COMPUTER-AIDED REASONING

An Approach

Advances in Formal Methods

Michael Hinchey
Series Editor

Other Series Titles:

The Object-Z Specification Language by Graeme Smith
ISBN: 0-7923-8684-1

Software Engineering with OBJ: Algebraic Specification in Action by Joseph A. Goguen and Grant Malcolm
ISBN: 0-7923-7757-5

Computer-Aided Reasoning: ACL2 Case Studies by Matt Kaufmann, Panagiotis Manolios and J Strother Moore
ISBN: 0-7923-7849-0

COMPUTER-AIDED REASONING

An Approach

by

Matt Kaufmann
Advanced Micro Devices, Inc.

Panagiotis Manolios
The University of Texas at Austin

J Strother Moore
The University of Texas at Austin

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

Kaufmann, Matt.
Computer-aided reasoning : an approach / by Matt Kaufmann, Panagiotis Manolios, J Strother Moore.
p. cm. -- (Advances in formal methods; 3)
Includes bibliographical references and index.
ISBN 978-1-4613-7003-1 ISBN 978-1-4615-4449-4 (eBook)
DOI 10.1007/978-1-4615-4449-4
1. Formal methods (Computer science) 2. Software engineering. 3. Expert systems (Computer science) I. Manolios, Panagiotis. II. Moore, J Strother, 1947- III. Title. IV. Series

QA76.9.F67 K38 2000
004'.01'51--dc21

00-038636

Printed on acid-free paper.

Series Foreword
Advances in Formal Methods

Michael Hinchey
Series Editor

University of Nebraska-Omaha
College of Information Science and Technology
Department of Computer Science
Omaha, NE 68182-0500 USA

Email: mhinchey@unomaha.edu

As early as 1949, computer pioneers realized that writing a program that executed as planned was no simple task. Turing even saw the need to address the issues of program correctness and termination, foretelling groundbreaking work of Edsger Dijkstra, John McCarthy, Bob Floyd, Tony Hoare, Sue Owicki and David Gries, among others, two and three decades later.

The term *formal methods* refers to the application of mathematical techniques for the specification, analysis, design, implementation and subsequent maintenance of complex computer software and hardware. These techniques have proven themselves, when correctly and appropriately applied, to result in systems of the highest quality, which are well documented, easier to maintain, and promote software reuse.

With emerging legislation, and increasing emphasis in standards and university curricula, formal methods are set to become even more important in system development. This Kluwer book series, Advances in Formal Methods, aims to present results from the cutting edge of formal methods research and practice.

Books in the series will address the use of formal methods in both hardware and software development, and the role of formal methods in the development of safety-critical and real-time systems, hardware-software co-design, testing, simulation and prototyping, software quality assurance, software reuse, security, and many other areas. The series aims to include both basic and advanced textbooks, monographs, reference books and collections of high quality papers which will address managerial issues, the development process,

requirements engineering, tool support, methods integration, metrics, reverse engineering, and issues relating to formal methods education.

It is our aim that, in due course, Advances in Formal Methods will provide a rich source of information for students, teachers, academic researchers and industrial practitioners alike. And I hope that, for many, it will be a first port-of-call for relevant texts and literature.

Computer-Aided Reasoning: An Approach and *Computer-Aided Reasoning: ACL2 Case Studies* are two complementary volumes in the Advances in Formal Methods series. The former provides an in-depth introduction to computer-supported reasoning with A Computational Logic for A Common Lisp (ACL2), the successor to Nqthm. It provides both technical details on computer-aided reasoning and an expository introduction to ACL2 and of how to apply it in system development and in reasoning about applications. The latter is a comprehensive collection of case studies, written by experts, illustrating the use of ACL2 in a range of problem areas. Individually, these volumes provide interesting reading and a firm foundation for those interested in computer-aided reasoning; together, they form the most authoritative source for information on ACL2 and its practical application.

Professor Mike Hinchey

To Holly, Helen, and Jo

Contents

Preface

This story grew in the telling. We set out to edit the proceedings of a workshop on the ACL2 theorem prover—adding a little introductory material to tie the research papers together—and ended up not with one but with two books. The subject of both books is computer-aided reasoning, from the ACL2 perspective. The first book, this one, is about *how* to do it; the second book is about *what* can be done.

The creation of ACL2, by Kaufmann and Moore, was the first step in the process of writing this book. It was a step that took many years and we acknowledge here the people who made it possible.

Foremost is Bob Boyer, whose creative insights, clarity of thought, and hard work helped us create the earliest versions of ACL2 from its predecessor system, the Boyer-Moore theorem prover, Nqthm. Bob's contributions are so numerous and pervasive that attempting to list them does him an injustice.

For the first eight years of ACL2's existence it was nurtured in the creative and relaxed environment provided by Computational Logic, Inc. (CLI). The company closed its doors in 1997, but Kaufmann and Moore owe a debt of gratitude to their former colleagues at CLI. In particular, we could not have had the opportunity there for sustained focus on our research were it not for the enlightened management and tireless fund-raising of Don Good, the founding president of CLI, and Warren Hunt, the vice president for hardware research.

ACL2 was blessed by having among its earliest users many expert users of its predecessor, Nqthm. They expected a certain level of reliability and performance and helped us achieve it. In addition, many of the earliest users helped create collections of useful theorems, helped port the system to various implementations of Common Lisp, or showed how to model new problems with it. These users include Ken Albin, Larry Akers, Bill Bevier, Bishop Brock, Alessandro Cimatti, Rich Cohen, John Cowles, Art Flatau, Ruben Gamboa, Warren Hunt, Bill Legato, Dave Opitz, Laurence Pierre, David Russinoff, Jun Sawada, Larry Smith, Mike Smith, Matthew Wilding, and Bill Young.

We wish to make special note of Bishop Brock's impact on ACL2. Among other contributions, he implemented a prototype for congruence-based rewriting, developed many of the public books of lemmas that enable

others to use the system, and repeatedly challenged us to improve ACL2. In a demonstration of his courage and faith in us, he pushed for CLI to agree to the Motorola CAP contract—which required formalizing a commercial DSP in the untested ACL2—and then he moved to Scottsdale, Arizona, to do the work with the Motorola design team. His demonstration of ACL2's utility was an inspiration.

Art Flatau, Noah Friedman, Laura Lawless, and Bill Young helped create the online documentation. David Greve, David Hardin, Robert Krug, William McCune, and Rob Sumners have suggested several important improvements, as have a number of users thanked elsewhere. Fausto Giunchiglia has been an enthusiastic critic and devoted supporter. Finally, Kaufmann and Moore would like to thank Panagiotis (Pete) Manolios for his careful scrutiny of parts of the system and his many helpful suggestions as an ACL2 user. With users like these, it would be hard not to have a useful and reliable system.

We thank the organizations that were willing to try out ACL2 on early projects, including Motorola Government Systems and Technology Group, in Scottsdale, Arizona, and the Computation Products Group of Advanced Micro Devices, Inc., in Austin, Texas. Calvin Harrison and Tom Lynch were early technical supporters who challenged us to make good the promise of formal methods and helped us meet the challenges.

Financial and moral support during the first eight years of ACL2's creation was provided by the U.S. Department of Defense, including DARPA and the Office of Naval Research, and Computational Logic, Inc. We are especially grateful to George Cotter, Norm Glick, Terry Ireland, Bill Legato, Robert Morris, Bill Scherlis, and Ralph Wachter. Subsequently, ACL2's development has been supported in part by the University of Texas at Austin, the Austin Renovation Center of EDS, Inc., Advanced Micro Devices, Inc., and Rockwell Collins, Inc. We especially thank Joe Hill and Dave Reed. The support provided by all these people transcends just paying the bills.

We could not have created ACL2 without the efforts of those in the Lisp community who helped define Common Lisp, the many people who used Nqthm, and the students who took Boyer and Moore's *Recursion and Induction* classes. We are grateful to Bill Schelter for GCL, which has provided a reliable, free Common Lisp platform for ACL2. Finally, we owe a continuing debt of gratitude to ACL2's user community.

Turning from the ACL2 system to this book, we owe our greatest debt to the participants in the 1999 workshop where the idea of these books was born. Ken Albin, Warren Hunt, and Matthew Wilding were among the first to push for a workshop. We thank those participants who wrote material for the companion book, including Vernon Austel, Piergiorgio Bertoli, Dominique Borrione, John Cowles, Art Flatau, Ruben Gamboa, Philippe Georgelin, Wolfgang Goerigk, David Greve, David Hardin, Warren Hunt, Damir Jamsek, William McCune, Vanderlei Rodrigues, David Russinoff, Jun Sawada, Olga Shumsky, Paolo Traverso, and Matthew Wilding.

In addition to all the contributors and many of the users named above, we thank Rajeev Joshi, Yi Mao, Jennifer Maas, George Porter, and David Streckmann for proof reading drafts of various parts of the book.

We thank the series editor, Mike Hinchey, and Lance Wobus at Kluwer, who patiently tolerated and adjusted to the increasing scope of this enterprise.

For many months now, much of our "free time" has been spent writing and editing these books. Without the cooperation and understanding support of our wives we simply would not have done it. So we thank them most of all.

Matt Kaufmann
Panagiotis Manolios
J Strother Moore

Austin, Texas
February 2000

Part I

Preliminaries

1

Introduction

This book is a textbook introduction to applied formal reasoning. We show how to use a formal logic to define concepts and to state and prove theorems. Moreover, we show how to carry out these tasks in cooperation with a particular computed-aided reasoning system.

This book is meant for students and professionals in hardware or software engineering, formal methods, or symbolic reasoning. It is also aimed at those practitioners who wish to learn how to apply mechanized formal methods. The book is linked to additional material on the Web (as described below), including the computer assisted reasoning engine used here. The book contains about 140 exercises for readers wishing to master the techniques. Solutions are on the Web, along with additional exercises. A companion book, *Computer-Aided Reasoning: ACL2 Case Studies* [22], continues this tutorial approach with case studies of more complex problems, often distilled from industrial applications. Each case study is accompanied, on the Web, by its full solution and the solutions to all of its exercises.

In this introduction we discuss the structure of this book, the material on the Web, the role of the exercises, and the intended audience in more detail.

While most work in logic is *about* logic, we *use* logic to reason about other systems—mostly computing systems. The name that has been given to this application of logic is *formal methods*. Formality can be confining and tedious. Every idea must be expressed as a formula. On the other hand, what is the alternative? Who checks that your definitions are meaningful? Are you free to introduce "obvious" axioms whenever you wish? Who keeps you honest? In traditional mathematics, the answer is: peer review. But in our applications that is often impractical.

Fortunately, formality offers a unique advantage over informal methods: it is possible to check mechanically that one's "proofs" are valid. Unfortunately, formal proofs tend to be extremely long. Consider, for example, attempting to prove $2^{64} + 3^{19} = 18446744074871813083$ in a system like Peano arithmetic! Thus, it is easy to advocate a formal system but then revert to traditional informal mathematics while claiming to be formal.

We put forth a formal system and use it. We stay honest by using a mechanized tool: ACL2. With it we can check that our "formulas" are for-

M. Kaufmann et al., *Computer-Aided Reasoning*

mulas, that our "definitions" define functions, and that our "theorems" are theorems. ACL2 does not check or create formal proofs. Rather, it checks that proof outlines (often just statements of theorems) can in principle be turned into formal proofs. In this way, we can use ACL2 to reason about large, industrial-scale systems.

We chose ACL2 because its authors are Kaufmann and Moore[1], but ACL2, *per se*, is not essential to this enterprise. There are other mechanized logics one can use, *e.g.*, Nqthm [7], HOL [16], Otter [29], and PVS [13]. But we believe that using some mechanized tool is critical for using formal methods effectively on industrial applications. Without automated assistance from a robust tool such as ACL2, many potential users of formal methods are driven towards informality.

Formal methods are useful only if they scale to problems more complex than you can manage with informal methods. Logic and mathematics scale. But do the mechanical tools? Is ACL2 useful or is it just a pedagogical toy?

Boyer, Moore, and Kaufmann designed ACL2 in response to the problems Nqthm users faced in applying that system to large-scale proof projects [23]. Those projects included the proof of Gödel's incompleteness theorem [36], the verification of the gate-level description of the FM9001 microprocessor [21], the KIT operating system [1], the CLI stack [2] (which consists of some verified applications written in a high-level language, a verified compiler for that language, a verified assembler/loader targeting the FM9001), and the Berkeley C string library (as compiled by `gcc` for the Motorola MC68020) [9]. For a more complete summary of Nqthm's applications, see [7]. Such projects set the standards against which we measure ACL2.

ACL2 has been successfully applied in projects of commercial interest, including microprocessor modeling, hardware verification, microcode verification, and software verification. Such projects are discussed extensively in the companion book [22]. If you doubt the practicality of computer-aided reasoning, we urge you to read the companion book and judge for yourself.

1.1 Book Layout

This book is divided into four parts.

- Part I is an overview of what is involved in using formalism to model computing systems and to reason about those models with mechanized assistance.
- Part II teaches you the ACL2 formalism: a practical functional programming language closely related to Common Lisp. You will learn how to write "programs" in this language. We put the word in quotation marks because in this setting all the programs are side-effect

[1] Bob Boyer made substantial early contributions.

free: they are functions. Important ideas here are the techniques of list and tree processing and the use of recursion. Of course, the same programming language can also be used to make statements about the values of functions, *e.g.*, "the output of this function is an ordered list" or "when this Boolean function returns true on x and y it returns true on y and x."

- Part III presents a framework in which you can prove statements such as those above. It begins by presenting ACL2 as a mathematical logic. Clear meaning is given to the notion of a *theorem*: a formula that can be derived from the axioms using the primitive rules of inference. But the emphasis here is not on constructing low-level proofs; it is on becoming familiar with relatively large inference steps such as induction, case analysis, and simplification by repeated substitution of equals for equals.

- Part IV explains how to use a mechanization of these large inference steps. The presentation includes a user-level model of how the ACL2 theorem prover "works." But of more importance is the view of the system as a single very large derived rule of inference that can be used to leap from a collection of theorems to a new theorem. We explain "The Method" by which many successful users interact with the system to find proofs of hard theorems.

This book has two appendices. The first covers some basics of using the ACL2 system. The second describes a number of important features of ACL2, most of which are not covered elsewhere in this book.

The Companion Book

The companion book [22] shows how the techniques described in this book can be applied to more complex problems. In many cases, the problems in that book are distillations of industrial applications, *e.g.*, the formalization of a hardware design language, the verification of a floating point multiplier, and the analysis of a compiler. Every case study in that book is accompanied on the Web by the complete ACL2 script of its solution.[2] Furthermore, most of those case studies contain exercises, the solutions to which are posted on the Web. Thus, after mastering the basics described here, you can try your hand at some problems of more realistic scale while still being able to obtain complete solutions.

[2]The solutions for Ruben Gamboa's chapter are for his extension of ACL2 that supports the real numbers.

1.2 The ACL2 System

ACL2 is an efficient functional programming language. ACL2 functions and system models can be compiled, by any compliant Common Lisp compiler. Models can be made to execute efficiently. Executability is not necessarily important in pedagogical projects, but it can be important in practical applications. There, the formal models do double duty as specifications and as simulation engines. The ACL2 system itself is written almost entirely in ACL2. Its size (6 megabytes of source code), reliability, and utility demonstrate the practicality of the language. Other such demonstrations are discussed in the companion book [22], where ACL2 is used to model hardware designs, microprocessors, algorithms, and other artifacts that are not only analyzed symbolically but tested by execution.

The ACL2 system is available for free on the Web (under the terms of the Gnu General Public License). The ACL2 home page is `http://-www.cs.utexas.edu/users/moore/acl2`. There you will find the source code of the system, downloadable images for several platforms, installation instructions, two guided tours, a quick reference card, tutorials, an online User's Manual, useful email addresses (including how to join the mailing list or ask the community for help), scientific papers about applications, and much more. The ACL2 home page also includes links to Web pages for this book and its companion book [22].

The ACL2 online documentation is almost 3 megabytes of hypertext and is available in several formats. The HTML version can be inspected from the ACL2 home page with your browser. Other formats are explained in the "Documentation" section of the installation instructions accessible from the ACL2 home page.

1.3 Important Conventions and Information

In this book, you will often see underlined strings in typewriter font in such phrases as "see defthm." These are references to the online documentation. To pursue them, go to the ACL2 home page, click on "The User's Manual" link, and then click on the "Index of all documented topics." You will see a list from A to Z. Click on the appropriate letter and scan the topics for the one referenced (in this case, defthm) and click on it.

There are exercises throughout the book. Some of these are meant to be done with pencil and paper. Most are meant to be done with the ACL2 system. Solutions to all these exercises are available online. Go to the ACL2 home page, click on the link to this book and follow the directions there.

You will note that on the Web page for this book there is a link named "Errata." As the name suggests, there you will find corrections to the printed version of the book. But more importantly, you may find differences

between the version of ACL2 described here (Version 2.5) and whatever version is current when you go to the home page. The ideas discussed here are fundamental. But we do display such syntactic entities as command names, session logs, etc. These may change. Therefore, look at the online Errata when you first begin to use ACL2 in conjunction with this book.

If, after reading Part I, you want to learn how to use formal methods to address problems of practical interest, we urge you to obtain and install ACL2 on your computer, read the online Errata and the first appendix here, and then read the rest of the book, doing the exercises as you go. There is no better way to learn these techniques than to apply them!

1.4 Intended Audience

We believe it is appropriate to use this book in graduate and upper-division undergraduate courses on Software Engineering or Formal Methods. It could be used in conjunction with other books in courses on Hardware Design, Discrete Mathematics, or Theory (especially courses stressing formalism, rigor, or mechanized support). It is also appropriate for courses on Artificial Intelligence or Automated Reasoning.

If you are teaching from this book you may want exercises whose solutions are not posted. These are provided through the Web page for this book mentioned above.

We assume that you are familiar with computer programming. We also assume you are familiar with traditional mathematical notation: for example, "$f(x, y)$" denotes the application of the function f to (the values denoted by) x and y, and "$|x|$" denotes either the absolute value of x or its cardinality, depending on the context. Finally, we assume you have had some exposure to mathematical proof, especially proofs employing mathematical induction.

Familiarity with Lisp or functional programming would be ideal but is not necessary. If you have had a course in mathematical logic, you may find some of the material a little easier because you know the terminology: Boolean logic, variable symbol, term, formula, instantiation, substitution of equals for equals, well-founded orderings, induction, and so on. But we do not think a course in logic is a necessary prerequisite either to read the book or to use ACL2. The basic skills on which we build in this book are how to program without side-effects, how to use recursion, how to simplify expressions, how to decide whether to use case analysis or induction in an argument, and how to frame a simple inductive argument. We believe most readers have an intuitive grasp of these basic ideas from exposure to programming, informal reasoning about programs, and high-school algebra. The book makes these ideas more precise while elaborating them and providing lots of chances to practice their application.

2

Overview

This book is about formal reasoning, with particular emphasis on reasoning about computing systems. The dream is that we should be able to write down mathematically precise conjectures about special-purpose hardware designs, microprocessors, microcode programs, compilers, algorithms, etc., and then to prove them with a mechanized reasoning system. This dream is realizable. This book is about how to use a tool that is being used today to prove such conjectures. But more generally, the book is about how to formalize systems, how to express their properties, and how to decompose the proofs of those properties into manageable steps.

2.1 Reasoning

The dream of machines that reason has been around a long time. Leibniz expressed it this way.

> If we had some exact language ... or at least a kind of truly philosophic writing, in which the ideas were reduced to a kind of alphabet of human thought, then all that follows rationally from what is given could be found by a kind of calculus, just as arithmetical or geometrical problems are solved. — Leibniz (1646–1716)

By the end of the nineteenth century Boole, Frege, Peano, and others had clearly established the foundations of what we now call *symbolic logic*. Leibniz's dream of a calculus of human thought was apparently coming true. Bits of the mathematical world, *e.g.*, propositional calculus and elementary arithmetic, were described formally by axioms written as formulas in a fixed syntax. Rules were proposed for generating new formulas from old formulas—rules with the property that the new formulas are true if the old formulas are.

In the hands of such giants as Russell, Whitehead, Skolem, Herbrand, and Hilbert it looked as though symbolic logic could place all of mathematics on a firm foundation. Then, in 1931, Gödel showed the limitations of formal logic: given any sufficiently powerful formal logic there are truths that cannot be proved in the logic.

M. Kaufmann et al., *Computer-Aided Reasoning*

Gödel's theorem might be thought of as sealing the fate of Leibniz's dream. But does its inability to deduce all truths really justify the abandonment of formality? Not at all. If formality allows us to deduce some truths about new ideas, it is a useful tool. There is a far more practical obstacle to the construction of a calculus of human thought: the nature of the physical world and its role in what we think about.

Few human artifacts or endeavors, beyond mathematics itself, can be so accurately described by axioms that we would profess to believe any consequence of those axioms no matter how deep. Too many aspects of the physical world are vague, random, or governed by processes best described quantitatively with continuous mathematics like differential equations. Formal logic is good for describing abstract symbolic systems—worlds inhabited by discrete objects manipulated by clearly specified rules determining the behavior of the system. For most of the twentieth century the majority of work in formal logic was the study of formal logic itself.

