “ordered” if the key in each non-leaf node is greater than all the keys that occur in the left subtree of the node and is less than all the keys that occur in the right subtree. A leaf is ordered by default. The following equations of predicate calculus define "ordp": (ordp s) is true if s is ordered and false otherwise.

(ordp nil) =	true	{*ord0*}
(ordp s) =	($\forall x$.((keyp x (lft s)) $\rightarrow x <$ (key s))) $\wedge$ (ordp (lft s)) $\wedge$	{*ord1*}
	($\forall y$.((keyp y (rgt s)) $\rightarrow y >$ (key s))) $\wedge$ (ordp (rgt s))	

From the definition of ordp, it's not a big step to guess that an ordered search tree cannot contain duplicate keys. However, saying exactly what that means turns out to be tricky. One approach is to prove that if a key occurs at the root, then it doesn't reside in either subtree, and if it occurs in one subtree, then it doesn't occur in the other subtree or at the root.

Theorem {*keys unique*}. (ordp s) $\rightarrow$
(((k = (key s)) $\rightarrow$ ((not(keyp k (lft s))) $\wedge$ (not(keyp k (rgt s))))) $\wedge$
(((keyp (lft s))) $\rightarrow$ (($k \neq$(key s)) $\wedge$ (not(keyp k (rgt s))))) $\wedge$
(((keyp (rgt s))) $\rightarrow$ (($k \neq$(key s)) $\wedge$ (not(keyp k (lft s))))))

Stating this theorem about as complicated as proving it. The proof simply applies the definition of ordp judiciously in each circumstance. For example, if k = (key s), then (keyp k (lft s)) must be false because if (keyp k (lft s)) were true, then $k <$ (key s), according to the definition of ordp. That is inconsistent with k = (key s), so (keyp k (lft s)) must be false ({*reductio ad absurdum*}). By a similar argument, (keyp k (lft s)) also must be false. So, the first operand of the $\wedge$ formula of the theorem is true. Proving that the other two operands are also true can be done with similar reasoning from the definition of ordp. Since each operand of the $\wedge$ formula is true when the hypothesis of the implication (ordp s) is true, the implication formula that comprises the theorem is true ({$\wedge$ implication}, page 25)

EXERCISES

Ex. 99 — Prove that (ordp s) $\rightarrow$ (((keyp k (lft s))) $\rightarrow$ (($k \neq$(key s)) $\wedge$ (not (keyp k (rgt s))))) is true.

Ex. 100 — Prove that (ordp s) $\rightarrow$ (((keyp k (rgt s))) $\rightarrow$ (($k \neq$(key s)) $\wedge$ (not (keyp k (lft s))))) is true.

10.7 BALANCED SEARCH TREES

Search trees must be ordered to make it convenient to find things. However, order is not enough. Trees must also be short, relative to the number of items in the tree. Otherwise, order doesn't help. It can take as long, on the average, to find an item as it would if they were completely unorganized. Figure 10.3 (page 128) compares some extreme cases to an optimally balanced case.

There are many ways to think about balance in trees. Size balancing, for example, requires that both sides of a tree have the same number of nodes, not only at the root, but also for every subtree. The tree of height seven in Figure 10.3 (page 128) is unbalanced at every level (except at the bottom). The tree of height four, on the other hand, has the name number of nodes on the left side of the root as on the right, but is unbalanced at the next level because the left subtree has two nodes on the left and none on the right. All subtrees

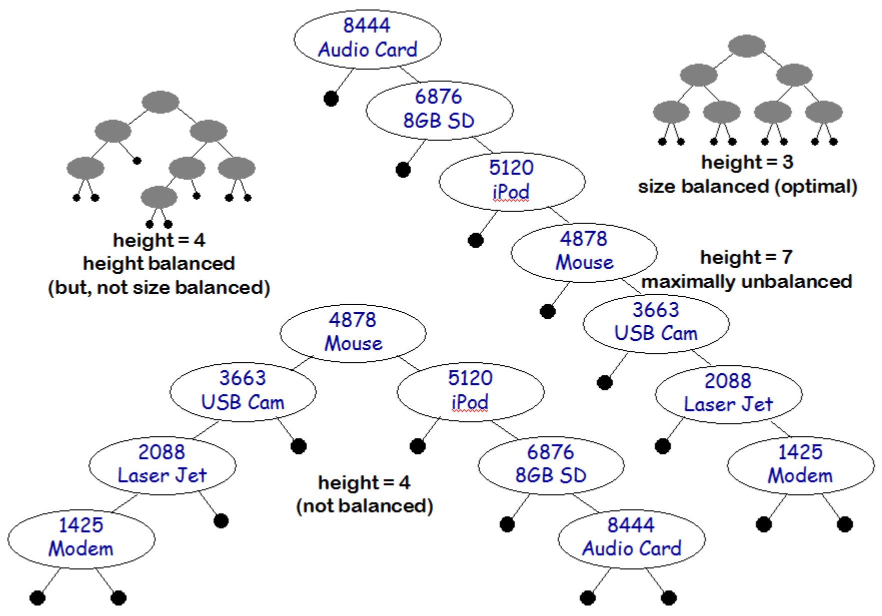


Figure 10.3: Balance Shortens Trees

in the tree of optimal height have the same number of nodes on the left as on the right, so it is size balanced. It is straightforward to demonstrate, using inductive construction, that the height of a size-balanced tree is $\lceil log_2 n \rceil$. This is optimal. The height of any tree with n nodes will be at least $\lceil log_2 n \rceil$.

It turns out that, however, that we can rely on a less strict notion of balance and still get what we need, which is logarithmic growth in tree height. A "height balanced" binary tree is

The height of a height-balanced tree is less than $1.5 \cdot log(n)$. That makes finding things take a few more steps than the optimal case, but the number of steps is still small. Finding a particular key among a billion nodes might require 45 steps instead of 30, but that's still plenty fast compared with half a billion steps, on the average, for unorganized data.

What we get for a few extra steps in finding things is an astonishing improvement in the number of steps required to insert a new node in an existing search tree. Instead of $n/2$ steps, on the average, for array-based methods (banks of filing cabinets), insertion (and deletion) can be done in logarithmic time. That is, the number of steps required to to insert or delete a node in a search tree will be proportional to $log(n)$, giving us the same advantage in insertion and deletion speed that binary search provides in look-up speed.

Exercises

Ex. 101 — Prove that it is possible to put $2^n - 1$ nodes in a binary tree of height n.

```
(defun balp (s) ; height balanced?
  (or (emptyp s)
      (and (<= (abs (- (height (lft s)) (height (rgt s)))) 1)
           (balp (lft s) (balp (rgt s))))))
```

Figure 10.4: Balance Predicate

10.8 Inserting a New Item in a Search Tree

A search tree, to be effective, must be both ordered and balanced. Since height balancing is good enough, we won't concern ourselves with optimal balancing. We will just make sure that both subtrees of any node in a search tree have the same height, or that the height of one of them is only one more than the height of the other. The function "balp" expresses this notion formally.

It is not difficult to maintaining order and balance while inserting new items into small search trees. To preserve order, simply find the place at a leaf node where the new key goes. You can do this by moving left or right down the tree according to whether the new key is smaller or larger than the key being considered. When you arrive at an empty tree, replace it with a one-node tree containing the new key, its associated data. Its subtrees will, of course, be empty. This automatically preserves order.

If you're lucky, it may also preserve balance. However, the replacement subtree has height 1, while the subtree it replaced (an empty tree) had height zero. If this takes place at a point where the tree was already a bit on the tall side, balance can get out of whack. If the trees goes out of balance, you have to rearrange it to bring it back into balance, without getting keys out of order. In small trees, it's easy to find a way to do this, as illustrated in Aside 10.1 (page 130).

How much out of whack? We have pointed out that putting the new node at the bottom can make the tree taller, but that doesn't necessarily happen. The insertion point might be on the empty side of a node with one empty side and one non-empty side. That would leave the tree height unchanged.

In other circumstances the height of the tree could change. How much could it change? Not by more than one. A proof using induction on the height of the tree confirms this assertion. We'll look into this later, but take it as a fact for now.

If the height of the new tree is one more than the height before the insertion, the new one could fail to be height balanced, even if the tree before the insertion was height balanced. However, because a height of the left subtree of a height-balanced tree does not differ from the height of the right subtree by more than one, and because the insertion of a new node cannot increase the height of either tree by more than one, the heights of the left and right subtrees in the new tree cannot differ by more than two.

So, if we can figure out how to deal with trees where balance is only off by one, we will have found a way preserve order and balance while inserting a new node.

The following example starts with a tree containing one item, then inserts three new items, one at a time. We use the formula (ins (k d) s) to denote the tree produced by inserting the key k and associated data d into the search tree s.

The end result is an ordered, balanced tree containing four items. It will aid your understanding of the insertion process if you draw diagrams similar to Figure 10.3 for the trees denoted by the formulas in the example. Verify, as you go, that each tree is both ordered and height balanced.

```
(ins (1125 "Modem")
     (8444 "Audio Card" nil nil))
     ⇓
(1125 "Modem" nil (8444 "Audio Card" nil nil))
```

```
(ins (4878 "Mouse")
     (1125 "Modem" nil (8444 "Audio Card" nil nil)))
     ⇓
(4878 "Mouse" (1125 "Modem" nil nil)
              (8444 "Audio Card" nil nil))
```

```
(ins (2088 "Laser Jet")
     (4878 "Mouse" (1125 "Modem" nil nil)
                   (8444 "Audio Card" nil nil)))
     ⇓
(2088 "Laser Jet" (1125 "Modem" nil nil)
                  (4878 "Mouse" nil (8444 "Audio Card" nil nil)))
```

Aside 10.1: Inserting New Nodes in Small Trees

Exercises

Ex. 102 — Given any three distinct keys, there is only one ordered, height balanced tree containing all of those keys and no others. Explain why.

Ex. 103 — Aside 10.1 (page 130) displays insertions leading to an ordered, height balanced search tree containing four items. The resulting trees were chosen from many, equally suitable alternatives. Except in the case where a new item was inserted into a tree with exactly two items, there was more than one way to configure resulting tree. Write formulas for ordered, height balanced trees different from the ones in the example that, like the ones in the example, contain the new item and the old ones.

Ex. 104 — Prove by induction on tree height that insertion of a new node does not increase height by more than one. That is, prove Theorem {*i-ht*}: (height (i-ord k d s)) is either (height s) or (height s)+1, where “i-ord” is the function defined as follows.

```
(defun i-ord (k d s) ; insert k/d in search tree s
  (if (empty s)
      (mk-tr k d nil nil)
      (let* ((z  (key s)) (c (dat s))
             (lf (lft s)) (rt (rgt s)))
        (if (< k z)
            (mk-tr z c (i-ord k d lf) rt)      ; i-ord<
            (if (> k z)
                (mk-tr z c lf (i-ord k d rt))  ; i-ord>
                (mk-tr k d lf rt))))))         ; i-ord=
```

Ex. 105 — Use induction on tree height to prove the following theorem (assume that k is a natural number): Theorem {*i-ord*} ((ordp s) $\rightarrow$ (ordp (i-ord k d s))) = *True*

10.9 Inductive Strategy

Although it is not difficult to find a way to rearrange a small tree to make it balanced, finding a way to rebalance a large tree can be tricky. We are going to consider some special cases of the general problem. Then, we are going to build a solution to the full problem based on these these special cases. Most of the special cases admit a straightforward solution to the rebalancing problem, but two of them require an ingenious insight.

We'll start with some easy cases. The first point to notice is that any tree of height zero, one, or two is balanced. The formula for any such tree must take one of the following forms: nil, [z c nil nil], [z c [x a nil nil] nil], [z c nil [y b nil nil]], or [z c [x a nil nil] [y b nil nil]], and all of these trees are balanced. We will refer to this fact as Theorem {*bal-ht2*}.

We are going to define an insertion operator "ins" to put a new key in an ordered, height-balanced search tree and preserve order and balance in the process. The definition of ins will, of course, be inductive. We will assume it works for trees under a certain height n, then define its value on a tree s of height $n+1$ by using it to insert the new key in one of the subtrees of s (which of course have height n or less), then forming a new ordered and balanced tree containing all the keys of s, plus the new key.

For any natural number n, let $B(n)$ stand for the proposition that if s is an ordered, height-balanced tree of height $m \leq n+2$, then the tree (ins [k d] s) is ordered, height balanced, contains all the key/data pairs in s and also the key/data pair [k d], and has height m or $m+1$.

We are going to verify that $B(n) \rightarrow B(n+1)$ is true. The hypothesis in this implication is that ins works properly on trees of height $n+2$ or less. We then need to verify that the tree it delivers when inserting a new key in a tree of height $n+3$ is ordered, balanced, and of height $n+3$ or $n+4$. We will also verify that $B(0)$ is true and conclude, by induction, that ins works properly on all trees of height two or more. Trees of height zero or one won't pose a problem because ins will deliver an ordered tree of one or two in those cases, an any tree of height one or two is balanced (theorem {*bal-ht2*}).