Much of that changed with the invention of the digital computer. Circuits, processors, programming languages, programs, networks, protocols, formats, ..., all are abstract symbolic systems. Such systems are fruitfully described and studied with symbolic logic.

2.2 Philosophic Writing

What kind of "philosophic writing," to use Leibniz's term, is appropriate for describing computing systems? Our choice is, in essence, a fairly conventional functional programming language. This book is about that language, how to use it to describe computing systems, how to reason in the language, and how to use a certain computer program to help you.

The name of the language, and of the reasoning engine for it, is "ACL2." It stands for "A Computational Logic for Applicative Common Lisp." The ACL2 language is a variant of Common Lisp. But instead of just providing an execution engine for the language, we describe the language as a mathematical logic, with axioms and rules of inference. And we provide a reasoning engine. Thus, in addition to being able to execute your programs, you can use the reasoning engine to prove things about them.

For example, here is a Lisp program to reverse a list.[1] It is not important that you understand this definition now. But we will use it just to talk about Lisp and logic.

```
(defun reverse (x)
  (if (consp x)
      (append (reverse (cdr x)) (list (car x)))
    'nil))
```

[1] ACL2 provides a function named `reverse` but it is not defined the way `reverse` is defined here.

This `defun` command defines a function named `reverse`, with one formal parameter, named `x`. The body of the definition is an expression, which is evaluated to determine the value of the function. In this case, the body is an `if`-expression. The `if` has three arguments, a test, a true branch, and a false branch, each of which is an expression. In this case, the test is `(consp x)`, the true branch is an `append` expression, and the false branch is the constant expression `'nil`. As you can see, Lisp expressions are either variables, constants, or function applications. To apply a function f to the values of some expressions, e_i, we write $(f\ e_1\ \ldots\ e_n)$, instead of the more traditional $f(e_1, \ldots, e_n)$.

The function defined above reverses its argument. More precisely, if applied to a linear list it returns a linear list with the elements in the opposite order. For example, to apply the function to the list `(5 6 7)` we could evaluate `(reverse '(5 6 7))`. `(5 6 7)`, by the way, is a list with three elements. The first is the number `5`, the second is the number `6`, and the third is the number `7`. The little "single quote" mark before it in the `reverse` expression is the way we write the literal constant whose value is the list `(5 6 7)`.

If you calculate the value of `(reverse '(5 6 7))` using the definition of `reverse` above the result is `(7 6 5)`. To compute the reverse of `(5 6 7)`, `reverse` recursively reverses `(6 7)`, obtaining `(7 6)`, and then concatenates it, with the function `append`, to the list `(5)`, producing `(7 6 5)`.

Here is a conjecture about our reverse function.

```
(equal (reverse (reverse x)) x)
```

Informally, this conjecture says that the reverse of the reverse of `x` is `x`. More formally, the conjecture says that, for any value of the variable `x`, `(reverse (reverse x))` and `x` evaluate to `equal` things.

You could test this expression in any Lisp (where the relevant functions are defined) and you would find that the value of the expression is often `t`.

But you can test the expression only on a finite number of cases. Will the expression always evaluate to `t`? One way to try to establish that is to *prove* it, *i.e.*, give a careful argument based on the definition of `reverse` and properties of other functions. We are not prepared to give such an argument here—we have not told you the relevant properties of all the symbols we use. But an attempt to construct a proof will lead you to realize that the conjecture is not always true!

However, a closely related formula can be proved. The related formula says that the reverse of the reverse of `x` is `x`, provided `x` is a list.[2] You might object and say, "Of course we meant that `x` was a list! Why else would we speak of reversing it?" But an aspect of formality is that one must say

[2]Here by "list" we actually mean the concept formalized by the predicate `true-listp`.

what one means and leave no hidden assumptions.[3] One consequence of proving a formula is that we know that its value, under any assignment of objects to its variables, will always be true.

Not only can the new formula be proved, but the ACL2 reasoning engine, or theorem prover, can prove it automatically.

It is good that the theorem prover can prove the "`reverse reverse`" theorem automatically because this theorem is absurdly simple compared to conjectures that are likely to be of interest to you. It is a long way from the definition of `reverse` to the definition of a microprocessor or programming language.

The bad news is that the `reverse reverse` theorem is pretty complicated from the theorem prover's perspective. In fact, this theorem is a perfect example of the mileage we humans get out of informal reasoning. Calling this concept "reverse" loads it with intuitive meaning that is not explicit in the definitional equation. The obviousness of the claim that reversing a list twice produces the original list probably owes more to the name "reverse" and experience with the physical world than it does to the definitional equation of `reverse`.

Suppose we introduced the concept with

```
(defun fn123 (x)
  (fn1 (fn2 x)
       (fn122 (fn123 (fn3 x)) (m8 (fn4 x)))
       'nil)),
```

where `fn1`, `fn2`, `fn3`, etc., are all described by their own equations. How impressive is it to discover or prove that "the `fn123` of the `fn123` of `x` is `x`, when `x` is a `fn98`?"

Whether we give our functions sensible names or not, their properties are derived from the axioms we write down about them, not the names we give them. The same is true of our circuits, components, programs, etc. Just because we call a component a "divider" does not mean it computes the quotient of one number by another! We tend to give our creations sensible names and we use those names to help us reason informally. But because those names mean more to us than to any mechanized reasoning engine, it is often necessary to tell the engine "obvious" properties. Sometimes when the engine fails to prove these "obvious" properties, we come to realize they are actually false!

2.3 System Models

To prove something it must first be expressed as a statement in the formal syntax. This means you cannot prove that a circuit, a microprocessor, or

[3] Syntactic typing sometimes makes it more convenient to express such assumptions. But ACL2 does not have syntactic typing.

any other physical artifact is correct. You can only prove that some model of it has a certain property. It is up to you to use the logic to create a model of the artifact in question and then to write down conjectures that express your understanding of correctness—conjectures that you believe, or at least hope, to be true. One of the main thrusts of this book is to teach you how to use the ACL2 logic to model digital artifacts. We sketch several models in this chapter.

Once you have defined a model of the artifact, you must express, in the formal syntax, the conjecture you wish to prove. Even this can be hard. Theorems proved informally are often never expressed as formulas! For example, "theorems" that otherwise might look like formulas often contain ellipses (...). Ellipses are usually an IQ test: can you guess what the author has in mind? Such informal notation must be made precise, which often requires defining additional concepts. Another common pitfall in converting from an informal statement to a formula you can prove is identifying all the hidden or implicit assumptions.

Sometimes the informal statement uses familiar mathematical concepts that are not yet defined in the logic you are using. The public distribution of ACL2 includes many online "books" that contain definitions and theorems about integer and rational arithmetic, finite sets, tables, and vectors. But we have only scratched the surface in formalizing the many operations and relations on these concepts. Often you will find yourself defining familiar concepts just to state your goal.

Sometimes you may be unable to define the concepts of interest. The ACL2 logic is in some ways quite restrictive, lacking for example infinite sets and higher order functions. However, more often than you might think—especially if you have heretofore used higher order logic or set theory—this problem can be overcome without undue violence to the intuitions you are trying to capture. Perhaps some formalizable concept can be substituted without weakening the meaning of the statement. But if not, you may need to choose a different mechanically supported formalism such as HOL [16] or PVS [34, 13].

The activity of casting your problem into the formal syntax of some logic is called *formalization*. One of the things you will learn in this book is how to use ACL2 to model systems and formalize their properties.

Consider the following ACL2 formula.

```
(implies (and (floating-point-numberp p 15 64)
              (floating-point-numberp d 15 64)
              (not (equal d 0))
              (rounding-modep mode))
         (equal (divide p d mode)
                (round (/ p d) mode)))
```

We do not give the definitions of the concepts here, but, intuitively, the formula may be read as follows. If `p` and `d` are floating point numbers

with 15 bits of exponent and 64 bits of significand, `d` is not 0, and `mode` is a rounding mode, then the output of the function `divide` on `p`, `d`, and `mode` is equal to the infinitely precise quotient of `p` divided by `d`, rounded according to `mode`.

The notions of `implies`, `and`, `not`, and `equal` are built into ACL2 and are fairly standard formalizations of implication, conjunction, logical negation, and equality. The function symbol '`/`' denotes the arithmetic operation of division of one rational number by another. It too is built into ACL2 and is axiomatized to satisfy the familiar properties of division in the rational field.

The other function symbols used in this formula were defined expressly for the purpose of stating this theorem. The question of whether they accurately capture the alleged intuitive notions is of great interest and can be difficult to settle. A careful reading of the IEEE floating point standard and its comparison with the definitions of `floating-point-numberp`, `rounding-modep`, and `round` gives some confidence that these functions capture the intended concepts.

The definition of the function `divide` was constructed, by hand, by studying the microcode implementing division on the AMD-K5 microprocessor of Advanced Micro Devices, Inc. (AMD[4]). The ACL2 function `divide` computes its final answer using a certain sequence of fixed format floating point additions and multiplications (and a few other operations). By talking to the designers of the microcode and of the AMD-K5 floating-point unit, we convinced ourselves and them that the definition of `divide` accurately captured the algorithm used by their microcode [33].

Using ACL2 we proved that the formula above, with all the definitions mentioned, is a theorem.

We therefore think of the theorem as establishing the correctness of the floating point division algorithm on the AMD-K5 microprocessor. But, of course, that characterization of the theorem is an informal one.

It is possible to state a theorem about the microcode itself—a certain fixed sequence of numbers representing instructions interpreted by the microcode engine. To state that theorem in ACL2 we would have to model the microcode engine as an ACL2 function that interprets microcode instructions and changes some state components accordingly. Such "microcode engine models" have been defined in ACL2 for other microprocessors, *e.g.*, the Motorola CAP digital signal processor [10], and "code proofs" have been done with ACL2. But that approach was not taken in the AMD-K5 division work, even though it arguably would have provided a more convincing model to study.

The reason is instructive. At the time the theorem above was formulated and proved, AMD was more concerned about the division algorithm than its expression in microcode. Furthermore, time was short because the AMD-

[4]AMD, the AMD logo and combinations thereof, AMD-K5, AMD-K7, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

K5 fabrication date was looming. The moral is: economic realities have a significant impact on formal modeling. Models must address the issues of concern and allow the analysis to be carried out in a timely fashion.

At the time of this writing, the AMD-K5 is two generations old. AMD has since released the AMD-K7^{TM} (or AMD Athlon^{TM}) microprocessor. What of ACL2's role in its development? ACL2 was used by David Russinoff to verify the AMD Athlon processor implementations of floating-point addition, subtraction, multiplication, division, and square root [35]. The implementations were done in hardware, not microcode, so no microcode engine was formalized. But the AMD hardware was described in a Verilog-like hardware description language referred to simply as "the RTL language." In the ACL2 work on the AMD Athlon processor, the RTL source was mechanically translated into ACL2 (using a translator written by Art Flatau). The correctness of the translation was not proved, because no formal semantics for the RTL was available. But the translation was tested: over 80 million floating-point test vectors were run through the ACL2 models and the results were compared to those computed by AMD's RTL simulator. The answers were identical. This highlights the value of ACL2's executability. It also highlights the inadequacy of testing: when correctness proofs were undertaken, bugs were found in the ACL2 models (and, more importantly, in the original RTL). These bugs were not exposed by the test suite. The bugs were fixed by changing the RTL, new models were mechanically produced, and those models were mechanically verified using ACL2. Russinoff and Flatau discuss their work on this in the companion book [22].

The point of this discussion is to drive home the fact that models of artifacts of commercial interest are not constructed in an ivory tower. They are constructed in the midst of a rapidly evolving design and implementation project. Such formal models may look *ad hoc* or incomplete. Instead of iterating on a given project and producing a *tour de force* model that answers every objection, a new project is undertaken where the issues are often completely different.

2.4 Truth and Proofs

Why prove formulas? The reason is that proof is a way to establish truth. A *theorem* is either an axiom or a formula produced by applying some rule of inference to other theorems. A *proof* is a finite structure showing the derivation of a theorem from the axioms.

But (most) ACL2 formulas can be executed. If you select concrete values for the variables and then evaluate a formula, some concrete value will be computed. (We here imagine an unbounded amount of memory.) Theorems

always evaluate to true.[5] The basic argument for this fundamental property of theorems is that (a) the axioms always evaluate to true and (b) the rules of inference preserve this property. Since a theorem is derived by applying a finite number of rules of inference to axioms, theorems must always evaluate to true.

But there are an infinite number of possible assignments to the variables of any formula containing a variable. So something wonderful has happened: We can determine that a formula will evaluate to true on the infinite number of possible cases by constructing a finite proof of it! That is why we prove formulas.

To use formal proofs to check your reasoning you must somehow translate your informal argument into a formal one. Informal proofs often involve important labor-saving metatheorems or derived rules of inference. Typical phrases suggesting the use of such rules are "without loss of generality we restrict our attention to," "by symmetry it suffices to prove," and "the other cases are analogous."

Suppose then that you have an informal proof of some formal statement. Imagine that you begin to present the proof to a colleague. Some of the steps in your proof are sufficiently small that your colleague follows them. But other steps are challenging and he or she interrupts and asks how you got from one formula to the next. When this happens you expand your explanation, perhaps discovering details that you had previously overlooked. A formal proof would be created if your colleague simply asked "how?" on every step requiring more than the most basic logical transformation.

2.5 The Reasoning Engine

The ACL2 theorem prover was designed for the kind of interaction sketched above. The theorem prover attempts to fill in every gap in your alleged proof of a formula. When successful, the computation done by the theorem prover is intended to guarantee that a formal proof of the formula exists, although you never actually see this formal proof. (Nevertheless, we often refer to ACL2's computation as its "proof" or "proof attempt.") When a gap is too big for the theorem prover to follow, you must break that step into smaller pieces.

As depicted in Figure 2.1, the theorem prover takes input from both you and a data base, called the *logical world* or simply **world**[6]. One view of the world is that it contains axioms, definitions, and previously proved theorems. But a more effective view of the world is that it embodies a theorem proving strategy, developed by you and codified into *rules* that direct certain aspects of the theorem prover's behavior. When trying to

[5] To be more precise, instances of theorems evaluate to non-nil.

[6] Recall our convention of underlining topics discussed in ACL2's online documentation. See page 4.

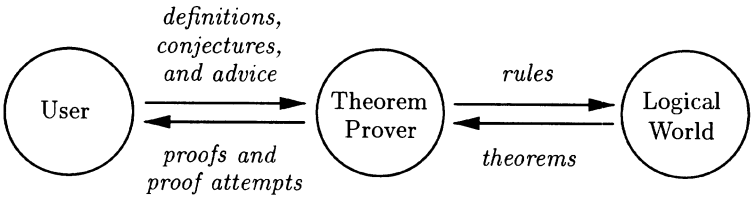

Figure 2.1: Data Flow in the Theorem Prover

prove a theorem, the theorem prover applies that strategy and prints its proof attempt. You can interrupt and abort or let the attempt continue. The theorem prover may eventually stop, in which case it will have either succeeded in proving the formula or failed to prove the formula. When it succeeds, your theorem is converted into a rule, using advice from you, and this new rule changes the theorem prover's future strategy. When it fails, you must either change the alleged theorem or give additional guidance to the theorem prover, often by inspecting its proof attempt to see what went wrong.

Note that the rules that control the theorem prover's behavior come from the logical world.

- Except by adding axioms, *you cannot directly add rules to the logical world.*
- *It is important to think about how your theorems will be interpreted as rules, if you want them used automatically.*

This design may at first frustrate you: you will feel powerless. But it relieves you of the burden of getting the rules logically correct! The theorem prover takes that responsibility.

With some experience you will feel empowered by this design. Why? Because you will see the theorem prover apply your strategy to prove theorems other than the ones for which you first designed it. However, you will also feel challenged to codify good strategies.

2.6 Effort Involved

How hard is it to use ACL2 to prove theorems of commercial interest? Modeling "the product" and capturing one or more important properties characterizing its "correctness" are the first and often hardest steps. These steps usually involve talking to the implementors, reading design documents, and experimenting with "toy models." You will often find that the implementors approach the project very differently than you do. They may

not have in mind a correctness criterion that can be expressed as a property you can formalize. Their notion of correctness is often simply the absence of "bugs." Bugs, like good art, can evidently be recognized when seen. But remember: the implementors are probably very good at their jobs, and are grappling with other problems, like power consumption, timing, production costs, time-to-market, etc. Furthermore, it is not by dumb luck that they have built things that you buy and use and rely on. We discuss the "sociology" of typical projects in Part I of [22].

As you get more familiar with the product you will find yourself thinking of it more abstractly. General properties will occur to you. In conversation with the team members you will learn whether these are intended to hold—or, more likely, under what conditions and at what level of abstraction they are intended to hold. Identifying and peeling apart co-mingled layers of abstraction may well become a big part of your job. Just being able to provide labels for the abstractions and a vocabulary to discuss the concepts unambiguously is of great benefit to the team.

Your experimentation with "toy models"—models that intentionally omit many features—is very important. The idea is to have a model that is so small you can try one modeling approach today, abandon it as impractical on your way home, and have a new approach up and running the next day. Such models allow you to develop the formalization of the basic ideas and properties, establish clear communication with the implementors, experiment with proof methods, and develop strategies for elaborating the models to interesting levels of complexity while only incrementally affecting proof complexity. See Moore's case study in [22]. Indeed, there are several more elaborate "toys" among the case studies in [22] (*e.g.*, the pipelined machine of Sawada, the floating point multiplier of Russinoff and Flatau, and the microprocessor of Greve, Wilding, and Hardin). These "toys" present the basic problems seen in applications of much more complexity.

In successful projects, you will eventually form a productive working relationship with the implementors, one based on mutual respect for the talents of "both sides." It is likely that you will produce a sequence of ever more elaborate models and the conjectured properties of these models and relations between them will become the goal theorems. Your models will evolve with the product design and theorems proved about one model will have to be "proved again" after modifying the model. "Proof maintenance" is an important aspect of ACL2's design. The ACL2 data base contains *strategies* for finding proofs, not individual proofs. Often, your old strategies will succeed on closely related new theorems. ACL2's ability to execute your evolving models, *i.e.*, use them as simulators, is also valuable. We describe some typical models and properties in Part I of [22].

You might start the project thinking that you are going to prove a particular theorem and be done. More likely, the "proof phase" lasts as long as the project and provides a continuing clarification and confirmation of some properties. The immediate impact is a better product. The legacy,

however, is a way of thinking, together with a set of concepts made precise and embodied in a collection of ACL2 books that can be used on the next project.

2.7 Requirements on the User

To use ACL2 to model a system and prove properties of the model, you must understand the system being modeled. In addition, you must understand

- how to use the ACL2 logic to formalize informally described concepts,
- how to do pencil-and-paper proofs of ACL2 formulas, and
- how to drive the ACL2 theorem prover.

We discuss these issues in this book. The book contains many exercises to help you develop these skills.

ACL2 will help construct the proof but its primary role is to prevent logical mistakes. The creative burden—the mathematical insight into why the model has the desired property—is your responsibility.

The "typical" ACL2 user

- has a bachelor's degree in computer science or mathematics,
- has some experience with formal methods,
- has had some exposure to Lisp programming and is comfortable with Lisp notation (reading some part of this book should suffice),
- is familiar with and has unlimited access to a Common Lisp host processor, operating system, and text editor (we use Gnu Emacs on Sun workstations and PCs running Unix and Linux),
- is willing to do some of the exercises in this book, and
- is willing to consult the ACL2 online documentation.

How long does it take for a novice to become an effective ACL2 user? Let us first answer a different question. How long does it take for a novice to become an effective C programmer? (Substitute for "C" your favorite programming language.) It takes weeks or months to learn the language but months or years to become a good programmer. The long learning curve is not due to the complexity of the programming language but to the complexity of the whole enterprise of programming. Shallow issues, like syntax and basic data structures, are easy to learn and allow you to write useful programs. Deep skills—like system decomposition, proper design of the interfaces between modules, and recognizing when to let efficiency impact clarity or vice-versa—take much longer to master. Once deep skills are

learned, they carry over almost intact to other languages and other projects. Learning to be a good programmer need not require using a computer to run your programs. The deep skills can be learned from disciplined reflection and analysis. But writing your programs in an implemented language and running them is rewarding, it often highlights details or even methodological errors that might not have been noticed otherwise, and, mainly, it gives you the opportunity to practice.

We hope that you find the above comments about programming noncontroversial because analogous comments can be made about learning to use ACL2 (or any other mechanized proof system).