Figure 10.5 (page 132) contains diagrams representing all ordered, balanced trees of height $n+3$. In the figure, the triangles represent ordered, height-balanced trees, and the letters x, y, and z stand for keys. The symbols *xL*, *xR*, *yL*, *yR*, *zL*, and *zR* designate the trees

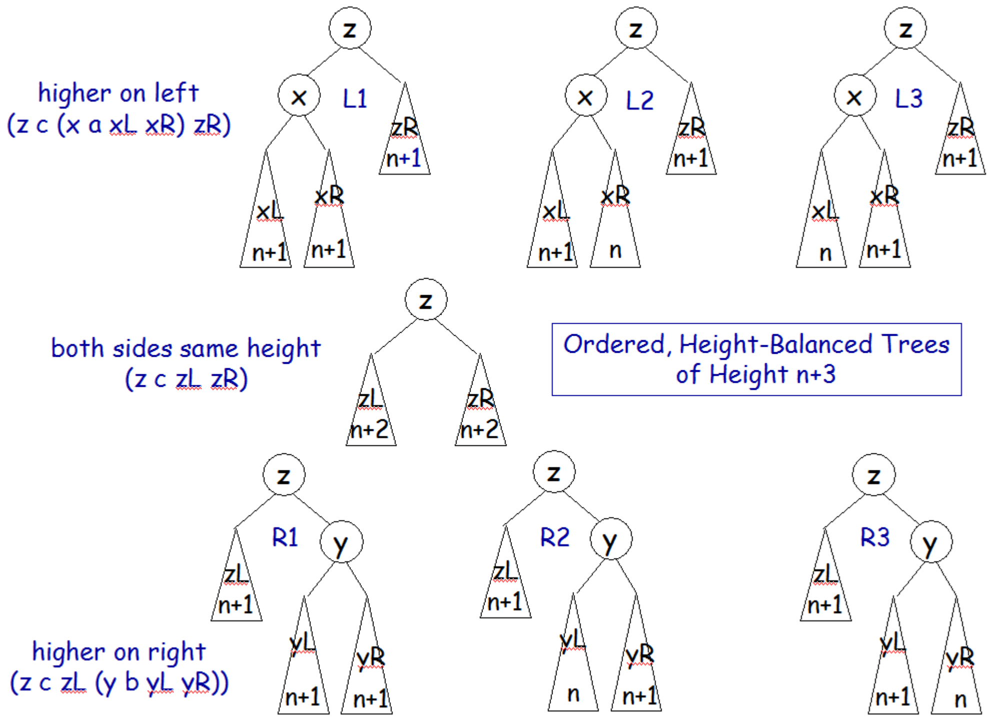


Figure 10.5: Ordered, Height-Balanced Trees of Height $n + 3$

that they label in the diagrams, and the formulas n, $n+1$, and $n+2$ in the triangles specify the heights of the trees that the triangles represent. Finally, the letters a, b, and c denote the data associated with keys x, y, and z, respectively.

We will define the ins operator to insert a new key into any of these trees. Maintaining order will dictate which subtree the new key goes into. The subtrees all have height $n + 2$ or less, so, by the induction hypothesis, ins will produce a new ordered, height-balanced subtree, either with the same height as before the insertion or taller by one. In some cases, this will force the tree out of balance, and in those cases, ins will be defined so that it rearranges the out-of-balance tree to put it back in balance.

Exercises

Ex. 106 — Prove theorem {*unbal-ht3*}: If s is an unbalanced tree of height three, then one subtree of s must be empty and the the other must have height two. That is, the following implication is a tautology.
((height s) = 3 $\wedge$ ($\neg$ (balp s))) $\rightarrow$
((height (lft s)) = 2 $\wedge$ (emptyp (rgt s))) $\vee$ ((emptyp (lft s)) $\wedge$ ((height (rgt s)) = 2))

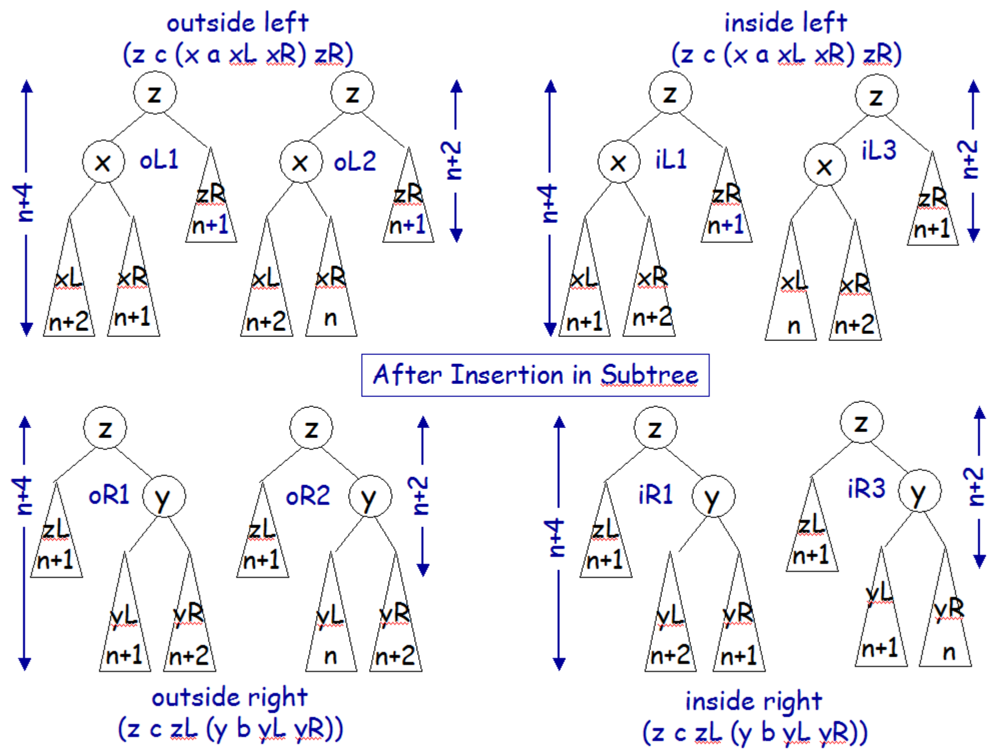


Figure 10.6: After Insertion into Tree of Height $n + 3$

10.10 Some Easy Cases

The complexity of inserting a new key in a search tree depends on where it goes, which is determined by the relationship between its value and those of the keys already in the tree. Suppose, for example, we want to insert the key/data pair (k d) into a tree s matching the tree the diagram in the middle of Figure 10.5 (page 132). If $k < z$, it will go into subtree *zL*, and if $k > z$, it will go into subtree *zR*. In either case, the new subtree will have height $n + 2$ or $n + 3$ because both *zL* and *zR* have height $n + 2$. So, the new tree will remain height balanced because all of its subtrees are balanced and the heights of its left and right subtrees will differ by one or zero.

Similarly, if s matches the tree *L1* in Figure 10.5 and $k > z$, the new key will go into subtree *zR*, and the tree will remain balanced. Likewise if s matches tree *L3* or *L3* and $k > z$, and there are several other cases where the resulting tree is balanced after the insertion, assuming $B(n)$ is true (that is, (ins (k d) *zR*) is an ordered, balanced tree with height $n + 1$ or $n + 2$.

The cases that need our attention are the ones where the tree goes out of balance after the new key is inserted into the appropriate subtree (that is, the subtree required to keep the keys of the tree in the proper order). The tree will go out of balance when the insertion into the subtree produces a taller subtree in any of the following cases:

1. $k < x$: (ins (k d) *xL*) in tree *L1* (Figure 10.5, page 132)
2. $k > x$: (ins (k d) *xR*) in tree *L1*
3. $k < x$: (ins (k d) *xL*) in tree *L2*
4. $k > x$: (ins (k d) *xR*) in tree *L3*
5. $k < y$: (ins (k d) *yL*) in tree *R1*
6. $k > y$: (ins (k d) *yR*) in tree *R1*
7. $k > y$: (ins (k d) *yR*) in tree *R2*
8. $k < y$: (ins (k d) *yL*) in tree *R3*

Some of these cases are easier to deal with than others. The easy ones are cases 1, 3, 6, and 7. In all of these cases, the tree will go out of balance after insertion of the new item in the subtree if the insertion delivers a taller subtree. If the tree goes out of balance, it has height $n + 4$ on one side and height $n + 2$ on the other. The factor that distinguishes these cases from the other four is that the tall part of the tree, the part that forces it out of balance, is a subtree on the outside boundary of the diagram, either on the far left (cases 1 and 3) or on the far right (cases 6 and 7).

The other four are "inside cases". That is when the tallest portion is a subtree that is not on the outside boundary of the diagram. Figure 10.6 (page 133) displays the status of the tree after the insertion of the new key into a subtree in each of the cases where the tree goes out of balance. As in Figure 10.5, balanced subtrees are drawn as triangles, x, y, and z are keys, etc. The inside cases are harder to rebalance. We'll deal with them later.

Let's first look at outside-left case 3 (page 134): the ins operator puts a new item in subtree *xL* of tree *L2* (Figure 10.5), producing tree *oL2* (Figure 10.6 (page 133). In this case, the tree is too tall on the left. Suppose we rearrange it with key x at the root and the other subtrees placed in the spots they have to go to keep the tree ordered. Tree *L2* Figure 10.7 (page 135) summarizes the situation before insertion, and tree *oL2* shows how it looks after insertion. Luckily, this rearrangement rebalances the tree, and not only for tree *oL2*. The same rearrangement works for tree *oL1* (Figure 10.6).

This type of rearrangement of an out-of-balance tree is known as a "rotation". For trees *oL1* and *oL2*, the rotation goes clockwise. We'll use the name "zig" for the operator that performs clockwise rotations. The operator "zag" will rotate in the other direction, which is what is needed to rebalance the outside right cases. Defining these operators in ACL2 is a matter of converting diagrams (Figure 10.7, page 135) to algebraic formulas.

```
(defun zig (s) ; rotate clockwise
  (let* ((z  (key s)) (c (dat s))
         (x  (key (lft s))) (a  (dat (lft s)))
         (xL (lft (lft s))) (xR (rgt (lft s)))
         (zR (rgt s)))
     (mk-tr x a xL (mk-tr z c xR zR))))
(defun zag (s) ; rotate counter-clockwise
  (let* ((z  (key s)) (c (dat s))
         (zL (lft s))
         (y  (key (rgt s))) (b  (dat (rgt s)))
         (yL (lft (rgt s))) (yR (rgt (rgt s))))
     (mk-tr y b (mk-tr z c zL yL) yR)))
```

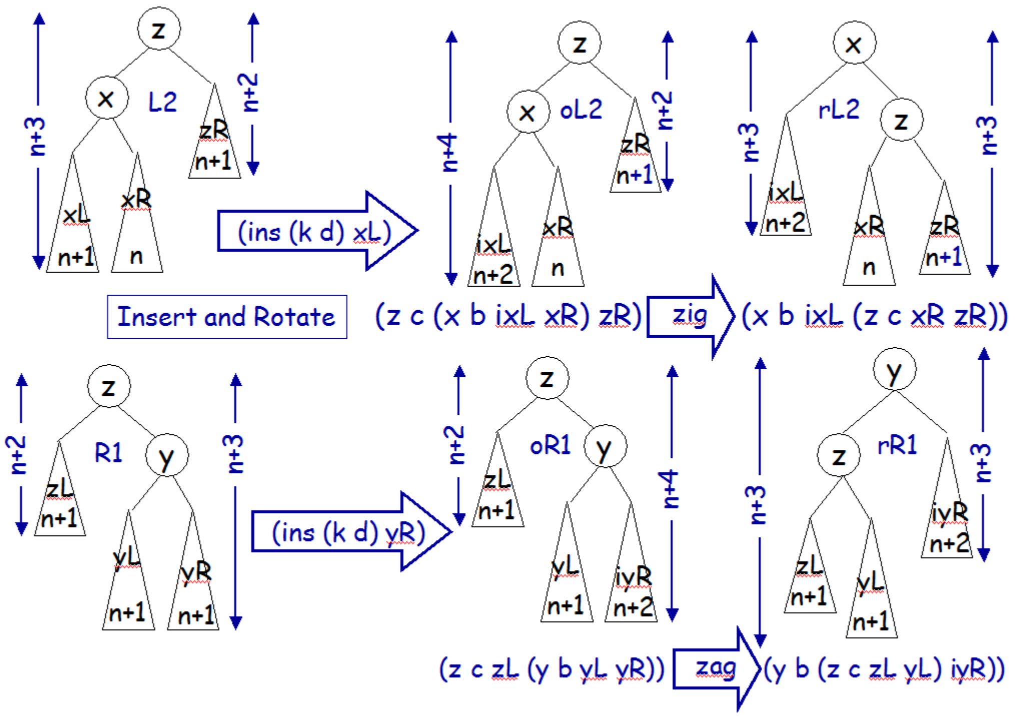


Figure 10.7: Insertion and Rotation, Outside Cases

To summarize, zig rebalances a tree with height $n + 4$ on the "outside left" and height $n + 2$ on the right, but with all subtrees balanced (Figure 10.6, 133), and zag rebalances trees that mirror this condition by being too tall on the outside right. Furthermore, all of these insertions start with a tree of height $n + 3$ and deliver a tree of the same height (see Figure 10.7, 135). The induction requires the tree, after insertion, to be either the same height as before or taller by one. We have met that requirement, so the proof by induction is complete for the easy cases (cases 1, 3, 6, and 7, page 133).