How long does it take for a novice to become an effective ACL2 user? It takes weeks or months to learn to use the language and theorem prover, but months or years to become really good at it. The long learning curve is not due to the complexity of ACL2—the logic or the system—but to the complexity of the whole enterprise of formal mathematical proof. Shallow issues, like syntax and how to give hints to the theorem prover, are easy to learn and allow you carry out interesting proof projects. But deep skills—like the decomposition of a problem into lemmas, how to define concepts to make proofs easier, and when to strive for generality and when not to—take much longer to master. These skills, once learned, carry over to other proof systems and other projects. You can learn these deep skills without doing mechanical proofs at all—indeed, you may feel that you have learned these skills from your mathematical training. Your appraisal of your skills may be correct. But writing your theorems in a truly formal language and checking your proofs mechanically is rewarding, it often points out details and even methodological errors that you might not have noticed otherwise, and, mainly, it gives you the opportunity to practice.

Part II

Programming

3

The Language

To reason formally you must know how to express your ideas in the formalism of the system you are using. In this book, we use the ACL2 system, which is a simple dialect of the Lisp programming language. Thus, it is important that you learn how to express yourself in this Lisp. We teach you that in this part of the book by presenting ACL2 as a programming language. A prerequisite for understanding this chapter is that you have some programming background, but you need not know Lisp. If you do know Lisp, we recommend that you skim this chapter quickly, confirming that the examples make sense and noting our terminology.

Most programmers who have come across the Lisp programming language probably remember it most for its peculiar parenthesis-laden syntax. To sum the values of the two variables `x` and `y`, the Lisp programmer writes `(+ x y)`, while virtually everybody else in the world writes `x+y`. Where the C programmer might write `8*f(x,a+3)`, and the mathematician might write $8f(x, a + 3)$, the Lisp programmer would write `(* 8 (f x (+ a 3)))`. It is easy to get from traditional notation to Lisp: eliminate infix (by introducing function applications), move the parentheses in front of the function symbols, and eliminate the commas.

Despite this verbose syntax, Lisp is one of the oldest programming languages still in everyday use. This is not because there is a lot of "legacy code" maintained in Lisp. Most Lisp code running today was probably written in the past few years and was written "from scratch." Lisp is still the language of choice for many applications in artificial intelligence, for which it was first designed. The reason is that it is extraordinarily easy to build systems in Lisp, to get them to run, and to experiment with modifications to the code. In short, Lisp is probably the best rapid prototyping language in the world today and has been since it was invented by John McCarthy in 1959 [28]. Many important features of other programming languages, *e.g.*, optimizing compilers, iterative forms, flexible typing systems, functional programming, declarative programming, and object oriented programming, were invented or first extensively explored in Lisp-based systems.

Part of the reason for Lisp's success in this arena is its peculiar syntax. Data structures in Lisp are written in the same parenthesis-laden notation mentioned above. Put a quote mark in front of a Lisp program and you have a Lisp data object that you can manipulate easily with another Lisp

M. Kaufmann et al., *Computer-Aided Reasoning*

program. This makes it convenient to extend the language. Furthermore, because Lisp provides a garbage collector, the programmer thinks in terms of objects rather than pointers or addresses.

In the first three decades after Lisp was invented, many different varieties were developed, including MacLisp [30], Interlisp [41], and Zetalisp [31]. In 1984, Guy Steele, in cooperation with the community of Lisp users and developers, described Common Lisp [39], a variant of MacLisp which standardized features found useful in various Lisp dialects. The second edition of the Common Lisp reference manual, [40], is now the *de facto* standard for Lisp.

The ACL2 programming language is in fact an extension of a non-trivial subset of Common Lisp. The subset contains none of the Common Lisp features that involve side effects, *e.g.*, global variables and destructive modification of data. That is, ACL2 focuses on the *functional* or *applicative* or *side-effect free* subset of Common Lisp.

In this book we use the word "function" the way mathematicians do: a function is a map from a domain to a range. If a function, f, is defined on some input, n, then its value is written $f(n)$ or, in Lisp, $(f\ n)$. If f is defined on n, then every time f is applied to n it returns the same result. For example, applying the factorial function to 5 always produces the same result, namely 120.

This need not be true of "functions" in programming languages that permit side-effects. In Java, for example, `factorial(5)` could be defined to be 120, except when it is evaluated for the 25^{th} time, upon which occasion it returns the local time of day. In such a setting the Java expression "`factorial(5) == factorial(5)`" could actually evaluate to false (because one of the calls happens to be the 25^{th} one and the local time then is not 120).

ACL2 "functions" are functions and that explains the sense in which ACL2 focuses on a subset of Common Lisp: we deal only with Common Lisp *functions*. We throw out, for example, the Common Lisp programs for destructively manipulating arrays, property lists, and files. To make ACL2 a useful programming language we must provide some efficient functional analogues of these facilities. Thus, we add to our subset of Common Lisp some ACL2 functions for manipulating arrays, property lists, and files. These ACL2 functions are truly functions. For example, ACL2 functions that write files take an explicit "state object" as an argument and instead of modifying their arguments they return a new explicit state.[1]

Some Common Lisp functions are only partially defined. For example, the Common Lisp reference manual [40] does not specify the behavior of the addition function, `+`, when it is applied to non-numeric arguments. But ACL2 functions are total: they are defined on all inputs. When an

[1] Under the hood there is truly a file being written, and no "state" object is actually constructed. However, ACL2 maintains an applicative semantics. See the online documentation for `state` and `stobj`.

ACL2 function is applied to arguments that are outside the domain of the corresponding Common Lisp function, some particular value is computed, usually by "defaulting" the "bad" arguments to "good" ones. For example, addition of non-numeric constants such as `(+ t nil)` is undefined in Common Lisp (and signals an error in most implementations), but in ACL2 `(+ t nil)` is defined to be `0` and, indeed, returns `0` if evaluated in ACL2.[2] ACL2 and Common Lisp agree where Common Lisp is defined. The guard[3] feature of ACL2 allows us to characterize the intended domain of a function and to prove theorems that show that a function or expression is compliant with Common Lisp. When an ACL2 function is shown to be compliant with Common Lisp, you know that it is "well-typed." In addition, ACL2 can often execute such functions faster by relying on the underlying Common Lisp engine. Thus, guards can be important to the successful modeling of complex systems and we discuss them briefly here (page 51). But in this chapter we use ACL2 as our execution engine and largely ignore the fact that some Common Lisp functions are partial.

In this chapter we introduce the functional subset of Common Lisp supported by ACL2. The Common Lisp reference manual, [40], is an excellent source book for the full language and is available online at the following URL: `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/clm/clm.html`. The online documentation for ACL2 contains additional material, both informal and formal. Please refer to these sources of information for the details omitted below.

We do not try to teach you how to use Lisp efficiently. We do not try to explain the ACL2 "state object" or how to use ACL2's functional facilities for arrays, property lists, and file input/output. These topics are explained in the ACL2 documentation. For the present purposes we are interested only in teaching you how to define simple recursive functions on lists and numbers. To use ACL2 to model computational systems, to write specifications, or to state conjectures you must master recursive definition. Furthermore, you will find that even in sophisticated applications, such simple definitions form the bulk of your ACL2 programming.

If you find it helpful to try things out as you read, we recommend that you read Appendix A before reading this chapter.

3.1 Basic Data Types

ACL2 supports five disjoint kinds of data objects, each of which is illustrated below.[4]

[2] See `set-guard-checking`.

[3] Recall the convention that words underlined in typewriter font are topics in the ACL2 online documentation.

[4] The real numbers have been added to a modification of ACL2. See the contributions by Gamboa and Kaufmann in the companion volume, [22].

- numbers: `0`, `-123`, `22/7`, `#c(2 3)`
- characters: `#\A`, `#\a`, `#\$`, `#\Space`
- strings: `"This is a string."`
- symbols: `nil`, `x-pos`, `smith::x-pos`
- conses: `(1 . 2)`, `(a b c)`, `((a . 1) (b . 2))`

Here is an informal description of the logical construction of the numbers. The non-negative integers (*i.e.*, the naturals) are built from 0 by the successor function. The negative integers are built from the naturals by the negation function. The rationals are built from the integers by division. The complex numbers are built from pairs of rationals.

Similarly, a character can be constructed from a natural number (less than 256). A string can be constructed from a finite sequence of characters. A symbol can be constructed from two strings, one naming the package of the symbol and the other naming the symbol within that package. Finally, conses are ordered pairs of objects.

Despite this constructive hierarchy, ACL2 encourages the view that the numbers, characters, strings, and symbols are "atomic" objects (by referring to objects of those types as "atoms"). With the exception of arithmetic on numbers, most ACL2 programs tend to pass atoms around and compare them, without delving into their structure. For that reason we give short shrift to the structure of atoms in this introduction.

Every ACL2 object can be written down and used as data in ACL2 programs.

3.1.1 About Atoms

Numbers are generally written in base 10. No decimal point is ever written. Rationals are either integers or are written in the form of fractions. Here are some examples of rationals: `0`, `-77`, `123`, `1/3`, `22/7`.

There are many different ways to write any given number. For example, `123` may also be written `0123`, `00123`, `+123`, and `456/2`. Fractions are reduced internally to lowest terms.

The complex number 3+5i is written `#c(3 5)`. The real and imaginary parts of a complex number are always rationals. `#c(3 0)` is 3.

Binary, octal, and hexadecimal notation are supported. `123` may also be written `#b1111011`, `#o173`, and `#x7b`. The negative rational `-2/5` could be written `#b-10/101`. Alphabetic characters in the notation may be capitalized.

Character objects are ACL2 objects representing the 256 ASCII characters. The object representing an upper case 'A' is written `#\A`. Its lower

case counterpart is `#\a`. Signs, punctuation, and digit characters are written analogously. "Whitespace" characters have names that are spelled out: `#\Space`, `#\Tab`, and `#\Newline`.

Strings are finite sequences of characters, *e.g.*, `"Hi there!"`, and are written by enclosing the desired sequence in "double quotation marks". To include a double quotation mark in a string, precede it with a backslash.

Symbols are objects like strings but with a richer structure. Symbols are composed of two parts: a package name and a symbol name. Both are strings. The symbol with package name `"SMITH"` and symbol name `"ABC"` is written `SMITH::ABC`. Note that unlike strings, symbols have no delimiters (quote marks or otherwise).

We say the symbol `SMITH::ABC` is "in" the package named `"SMITH"`, or more succinctly, in the package `SMITH`. Packages that are not built-in must be declared in advance with `defpkg`.

Because of the absence of delimiters, the strings naming packages and symbols must satisfy certain syntactic requirements to allow unambiguous parsing. Let us say a string is "simple" if it begins with an upper case alphabetic character, contains only upper case alphabetic characters, digits, and signs other than the backslash, vertical bar, and hashmark (#), and contains no punctuation (colon, comma, dot, quote marks, or parentheses) or whitespace. A "simple symbol" is one in which both the package and symbol names are simple strings and the package name is not `"KEYWORD"`. When parsing a simple symbol, the double colon ends the package name (and is not part of it) and the first non-simple character ends the symbol name (and is not part of it). Lower case characters encountered while parsing a simple symbol are automatically converted to upper case. Thus `SMITH::ABC`, `Smith::Abc`, and `smith::abc` are three ways to write the same simple symbol. We tend to write symbols in lower case except when they are the first word of a sentence, where we capitalize them.

Symbols are parsed with respect to a user-selected "current package." Suppose the current package is named `"SMITH"`. Then symbol `smith::abc` could be written simply `abc`. When a package name is omitted, the current package is used by default. Unless otherwise noted, the current package in this book is named `"ACL2"`. Thus, when we refer to the symbol `car`, say, we mean `ACL2::CAR`.

Two commonly used symbols are `t` and `nil`. These are the "Boolean symbols" and are used to denote true and false in ACL2. However, all ACL2 primitive conditional and propositional operators test against `nil`. Any object other than `nil` may serve as an indicator of "truth." When we say some test "is true" we mean that the value returned is not `nil`. The symbol `t` is just a convenient choice.

Symbols with package name `"KEYWORD"` are called *keywords*. Keywords are not simple symbols. Keywords may be written by preceding the symbol name by a single colon. Thus, while `abc` is a simple symbol in package `ACL2`, `:abc` is a keyword in package `KEYWORD` and could be written `KEYWORD::ABC`.

It is possible to write other non-simple symbols. For example, the symbol `|Hi pal!|`, which is in the current package, has a name starting with an upper case "H" that includes whitespace and lower case letters. It is also possible to import symbols from one package into another so that, for example, `smith::car` is the same symbol as `acl2::car`. See defpkg.

Exercise 3.1 *Which of the utterances below denote ACL2 atoms? For those that do denote atoms, indicate whether the atom is an integer, rational, complex number, character, string, or symbol.*

1. `0.33`
2. `#b-1101`
3. `+123`
4. `#b+001/011`
5. 1101_2
6. `12E-7`
7. `#\A`
8. `#\Umlatt`
9. `#\Space`
10. `#\`σ
11. `"Error 33"`
12. `"No such name: "Smithville""`
13. `ab`
14. `:question`
15. `1,y`
16. `nil`
17. `ACL2::SETQ`
18. `ACL2::FOO`

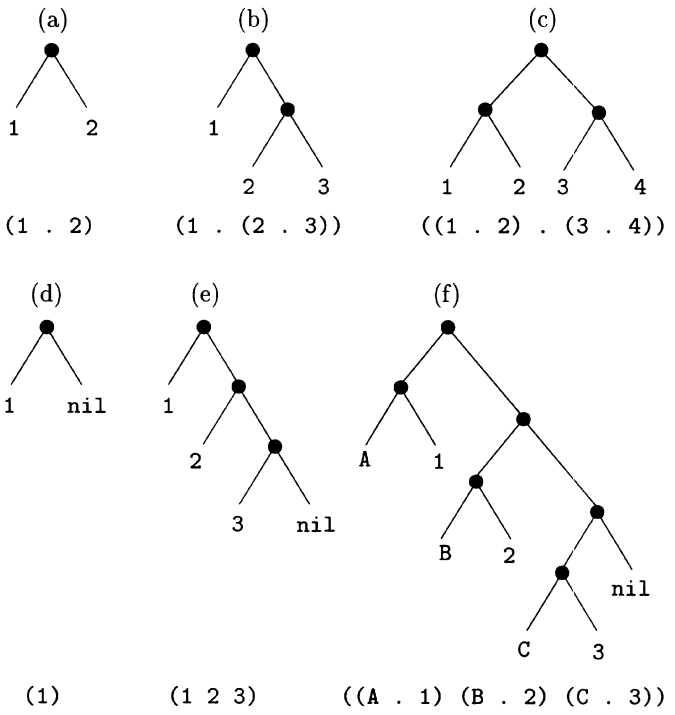

Figure 3.1: Some binary trees and linear lists

3.1.2 About Conses

Conses are ordered pairs of objects. They are by far the most commonly used objects in ACL2. Conses are sometimes called "lists," "cons pairs," "dotted pairs," or "binary trees."

There are many ways to write a given list. This should not be surprising. Consider the fact that `123`, `000123`, `+0123`, `246/2`, and `#b1111011` are all ways to write down the same numeric constant and stylistic considerations determine which form you use. So too with list constants.

Any two objects may be put together into a `cons` pair. The cons pair containing the integer 1 and the integer 2 might be drawn as shown in Figure 3.1(a) or might be written in Cartesian coordinate notation as ⟨1,2⟩. But in ACL2 it is written as `(1 . 2)`. This is a single cons object in ACL2 with two integer objects as constituents. The left-hand constituent is called the `car` of the pair. The right-hand constituent is called the `cdr`.[5]

[5]These names are venerable historical artifacts. They stand for the "contents of the

The tree shown in Figure 3.1(b), which in Cartesian notation is ⟨1,⟨2,3⟩⟩, may be written in ACL2 as `(1 . (2 . 3))`. Similarly, the tree in Figure 3.1(c), ⟨⟨1,2⟩,⟨3,4⟩⟩, may be written in ACL2 as `((1 . 2) . (3 . 4))`. The car of tree (c) is tree (a), *i.e.*, `(1 . 2)`, and the cdr of tree (c) is `(3 . 4)`.

The notation we are using to write list constants is called *dot notation.* It is a straightforward translation of the familiar Cartesian coordinate notation in which parentheses replace the brackets and a dot (which must be surrounded by whitespace) replaces the comma.

ACL2 has two syntactic rules that allow us to write cons pairs in a variety of ways. The first rule provides a special way to write trees like that shown in Figure 3.1(d), *i.e.*, the tree `(1 . nil)`. This tree may be written as `(1)`. That is, if a cons pair has the symbol `nil` as its cdr, you can drop the "dot" and the `nil` when you write it.

The second rule provides a special way to write a cons pair that contains another cons pair in the cdr: you may drop the dot and the balanced pair of parentheses following it. Thus $(x\ .\ (\ldots))$ may be written as $(x\ \ldots)$. For example, the tree shown in Figure 3.1(e) may be alternatively written in any of the following ways.

```
(1 . (2 . (3 . nil)))
(1 . (2 . (3)))
(1 . (2 3))
(1 2 3)
```

It may also be written in many other ways, *e.g.*, `(1 2 . (3 . nil))`.

Binary trees that terminate in `nil` on the right-most branch, such as the trees in Figure 3.1(d-f), are often called *linear lists* or *true lists*. By extension, the symbol `nil` is also called a (true) list or, in particular, the *empty list*, and is sometimes written `()`. Note that the empty list is a symbol, not a cons.

The *elements* of a list are the successive cars along the "cdr chain." That is, the elements are the car, the car of the cdr, the car of the cdr of the cdr, etc. The elements of the list in Figure 3.1(e) are `1`, `2`, and `3`, in that order. There are no elements in the empty list.

Combinations of car and cdr are so common that a naming scheme exists. For example, the *cadr* of a list is the c*a*r of the c*d*r and the *caddr* is the c*a*r of the c*d*r of the c*d*r, etc.

If we let x denote the tree `(1 2 3)`, then the car of x is `1`, the cdr of x is `(2 3)`, the cadr of x is `2`, the caddr of x is `3`, and the cdddr of x is `nil`.

Of course, the elements of a list may be conses. Consider the tree in Figure 3.1(f). This list may be written `((A . 1) (B . 2) (C . 3))`. Its car is the cons pair `(A . 1)`, a pair whose car is the symbol `A`.

address register" and the "contents of the decrement register," and have to do with the original implementation of Lisp on the IBM 704.

Lists such as the one in Figure 3.1(f) are so common they have a name. They are called *association lists* or *alists* and are used as keyed tables. The list above associates the key A with the value 1, the key B with the value 2, etc. Later we show functions for manipulating alists. Here is another alist.

```
((NAME . ((FIRST . "John")
          (MIDDLE-INITIAL . #\L)
          (LAST . "Smith")))
 (AGE . 35)
 (HEIGHT-IN-METERS . 24/13)
 (ADDRESS . ((STREET . "1404 Elm Street")
             (CITY . "San Lajitas")
             (STATE . TX)
             (ZIP . 79834))))
```

Observe that some of the "values" are themselves alists. Whitespace may be inserted into the display of a list to improve appearance and readability, as long as it does not destroy the parsing of the atoms. When the elements of a linear list cannot be displayed on one line, many programmers "stack" the elements one under the other. This is called "pretty printing" and involves questions of aesthetics.

By convention in Common Lisp, the notions of car and cdr are defined also on the atom `nil`: the car and cdr of `nil` is `nil`. Beware: this does not mean that `nil` is produced by consing `nil` onto `nil`!

Exercise 3.2 *For each list below, how many elements are in the list? How many distinct elements?*

1. `(A 6 A-6 a "A" #b110 #\A)`
2. `(NIL () (NIL) (()))`
3. `((A B) (A b) (A . b) (A B . NIL))`

Exercise 3.3 *What is the length of the longest branch in each of the following binary trees? Also, for each tree, write down the list of atoms in the leaves, from leftmost to rightmost. Does either tree contain the same atom twice as a leaf? It sometimes helps to draw the tree.*

1. `((a b . c) . d)`
2. `((a b c) 1 2 (3 . 4) 5)`

3.2 Expressions

ACL2 programs are composed of *expressions*, also called *terms*. While many programming languages have expressions, statements, blocks, procedures,

modules, etc., ACL2 just has expressions. For convenience we first introduce the notion of a simple expression and later introduce "special forms" and "macros" that are just abbreviations for simple expressions.

3.2.1 Simple Expressions

A *simple expression* is

- a variable symbol,
- a constant symbol,
- a constant expression, or
- the application of a function expression, f, of n arguments, to n simple expressions, $a_1, \ldots, a_n$, written $(f\ a_1\ \ldots\ a_n)$.

Comments may be written where whitespace is allowed. A comment begins with a semi-colon and ends with the end of the line.

Here is an example expression.

```
(if (equal date '(august 14 1989))   ; Comments are written
    "Happy birthday, ACL2!"          ; like this.
  nil)
```

This expression contains the variable symbol `date`, the constant symbol `nil`, two constant expressions (a list and a string), and two function applications (of the function symbols `if` and `equal`).

To make the definition of a simple expression precise we must define the variable and constant symbols, the constant expressions, and the function expressions. Technically, these concepts depend upon the history of the session. For example, whether something is an expression or not depends on which functions have been defined. We leave the history implicit in our discussion here.

What does an expression mean? Given an assignment of ACL2 objects to its variable symbols, an expression can be *evaluated*. The value is always an ACL2 object. In this discussion, we let the assignment be implicit. The value of an expression is given recursively in terms of the values of the subexpressions. We discuss this as we define the syntax more precisely.

A *variable symbol* is a symbol other than a constant symbol (defined below).[6] For example, `x`, `entry`, and `address-alist` are variable symbols. Variable symbols are given values by the (implicit) assignment.

A *constant symbol* is `t`, `nil`, a keyword, or a symbol declared with `defconst`. The value of `t` is the ACL2 object `t`; the value of `nil` is `nil`; the value of a keyword is itself. Symbols declared with `defconst` have

[6] We also disallow certain variable symbols in the LISP package, but we will feel free to avoid similarly obscure restrictions in the remainder of this discussion.

names that start and end with asterisks. The value of such a symbol is some ACL2 object specified by the `defconst` declaration. For example, `*standard-chars*` is a constant symbol in ACL2. Its value is a certain list of characters.

A *constant expression* is a number, a character, a string, or a single quote mark (') followed by a ACL2 object. In the last case, we call the constant expression a *quoted constant*. The value of a number, character, or string is itself. The value of a quoted constant is the ACL2 object quoted. Thus, `123` and `"Error"` are constant expressions with themselves as their values. `'Monday` is a quoted constant whose value is the symbol `Monday`. `'(August 14 1989)` is a quoted constant whose value is a list of three elements, the first of which is the symbol `august`.

Note that `'123` and `'"Error"` are quoted constants and have `123` and `"Error"`, respectively, as their values. More generally, when writing numeric, character, and string data in expressions the quote mark is optional. We generally do not quote numbers, characters, or strings since they evaluate to themselves anyway. But when symbols (other than `t` and `nil`) are used as data in an expression they must be quoted because otherwise they might be confused with variables. That is, the expression `x` is a variable whose value is specified by the context. The expression `'x` is a constant expression whose value is the symbol `x`. You must write what you mean!

Cons objects must be quoted because otherwise they might be confused with function applications. We will return to this point in a moment.

The quoted constant 'α may also be written (`quote` α). The symbol `quote` is not a function symbol but a special marker indicating that its "argument" is to be taken literally rather than treated as an expression.

A *function expression* of n arguments is either a function symbol of n arguments or a lambda expression of the form (`lambda` (v_1 ... v_n) *body*), where the v_i are n distinct variable names, *body* is a simple expression, and no variable other than the v_i occurs "freely" in *body*. The v_i are the *formal parameters* of the lambda expression.

An occurrence of a variable symbol v is a *free occurrence* in expression x if either x is v or else x is a function application (f a_1 ... a_n) and the occurrence is a free occurrence of v in one of the a_i. An occurrence of a variable symbol v is a *binding* occurrence if the occurrence is in the formals list of a `lambda` expression. If an occurrence of a variable in an expression is neither a free occurrence nor a binding occurrence, we say it is a *bound* occurrence.

A variable is *free* in expression x if there is a free occurrence of it in x. A variable is *bound* in an expression x if there is a bound occurrence. It is possible for a variable to be simultaneously free and bound in an expression, in different occurrences of the variable. For example, `x` is both free and bound in `((lambda (x) (car x)) (cdr x))`. The variable `x` occurs three times. The first (reading from left to right) is a binding occurrence, the second is a bound occurrence, and the last is a free occurrence.

The value of a *function application*, $(f\ a_1\ \ldots\ a_n)$, under a given variable assignment is described in terms of the values, c_i, of the a_i under that assignment. How the value is computed depends on whether f is a primitive (built-in) function symbol, a defined function symbol, or a lambda expression.

Associated with every primitive function symbol of n arguments is a function of arity n. Such a function maps every combination of n ACL2 objects to some ACL2 object. For example, equal is a primitive function symbol of two arguments. The associated function returns `t` if its two arguments are the same object and returns `nil` otherwise. The primitive functions are described by the ACL2 axioms. We informally describe many primitive functions below. If f is a primitive function symbol with associated function g, then the value of the primitive function application $(f\ a_1\ \ldots\ a_n)$ is $g(c_1, \ldots, c_n)$.

If f is a defined function symbol, then the definition provides f with a *formal parameter list* and a *body*. The formal parameter list is a list of n distinct variable symbols, $v_1, \ldots, v_n$. The body is a simple expression. The value of $(f\ a_1\ \ldots\ a_n)$ is the value of *body* under the assignment in which each v_i has the value c_i.

Similarly, if f is a lambda expression, `(lambda` $(v_1\ \ldots\ v_n)$ *body*$)$, then the value of $(f\ a_1\ \ldots\ a_n)$ is the value of *body* under the assignment in which v_i has the value c_i.

As described above, `equal` is a primitive function symbol of two arguments. `If` is a primitive function symbol of three arguments. The function associated with if returns its second argument or third argument depending on its first argument. In particular, if the first is `nil`, the function returns its third argument; otherwise, it returns its second. The value of the simple expression

```
(if (equal date '(august 14 1989))  ; Comments are written
    "Happy birthday, ACL2!"         ; like this.
  nil)
```

is either the string `"Happy birthday, ACL2!"` or else the symbol `nil`, depending on whether the value assigned to the variable symbol `date` is the list `(august 14 1989)` or not. The list object in question is shown as a binary tree in Figure 3.2.

Many novice ACL2 programmers are confused about when to use the single quote mark. If numbers, characters, and strings evaluate to themselves, why not treat list constants similarly? The reason is that list constants can look just like function applications. We now return to the question of why we must write single quote marks before list constants. Reconsider the simple expression above, except remove the single quote mark.

```
(if (equal date (august 14 1989))   ; Comments are written
    "Happy birthday, ACL2!"         ; like this.
  nil)
```

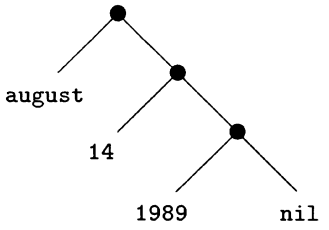

Figure 3.2: A list constant

Note that the value of the subexpression `(august 14 1989)` in this expression bears no *a priori* relationship to the similar expression in the earlier example. Here, `(august 14 1989)` is treated as the application of the function `august` to the arguments `14` and `1989` and its value is not (necessarily) the tree in Figure 3.2 but is determined by the definition of the function symbol `august`.

Two wonderful facts about ACL2 should be noted here. The first is that every expression is a ACL2 object. For example, we can write a program that operates on the list constant.

```
'(if (equal date (august 14 1989))     ; Comments are written
     "Happy birthday, ACL2!"           ; like this.
   nil)
```

The second is that the symbols used in the syntax, *e.g.*, function and variable names, are *symbols*. They are in packages. If `smith::step` is a function symbol of one argument then `(smith::step 1)` is an expression. A user who selects the package named `"SMITH"` as the current package could write this expression more simply as `(step 1)`. With `SMITH` as the current package, `(equal (step 1) t)` is the same as `(smith::equal (smith::step 1) smith::t)`. If package `SMITH` imports `acl2::equal` and `acl2::t`, this expression is the same as `(equal (smith::step 1) t)`. It is common when defining new packages to import all of the primitive function symbols. If programmers Smith and Jones behave analogously, each defining his or her own package and selecting it as current, then each can define new functions, *e.g.*, `step`, and for Smith it will mean `smith::step` but for Jones it will mean a different symbol, `jones::step`. Thus, the two systems of definitions can be loaded into a single ACL2 session without conflict. Furthermore, a programmer can refer to either function by its full name, *e.g.*, `smith::step` or `jones::step`, regardless of the current package.

Exercise 3.4 *In the following, assume* `car` *and* `cdr` *are function symbols of one argument,* `equal` *is a function symbol of two arguments, and* `if` *is*

a function symbol of three arguments. For the moment, assume there are no other function symbols. Which of the following are expressions?

1. `x`
2. π
3. `'(a b c)`
4. `(equal (car x) (cdr y))`
5. `(car x y)`
6. `(car (equal x y))`
7. `(if (if a b c) "One" "Two")`
8. `(car (a b c))`
9. `(car (cdr b c))`
10. `(car (cdr car))`
11. `(equal (if 1 (car if) cdr) equal)`
12. `((lambda (x) (equal x x)) (car a))`
13. `((lambda (x) (equal x a)) (car x))`

Exercise 3.5 *For each variable occurrence in the expressions below, say whether the occurrence is a free, bound, or binding occurrence.*

1. `x`
2. `(equal (car x) (cdr y))`
3. `((lambda (x y) (equal x y)) (car a) (car b))`
4.
```
(if (equal x y)
      ((lambda (x) (equal x 1)) (car a))
    (cdr z))
```
5. `((lambda (x y) '(equal x y)) (car a) (car b))`
6. `((lambda (x) (equal x 'x)) x)`
7.
```
(if (x x)
      ((lambda (x) (equal x 'x)) x)
    (x 'x))
```

3.2.2 Macros and Special Forms

Certain constructions are so common that succinct abbreviations have been introduced for them. For example, it is convenient to be able to write both `(+ x y)` and `(+ x y z)`. But they cannot both be simple expressions because function symbols, such as +, must have a fixed number of arguments in ACL2. We deal with this by introducing a function of two arguments named `binary-+` and then arranging for the first + expression above to be an abbreviation for `(binary-+ x y)` and the second to be an abbreviation for `(binary-+ x (binary-+ y z))`.

This "arrangement" is made via the *macro* facility of ACL2. Macros are just functions on ACL2 objects, but the objects are the syntax trees of the expressions in question. Consider an object $(f\ obj_1\ \ldots\ obj_n)$ that is used as an expression but is not a simple expression because f is a symbol but not a function symbol. Rather than treat the object as an ill-formed simple expression, ACL2 determines whether f is defined as a macro. If so, we call the object a *macro application*. To check whether a macro application is syntactically well-formed, ACL2 evaluates the function corresponding to the macro symbol f on the obj_i (not on their values but on the objects themselves) obtaining an object, *val*, to be used as an expression in place of the original macro application. We say *val* is the *immediate expansion* of $(f\ obj_1\ \ldots\ obj_n)$.

Objects built up from simple expressions and macro applications are called *expressions*. To every expression there corresponds a simple expression obtained by repeatedly expanding all the macro applications involved, outside-in, until there are none left. That simple expression is called the *expansion* or *translation* of the expression. The value of an expression is the value of its expansion.

We write "$expr_1 \Longleftrightarrow_{Syn} expr_2$" to mean that $expr_1$ and $expr_2$ expand to identical simple expressions. We have subscripted the arrow with "Syn" (for "Syntactic") to help remind you that this is not logical implication or equivalence but a relationship between expressions in the syntax.

Since users can define their own macros, the syntax of ACL2 can vary greatly from one user to another. However, we recommend that you not define your own macros until you are thoroughly familiar with the basic syntax and semantics of ACL2. Macros are defined via the `defmacro` command (see page 52).

ACL2 provides many built-in macros. Some, like +, provide for expressions that look like simple expressions but with "function symbols" that take a varying number of arguments. Others provide expressions that do not look like simple expressions at all. These macro applications are often called "special forms" in ACL2. Here are some commonly used special forms.

A `cond` expression evaluates its test expressions, p_i, sequentially until one returns non-`nil` and then evaluates the corresponding x_i.

```
(cond (p1 x1)                          (if p1 x1
      (p2 x2)                           (if p2 x2
      ...                                ...
      (pn xn)                                (if pn xn
      (t y))                                     y)...))
```

$\Longleftrightarrow_{Syn}$

A **case** expression compares the value of its switch, v, to each of the keys, e_i, in turn, and evaluates the first x_i whose key is equal.

```
(case v                                (cond ((equal v 'e1) x1)
 (e1 x1)                                     ((equal v 'e2) x2)
 (e2 x2)                                     ...
 ...                                         (t y))
 (otherwise y))
```

$\Longleftrightarrow_{Syn}$

Note the introduction of the ' marks on the e_i. The expansion scheme shown here is accurate if the e_i are numbers, symbols or characters. If they are lists, a different scheme is used. See **case**.

A **let** expression binds its local variables v_i, in parallel, to the values of the x_i and evaluates its *body*.

```
(let ((v1 x1)                          ((lambda (v1 ... vn)
      ...                                 body)
      (vn xn))                          x1
  body)                                 ...
                                        xn)
```

$\Longleftrightarrow_{Syn}$

A **let*** expression binds its locals sequentially and evaluates its *body*.

```
(let* ((v1 x1)                         (let ((v1 x1))
       ...                               (let* (...
       (vn xn))                                (vn xn))
  body)                                      body))
```

$\Longleftrightarrow_{Syn}$

Recall the restriction on `lambda` expressions that the body mention no variables freely other than those in the `lambda` formals. ACL2 provides a convention that lifts this restriction. This convention is implemented during macro expansion, though it is not implemented as a macro. Consider the application of a `lambda` expression to some actual expressions, ((`lambda` (v_1 ... v_n) *body*) a_1 ... a_n) and suppose that *body* mentions the variable x freely and that x is not among the v_i. Then ACL2 adds x to the list of formals and also adds x in the corresponding position of the actuals. Thus, the `lambda` application in question becomes ((`lambda` (v_1 ... v_n x) *body*) a_1 ... a_n x). Thus, you may write `lambda` expressions and, more commonly, `let` and `let*` expressions, that involve variables other than the `lambda` formals.

You can see how a special form expands in ACL2 by using the :trans command.

```
ACL2 >:trans (+ x y z)

(BINARY-+ X (BINARY-+ Y Z))

=> *
```

The ACL2 > above is ACL2's prompt. Its format depends on ACL2's mode. After the prompt is a user-supplied command. In this case, the user typed ":trans (+ x y z)" followed by a return. ACL2 printed the rest. The binary-+ expression is the expansion of (+ x y z). The "=> *" indicates that the expression returns one result. We talk about "multiple values" later. For more on how to interact with ACL2, see Appendix A.

Exercise 3.6 *Write the simple expression abbreviated by the following special forms. Check your expansions using* :trans.

1. ```
   (cond ((equal op 'incrmt) (+ x 1))
         ((equal op 'double) (* x 2))
         (t 0))
   ```

2. ```
   (let ((x 1)
         (y x))
     (+ x y))
   ```

Exercise 3.7 *Write the* cond *expression of Exercise 3.6 as a* case *statement.*

Exercise 3.8 *Suppose the* let *expression of Exercise 3.6 is evaluated in a context in which* x *has the value* 3. *What is the result? Now replace the* let *with* let*. *What is the expansion? What is the result in that same context?*

3.3 Primitive Functions

In this section we describe very briefly many of the primitive functions of ACL2. Some of the "functions" below are actually macros. For complete documentation see the ACL2 manual.

Boolean	Description
(and p1 p2 ...)	Logical conjunction operator
(or p1 p2 ...)	Logical disjunction operator
(implies p q)	Logical implication
(not p)	Logical negation
(iff p q)	Logical equivalence

Arithmetic	Description
`(acl2-numberp x)`	Recognizer for any type of ACL2 number
`(integerp x)`	Recognizer for integers
`(rationalp x)`	Recognizer for rationals
`(complex-rationalp x)`	Recognizer for complex numbers
`(zerop x)`	`x`= 0
`(zip x)`	`x`= 0 or `x` is not an integer
`(zp x)`	`x`= 0 or `x` is not a natural
`(< x y)`	Less than relation
`(<= x y)`	Less than or equal relation
`(> x y)`	Greater than relation
`(>= x y)`	Greater than or equal relation
`(+ x1 x2 ...)`	Addition
`(* x1 x2 ...)`	Multiplication
`(- x y)`	Subtraction
`(- x)`	Arithmetic negation
`(/ x y)`	Division
`(1- x)`	Decrement by 1
`(1+ x)`	Increment by 1
`(numerator r)`	Numerator of a rational
`(denominator r)`	Denominator of a rational
`(realpart c)`	Real part of a complex
`(imagpart c)`	Imaginary part of a complex
`(complex r i)`	Make complex number with rational components

Characters	Description
`(characterp x)`	Recognizer for character objects
`(char-code char)`	Convert character to integer
`(code-char n)`	Convert integer to character

Strings	Description
`(stringp x)`	Recognizer for string objects
`(char str n)`	$\texttt{N}^{th}$ character
`(coerce str 'list)`	Convert string to a list of chars
`(coerce charlist 'string)`	Convert list of chars to string
`(length str)`	Length of a string (or list)

Symbols	Description
`(symbolp x)`	Recognizer for symbols
`(symbol-name sym)`	Name (string) of symbol
`(symbol-package-name sym)`	Package (string) of symbol
`(intern-in-package-of-symbol str psym)`	Symbol with name `str` in the same package as `psym`

Cons Pairs and Lists	Description
`(consp x)`	Recognizer for ordered pairs
`(cons x y)`	Construct an ordered pair
`(car pair)`	First component of a pair
`(cdr pair)`	Second component of a pair
`(endp x)`	Recognizer for non-pairs
`(atom x)`	Recognizer for non-pairs
`(list x1 ... xn)`	Linear list of n objects
`(list* x1 ... xn z)`	List of n objects with `z` as the "last" cdr
`(caar pair)`	Car of the car
`(cadr pair)`	Car of the cdr
`(cdar pair)`	Cdr of the car
`(cddr pair)`	Cdr of the cdr
...	...
`(cddddr pair)`	Cdr of the cdddr
`(append x1 x2 ...)`	Concatenate linear lists
`(member-equal x lst)`	Check for membership
`(assoc-equal x alist)`	Lookup in an alist
`(nth n lst)`	$\mathtt{N}^{th}$ element of a list (0-based)
`(length lst)`	Length of a list (or of a string)
`(len lst)`	Length of a list
`(true-listp x)`	Recognizer for linear lists

There are efficiency issues, often quite minor but not always, that are not addressed in the table above. For example, `endp` is logically the same function as `atom`, but `endp` has a guard specifying its argument to be a cons or `nil`, while `atom` has no expectations on its argument. Some Common Lisp implementations may execute `endp` more efficiently than `atom`. Also not discussed in the table above are equality predicates. We have already mentioned the function `equal`, which returns `t` if its two arguments have the same value, else `nil`. But there are other functions for testing equality, each of which is logically the same as `equal`, and is potentially more efficient than `equal`, but has guards restricting the types of arguments to which it may be applied. The function `eq` can be used when at least one argument is a symbol; the function `eql` can be used when at least one argument is a number, symbol, or character; and the functions = and `int=` may be applied to pairs of numbers and to pairs of integers, respectively. With this variety of equality tests, it is not surprising that there are logically equivalent primitive functions that use different equality tests. For example, `assoc`, `assoc-equal`, and `assoc-eq` are all logically the same function, but `assoc` uses `eql` for its equality tests while `assoc-equal` uses `equal` and `assoc-eq` uses `eq`. We mention all these variations in order to acquaint the reader with their existence, but suggest that those new to ACL2 take a relaxed attitude towards the choice of primitive function where such efficiency considerations are concerned.

Exercise 3.9 *Write down an ACL2 expression formalizing the phrases below.*

1. *twice the sum of x and y*
2. *the car of the cdr of x*
3. *x is y*
4. *x is a non-integer rational number*
5. *x is a symbol in the package* SMITH
6. *0, if x is a string; 1, otherwise*

3.4 The Read-Eval-Print Loop

ACL2 presents itself to the user as a "read-eval-print loop." That is, it repeatedly reads an expression from the user, evaluates it, and prints the result. ACL2 prompts you for the next input by printing the prompt "ACL2 >". The prompt tells you the name of the current package and the mode. ACL2's mode determines, among other things, how ACL2 deals with function applications on "unexpected" inputs. The expressions typed into the read-eval-print loop are called *top-level* expressions. *Top-level expressions in ACL2 are not allowed to contain unbound variable symbols.*

Evaluating expressions is a good way to check your understanding of the language. Here is how one interaction looks.

```
ACL2 !>(+ 2 2)
4
```

Note that the value is printed on a new line, immediately below the form. In Figure 3.3 we show some other examples of top-level input, in a more compact form. Examples 12 and 13 illustrate that fractions are always reduced. Examples 16 and 27 illustrate that indexing is 0-based. Example 19 illustrates that the symbol LISP::CAR is imported into the ACL2 package.

For more information on ACL2's read-eval-print loop, see Appendix A.

3.5 Useful ACL2 Programming Commands

Here are the commands used most commonly when programming in ACL2. Note that because keywords evaluate to themselves they are not very interesting as top-level expressions. Thus, ACL2's read-eval-print loop gives special significance to input that begins with a keyword, known as a *keyword command.* See the online documentation for more about these commands. See also keyword-commands.