10.11 Double Rotations and Insertion

When the insertion takes place on an "inside" subtree (trees *iL1*, *iL3*, *iR1*, and *iR3* in Figure 10.6, page 133), it's not so easy to rebalance the tree. Consider tree *iL3*, for example. It looks like it needs a clockwise rotation, but zig would produce a tree with a left subtree of height n and a right subtree of height $n + 3$, which is even more out of balance than the tree we started with (*iL3*). Obviously, we need a different strategy.

Let's stick with tree *iL3* (Figure 10.6) to get our bearings. Subtree *xR* of tree *iL3* is a balanced tree of height $n + 2$. That means it cannot be empty. Consequently, it is possible to apply zag, the counterclockwise rotation to the tree [*x a xL xR*]. It's a counterintuitive move, might not help, might even make things worse, but it's possible, nonetheless. (Note: It would not be possible to apply zag if *xR* were empty.)

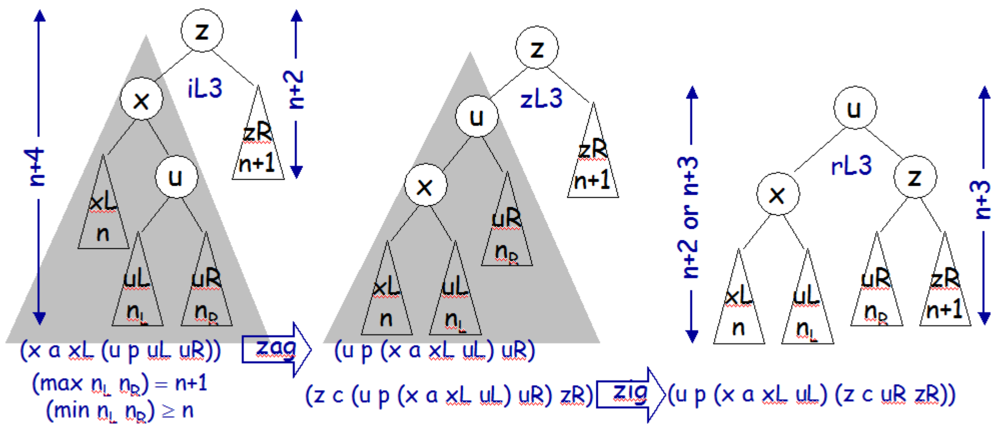


Figure 10.8: Double Rotation

To see what happens if we apply zag to xR, let's conjure up some names for the parts of *xR*: u for (key *xR*), p for (key *xR*), *uL* and *uR* for the left and right subtrees of *xR*, and n_L and n_R for the heights of *uL* and *uR* (see Figure 10.8, page 136). Then, (zag *xR*) = (zag [x a *xL* [u p *uL* *uR*]]) = [u p [x s *xL* *uL*] *uR*].

At this point, we have a tree with key z at the root and a left subtree that is not empty. This is the tree labeled *zL3* Figure 10.8 (page 136). We can use the zig operator to rotate *zL3* clockwise, producing the tree *rL3* in Figure 10.8).

Is *rL3* balanced? That depends on the values of n_L and n_R. Fortunately, we know how these heights are constrained. First, observe that $n + 2$ = (height *xR*) = (height [u p *uL* *uR*]) = (max n_L n_R) + 1, so (max n_L n_R) = $n + 1$. Furthermore, (min n_L n_R) is either n or $n + 1$, since *xR* is balanced. That ensures that at least one of the heights n_L and n_R must be $n + 1$. The other will be either n or $n + 1$.

These constraints on n_L and n_R guarantee that the left and right subtrees of *rL3* (Figure 10.8) are balanced. In addition, the height of the left subtree of *rL3* is either $n + 1$ or $n + 2$, depending on whether n_L is n or $n + 1$, and the height of the right subtree is $n + 2$. That means *rL3* is balanced and has height $n + 3$, which completes the induction for *iL3* (case 4, page 133).

This trick is a double rotation, first a counter-clockwise rotation (zag) on a subtree, then a clockwise rotation (zig) on the tree as a whole. Exactly the same double rotation works for case 2 (page 133), which is the other inside-left case (tree *iL1*, Figure 10.6, page 133).

A mirror reflection of this double rotation rebalances the inside-right cases: case 5 (page 133) rotating tree *iR1* (Figure 10.6) and case 8 (page 133) rotating tree *iR3* (Figure 10.6).

There are no other cases in which insertion can cause the tree to go out of balance, so the induction is complete. We have, at long last, proved that $B(n) \to B(n+1)$ is true. To complete the proof by induction of $\forall n.B(n)$, we need to verify $B(0)$. We would then know that the operator ins inserts a new item into an ordered, height balanced tree of height two or more and delivers an ordered, height-balanced tree of the same height or at most one taller than what we started with. Then, we would be left with a few small cases to clean up, namely that insertion works properly on trees of height zero, one, and two. We are going to leave these proofs to the exercises.

Expressing these ideas in terms of ACL2, rather than the diagrams in used the proof of $B(n) \to B(n+1)$, is a matter of converting the diagramed rotations to algebraic formulas, the applying them to define the insertion operator, ins. The function rot-R, defined as follows, performs a clockwise rotation to rebalance a tree with a left subtree whose height is two more than that of its right subtree, assuming that all of the subtrees are balanced. If the left subtree is the same height as the right subtree, or possibly taller by one, rot-R does nothing because the tree is already in balance. The counter-clockwise rotation, rot-L, is the mirror reflection of rot-R.