	Top-Level Expression	Value
1.	`(and t t t)`	`T`
2.	`(not t)`	`NIL`
3.	`(not nil)`	`T`
4.	`(integerp 3)`	`T`
5.	`(integerp 1/5)`	`NIL`
6.	`(rationalp 1/5)`	`T`
7.	`(equal 23 46)`	`NIL`
8.	`(equal 23 46/2)`	`T`
9.	`(+ 1 2 3 4)`	`10`
10.	`(* 51 2/17)`	`6`
11.	`(+ 1/2 #c(1/2 3))`	`#C(1 3)`
12.	`(denominator 25/15)`	`3`
13.	`(numerator 25/15)`	`5`
14.	`(char-code #\A)`	`65`
15.	`(code-char 66)`	`#\B`
16.	`(char "Abc" 2)`	`#\c`
17.	`(symbol-name 'abc)`	`"ABC"`
18.	`(symbol-package-name 'abc)`	`"ACL2"`
19.	`(symbol-package-name 'car)`	`"LISP"`
20.	`(intern-in-package-of-symbol "ABC" 'car)`	`LISP::ABC`
21.	`(cons 1 (cons 2 3))`	`(1 2 . 3)`
22.	`(car '(1 2 . 3))`	`1`
23.	`(cdr '(1 2 . 3))`	`(2 . 3)`
24.	`(list 1 2 3)`	`(1 2 3)`
25.	`(len '(a b c))`	`3`
26.	`(append '(1 2) '(a b))`	`(1 2 A B)`
27.	`(nth 3 '(0 1 2 a b))`	`A`

Figure 3.3: Top-Level input and output for prompt `ACL2 !>`

Definitions	Description
`(defpkg` *pkg* `'(`s_1 ... s_n`))`	Define package *pkg*, importing symbols s_i
`(defconst` **sym** `'const)`	Define constant **sym**
`(defmacro` *f args body*`)`	Define macro *f*
`(defun` *f* `(`v_1 ... v_n`)` *body*`)`	Define function *f*

`(mutual-recursion`
`(defun` f_1 ...`)`
...
`(defun` f_n ...`))`
	Define n mutually-recursive functions

`(ld "`*file*`")`	Load a file of definitions

Modes	Description
`:program`	Enter programming mode
`:logic`	Enter logic mode
`:redef`	Allow redefinitions
`:set-guard-checking t/nil`	Turn guard checking on/off
`(in-package` *pkg*`)`	Select *pkg* as current package

History (see also `history`)	Description
`:u`	Undo last command
`:pbt` *k*	Print back through command *k*
`:ubt` *k*	Undo back through command *k*
`:pc` *name*	Print command defining *name*

3.6 Common Recursions

As noted above, `(defun` *f* `(`v_1 ... v_n`)` *body*`)` defines the function *f*. The function has *n* formal parameters, v_1, ..., v_n, and its value is computed by evaluating *body* (in an environment in which the formals are bound to the objects to which the function is applied). Because ACL2 is not only a programming language but a logic, there are some restrictions on function definitions that you may find surprising.

- "Global variables" are not allowed, *i.e.*, *body* may contain no free variables other than the v_i.
- Any function used in *body*, other than the one(s) being defined [7], must have been introduced earlier. Thus, a system of definitions must be presented "bottom up."
- Recursive definitions must be proved to terminate.

We discuss the formal aspects of function definition in the next part of this book.

Because of these restrictions, the system may often reject the definitions you invent in your early experiments with ACL2. While it is not difficult to cope with the first two restrictions, once you know they are enforced, the third restriction can have a major impact. Until you learn how termination is proved and how to lead the theorem prover to proofs it cannot discover by itself, it is best to confine yourself to certain recursive schemes (or perhaps even to "turn off" logical processing altogether using `:program`). In fact, the commonly used schemes are so prevalent in ACL2 modeling that it is extremely useful to get to know them anyway.

Counting Down from a Natural Number:

[7] The `mutual-recursion` command allows several functions to be defined simultaneously.

```
(defun f (... n ...)
  (if (zp n)
      ⟨...⟩
      ⟨... (f ... (- n 1) ...) ...⟩))
```

Visiting Elements of a Linear List:

```
(defun f (... list ...)
  (if (endp list)
      ⟨...⟩
      ⟨... (car list)
           (f ... (cdr list) ...) ...⟩))
```

Visiting Tips of a Binary Tree:

```
(defun f (... tree ...)
  (if (atom tree)
      ⟨...⟩
      ⟨... (f ... (car tree) ...)
           (f ... (cdr tree) ...) ...⟩))
```

We illustrate and discuss these three schemes below.

3.6.1 Counting Down

Here is a definition of the factorial function. This example shows how to count down from some natural number `n` to `0`.

```
(defun fact (n)
  (if (zp n)                ; If n is 0 (or not a natural)
      1                     ;  return 1;
    (* n                    ; else multiply n by
       (fact (- n 1)))))    ;  fact of n-1.
```

If you type `(fact 6)` as a top-level expression, 720 is returned and printed. `(Fact 30)` returns 265252859812191058636308480000000.

ACL2 functions are total. So what is the value of `(fact 'abc)`? The answer can be derived by substituting `'abc` for `n` in the body of `fact` and evaluating. Because the value of `(zp 'abc)` is `t`, the value of `(fact 'abc)` is `1`. If you type `(fact 'abc)` as a top-level expression after defining `fact` this way, it will either print an error message or return the correct logical answer, `1`, depending on ACL2's mode. See `set-guard-checking`.

Informally, we think of `(zp n)` as answering the question "is `n` zero?" This would suggest that `(zp n)` could be replaced by `(equal n 0)`. But if we made that replacement above, what would be the value of `(fact -1)`? It would be undefined because the recursion would not terminate. This would violate the termination condition on definitions and so such a definition of `fact` would be rejected. Nonnumeric input raises similar worries and would also force the programmer to contemplate how the arithmetic

operations behave on nonnumeric data. To make it easy to define total functions that recur by counting down to zero, we define `(zp n)` to return `t` if `(equal n 0)` or if `n` is not a natural number. Using `zp` to terminate a recursion towards `0` means that all "unnatural" arguments are handled as `0` is handled.

Here we determine if a natural number is even.

```
(defun even-natp (n)
  (if (zp n)
      t
    (not (even-natp (- n 1)))))
```

Alternatively, we could have used the definition

```
(defun even-natp (n)
  (if (zp n)
      t
    (if (equal n 1)
        nil
      (even-natp (- n 2)))))
```

or even

```
(mutual-recursion
  (defun even-natp (n)
    (if (zp n)
        t
      (odd-natp (- n 1))))
  (defun odd-natp (n)
    (if (zp n)
        nil
      (even-natp (- n 1)))))
```

All three definitions of `even-natp` are equivalent. But the first is usually the simplest to manipulate formally. If you type `(even-natp 146)` the result is `t`, whereas `(even-natp 149)` produces `nil`.

Perhaps surprisingly, `(even-natp -3)` returns `t`, because `-3` is not a natural. Here is a function that recognizes even integers.

```
(defun even-intp (n)
  (if (< n 0) (even-natp (- n)) (even-natp n)))
```

The following function computes how many times the positive integer `j` can be subtracted from the natural `i` before reaching `0`, *i.e.*, the integer quotient of `i` divided by `j`.

```
(defun nat-quo (i j)
  (if (or (zp i)
          (zp j)
          (< i j))
```

```
    0
  (+ 1 (nat-quo (- i j) j))))
```

`(Nat-quo 21 5)` returns 4, `(nat-quo 25 5)` returns 5, and `(nat-quo 3 0)` returns 0. The last result may be surprising. We could have defined division by 0 to return some other result, but since ACL2 functions are total, it has to be defined to be something and it was convenient here to let the base case of the recursion specify the value.

Exercise 3.10 *Define the Fibonacci function,* `(fib n)`*, so that on successive integer values of* `n` *starting at* 0*, the function returns 1, 1, 2, 3, 5, 8, 13, ...*

Exercise 3.11 *Define* `(pascal i j)` *to be the binomial coefficient,* $\binom{i}{j}$*,* i.e.*, so that it returns the* $\mathtt{j}^{th}$ *entry in the* $\mathtt{i}^{th}$ *row of "Pascal's triangle,"*

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5  10  10   5   1
        1   6  15  20  15   6   1
  ...                ...           ...
```

For example, `(pascal 6 2)` *is* 15.

3.6.2 Linear Lists

This function sums the elements of a linear list (or true list).

```
(defun sum-list (x)
  (if (endp x)
      0
    (+ (car x) (sum-list (cdr x)))))
```

`(Sum-list '(1 2 3 4))` evaluates to 10. Perhaps surprisingly, `(sum-list '(1 2 abc 4))` evaluates to 7. The non-number `'abc` is coerced to 0 by `+`. Also, `(sum-list '(1 2 3 . 4))` evaluates to 6; the list is coerced to a true-list by the use of `endp` in the termination check. Naively, `(endp x)` checks whether `x` is the empty list, `nil`, but actually it checks whether `x` is any atom.

This function determines whether `e` is an element of a linear list.

```
(defun mem (e x)
  (if (endp x)
      nil
```

```
    (if (equal e (car x))
        t
      (mem e (cdr x)))))
```

`(Mem 'abc '(1 2 abc 4))` evaluates to `t` and `(mem 'xyz '(1 2 abc 4))` evaluates to `nil`. Note that `(mem 2 23)` evaluates to `nil` because 23 is an atom.

This function determines the position (if any) at which `e` first occurs in a linear list.

```
(defun mempos (i e x)
  (if (endp x)
      nil
    (if (equal e (car x))
        i
      (mempos (+ 1 i) e (cdr x)))))
```

`(Mempos 0 'abc '(1 2 abc 4))` evaluates to 2 and `(mempos 0 'xyz '(1 2 abc 4))` evaluates to `nil`.

This function inserts n immediately before the first element of `x` that is bigger than n or at the end.

```
(defun insert (n x)
  (cond ((endp x) (list n))
        ((< n (car x)) (cons n x))
        (t (cons (car x) (insert n (cdr x))))))
```

`(Insert 3 '(0 1 2 5 7))` evaluates to `(0 1 2 3 5 7)` and `(insert 2 nil)` evaluates to `(2)`.

Exercise 3.12 *Define* `(subset x y)` *to return* `t` *if every element of* `x` *is an element of* `y`*, and* `nil`*, otherwise.*

Exercise 3.13 *Define* `(un x y)` *to return a list with the property that e is an element of the list iff either e is an element of* `x` *or e is an element of* `y`*, and moreover, if neither* `x` *nor* `y` *has duplicate elements then* `(un x y)` *has no duplicate elements.*

Exercise 3.14 *Define* `(int x y)` *to return a list with the property that e is an element of the list iff e is an element of* `x` *and e is an element of* `y`*.*

Exercise 3.15 *Define* `(diff x y)` *to return a list with the property that e is an element of the list iff e is an element of* `x` *and e is not an element of* `y`*.*

Exercise 3.16 *Define* `(rev x)` *to return the list containing the elements of* `x` *in the reverse order. For example,* `(rev '(a b a c d))` *should return* `(d c a b a)`*.*

Exercise 3.17 *Define* `(isort x)` *to return a list containing the elements of* `x` *in ascending order. You may assume* `x` *is a list of numbers. For example,* `(isort '(4 1 0 9 7 4))` *should return* `(0 1 4 4 7 9)`*.*

3.6.3 Binary Trees

This function sums the tips of a binary tree.

```
(defun sum-tips (x)
  (if (atom x)
      x
    (+ (sum-tips (car x))
       (sum-tips (cdr x)))))
```

`(Sum-tips '((1 . 2) . (3 . 4)))` evaluates to 10 and `(sum-tips 23)` to 23.

This function visits the tips of tree `x` and whenever it finds one equal to `old` it replaces it by the tree `new`.

```
(defun replace-tips (old new x)
  (if (atom x)
      (if (equal x old) new x)
    (cons (replace-tips old new (car x))
          (replace-tips old new (cdr x)))))
```

`(Replace-tips 2 '(7 . 8) '((1 . 2) . (3 . 2)))` returns `((1 . (7 . 8)) . (3 . (7 . 8)))`. If you try this evaluation in ACL2 you will see that the result prints differently, but it is the same tree, `((1 7 . 8) 3 7 . 8)`. Perhaps surprisingly, `(replace-tips nil 'hi '(nil 1 2))` returns `(hi 1 2 . hi)` because `'(nil 1 2)` abbreviates `'(nil 1 2 . nil)`.

Exercise 3.18 *Define* `(flatten x)` *to return the list of tips of the binary tree* `x`*, in the order in which they are encountered in a left-to-right sweep. For example,* `(flatten '((a . b) . c))` *should return* `(a b c)`.

Exercise 3.19 *Define* `(swap-tree x)` *to return the mirror image of the binary tree* `x`*. The mirror image of* `((a . b) . c)` *is* `(c . (b . a))`.

Exercise 3.20 *Define* `(depth x)` *so that it returns the length of the longest branch in the binary tree* `x`.

Exercise 3.21 *Consider the notion of a "path" down a binary tree to a given subtree. Let a path be given by a list of* `A`*'s and* `D`*'s indicating which way you should go at each cons. The length of the path indicates how many steps to take to reach the subtree in question. For example, the path to the second* `2` *in* `((1 . (2 . 2)) . 3)` *is* `(A D D)`*. Define the function* `(subtree p x)` *to return the subtree at the end of path* `p` *in tree* `x`.

Exercise 3.22 *Define the function* `(replace-subtree p new x)` *to replace the subtree at the end of path* `p` *with the subtree* `new` *in tree* `x`*. See Exercise 3.21.*

3.7 Multiple Values

It is sometimes useful for a function to compute and return multiple results. Here is a function that scans the tips of a binary tree once and returns two results: the number of symbols among the tips and the number of strings among the tips. In the definition below we use `cond` instead of the simpler `if`. This is merely stylistic.

```
(defun classify-tips (x)
  (cond ((atom x)
         (cond ((symbolp x) (mv 1 0))
               ((stringp x) (mv 0 1))
               (t           (mv 0 0))))
        (t (mv-let (syms1 strs1)
                   (classify-tips (car x))
                   (mv-let (syms2 strs2)
                           (classify-tips (cdr x))
                           (mv (+ syms1 syms2)
                               (+ strs1 strs2)))))))
```

The two key forms for dealing with multiple values are:

- (mv e_1 ... e_n) evaluates the e_i and returns a vector of the n results.
- (mv-let (v_1 ... v_n) *res body*), evaluates *res*, obtaining a vector of exactly n results, binds the v_i to the successive results, and evaluates *body* under that assignment.

Syntactic restrictions insure that if a function returns n results along one output path it returns n results along all paths. The number of results a function returns is called its *multiplicity*. An `mv-let` expression binding n variables must have a *res* expression of multiplicity n. If an expression of multiplicity n is evaluated at the top-level, a list of the n results is printed.[8]

Exercise 3.23 *Define* `(deep-tip x)` *so that it returns a tip (leaf) of the binary tree* `x` *with the property that the tip occurs maximally deep in the tree. Thus,* `(deep-tip '((a . b) . c))` *might return* `a` *or might return* `b` *(both occur at depth 2), but it would not return* `c`*, which occurs at depth 1.*

[8]There is an exception to this convention. Certain results of multiplicity 3, called "error triples," are treated differently by the top-level printer. When the result (`mv` *flag val* `state`) is returned to the top-level and `state` is ACL2's state object `state`, the printer prints nothing if *flag* is true and prints only *val*, preceded by a space, otherwise. Within the ACL2 system code, such triples are used to signal error conditions to their callers; see `ld-error-triples`.

3.8 Guards and Type Correctness

Lisp is syntactically untyped. But the Common Lisp standard [40] specifies the intended domain of each Lisp primitive. Common Lisp functions are undefined outside their intended domain. What they do when evaluated "out there" is up to the implementation. For example, it is legal to write `(+ t 3)` in Common Lisp; the standard does not define its value, but most implementations cause an error when the expression is evaluated.

All ACL2 functions are total: functions may be applied to arbitrary arguments. The ACL2 axioms specify the value returned by each Lisp primitive on every possible input. That is, we have "completed" the Common Lisp specification by defining the primitives everywhere. But ACL2 has built into it the "intended domains" of the primitives, and, when expressions are evaluated, ACL2 checks that the computation stays within the intended domains of the functions concerned and signals an error otherwise (if guard checking mode is turned on, which it is by default or with `:set-guard-checking`).

The intended domain of a function is specified by its *guard*. For example, the guard of the factorial function, `fact`, defined above, could be that `n` must be a natural number. ACL2 permits you to specify the guard using declarations (page 55).

If you apply a function outside the domain specified by its guard, an error is signalled as a "courtesy." The function *is defined everywhere.* If you wish to ignore the guards and compute according to ACL2's axioms, turn guard checking off with the keyword command `:set-guard-checking nil`.

Guard verification is the process of proving theorems that establish that a given function respects the guards of all of the functions used in its body. Such a function is said to be *Common Lisp compliant* and the concept lifts from functions to expressions. Common Lisp compliant ACL2 expressions evaluate to the same values as in any Common Lisp (with the appropriate files loaded).

Common Lisp compliant ACL2 functions and expressions are "well-typed." Evaluation never ventures outside the intended domains of the concerned functions. In addition, when a function has been shown to be compliant it can often be executed faster, because runtime type checks can be avoided.

Nevertheless, we recommend that you avoid guards and guard verification until you understand the semantics of ACL2 and how to use the theorem prover. See `guard`.

3.9 Introduction to Macros

As noted earlier, newcomers should not define macros. But once you can define recursive functions on cons trees, you might experiment with macros.

Macros are like functions except that they operate at the syntactic level. To make the analogy clear, we summarize functions and macros in parallel terms.

A function definition is of the form (defun f *formals body*). A function application is an expression of the form (f a_1 ... a_n), where the a_i are expressions. The value of a function application is computed by binding the formals of f to the values of the a_i and evaluating the body of f.

A macro definition is of the form (defmacro m *formals body*). A macro application is an expression of the form (m a_1 ... a_n). The (first level) expansion of a macro application is the expression produced by binding the formals of m to the a_i and evaluating the body of m.

Observe the key differences:

- In a function application, the a_i are expressions, but in a macro application they need not be expressions. They are just arbitrary ACL2 objects.
- Function applications have values while macro applications have expansions.
- In function evaluation, the formals are bound to the values of the a_i, but in macro expansion, the formals are bound to the a_i themselves.
- Function bodies produce arbitrary objects while macro bodies must produce expressions.

Roughly speaking, functions map objects to objects while macros map objects (which are often expressions) to expressions.

Since the expressions produced by macros may contain additional macro applications, they may have to be expanded still further. That is why in defining expressions (page 37) we say:

> Objects built up from simple expressions and macro applications are called *expressions*. To every expression there corresponds a simple expression obtained by repeatedly expanding all the macro applications involved, outside-in, until there are none left. That simple expression is called the *expansion* or *translation* of the expression. The value of an expression is the value of its expansion.

Here is a simple example of an ACL2 macro definition.

```
(defmacro cadr (x)
  (list 'car (list 'cdr x)))
```

Thus `(cadr (rev a))` is a macro application. The expansion of this application is the expression `(car (cdr (rev a)))`. In the previously introduced notation we could express this with `(cadr (rev a))` $\Longleftrightarrow_{Syn}$ `(car (cdr (rev a)))`. The value of `(cadr (rev a))` is, by definition, the value of `(car (cdr (rev a)))`, which, of course, depends on the value of `a`.

While `cadr` illustrates how macros are defined and expanded, it does not motivate the introduction of macros very well because much the same effect could be achieved by defining the function below.[9]

```
(defun cadr (x)
  (car (cdr x)))
```

Had we done that instead of defining `cadr` as a macro, then `(cadr (rev a))` would be a function application, we would not be able to speak of its expansion (nor would be interested in doing so), and its value for any given assignment to `a` would be the same as it would have been had `cadr` been defined as a macro.

So why might we want macros?

Suppose you wanted an if-then-else operator that tests equality with `0` instead of inequality with `nil`. That is, suppose you wished to write `(ifz` α β γ`)` and mean: if α evaluates to 0, then evaluate β, else evaluate γ. You might try to do this with the function definition below.

```
(defun ifz (a b c)
  (if (equal a 0) b c))
```

But this would not work as described because all the arguments to a function are evaluated before the body is evaluated. Thus `(ifz 0 1 (/ 2 0))` would evaluate `(/ 2 0)`, dividing 2 by 0, before it detected that the first argument to the `ifz` was 0.

However, if you defined the macro

```
(defmacro ifz (a b c)
  (list 'if
        (list 'equal a 0)
        b
        c))
```

then `(ifz 0 1 (/ 2 0))` $\Longleftrightarrow_{Syn}$ `(if (equal 0 0) 1 (/ 2 0))`. By definition, the value of the former expression is the value of the latter, namely 1, and `(/ 2 0)` is never evaluated.

To see the one-level expansion of a macro application, use the keyword command `:trans1`.

```
ACL2 >:trans1 (ifz (car x) nil (cadr x))
 (IF (EQUAL (CAR X) 0) NIL (CADR X))
```

[9] In some Lisps `cadr` is in fact defined as a function rather than as a macro.

To see the (full) expansion or translation of an expression, use :trans.