With rot-R and rot-L in place, an inductive definition of the insertion operator, ins, simply makes sure the insertion is performed in the proper subtree to maintain order. Of course, the new item goes into the left subtree if the new key is smaller than the key at the root, and into the right subtree if it's greater. If the new key is the same as the one at the root, we simply replace the data at the root with the new data supplied for the insertion. We could have made other choices in the case of equal keys, but this choice insures that the new tree contains the data supplied for the insertion.

```
(defun rot-R (s)     ; rebalance clockwise if ht +2 left
  (let* ((z  (key s)) (c  (dat s))         ;NOTE: s is not empty
         (zL (lft s)) (zR (rgt s)))
    (if (> (height zL) (+ (height zR) 1))           ; unbalanced?
        (if (< (height (lft zL)) (height (rgt zL))) ; inside lft?
            (zig(mk-tr z c (zag zL) zR))            ; dbl rotate
            (zig s))                                ; sngl rotate
        s)))                                        ; no rotate
(defun ins (kd s)
  (if (emptyp s)
      (mk-tr (first kd) (second kd) nil nil)      ; one-node tree
      (let* ((k (first kd)) (d (second kd))
             (z  (key s)) (c  (dat s))
             (zL (lft s)) (zR (rgt s)))
        (if (< k z)
            (rot-R (mk-tr z c (ins kd zL) zR))    ; insert left
            (if (> k z)
                (rot-L (mk-tr z c zL (ins kd zR))); insert right
                (mk-tr z d zL zR))))))            ; new root data
```

To make search trees really useful, we need to be able to delete items as well as insert them. Deletion is a little more complicated than insertion, but uses the same basic rotation operators. You know enough, now, to explore that idea on your own. Check it out on Wikipedia.

EXERCISES

Ex. 107 — A comment on page 135 points out that is it not possible to apply the zag operator to a tree whose right subtree is empty. Similarly, zig does not apply to a tree whose left subtree is empty. Explain why.

Ex. 108 — Prove $B(0)$ (see page 131 for a defintion of $B(n)$).

Ex. 109 — Prove that (ins k d nil) works properly.

Ex. 110 — Prove that (ins k d s) works properly when (height s) = 1.

Ex. 111 — Define rot-L (see the definition of rot-R, page 137).

10.12 Fast Insertion

The rotation operators, rot-L and rot-R, need to know the heights of subtrees to decide whether a rotation is needed and if so, whether it should be a single rotation or a double rotation. Computing the height of a tree takes time because it requires counting all the way to the leaves on every branch. To do this, all of the nodes must be examined, so the number of steps in the computation is proportional to the number of nodes in the tree. Since ins has to check for possible rotations at every level in the tree, there are a lot of height computations to perform, and this makes the insertion process take a lot of computation steps.

Fortunately, there is a way to avoid the height computation by recording tree height in the tree itself, along with keys, data, and subtrees. When a new tree is formed from a key, data, and two subtrees, its height can be recorded quickly by extracting the heights of the subtrees and adding one to the larger of those heights. We use the term "extracting" instead of "computing" because the height of a subtree is right there with its key. Extracting the height is a non-inductive operation, like extracting the key.

The definitions of ins and the rotations stay the same as before. The functions for tree construction and height need to be modified to use recorded height instead of computed height. Below are the new definitions of these functions.

```
(defun height (s) ; tree height
  (if (emptyp s)
      0          ; ht0
      (fifth s))) ; ht1
(defun mk-tr (k d lf rt)
  (list k d lf rt (+ 1 (max (height lf) (height rt)))))
```

In addition, the function treep, which distinguishes trees from other kinds of things, needs one minor modification. Instead of checking for a four-element list, it must check for a five-element list, since the representation of a tree now includes the height as an additional element.

The new definitions are no more complicated than the originals, but they make a huge difference in the speed of insertion. The number of computation steps required to insert a new element with the original definitions is proportional to the number of items already in the tree. Recording heights directly in the tree, rather than computing the height of the tree, reduce the number of computation steps for insertion to something proportional to the logarithm of the number of items already in the tree.

The reason the function run-to-measure-time delivers only the height of the tree it builds, rather than the tree itself, is because it would take a lot of time and space to print the tree. We're just interested in the amount of time it takes, not the tree itself. We don't have a way to deliver the computation time, directly, so we just write a formula that will cause the computation to take place. Then, we measure the amount of time the computer takes to do it.

Incidentally, the height of the tree that (tree-for-timing n) delivers should be $\lceil log_2(n+1)\rceil$. If it's not, something is wrong. In other words, the following function should always deliver the value true.

```
(defun height-right (n)
  (let* ((h     (height (tree-for-timing n)))
         (2^h   (expt 2 h))
         (2^h/2 (/ 2^h 2)))
    (and (<  2^h/2 (+ n 1))
         (<= (+ n 1) 2^h))))
```

Aside 10.2: Timing Tricks

To get a feeling for the difference, run the function measure-time, defined below, for larger and larger trees, and measure the time it takes. Start with, say 100 elements, then 200, 400, 800, and so on, doubling the size of the tree each time. Chart the number of elements and the time it takes to build the tree. One way to do measure the time is to count the number of times the Dracula cursor flashes between the time you enter the (measure-time n) formula and the time it delivers the result.

```
(defun tree-for-timing (n)
  (if (zp n)
      nil
      (ins (list n nil) (tree-for-timing (- n 1)))))
(defun measure-time (n)
  (height (tree-for-timing n)))
```

Then, change the mk-tr and height definitions from the ones that record height to the original ones that compute height. Do the timings again. You will see that the time required by the original functions soon gets unacceptably long. Inserting items one by one, starting from an empty tree, to build a tree with n items takes time proportional to $nlog(n)$ with recorded height. With computed height, it takes time proportional to n^2.

We refer to the faster algorithm as an $nlog(n)$ algorithm and the slow one as an n-squared algorithm. These are the kinds of differences in speed that turn software that isn't practical to use into software that is practical. Computer scientists expend a lot of intellectual effort figuring out how to design ways to do things that lead to performance improvements like this.

EXERCISES

Ex. 112 — Carry out the timing experiment described at the end of this section and report the results.

Eleven

Hash Tables

Part IV

Computation in Practice

Twelve

Sharding with Facebook

Facebook is a website that redefined social networking. As of this writing in 2011, just seven years after its launch in 2004, it has grown to connect more than 600 million users—about twice the total population of the United States or nearly one out of every 11 people in the entire world.

Having so many users poses tremendous technical challenges, and Facebook's success is partly due to their ability to handle these problems. Facebook may look like an ordinary website, but behind it lies incredible technical innovation. In this chapter, we'll discuss some of these innovations.

12.1 The Technical Challenge

To understand the challenges facing Facebook, let's consider just one of their main features. Facebook users can post short status updates, and they can view the status updates posted by their friends. Implementing this feature requires two pieces of information for each user:

1. a list of each user's friends, and
2. a list of each user's status updates.