```
ACL2 >:trans (ifz (car x) nil (cadr x))

(IF (EQUAL (CAR X) '0)
    'NIL
    (CAR (CDR X)))

=> *
```

Observe that `(cadr x)` was translated to `(car (cdr x))`. In addition, quotes were introduced on the constant symbol `nil` and the numeric constant `0`. Internally, ACL2 always keeps quote marks on constants, but you may write numeric, character, and string constants without quote marks.

The macro expansion process may not always terminate. The macro

```
(defmacro macloop (a)
  (list 'macloop a))
```

is such that `(macloop 1)` expands to itself, so the expansion process repeats indefinitely. It is up to you, the programmer, to avoid such loops.

In describing how the macro application $(m\ a_1\ \ldots\ a_n)$ is expanded we said that the formals of m are bound to the a_i. Normally, the formals are bound to successive a_i in a left-to-right sweep and there must be exactly as many a_i as there are formals in the `defmacro`.

But the sweep through the formals and the a_i is more complicated because some "formals" are given special treatment. These symbols are called *ampersand markers* in Common Lisp. These symbols affect how other formals are bound. We explain this in more detail on page 65. But one ampersand marker, `&rest`, is used so often that it merits note in this introduction to ACL2.

Suppose you wish to define a macro that takes an arbitrary number of arguments. That is done by using a formals list like `(&rest args)`. The "formal" `&rest` is just an ampersand marker that means that the next formal, `args`, should be bound to the list of all of the remaining arguments. Below we define `expt*` as a macro that takes an arbitrary number of arguments and expands to a stack of exponents, *e.g.*, `(expt* i j k)` expands to `(expt i (expt j (expt k 1)))`.

```
(defmacro expt* (&rest args)
  (cond ((endp args) 1)
        (t (list 'expt (car args)
                       (cons 'expt* (cdr args))))))
```

Observe that

```
(expt* i j k)
```
$\Longleftrightarrow_{Syn}$
```
(expt i (expt* j k))
```

$\Longleftrightarrow_{Syn}$
```
(expt i (expt j (expt* k)))
```
$\Longleftrightarrow_{Syn}$
```
(expt i (expt j (expt k (expt*))))
```
$\Longleftrightarrow_{Syn}$
```
(expt i (expt j (expt k 1))).
```

The `list` "function" in ACL2 is really a macro. The expression `(list a b c)` expands to `(cons a (cons b (cons c nil)))`. Here is a definition.

```
(defmacro list (&rest args)
  (cond ((endp args) nil)
        (t (cons 'cons
                 (cons (car args)
                       (cons (cons 'list (cdr args))
                             nil))))))
```

(Since `list` is already defined in ACL2 you cannot experiment with this `defmacro` literally, but you could change both occurrences of `list` above to, say, `my-list` and experiment.)

We discuss ampersand markers and give more examples of macros in Chapter 5.

3.10 Declarations

ACL2 permits the use of *declarations* in certain expressions. Declarations are used to inform the Lisp compiler and the ACL2 system about various pragmatic issues. Declarations do not affect the meaning of an expression. Their use by the compiler is documented in Chapter 9 of [40]. Their use by the ACL2 system is documented in this book and in the online documentation (see `declare`).

In ACL2, declarations can only be written in the following contexts: immediately after the formals in a function or macro definition and immediately after the bound variables in a `let`, `let*`, or `mv-let` expression.

Except in three situations, the novice ACL2 programmer will probably not need to use declarations. The common exceptions are:

- `(declare (ignore` v_1 ... v_n`))` – to declare that the indicated formal parameters or `let`, `let*`, or `mv-let` variables are intentionally not used in the body of the form. The declaration should appear immediately after the ignored variables are bound.

- `(declare (xargs :guard` g`))` – to declare that the guard for a defined function or macro is the expression g. The declaration should occur immediately after the formals of the definition.

- `(declare (xargs :measure` *m*`))` – to declare the "measure" to be used to prove termination. The declaration should occur immediately after the formals of the definition. Here, *m* is an expression, in the formal parameters of the function, that must always evaluate to an ACL2 "ordinal" and that must decrease as the function recurs. The most common measure used is `acl2-count`, a function of one argument that measures the "size" of an object. We discuss the notions further on page 85. When a definition is supplied without a `:measure`, the system tries to supply one. It looks for a formal, *v*, that is changed in every recursive call and that is tested somewhere on every branch leading to a recursion. If it finds such a *v*, it guesses the measure `(acl2-count` *v*`)`, for the first such *v*.

Many other keywords are permitted in the `xargs` declare form. See `xargs`. Multiple keywords may be used, *e.g.*, `(declare (xargs :guard` *g* `:measure` *m*`))` is legal. `Ignore` and `xargs` may be used in the same declaration, *e.g.*, `(declare (ignore x) (xargs :measure` *m*`))`.

Below we declare the guard for `fact`: its intended domain is the non-negative integers. We also declare the measure to use in its admission. The measure declaration is unnecessary in this case, since the system "guesses" this measure.

```
(defun fact (n)
  (declare (xargs :guard (and (integerp n) (<= 0 n))
                  :measure (acl2-count n)))
  (if (zp n)
      1
    (* n (fact (- n 1)))))
```

4

Programming Exercises

We strongly suggest that you work these exercises, especially if you are not familiar with ACL2 or you are not completely comfortable with recursive definition. Solutions can be found on ACL2's home page; just follow the link for this book.

A number of addition programming examples can be found in the documentation under `acl2-tutorial`. Further exercises can be found on the Web page for this book.

All of these exercises may be done in `:program` mode (see page 230 in Appendix A).

Once you have learned to use the ACL2 theorem prover, you might do these exercises again, this time while operating in `:logic` mode. In `:logic` mode, your definitions become axioms—but you must first prove that each recursive definition terminates. It is also instructive to add guards (see `guard`) to the definitions introduced here and to verify that your definitions respect your guards. We recommend that you not use `:logic` mode or add guards to your definitions until you are prepared to interact with the theorem prover.

Remark. We allow ourselves to refer to a formal parameter when we really intend to refer to its value. For example, we may say "`(f a)` returns `a`+1" rather than the more verbose: "`(f a)` evaluates to $v + 1$, where v is the value of `a` (in the implicit environment)."

4.1 Non-Recursive Definitions

The following exercises are intended to be very simple. Their primary purpose is to get the reader used to interacting with ACL2. Exercise 4.3 is used in a more interesting exercise (4.8) that comes later.

Exercise 4.1 *Use the functions* `nth` *and* `reverse` *in order to define a function* `(from-end n lst)` *that picks out the* $\mathtt{n}^{th}$ *element from the end of the given list* `lst`*, using zero-based indexing.*

For example:

```
ACL2 !>(nth 4 '(a b c d e f))
```

```
E
ACL2 !>(from-end 4 '(a b c d e f))
B
ACL2 !>
```

Exercise 4.2 *Define a function* `(update-alist key val a)` *that returns the result of "updating" association list* `a` *(see page 31) so that* `key` *is associated with the value* `val`*. Do not use recursion.*

Note that `(assoc-equal key1 (update-alist key2 val a))` should return `(cons key1 val)` if `key1` equals `key2`, and otherwise should return the same value as `(assoc-equal key1 a)`. Some examples follow.

```
ACL2 !>(assoc-equal 'b '((c . 5) (b . 4) (a . 3)))
(B . 4)
ACL2 !>(assoc-equal 'c '((c . 5) (b . 4) (a . 3)))
(C . 5)
ACL2 !>(assoc-equal 'b
                    (update-alist 'b 17 '((c . 5) (b . 4) (a . 3))))
(B . 17)
ACL2 !>(assoc-equal 'c
                    (update-alist 'b 17 '((c . 5) (b . 4) (a . 3))))
(C . 5)
ACL2 !>(assoc-equal 'd '((c . 5) (b . 4) (a . 3)))
NIL
ACL2 !>(assoc-equal 'd
                    (update-alist 'b 17 '((c . 5) (b . 4) (a . 3))))
NIL
ACL2 !>
```

Exercise 4.3 *Define a function* `(next-k k)` *that returns* $3k+1$ *if its positive integer input* `k` *is an odd number, and otherwise returns* $k/2$*. (*Hint: *Execute* `:pe evenp` *in the ACL2 loop or see* `evenp` *in the ACL2 documentation.)*

4.2 Some Recursive Definitions

Exercise 4.4 *Use the function* `mem` *(defined on page 48) to define a function* `(no-dupls-p lst)` *that returns* `t` *if the input list* `lst` *has no duplicate elements, and returns* `nil` *otherwise.*

For example:

```
ACL2 !>(no-dupls-p '(a b c a d))
NIL
ACL2 !>(no-dupls-p '(a b c d))
T
ACL2 !>
```

Exercise 4.5 *Define a function* `(get-keys alist)` *that returns the list obtained by taking the* `car` *of each pair in the given* `alist`.

For example:

```
ACL2 !>(get-keys '((c . 5) (b . 4) (a . 3) (c . 2)))
(C B A C)
ACL2 !>
```

Exercise 4.6 *Define a function* `(perm x y)` *that is true (returns* `t`*) for those true lists* `x` *and* `y` *such that* `y` *can be obtained from* `x` *by rearranging elements. (*Hint: *Use the function* `mem` *defined above on page 48. You may also need to define an auxiliary function.)*

For example:

```
ACL2 !>(perm '(a b c a) '(b a a c))
T
ACL2 !>(perm '(a b c a) '(b a a c b))
NIL
ACL2 !>(perm '(a b c a) '(a b c))
NIL
ACL2 !>
```

We need the following notion for the next exercise. A key `k` is said to be *bound* in an alist `a` if `(assoc-equal k a)` is not `nil` (or equivalently if it is a cons pair).

Exercise 4.7 *Define* `(update-alist-rec key val alist)` *to satisfy the specification in Exercise 4.2 above, but with the additional property that if* `key` *is bound in* `alist`*, then function* `get-keys` *(defined in Exercise 4.5) when applied to* `alist` *returns the same thing it returns when applied to* `(update-alist-rec key val alist)`.

For example:

```
ACL2 !>(update-alist-rec 'b 17 '((c . 5) (b . 4) (a . 3)))
((C . 5) (B . 17) (A . 3))
ACL2 !>(update-alist-rec 'c 17 '((c . 5) (b . 4) (a . 3)))
((C . 17) (B . 4) (A . 3))
ACL2 !>(update-alist-rec 'd 17 '((c . 5) (b . 4) (a . 3)))
((C . 5) (B . 4) (A . 3) (D . 17))
ACL2 !>(equal
         (get-keys '((b . 4) (a . 3)))
         (get-keys (update-alist-rec 'a 17
                                     '((b . 4) (a . 3)))))
T
ACL2 !>
```

The following exercise is an open problem for :logic mode; it is not known if the computation always terminates [42]. Hence, it is a good idea to put :mode :program in the xargs of your defun; see also defun-mode.

Exercise 4.8 *Define a function* (next-k-iter-list k) *that returns the sequence obtained by iterating the function* next-k, *defined in Exercise 4.3, starting with input* k *and stopping with the value 1. Also define a function* (next-k-iter k) *that returns the number of iterations required to reach 1 using* next-k, *without constructing a list. Can you state a relationship between these two functions?*

For example:

```
ACL2 !>(next-k-iter-list 11)
(11 34 17 52 26 13 40 20 10 5 16 8 4 2 1)
ACL2 !>(next-k-iter 11)
14
```

Exercise 4.9 *Define the function* (next-k-max-iterations n) *that returns the maximum value of* (next-k-iter i) *for* $1 \leq i \leq$ n.

Consider the following examples. The last form may cause stack overflows in some systems unless next-k-max-iterations is compiled (see compilation).

```
ACL2 !>(next-k-iter 1)
0
ACL2 !>(next-k-iter 2)
1
ACL2 !>(next-k-iter 3)
7
ACL2 !>(next-k-iter 4)
2
ACL2 !>(next-k-max-iterations 4)
7
ACL2 !>(next-k-max-iterations 10000)
261
```

Exercise 4.10 *The popular method of "casting out nines" checks for divisibility by 9 by adding the digits of a positive base-10 integer to get a new number, iterating until the result is a single digit. The integer is divisible by 9 exactly when the single digit is 9. Define a function that tests for divisibility by 9 using this algorithm.* Hint: *Use* floor *and* mod.

For example:

```
ACL2 !>(cast-out-nines 99998)
NIL
ACL2 !>(cast-out-nines 99999)
T
ACL2 !>
```

4.3 Tail Recursion

Many Lisp compilers can compute more efficiently when a recursive function call returns the value of the enclosing call. In fact, such *tail recursion* may be necessary when the call stack otherwise grows too large. Consider for example the following function.

```
(defun zeros (n)
  (if (zp n)
      nil
    (cons 0 (zeros (1- n)))))
```

If we compile (using `:comp zeros`) and then attempt to call this function with input 10000, we get a stack overflow, illustrated as follows. (Here, the underlying Lisp is GCL.)

```
ACL2 !>(len (zeros 10000))

Error: Value stack overflow.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by LP.
Broken at COND.  Type :H for Help.
ACL2>>
```

We can define a *tail recursive* function that avoids the above problem, and use it to define a new version `zeros1` of `zeros`, as follows.

```
(defun zeros-tailrec (n acc)
  (if (zp n)
      acc
    (zeros-tailrec (1- n) (cons 0 acc))))
(defun zeros1 (n)
  (zeros-tailrec n nil))
```

We may now do the desired computation, even with much larger inputs.

```
ACL2 !>(len (zeros1 100000))
[SGC for 86 FIXNUM pages.. $\ldots$ GC finished]
[SGC for 86 FIXNUM pages.. $\ldots$ GC finished]
100000
ACL2 !>
```

The `SGC` message is printed by GCL when it does a garbage collection. Your system may not print these messages.

Exercise 4.11 *A definition of the factorial function,* `fact`*, is given above (page 56). Give an alternate definition that uses tail recursion.*

Exercise 4.12 *Give tail recursive definitions of the following functions:*

- `get-keys` *(Exercise 4.5),*
- `update-alist-rec` *(Exercise 4.7),*
- `next-k-iter` *(Exercise 4.8), and*
- `next-k-max-iterations` *(Exercise 4.9).*

4.4 Multiple Values and Sorting

Although there is no general connection between multiple values and sorting, the following related exercises will take the reader through both of these concepts.

Exercise 4.13 *Define a function* `(split-list x)` *that returns two values: every other element of the list starting with the first, and every other element of the list starting with the second.*

For example:

```
ACL2 !>(split-list '(3 6 2 9 2 8 7 3/2))
((3 2 2 7) (6 9 8 3/2))
ACL2 !>(mv-let (odds evens)
               (split-list '(3 6 2 9 2 8 7 3/2))
               (append odds evens))
(3 2 2 7 6 9 8 3/2)
ACL2 !>
```

Exercise 4.14 *Define a function* `(merge2 x y)` *such that if* `x` *and* `y` *are true lists of rational numbers (see* `rational-listp`*) sorted in non-decreasing order, then* `(merge2 x y)` *is a permutation (see Exercise 4.6) of* `(append x y)` *but with elements in non-decreasing order.*

For example:

```
ACL2 !>(merge2 '(2 2 3 7) '(3/2 6 8 9))
(3/2 2 2 3 6 7 8 9)
ACL2 !>
```

Exercise 4.15 *Define a function* `(mergesort x)` *that returns a permutation of* `x`*, a true list of rational numbers, such that the result is sorted in non-decreasing order. Use the functions of the preceding two exercises.*

For example:

```
ACL2 !>(mergesort '(3 6 2 9 2 8 7 3/2))
(3/2 2 2 3 6 7 8 9)
ACL2 !>
```

4.5 Mutual Recursion

The following exercises use a simplified version of the ACL2 expression syntax to explore mutual recursion. See page 46 and `mutual-recursion`. For these exercises, let us define the class of *basic terms* to be the least class that includes all symbols, numbers, characters, and strings, as well as all true lists containing a symbol (conceptually, a function) followed by a list of basic terms.

Exercise 4.16 *Define a function* `(acl2-basic-termp x)` *that recognizes basic terms, as defined just above. That is, this function should return* `t` *if its input is a basic term, otherwise it should return* `nil`. Hint: *Use* `mutual-recursion` *to define a recognizer for lists of basic terms at the same time.*

Note that ACL2 syntax additionally allows quoted constants, *e.g.*, `(quote x)` or equivalently, `'x`. Feel free to extend your functions appropriately to recognize a larger class of ACL2 terms.

Exercise 4.17 *Define a function* `(acl2-sub new old x)` *which, for basic terms* `new`, `old`, *and* `x`, *returns the result of replacing each occurrence of* `old` *in* `x` *with* `new`.

For example:

```
ACL2 !>(acl2-sub '(f x) '(g y) '(h (g x) (g y) 17))
(H (G X) (F X) 17)
ACL2 !>
```

Exercise 4.18 *Write an alternate recognizer for basic terms that takes a list of legal function symbols as a formal parameter and checks that every function symbol used in the term is a member of this list. Then define a function* `(acl2-value x a)` *that returns, for any alist* `a` *and basic term* `x` *with respect to function symbols* `+`, `*`, *and* `=`, *the value of* `x` *obtained by assigning symbols their values from* `a`. *Exception:* `t` *and* `nil` *are their own values, and symbols not bound in* `a` *are assigned the value 0.*

For example:

```
ACL2 !>(acl2-value 22 nil)
22
ACL2 !>(acl2-value '(= 22 (+ x (* y 3))) '((x . 1) (y . 7)))
T
ACL2 !>(acl2-value '(= 22 (+ x (* y 3))) '((x . 3) (y . 7)))
NIL
ACL2 !>
```

4.6 Arrays and Single-Threaded Objects

The exercises in this section are relatively advanced, and optional. The reader is referred to the documentation on `arrays` and `stobj`, respectively, for information on arrays and single-threaded objects.

Exercise 4.19 *Redo Exercise 4.9 using ACL2* `arrays` *to store values of* `next-k-iter` *computed along the way. The technique of remembering intermediate values is called* memoization. *Use an array of length 100,000.*

Exercise 4.20 *Redo the above exercise using ACL2 single-threaded objects (see* `stobj`*) instead of arrays.*

Exercise 4.21 *Use an input of 50000 to compare the running times of your solutions to Exercises 4.9, 4.12, 4.19, and 4.20.*

For example, in raw Lisp (running GCL) after compiling (see `:comp`), we compute the running time as follows.

```
ACL2>(time (next-k-max-iterations-stobj 50000 st))
real time : 2.000 secs
run time  : 1.690 secs
323
```

You should see a significant speed-up with arrays and stobjs.

Single-threaded objects are especially useful in defining "machine interpreters," functions that model state machines. In these functions, states are represented as ACL2 objects—often single-threaded objects—and state transitions are carried out on this object. We define a simple stack machine on page 205, but that machine does not use single-threaded objects. A machine using a single-threaded object for its state is presented in the article by Greve, Wilding, and Hardin in [22].

5

Macros

Here we briefly offer some additional remarks on defining macros. You may want to skip this chapter on your first reading of the book and come back to it if you want to define more sophisticated macros than those allowed by the sketch given on page 52.

Macros are part of Common Lisp. ACL2 supports some, but not all, of the macro features and adds no new features. ACL2 signals errors when unsupported features are used. This chapter summarizes completely the features ACL2 supports. For details, see [40]. To see how macros are used in ACL2, see the case studies in the companion volume [22], especially those by Borrione *et al.*, Greve *et al.*, and Jamsek.

5.1 Advice on Using Macros

Macros are just syntactic sugar. In ACL2, macro applications are expanded away during function definition, during top-level evaluation, and before theorem proving begins. Since every macro application is just an abbreviation for some simple expression, macros add nothing to the power of the language. If you cannot state a property using just simple expressions, then you cannot state it using macros either. We have seen users struggle to use macros to add fundamental extensions to ACL2 such as higher-order functions. Such efforts are doomed.

The most common complaint of ACL2 programmers when they confront the limitations of ACL2 macros is "I want macros that are sensitive to the state of the ACL2 data base." This is not allowed. On page 193 of *Common Lisp the Language*, [40], we see:

> More generally, an implementation of Common Lisp has great latitude in deciding exactly when to expand macro calls within a program. ... Macros should be written in such a way as to depend as little as possible on the execution environment to produce a correct expansion.

One implementation of Common Lisp may expand macro applications at `defun` time while another may expand them at evaluation or compile time.

For macros to be mere syntactic sugar, as they are in ACL2, it is essential that a given macro application expand the same way every time, regardless of context. The ACL2 reflection of this remark is that we insist that macros be functions of their arguments.

Given these caveats, why have macros? Macros are very useful for making your formulas more succinct. They allow you to codify and abbreviate certain programming and theorem-proving styles or strategies. They can be used to make your formulas "prettier" for presentation in papers and talks. We recommend that you not use macros until you are very familiar with the functions you have defined and how they are typically used in your applications. Only then should you introduce macros to abbreviate frequently used combinations.

5.2 Details on Ampersand Markers

Macros allow you to compute an expression from the arbitrary objects provided in the macro application. The computation proceeds roughly as follows: the macro formals are bound to the elements in the macro application and then the macro body is evaluated to produce the desired expression. The evaluation of the macro body is straightforward. However, we have already seen that the ampersand marker `&rest` complicates the binding of the macro formals. On the other hand, `&rest` allows you to write forms that appear to have varying numbers of arguments from one application to the next. There are a variety of other ampersand markers. None increase the expressive power of macros—after all, you could use a single `&rest` argument and decompose it at will in the macro body. But for convenience, Common Lisp provides many ampersand markers.

ACL2 only supports `&whole`, `&rest`, `&optional`, `&key`, `&body`, and `&allow-other-keys`. `&Body` is like `&rest` and we do not discuss it.

Below we define the macro `silly1` to illustrate some of the issues in binding of macro formals. This definition uses three of the common ampersand markers: `&whole`, `&optional`, and `&rest`. It has six formals. The body of `silly1` constructs a `quote` expression that exhibits the bindings of the six formals. Thus, we can evaluate applications of `silly1` to see how the formals are handled.