That amounts to 1.2 *billion* lists, and it's not unusual for a list to have hundreds of items.

Traditionally, this data would be stored in a database using two "tables," one for the status updates, and one for the friends. Figure 12.1 shows what these tables could look like. Databases can manage tables, taking care of the details of inserting, deleting, and modifying rows. They also support sophisticated queries that can retrieve information from the tables. For example, Facebook could use the following query to determine what status updates to display when a specific user, say John, logs in:

```
SELECT   s.User, s.Time, s.Status
FROM     Friends f, Statuses s
WHERE    f.User='John'
  AND    f.Friend = s.User
ORDER BY s.Time DESC
```

This approach would work very well if Facebook had a small number of users, but it simply does not scale to 600 million users. The problem is that the database cannot process this query quickly enough to keep up with the demand from Facebook's users.

Facebook is not the only company that has this problem. For example, Amazon offers over ten million items for sale. It encourages customers to write reviews for each item, and it keeps a history of purchases made by its customers so it can make suggestions for

FRIENDS	
User	**Friend**
John	Sally
John	Mary
Mary	Sally
Sally	David

STATUSES		
User	**Time**	**Status**
John	Apr 21, 2011, 10:27 am	Checked into Starbucks in Norman.
Sally	Apr 21, 2011, 10:29 am	Saw *Battle: Los Angeles* last night. What a waste!
Mary	Apr 21, 2011, 10:32 am	Is anybody going to the carnival this weekend?
Sally	Apr 21, 2011, 10:33 am	Looks like the fires are getting closer to our house. Thinking about evacuating.

Figure 12.1: Facebook tables for storing status updates

products to buy. This information could easily be stored in tables in a traditional database, but retrieving it would take too long.

In fact, this is a problem faced by many Web 2.0 companies. By "Web 2.0" we mean a website that it allows users, as well as website owners, to produce content for the website. This can be directly in the form of passages, as in Facebook updates or Amazon reviews, or indirectly, as in Amazon recommendations. When a Web 2.0 company is successful, its users produce so more content than traditional databases can handle.

12.2 Stopgap Remedies

12.2.1 Caching

The basic problem is that traditional databases take too long to retrieve the information needed for a user's welcome screen. One way to address this is to limit the number of queries that the database needs to carry out. For example, if a user checks her home screen several times in a short time span, the computer can simply remember the previous results, instead of asking the database to retrieve them again and again, each time the user checks her screen.

That's called *caching* the data. A cache is a key/value store, like the search trees of chapter **??** or the hashtables of chapter **??**. When a query arrives, the database first checks the cache to see of the information is already there. If it is, it is simply reused. Otherwise, the database executes the query, and puts the results in the cache, so they can be reused later.

Caching is useful when three conditions are met. First, putting information in the cache and retrieving must be faster than ordinary database operations. Much faster, so there is a significant gain when the results are in the cache and only a tiny delay when they're not. Second, interactions with the database must frequently repeat the same transactions, so that results stored in the cache are often reused. Third, retrieval, rather than update, must be the dominant type of database transaction. Updates can make the data in the cache inconsistent with information in the database, forcing it to be retrieved anew, just as it would be if there were no cache at all. So, if updates are the dominant transaction, caching is a waste of time and effort.

Caching is used throughout the web. It is especially successful in storefront applications, where database queries are returning details about a particular product. Storefronts often hundreds of thousands of products, but only a handful of them are really popular. So, there are frequent requests for the same information, which makes the cache effective.

Caching worked well for Amazon, at least before product reviews and recommendations became prevalent in their product pages. But caching cannot work well for Facebook. The problem users look at their welcome pages on an individual basis, and they make frequent postings, so retrieving information does not dominate the pattern of transactions the way it does with Amazon product pages. Besides that, one update on Facebook can trigger changes in many pages of the website. This is typical in Web 2.0 applications, and this need for frequent updates of customized information for every user eliminates the advantages of caching. Another solution is needed.

12.2.2 Sharding

Sharding splits the database into many different databases. For example, John's Facebook friends and status updates may be stored in machine J, while Mary's friends and status updates are stored in machine M. Machines, like J and M, which store just a portion of the data are called *shards*. Because the database is stored on many different computers, no one computer has to shoulder the entire load. That's the upside. The downside is that it makes it harder to automate transactions with the database. Programmers have to specify how to distribute individual queries across all of the many computers that may be involved in resolving the query.

To see how this might work, suppose that the computer needs to generate John's welcome page. The first step is to find John's users, which can be done by executing a query on machine J:

```
SELECT   f.Friend
FROM     Friends f
WHERE    f.user='John'
```

The query returns John's friends, Sally and Mary. The next step is to find Sally's and Mary's status updates. That leads to the the following queries:

```
SELECT   s.Time, s.Status
FROM     Statuses s
WHERE    s.User='Sally'
ORDER BY s.Time DESC

SELECT   s.Time, s.Status
```

```
FROM     Statuses s
WHERE    s.User='Mary'
ORDER BY s.Time DESC
```

Of course, the first query should run on machine S, while the second query should on machine M. The final step is to combine the results from the two queries, and then the results need to be merged in such a way that the combined list stays in reverse chronological order.

Queries like this show one of the shortcomings of sharding. Notice that each query retrieves results from only one table. The reason is that the related records are not necessarily stored in the same shard. In this particular example, the information needed to answer the query was distributed across shards J, S, and M. So the program had to collect the information from all the different sources and then combine it. Clearly, using sharding is much more difficult than keeping all the information in one place, as in a traditional database.

It also suffers from uneven distribution. What if, for example, one of the shards ends up with too many records? That shard could be split into pieces. For example, the system could split the Ms shard into to shards, one for Ma-Mp, and another for Mq-Mz. That seems like a good idea, but in practice splitting shards is complicated because the software on all the computers that access the data need to be modified, so that they can find the shard that contains the information they're looking for.

12.3 The Cassandra Solution

Faced with these difficulties, Facebook engineers developed a solution that retained the benefits of sharding, but avoided some of the difficulties. The goal was to make easy to split a shard into multiple pieces, and to hide from the software the complexity of sharding.

Cassandra, the solution they devised, combines features from Amazon's Dynamo project and Google's BigTable. From Dynamo, Cassandra borrows the idea of a replication ring, and from BigTable a data model.

Cassandra's data model groups records into into different tables. Each record in a table is identified with a key. The key of key must be unique in a given table, but the same key may be used in different tables. Each record consists of one or more key/value pairs.