```
(defmacro silly1 (&whole appl
                  x y
                  &optional opt1 opt2
                  &rest rst)
  (list 'quote
        (list (list 'appl appl)
              (list 'x    x)
              (list 'y    y)
```

```
        (list 'opt1 opt1)
        (list 'opt2 opt2)
        (list 'rst  rst))))
```

The **&whole** marker means that the next formal, in this case **appl**, should be bound to the entire macro application. The **&whole** marker should come first, if it is present at all. The "normal" formals **x** and **y** are then bound to the elements of the macro application after the macro name. Since **silly1** has two normal formals, at least two elements must follow the macro name. If an application has no "extra" elements, the other formals of the macro are bound to **nil**.

Below we evaluate **(silly1 a b)**. Observe that **appl** is bound to the entire application, **x** and **y** are bound to **a** and **b** respectively, and all other formals are bound to **nil**.

```
ACL2 >(silly1 a b)
((APPL (SILLY1 A B))
 (X A)
 (Y B)
 (OPT1 NIL)
 (OPT2 NIL)
 (RST NIL))
```

A syntax error would occur if you tried to evaluate **(silly1 a)** because it does not have enough elements to bind the normal formals.

After the normal formals, you may declare some optional formals with **&optional**. **Silly1** has two. They are bound successively to the "extra" elements in the application. After the optional formals you may insert the **&rest** argument to bind to the list of all remaining elements.

```
ACL2 >(silly1 a b c)
((APPL (SILLY1 A B C))
 (X A)
 (Y B)
 (OPT1 C)
 (OPT2 NIL)
 (RST NIL))
ACL2 >(silly1 a b c d e f g)
((APPL (SILLY1 A B C D E F G))
 (X A)
 (Y B)
 (OPT1 C)
 (OPT2 D)
 (RST (E F G)))
```

After the **&rest** marker you may write the **&key** marker. Here is another silly macro to illustrate that.

```
(defmacro silly2 (&whole appl
                  x y
                  &optional opt1 opt2
                  &rest rst
                  &key g h)
  (list 'quote
        (list (list 'appl appl)
              (list 'x    x)
              (list 'y    y)
              (list 'opt1 opt1)
              (list 'opt2 opt2)
              (list 'rst  rst)
              (list 'g    g)
              (list 'h    h))))
```

If the binding process gets as far as the `&key` marker, then the subsequent elements of the macro application must be an alternating list of keywords and elements. In the case of `silly2`, if the keyword `:g` is used, then `g` is bound to the next element, and if the keyword `:h` is used, then `h` is bound to the next element. Otherwise `g` and `h` are `nil` by default. Here are some examples of `silly2`.

```
ACL2 >(silly2 a b c d :h i)
((APPL (SILLY2 A B C D :H I))
 (X A)
 (Y B)
 (OPT1 C)
 (OPT2 D)
 (RST (:H I))
 (G NIL)
 (H I))
ACL2 >(silly2 a b c d :h i :g k)
((APPL (SILLY2 A B C D :H I :G K))
 (X A)
 (Y B)
 (OPT1 C)
 (OPT2 D)
 (RST (:H I :G K))
 (G K)
 (H I))
```

Observe that the order of `:g` and `:h` in the application does not matter. Observe also that the `&rest` argument picks up the entire list of alternating keywords in this application.

If you write `(silly2 a b c d e)` a syntax error occurs, because `e` is not a keyword. If you write `(silly2 a b c d :g)` a syntax error occurs because the keyword portion of the application is not an alternating list

of keywords and elements. If you write `(silly2 a b c d :k k)` a syntax error occurs because `silly2` does not allow any other keys besides `:g` and `:h`. However, if the `&allow-other-keys` marker had been included in the `defmacro`, no error would have been caused and the `&rest` formal could have been used to find the other keys and their values.

If the `&optional` and `&key` markers appear in the `defmacro` then all the optional arguments must be bound before the keywords are processed. That is, while you might think `(silly2 a b :g k)` would bind the two optional formals to `nil` and bind the keyword formal `g` to `k`, it actually binds the two optional formals to `:g` and `k`.

If ampersand markers are used in an ACL2 macro definition, they must appear in the order `&whole`, `&optional`, `&rest`, `&body`, `&key`, and `&allow-other-keys`. Markers may be omitted, of course.

If v is an optional or keyword formal, then we have seen that it is bound to `nil` if no element is provided for it in the macro application. It is sometimes useful to allow a formal to default to some other constant. It may also be useful to know whether the value of such a formal was explicitly provided in the application or was provided by default. Here is a third silly macro that illustrates how this is done for an optional formal. The same convention is implemented for keyword formals.

```
(defmacro silly3 (&optional x (y '23 yflag))
  (list 'quote
        (list (list 'x     x)
              (list 'y     y)
              (list 'yflag yflag))))
```

Here are some examples. Note that `yflag` is `t` or `nil` according to whether a value was explicitly supplied. When a value is not supplied, `y` defaults to 23. In ACL2, the default value must be quoted.

```
ACL2 >(silly3)
((X NIL) (Y 23) (YFLAG NIL))
ACL2 >(silly3 a)
((X A) (Y 23) (YFLAG NIL))
ACL2 >(silly3 a b)
((X A) (Y B) (YFLAG T))
```

5.3 Backquote

It is very common for macros to use `list` to construct the expressions they return. All the macros we have shown use `list`. Recall `ifz` from page 53.

```
(defmacro ifz (a b c)
  (list 'if
        (list 'equal a 0)
        b
        c))
```

Consider the `list` expression in the body of `ifz`. If the formals `a`, `b`, and `c` are bound, respectively, to α, β, and γ, then we might write the value of the body as:

```
(if (equal α 0)
    β
  γ)
```

It would be very convenient, and perhaps clearer, to be able to write the body of `ifz` as a quoted constant

```
'(if (equal a 0)
     b
     c)
```

except that it would be wrong to do so since we want `a` replaced by the value of `a`, `b` replaced by the value of `b`, etc.

Common Lisp and ACL2 provide a notation for such "near constants." It is called *backquote* notation. We recommend that you read about it in [40] if you define a lot of macros. Using backquote notation, `ifz` could be written as follows.

```
(defmacro ifz (a b c)
  `(if (equal ,a 0)
       ,b
       ,c))
```

Backquote notation can be used anywhere an expression is expected. The meaning of the notation is to construct a list object "like" the one following the backquote but with certain components computed by evaluating expressions. Those components are marked by commas. Following each comma is an *expression* whose value is used as the component. Thus, `` `(nth ,(+ 1 j) ,e) `` is another way to write `(list 'nth (+ 1 j) e)`.

If a comma is followed by an at-sign (`@`), the value of the following expression is spliced into the list. Thus, `` `(+ ,(+ 1 j) ,@args k ,m) `` is another way to write `(list* '+ (+ 1 j) (append args (list 'k m)))`. For example, if `j` is 7, `args` is the list `(a b c)`, and `m` is the list `(car x)`, then `` `(+ ,(+ 1 j) ,@args k ,m) `` expands to `(+ 8 a b c k (car x))`.

Here, for example, is a definition of the macro `list`, which expands into a nest of `cons` expressions.

```
(defmacro list (&rest args)
  (cond ((endp args) nil)
        (t `(cons ,(car args)
                  (list ,@(cdr args))))))
```

Contrast this with the definition on page 55.

Sometimes it is helpful to use auxiliary functions when defining macros. For example, the ACL2 source code defines `list` as follows.

```
(defmacro list (&rest args)
  (list-macro args))
```

`List-macro` is the following function.

```
(defun list-macro (lst)
  (declare (xargs :guard t))
  (if (consp lst)
      (cons 'cons
            (cons (car lst)
                  (cons (list-macro (cdr lst)) nil)))
    nil))
```

These two definitions of `list` give the same expansions, but it is interesting to note a difference in how they do so. The earlier definition of `list` causes the one-level expansion on one or more arguments to introduce a subsidiary call of `list`, for example: `(list a b c)` produces `(cons a (list b c))`. The ACL2 definition introduces all the appropriate calls of `cons` in a single one-level expansion, so that in this same example the one-level expansion is `(cons a (cons b (cons c nil)))`.

5.4 An Example

Suppose you are formalizing a state machine. Suppose a state has four components: a program counter, *pcnt*, some registers, *regs*, some flags, *flags*, and a memory, *mem*. Suppose states are constructed by the function `make-state` of four arguments and suppose `pcnt`, `regs`, `flags`, and `mem` are each functions of one argument that return the indicated component of a state.

In describing the state machine it will be common to write expressions that build new states from old. Here, for example, is an expression that builds a new state from `s` by advancing the program counter and modifying nothing else: `(make-state (+ 1 (pcnt s)) (regs s) (flags s) (mem x))`. Here is an expression that sets the program counter to a new location `j` and stores register `i` into memory location `m`: `(make-state j (regs s) (flags s) (put-nth (nth i (regs s)) m (mem s)))`.[1]

It is convenient to define a macro that produces a new state from an old one by "modifying" some components and leaving others fixed. For example, the form `(modify s :pcnt (+ 1 (pcnt s)))` might expand to the first `make-state` expression above and the form `(modify s :pcnt j :mem (put-nth (nth i (regs s)) m (mem s)))` might expand to the second. If states have many components and typical transitions only change a few, this notation would be quite succinct. Furthermore, with this macro the reader need not recall the order in which `make-state` takes its arguments.

[1] The function `put-nth` would have to be defined appropriately.

Here is a macro that implements this.

```
(defmacro modify (s &key
                    (pcnt 'nil pcntp)
                    (regs 'nil regsp)
                    (flags 'nil flagsp)
                    (mem 'nil memp))
  `(make-state ,(if pcntp pcnt `(pcnt ,s))
               ,(if regsp regs `(regs ,s))
               ,(if flagsp flags `(flags ,s))
               ,(if memp mem `(mem ,s))))
```

Each component has a keyword argument. If a new value is not explicitly provided, the macro fills in that slot with the corresponding component of the old state.

Exercise 5.1 *Define the macro* `cadn` *so that*

`(cadn 0 x)` $\Longleftrightarrow_{Syn}$ `(car x)`
`(cadn 1 x)` $\Longleftrightarrow_{Syn}$ `(car (cdr x))`
`(cadn 2 x)` $\Longleftrightarrow_{Syn}$ `(car (cdr (cdr x)))`
etc.

Hint: *This macro does not require ampersand markers. You might consider defining a recursive function first.*

Exercise 5.2 *What expression does* `(cadn t x)` *abbreviate? What expression does* `(cadn i x)` *abbreviate? In many solutions, these expressions expand to* `(car x)` *because the first argument to* `cadn` *is not a natural number. Suppose you wished to write the* `cadn` *macro so that an error is caused if its first argument is not an explicitly given natural. How can you do that?* Hint: *Macros can have guards (see* guard*) and they are evaluated after the macro formals are bound and before the macro body is evaluated.*

Exercise 5.3 *Define the macro* `ffix` *so that* `(ffix` *expr*`)` $\Longleftrightarrow_{Syn}$ `(fix` *expr*`)` *unless expr is an application of* `+`, `*`, *or* `fix`, *in which case* `(ffix` *expr*`)` $\Longleftrightarrow_{Syn}$ *expr*. Hint: *This macro does not require ampersand markers.*

Exercise 5.4 *Suppose you must frequently pair some computed object with itself. You write the macro*

```
(defmacro pairit (x)
  `(cons ,x ,x)).
```

Why is this a poor design for this macro? Hint: *Consider the time it takes to compute the value of* `(pairit (fact 100))`.

Exercise 5.5 *The following* `defmacro` *is intended to provide the abbreviation* `(if-nzp` *x expr*$_1$ *expr*$_2$ *expr*$_3$`)` *so that expr*$_1$ *is evaluated and returned if the value of x is negative, expr*$_2$ *is evaluated and returned if the value of x is zero, and expr*$_3$ *is evaluated and returned otherwise.*

```
(defmacro if-nzp (x expr1 expr2 expr3)
  `(let ((val ,x))
     (cond ((< val 0) ,expr1)
           ((equal val 0) ,expr2)
           (t ,expr3))))
```

The definition cleverly avoids the re-evaluation of its first argument. What is wrong with this expansion scheme? How can you avoid the problem? Hint: *Consider the variables that occur freely in the* `expri`. *ACL2 does not support the Common Lisp "function"* `gensym`. *The best we can do is to insure that certain symbols do not occur freely in other forms. See the macro* `check-vars-not-free` *in the ACL2 sources.*

Exercise 5.6 *Define* (`type-case` x $:type_1$ $expr_1$ ... $:type_n$ $expr_n$) *to evaluate and return* $expr_i$, *where* $type_i$ *is the first type listed describing* x. *The form should return* `nil` *if no type listed describes* x. *The expected* $type_i$ *are* `integer`, `rational`, `string`, `symbol`, `cons`, `Boolean`, *and* `other`. *For example,*

```
(type-case (car x)
  :integer  (+ 1 i)
  :rational (+ 2 i)
  :symbol   (+ 3 i)
  :other    (+ 4 i))
```

should evaluate and return `(+ 1 i)`, *if the value of* `(car x)` *is an integer,* `(+ 2 i)`, *if the value of* `(car x)` *is not an integer but is a rational, etc.*

Exercise 5.7 *Suppose you have a lot of functions that take* `s` *as an argument and return a "new value" of* `s` *as a result. Here we are thinking of* `s` *as a "state" and these functions as state transformers. For example,* `(do-f x s)` *might return a "new" state and* `(do-g x y s)` *might return a "new" state. You will frequently need to create sequential compositions of these functions,* e.g., `(do-f x (do-g i2 j2 (do-g i1 j1 s)))`. *To avoid writing such compositions define the macro* `do-sequentially` *so that*

```
(do-sequentially (do-g i1 j1 s)
                 (do-g i2 j2 s)
                 (do-f x s))
```

evaluates to the same thing as the composition above.

Part III

Reasoning

6

The Logic

In this chapter we present ACL2 as a mathematical logic. Traditionally, one begins by specifying the *syntax* of formulas, identifying some formulas as *axioms*, and precisely specifying some *rules of inference*. The rules of inference are formula transformers; they permit one to produce new formulas from old formulas. A *theorem* is either an axiom or the formula produced by applying some rule of inference to other theorems. A *proof* is a finite tree showing the derivation of a theorem from the axioms. The traditional presentation of a mathematical logic proceeds by assigning a meaning to formulas as follows. A *structure* is a pair consisting of a *domain* and a map assigning functions and predicates on the domain to the function and predicate symbols of the logic. An *interpretation* is a pair consisting of a structure and an *assignment* of elements in the domain to the variable symbols. The *semantics* of a formula under an interpretation is the *truth-value* of the formula under the interpretation. An interpretation is a *model* of a set of formulas if every formula in the set is true in the interpretation. If the rules of inference are *truth preserving* then every theorem is true in every model of the axioms.

This book is about using logic, not about exploring its foundations; hence, we only outline the traditional development and do not give the details. But the tradition is well considered because it justifies the use of formal logic: it is possible to show that a formula is *valid*, *i.e.*, true in all interpretations, by constructing a formal proof. This is wonderful because the set of all interpretations is infinite, but the formal proof is finite. Furthermore, the traditional presentation of a formal logic makes it exceedingly clear what constitutes a proof: it is a finite tree of formulas, each formula being either an axiom or derived from its immediate ancestors by a rule of inference. The rules of inference are also exceedingly clear, *e.g.*, from $(\phi_1 \vee \phi_2)$ and $(\neg\phi_1 \vee \phi_3)$ you may derive $(\phi_2 \vee \phi_3)$. Indeed, formal logic was developed in an attempt to make clear the rules by which people "reason."

But while formal logic may be said to model mathematical reasoning, few people use it to determine validity. Even when dealing with formulas in some formal syntax, all but the most logically trained among us tend to make huge leaps from one formula to the next, often justifying our leaps by appeal to intuition or unarticulated properties of an intended model.

M. Kaufmann et al., *Computer-Aided Reasoning*

When suitably challenged, we can (usually) express these intuitions and properties as formulas and we can break up huge leaps into smaller ones that are easier to follow.

The design philosophy of the ACL2 theorem prover is to aid and abet this style of reasoning while maintaining logical soundness. When successful, the computation done by the theorem prover should guarantee that a formal proof of the formula exists. When a gap is too big for the theorem prover to follow, you must break it into smaller pieces. We discuss the theorem prover in the next part of the book.

As a user of ACL2, you will not deal with the details of the formal logic nearly as often as you will deal with the meaning of formulas. You must be able to express (alleged) truths in the ACL2 language and you must be able to break them down into smaller truths that can be pieced together appropriately. Such an understanding of ACL2 probably comes faster by using ACL2 as a programming language than it does by studying it as a formal logic.

But there is no avoiding the syntactic nature of proofs. To design formulas that can be "pieced together appropriately" you must think in terms of the fundamentals of logic, whether you have studied logic or not. For example, you must understand what it is to *instantiate* a formula by replacing the variables in it with expressions. You must understand how to *substitute equals for equals.* You must understand how to *chain* theorems together so that you can prove that p implies r by proving that p implies q and that q implies r. A little clear talk about basic logic is therefore appropriate.

To reason formally about ACL2 functions we need a way to describe properties of functions. One might think that we need a new formalism to do this, but we do not.

For example, consider the problem of arguing that `mergesort`, the function described on page 62, is "correct." An obvious aspect of its correctness is that its output is ordered. Another aspect is that the function does not insert or delete elements. It is this second property we focus on here. We might start by testing this vague conjecture by running `mergesort` on some examples. The expression `(mergesort '(1 3 2 1))` evaluates to `(1 1 2 3)`. We can check by hand that every element in the input occurs just as many times in the output, and no new elements occur in the output. After a few such tests we might tire of checking this relationship by hand. To ease the burden, we might define a function to check it for us. This function is, of course, `perm`, as discussed on page 59. We could then evaluate expressions that test whether the input to `mergesort` is a `perm` of the output. Here is one such test.