For example, John's friends may be stored in a record like the one shown in Figure 12.2. The important thing is that a program can retrieve all of John's friends by requesting the single record with ID John. (Note: Different records in a Cassandra table may have different keys.)

Once John's friends are known, it is necessary to retrieve their status updates. This can be done by looking for the records in the status table that have the appropriate record IDs. Figure 12.3 shows what the status table could look like.

The table structures we have presented assume that all fields will fit in a single record. That is, we assume that a single record can hold all of a user's friends, or all of a user's status updates. Cassandra tables are designed to support thousands of fields. This will be enough for most users, but not for the heaviest users of Facebook. To deal with the heaviest users, Facebook can reuse the same idea with column names. To support an arbitrary number of friends or status updates, the values can be spread across multiple records, with IDs such as John1, John2, etc.

FRIENDS		
Record ID	**Friend1**	**Friend2**
John	Sally	Mary
Mary	Sally	
Sally	David	

Figure 12.2: Storing friends lists in Cassandra

STATUSES				
Record ID	**Time1**	**Time2**	**Status1**	**Status2**
Sally	Apr 21, 2011, 10:29 am	Apr 21, 2011, 10:33 am	Saw *Battle: Los Angeles* last night. What a waste!	Looks like the fires are getting closer to our house. Thinking about evacuating.

Figure 12.3: Storing status updates in Cassandra

The upshot is that the workflow for retrieving information from Cassandra is very similar to the workflow for sharding, but with a major difference. The queries that are generated do not need to be aware of which shard contains the information they need.

In fact, Cassandra relies on sharding both for performance and for replication. The key innovation is that the shards are arranged in a ring. For simplicity, assume that the shards are labeled A, B, C, . . . , Z. The ring arrangement means that each shard is connected to the next, and eventually the last shard is connected to the first. For example, A could be connected to B, B to C, and so on, until Z is connected back to A.

The records are arranged in an order determined by a hash function. The record's key is hashed, and the resulting hash value is mapped to the shard labels (A-Z in the example). That is not to say that the hash function can only take on the values A, B, . . . , Z. Rather, it is that all hash values up to A are mapped to shard B, those between A and B to shard C, and so on.

The hash function and mapping of hash values to shards is known to every shard. So a program can ask any of the machines to retrieve a given value. If the shard does not have the value, it can easily determine which is the correct shard, and it forwards the request to that shard.

The ring makes it easier to rebalance the shards, in case one of them becomes too large. Suppose, for example, shard B gets too large. To balance it, a new shard, say BM, is created and inserted between B and C. During the insertion process, B sends all of its records between BM and C to shard BM. When this process completes, all shards are notified of the new shard's existence, and shard BM joins the ring. This can happen without the knowledge of any programs that are retrieving data from the system.

Cassandra also uses the ring for replication. What this means is that records that should be stored in A are *also* stored in B and C. This is important, because computers and disks can fail, but if shard A should fail, there are two more copies of its data. It can also serve to improve performance during spikes. If shard A becomes busy, shards B and C can take over some of the load.

Replication complicates the splitting of shards somewhat. When shard B is split into B and BM, for example, this also affects shards C and D. Shards C and D replicated all the records for shard B. Now this needs to be restructured, so that B's records are replicated in shards BM and C. Moreover, C and D should replicate the records for shard BM. Effectively, this means that shards C and D need to participate in the insertion of BM into the ring. In the case of C, it simply needs to know that some of the records it replicated for B are now associated with BM instead. For D, it means that some of the records it was replicating on behalf of B can now be forgotten, and the rest need to be associated with BM. Finally, BM needs to receive not just its share of B's records, but all of B's records, so that it can replicate B's remaining data.

12.4 Summary

Web 2.0 applications bring up many challenges due to scaling. These challenges go beyond what traditional database solutions can offer. So, many of the leading Web 2.0 companies have developed custom solutions based on the idea of sharding. Cassandra, Facebook's solution, represents the current state of the art. Fortunately, Facebook decided to make Cassandra available to the programming community via an open source process. Programmers can download Cassandra from `http://cassandra.apache.org` and use it to develop new software.

Thirteen

Parallel Execution with MapReduce

13.1 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.

13.2 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 not 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.

13.3 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 be 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.

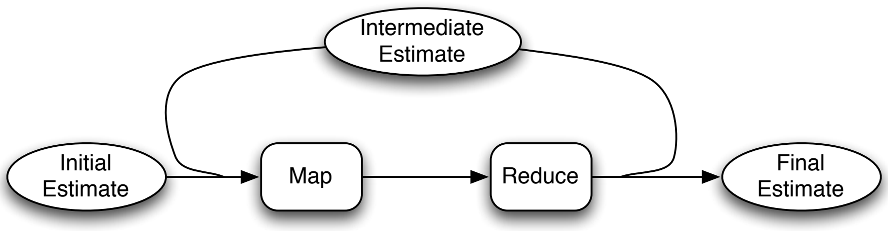


Figure 13.1: Iterative Map Reduce Operation

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 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.

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

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.

13.4 Summary

Some problems are too large to solve on a single, conventional machine. That leaves us with two options. We can either get a large machine, e.g., a supercomputer, or we can break the problem down into many tasks and execute each task on a separate computer.

The first approach, vertical scaling, is limited by the size of the largest computer that we can afford. It is also difficult to use in practice, because as the problem grows, e.g., as a website attracts a growing number of users, several scaling steps may be necessary, and each step requires an expensive migration to a more capable computer.

The alternative approach, horizontal scaling, offers the promise of virtually unlimited scalability and a gradual increase in the cost of the solution as the problem size grows. But it is much more difficult to write programs that are split across multiple machines. Also, this solution only applies to problems that can be reasonably split. Such problems usually involve a large data set, and portions of this data can be processed independently.

Based on equational programming, MapReduce is a framework that makes it easy to write programs that can be distributed across a large number of computers. MapReduce programs are organized around two functions. The Map function applies to each of the input values, and it generates an intermediate result. The intermediate results are subsequently combined using the Reduce function. If the results generated by the Reduce function are in the same format as the input to the Map function, multiple MapReduce passes can be performed. This is useful in programs that estimate optimal results by refining an initial estimate, such as programs that find clusters in a large dataset.

Fourteen

Generating Art with Computers

Colophon

This manual was typeset using the LaTeX typesetting system created by Leslie Lamport and the memoir class. The body text is set 10/12pt on a 33pc measure with Palatino designed by Hermann Zapf, which includes italics and small caps. Other fonts include Sans, Slanted and Typewriter from Donald Knuth's Computer Modern family.