```
ACL2 !>(let ((x '(3 6 2 9 2 8 7 3/2)))
         (perm x (mergesort x)))
T
ACL2 !>
```

The answer is `T`, which is supportive of our conjecture and easier to recognize. No finite number of tests will settle the issue, since there are an infinite number of possible values for `x`. So, after enough testing to convince ourselves of the plausibility of the conjecture, we might try to *prove* `(perm x (mergesort x))`. If we succeed, then we know that `x` and `(mergesort x)` are related by `perm`, regardless of the value of `x`. Note that we used another defined function, `perm`, to characterize an important property of `mergesort`. We can similarly handle the property of being ordered.

Of course, we might wish to investigate whether our definition of `perm` is "correct," in the sense of having the properties we expect of permutations. Is it reflexive? Symmetric? Transitive? Each of these can be tested and/or proved. For example, to test symmetry, we might evaluate the expression `(if (perm x y) (perm y x) t)`, which we can also write as `(implies (perm x y) (perm y x))`, for various values of `x` and `y`. We expect the answer will always be `t`. We could also try to prove it.

Up until now, we have presented function definitions as rules for computing the values of functions on specific inputs. In this chapter, we show that function definitions can be thought of as axioms, which can be used to reason about the functions.

Since defining a function introduces a new axiom, ACL2 enforces certain restrictions on function definitions to insure soundness.[1] One of these restrictions is that functions must be *total*, *i.e.*, functions must terminate on every input. Termination arguments are based on the notion of *well-founded* structures. A well-founded structure is a pair $\langle W, \prec \rangle$ consisting of a set W and a relation $\prec$, such that for no infinite sequence $\langle x_i \rangle$ do we have $x_{i+1} \prec x_i$ and $x_i \in W$ for all i. (You can think of this condition as: there are no infinite sequences of "decreasing" W elements.) The ACL2 *ordinals* (with their ordering relation) constitute a well-founded structure and are used to prove termination of defined functions. The ordinals have enough structure to allow proofs of termination based on many of the standard well-founded structures used for this purpose (*e.g.*, the lexicographic ordering on n-tuples of natural numbers).

Our presentation of the ACL2 logic is organized as follows. In the next section, we present a quantifier-free subset of the ACL2 logic. ACL2 allows quantification, but this topic is not discussed until Section 6.6.2 (see `defun-sk`). In Section 6.2, we give axioms for some of the primitive ACL2 functions. We describe the ordinals in Section 6.3. In Section 6.4, we give a detailed description of the restrictions ACL2 enforces on function definitions.

Finally, we explain induction as a rule of inference. Induction is such a pervasive technique in building and reasoning about computational systems that we expect you are already familiar with it. In Section 6.5 we discuss

[1] A stronger condition holds, namely that each axiom gives a *conservative extension* of the existing set of axioms. This means that no new theorems can be proved about the existing set of function symbols when the new axiom is added.

mathematical induction and a very powerful type of induction called *well-founded induction.* Most importantly, we describe the ACL2 Induction Principle.

Formal Reasoning and Meta-Reasoning

The focus of this book is on formal reasoning: an activity which, here, consists of proving formulas in the ACL2 logic, using methods derivable from the rules of inference. This includes using the ACL2 theorem prover, which we regard as a powerful derived rule of inference.

Another type of reasoning is also employed in this book and we want to make a distinction to avoid any misunderstanding. When describing the ACL2 logic, it helps to step back and consider the logic itself as a mathematical object of study. For example, we introduce the principle of well-founded induction and prove it correct. You are asked to prove similar results. This type of reasoning is called *meta-reasoning* because it is not about statements in the ACL2 logic; it is about the logic itself. Meta-reasoning is not required to define the logic nor to define the notion of formal proof. We stress that formal reasoning is an entirely syntactic process, meaning that a computer program can be written to determine if a purported "proof" really is a proof. The sole purpose of meta-reasoning here is to shed light on the logic so that concepts such as induction and the ordinals do not seem like magic.

6.1 Quantifier-Free First-Order Logic

ACL2 is a first-order logic, but we begin by describing a quantifier-free subset of that logic. In logic the word "term" is often used synonymously with the ACL2 notion of "expression," *i.e.*, a *term* is a variable symbol, a constant symbol, a quoted constant, or an n-ary function expression applied to n terms. The *logical operators* are conjunction, $\wedge$, read "and"; disjunction, $\vee$, read "or"; negation, $\neg$, read "not"; implication, $\rightarrow$, read "implies"; and propositional equivalence, $\leftrightarrow$, read "iff" or "if and only if." *Atomic formulas* are equalities $\tau_1 = \tau_2$, between expressions τ_i. *Formulas* are built recursively starting from atomic formulas using the logical operators. We use parentheses and brackets to disambiguate formulas. An example formula is thus

```
((integerp x) = t) ∧ ((integerp y) = t)
  → ((integerp (+ x y)) = t).
```

"$\phi_1 \wedge \phi_2$" is an abbreviation of "$\neg[(\neg\phi_1) \vee (\neg\phi_2)]$". Implication and equivalence are handled similarly. "$(\tau_1 \neq \tau_2)$" is an abbreviation of "$\neg(\tau_1 = \tau_2)$".

Some of the axioms and rules of inference of the logic are taken essentially from Shoenfield [37]. We have one

- *Propositional Axiom Schema*: $(\neg\phi \vee \phi)$,

meaning we have such an axiom for every formula ϕ, and four rules of inference:

- *Expansion*: derive $(\phi_1 \vee \phi_2)$ from ϕ_2;
- *Contraction*: derive ϕ from $(\phi \vee \phi)$;
- *Associativity*: derive $((\phi_1 \vee \phi_2) \vee \phi_3)$ from $(\phi_1 \vee (\phi_2 \vee \phi_3))$; and
- *Cut*: derive $(\phi_2 \vee \phi_3)$ from $(\phi_1 \vee \phi_2)$ and $(\neg\phi_1 \vee \phi_3)$.

An easily derived rule of inference is *Modus Ponens* which allows us to derive ϕ_2 from ϕ_1 and $\phi_1 \rightarrow \phi_2$.

Equality is introduced with the following.

- *Reflexivity Axiom*: $(\mathtt{x} = \mathtt{x})$
- *Equality Axiom Schema for Functions*: For every function symbol f of arity n we add the axiom

$$[(\mathtt{x1} = \mathtt{y1}) \wedge \cdots \wedge (\mathtt{x}n = \mathtt{y}n)] \rightarrow [(f\ \mathtt{x1} \ldots \mathtt{x}n) = (f\ \mathtt{y1} \ldots \mathtt{y}n)]$$

- *Equality Axiom*:

$$[(\mathtt{x1} = \mathtt{y1}) \wedge (\mathtt{x2} = \mathtt{y2})] \rightarrow [(\mathtt{x1} = \mathtt{x2}) \rightarrow (\mathtt{y1} = \mathtt{y2})]$$

Finally, we have the fundamental Rule of Inference:

- *Instantiation*: Derive ϕ/σ from ϕ,

where σ is any substitution mapping variable symbols to terms and "ϕ/σ" denotes the application of that substitution to formula ϕ. If a substitution σ maps variable v to term τ we say v is a *target variable* of σ and its *image* is τ. The application of a substitution σ to formula ϕ uniformly replaces, in ϕ, every free occurrence of a target variable by its image. In this document we write substitutions like this $[v_1 \lhd \tau_1; \ldots; v_n \lhd \tau_n]$.[2]

If, for example, `(car (cons x y))` = `x` is a theorem then, by instantiation, so is `(car (cons (+ u v) nil))` = `(+ u v)`. The substitution in question, [`x` ◁ `(+ u v)`; `y` ◁ `nil`], replaces the target variable `x` by the image term `(+ u v)` and the target variable `y` by the image term `nil`.

A *formal proof* is a finite tree of formulas, each of which is either an axiom or is derived from its immediate ancestors in the tree by one of the

[2]However, the user of the mechanized ACL2 writes this as $((v_1\ \tau_1)\ \ldots\ (v_n\ \tau_n))$.

rules of inference above. A *theorem* is any formula in a proof, but most especially the root of the tree.

The foregoing formalization of first-order logic is completely standard. Classic results of mathematical logic apply and allow us to construct proofs with much larger steps.

For example, every *propositional tautology* has a proof in the above sense. A tautology can be recognized by using a truth table to test the value of the formula on each of a finite number of assignments. Rather than exhibit the formal proof, we can just exhibit the truth table and appeal to this well-known result.

Similarly, we may prove theorems by *case analysis*. That is, we can prove ϕ by proving it under each of a finite and exhaustive set of cases $\alpha_1, \ldots, \alpha_k$, by proving, for each $1 \leq i \leq k$, $\alpha_i \rightarrow \phi$ and proving $(\alpha_1 \vee \cdots \vee \alpha_k)$.

We may use *substitution of equals for equals*. That is, if $\tau_1 = \tau_2$ has been proved or is given, then in any formula we may replace one or more occurrences of τ_1 by τ_2.

We may use the *deduction law* so that we may assume the hypotheses of a formula as "givens" while trying to prove the conclusion. For example, suppose we have proved $\phi_1 \rightarrow \phi_2$ and $\phi_2 \rightarrow \phi_3$, which we refer to as lemmas 1 and 2 below. Then we might offer the following proof of $\phi_1 \rightarrow \phi_3$:

Proof

ϕ_1	{Given}
$\phi_1 \rightarrow \phi_2$	{lemma 1}
ϕ_2	{Modus Ponens}
$\phi_2 \rightarrow \phi_3$	{lemma 2}
ϕ_3	{Modus Ponens}

□

When using the deduction law here care must be taken with the variables in the "givens." These variables cannot be instantiated. The rule of instantiation allows you to instantiate the variables of a theorem, not of a hypothesis.

Consider the formula $(\mathtt{m} = \mathtt{1}) \rightarrow (\mathtt{2} = \mathtt{1})$. This formula is clearly invalid. If `m` is `1`, then the truth-value of the formula is false. But we can "prove" it using the bogus rule of "instantiation of the hypothesis."

"Proof"

$\mathtt{m} = \mathtt{1}$	{Given}
$\mathtt{2} = \mathtt{1}$	{"Instantiation" of line 1}

□

Because this section is a review of basic logic, we have taken certain liberties, *e.g.*, proof outlines are presented as sequences, instead of trees, of formulas, and we have merely hinted at deep results, such as the derivability of the deduction law. Readers interested in the classic study of formal systems such as ours should see [37] or any other good textbook on mathematical logic.

6.2 The Axioms of ACL2

As mentioned at the beginning of the chapter, the primitive functions of ACL2 can be thought of as being defined by axioms. We do not give a complete account of the axioms of ACL2. This would be too great a digression and such an account is given elsewhere (see [25]). Our purpose is to show you how the meaning of functions can be captured by axioms. If you have read the previous chapters on programming, then after reading this chapter, you should be able to axiomatize the other primitive functions (*e.g.*, functions dealing with strings, characters, and numbers; see page 34).

We now enumerate some of the axioms regarding the primitive functions of ACL2. We start with the following.

- `x` = `nil` → `(if x y z)` = `z`
- `x` ≠ `nil` → `(if x y z)` = `y`
- `x` = `y` → `(equal x y)` = `t`
- `x` ≠ `y` → `(equal x y)` = `nil`

Here, `t` and `nil` are constant symbols; `x`, `y`, and `z` are variable symbols; `if` is a function symbol of arity three and `equal` is a function symbol of arity two.

We introduce the abbreviation `(and` p_1 $p_2 \ldots$`)` $\Longleftrightarrow_{Syn}$ `(if` p_1 `(and` $p_2 \ldots$`) nil)`, where `(and` p_1`)` $\Longleftrightarrow_{Syn}$ p_1. Analogously, `(or` p_1 $p_2 \ldots$`)` $\Longleftrightarrow_{Syn}$ `(if` p_1 `t (or` $p_2 \ldots$`))`, where `(or` p_1`)` $\Longleftrightarrow_{Syn}$ p_1. (See page 37 for a definition of $\Longleftrightarrow_{Syn}$.)

We define the "propositional functions" so that

- `(not p)` = `(if p nil t)`
- `(implies p q)` = `(if p (if q t nil) t)`
- `(iff p q)` = `(if p (if q t nil) (if q nil t))`

We add the following axiom.

- `(equal (car (cons x y)) x)` ≠ `nil`

Exercise 6.1 *Use the axioms and rules of inference to prove each of the following.*

- `t` ≠ `nil`
- `(equal x y)` = `t` ↔ `x` = `y`
- `(equal x y)` = `nil` ↔ `x` ≠ `y`
- `(car (cons x y))` = `x`

Notice that there is a correspondence between formulas and terms: formulas can be expressed as terms whose outmost function symbol is `equal`, `and`, `or`, `not`, `implies`, or `iff`. Such a term is true (in the sense that its value is not equal to `nil`) precisely when the corresponding formula is true (in the traditional sense of first-order logic). For example, `(or p (not p))` is true (in the term sense) precisely when $((\texttt{p} \neq \texttt{nil}) \vee \neg(\texttt{p} \neq \texttt{nil}))$ is a true formula.

Exercise 6.2 *Prove each of the following.*

- $\texttt{(not p)} \neq \texttt{nil} \quad \leftrightarrow \quad \neg\, (\texttt{p} \neq \texttt{nil})$
- $\texttt{(or p q)} \neq \texttt{nil} \quad \leftrightarrow \quad (\texttt{p} \neq \texttt{nil}) \vee (\texttt{q} \neq \texttt{nil})$
- $\texttt{(and p q)} \neq \texttt{nil} \quad \leftrightarrow \quad (\texttt{p} \neq \texttt{nil}) \wedge (\texttt{q} \neq \texttt{nil})$
- $\texttt{(implies p q)} \neq \texttt{nil} \quad \leftrightarrow \quad (\texttt{p} \neq \texttt{nil}) \rightarrow (\texttt{q} \neq \texttt{nil})$

Thus we may use terms rather than formulas. If the term τ is used where a formula is expected we mean instead to use the formula $\tau \neq \texttt{nil}$. Thus, for example, we might refer to the "theorem" `(implies (and p q) p)`. As we will see in the next part, when dealing with the theorem prover, we use terms rather than formulas.

We now state the remaining axioms for the familiar ACL2 functions.

- `(cdr (cons x y)) = y`
- `(consp (cons x y)) = t`

We can similarly axiomatize four other data types: the complex rationals (with the rationals and integers as subsets), the characters, the strings, and the symbols (with packages). These data types are pairwise disjoint. For example we have

- `(implies (consp x) (not (symbolp x)))`

which together with

- `(symbolp nil)`

implies that `(consp nil) = nil`.

The axioms for arithmetic are essentially the usual ordered field axioms for constants `0` and `1` and function symbols `+`, `-`, `*`, `/`, and `<`. Since the language is not typed, terms such as `(+ nil 1)` are legal. Our axioms "complete" the primitive functions. For example, when an argument to `+` is not numeric, that argument is treated as though it were `0`. You can find all the axioms of ACL2 by searching for "`defaxiom`" in the ACL2 source file `axioms.lisp`.

Our axioms are consistent with those of Common Lisp [40] modulo the notion of *guard* (see [24]), which limits our claims of correspondence with Common Lisp. *Guard verification* provides mechanized support for proving that functions are used in compliance with Common Lisp. See guard.

6.3 The Ordinals

The principles of recursive definition and of induction both rely on the notion of a well-founded structure. ACL2 has one well-founded structure that is used in the two principles, namely, the ACL2 ordinals. We examine the ACL2 ordinals in detail in this section, but we start with the (usual, set-theoretic) ordinals.

Consider the natural numbers, *i.e.*, $0, 1, 2, \ldots$. Notice that the natural numbers are linearly ordered: $0 < 1 < 2 < \cdots$, *i.e.*, given distinct numbers x and y, exactly one is smaller than the other. In addition, the set of natural numbers is (countably) infinite and under this ordering, it has no biggest element and it is well-founded. These are basic facts about the natural numbers.

The ordinals are an extension of the natural numbers. The idea is to add new "numbers" after the naturals. For example, if we add ω, we get $0, 1, 2, \ldots, \omega$, where $0 < 1 < 2 < \cdots < \omega$. Notice that this new set extends the naturals and is countable. In addition, under this ordering it has a biggest element (ω), is linearly ordered, and well-founded. The ordinals are what you get if you keep adding elements. The first few ordinals are: $0, 1, \ldots, \omega, \omega + 1, \ldots, \omega \times 2$. Notice that (the ordinals less than) $\omega \times 2$ is what you get if you put two copies of (the ordinals less than) ω next to each other, *i.e.*, $0, 1, \ldots, 0, 1, \ldots$ (and then rename the elements in the second copy). Continuing to add "numbers," we get $\omega, \omega \times 2, \omega \times 3, \ldots, \omega^2$. Now we have introduced ordinal exponentiation. Note that ω^2 is what you get if you put ω copies of ω, one after the other (and rename elements). We will pick up where we left off, but will now start extending the ordinals at a faster rate! $\omega^2, \omega^2 + 1, \ldots, \omega^2 + \omega, \omega^2 + \omega + 1, \ldots, \omega^2 + (\omega \times 2), \omega^2 + (\omega \times 2) + 1, \ldots, \omega^2 + (\omega \times 3), \ldots, \omega^2 + (\omega \times 4), \ldots, \omega^2 \times 2, \ldots, \omega^2 \times 3, \ldots, \omega^3, \omega^4, \ldots, \omega^\omega, \ldots, \omega^{(\omega^\omega)}, \ldots, \omega^{(\omega^{(\omega^\omega)})}, \ldots, \epsilon_0$. Even though this is only a very small initial segment of the ordinals, this is all you need to know because the ACL2 ordinals correspond to the set of ordinals less than ϵ_0. If you find the ordinals interesting, then prove that the set of ordinals less than ϵ_0 is countable and is well-founded under the appropriate relation. We also invite you to contemplate: How long can we keep extending the ordinals? The interested reader is encouraged to consult a good text on set theory (*e.g.*, [17] or [14]) for a discussion of such issues.

As pointed out above, the ACL2 ordinals are essentially the ordinals less than ϵ_0. We define a sequence of ACL2 objects in one-to-one correspondence with these ordinals. The ACL2 objects in question are just certain binary trees of natural numbers but we refer to them as "the ordinals" in the context of ACL2. Our formalization of the (ACL2) ordinals follows Goodstein [15]. Table 6.1 shows the correspondence between the ordinals and certain ACL2 list constants. We axiomatize `(e0-ordinalp x)` so that it returns `t` or `nil` according to whether `x` is a object representing an ACL2 ordinal. Note that the natural numbers are ordinals. The elements of a list

structured ordinal should be thought of as powers of ω. The powers must be listed in non-increasing order. Here is the equation defining `e0-ordinalp`.

Axiom.

```
(e0-ordinalp x)
  =
(if (consp x)
    (and (e0-ordinalp (car x))
         (not (equal (car x) 0))
         (e0-ordinalp (cdr x))
         (or (atom (cdr x))
             (not (e0-ord-< (car x) (cadr x)))))
  (and (integerp x) (>= x 0)))
```

We axiomatize `(e0-ord-< x y)` so that if `x` and `y` are two such ordinals, then the expression is `t` or `nil` according to whether `x` represents a smaller ordinal than `y`. Integer ordinals are compared with the `<` operator of arithmetic. List valued ordinals are compared by comparing their respective first elements. Only if their first elements are equal are the remaining elements compared. It is a theorem of set theory that `e0-ord-<` is well-founded on the ordinals; that theorem is not formalized in the ACL2 logic, but is essential to the soundness of the logic.[3]

Axiom.

```
(e0-ord-< x y)
  =
(if (consp x)
    (if (consp y)
        (or (e0-ord-< (car x) (car y))
            (and (equal (car x) (car y))
                 (e0-ord-< (cdr x) (cdr y))))
      nil)
  (if (consp y)
      t
    (< (if (rationalp x) x 0)
       (if (rationalp y) y 0))))
```

Finally, we axiomatize `(acl2-count x)` to return a natural number that measures the "size" of the object `x`. The size of a cons is one more than the sum of the sizes of the car and cdr. The size of an integer is its absolute value. The size of a non-integer rational is the sum of the sizes of the numerator and denominator. The size of a complex number is one more than sum of the sizes of its real and imaginary parts. The size of a string is its length. The size of a character or symbol is 0.

[3] In the definition of `e0-ord-<`, atoms are coerced to rationals so that the guards of the function can be verified. The Common Lisp Language does not define `<` on non-numeric arguments.

Ordinal	ACL2 object
0	0
1	1
2	2
3	3
...	...
ω	(1 . 0)
$\omega + 1$	(1 . 1)
$\omega + 2$	(1 . 2)
...	...
$\omega \times 2 = \omega + \omega$	(1 1 . 0)
$(\omega \times 2) + 1$	(1 1 . 1)
...	...
$\omega \times 3 = \omega + (\omega \times 2)$	(1 1 1 . 0)
$(\omega \times 3) + 1$	(1 1 1 . 1)
...	...
ω^2	(2 . 0)
...	...
$\omega^2 + (\omega \times 4) + 3$	(2 1 1 1 1 . 3)
...	...
ω^3	(3 . 0)
...	...
ω^ω	((1 . 0) . 0)
...	...
$\omega^\omega + \omega^{99} + (\omega \times 4) + 3$	((1 . 0) 99 1 1 1 1 . 3)
...	...
$\omega^{(\omega^2)}$	((2 . 0) . 0)
...	...
$\omega^{(\omega^\omega)}$	(((1 . 0) . 0) . 0)
...	...

Table 6.1: Some Ordinals in ACL2

Exercise 6.3 *What is the (ACL2 representation of the) first ordinal larger than all natural numbers?*

Exercise 6.4 *Which of the following represent ordinals?*

1. `(2 1 . 27)`
2. `(2 1 1 . 27)`
3. `(2 1 0 . 27)`
4. `(1 2 2 . 27)`

Exercise 6.5 *There are an infinite number of ordinals less than ω. This may encourage hope that an infinite decreasing sequence $\langle x_i \rangle$ can be constructed, where x_0 is ω and x_1 is one of the infinitely many ordinals less than it. Try to construct such a sequence by choosing some ordinal less than ω for x_1. What is the maximum length of the resulting sequence? Try to construct an infinite descending sequence from $\omega \times 2$.*

Exercise 6.6 *Suppose* `i1`, `i2`, `j1`, *and* `j2` *are natural numbers.*

1. *Is the value of* `(cons i1 i2)` *an ordinal?*
2. *Is the value of* `(cons (+ 1 i1) i2)` *an ordinal?*
3. *Under what conditions is the value of* `(cons (+ 1 i1) i2)` *less than* `(cons (+ 1 j1) j2)` *in the sense of* `e0-ord-<`*?*

Exercise 6.7 *The ACL2 ordinals with relation* `e0-ord-<` *form a well-founded structure. Exhibit a set, S, of ACL2 objects such that the structure $\langle S,$* `e0-ord-<`$\rangle$ *is not well-founded.*

Exercise 6.8 *Show that for any positive integer n, the set of n-tuples of natural numbers, ordered lexicographically, can be embedded into the ACL2 ordinals in an order-preserving way. The lexicographic ordering, $<_n$, on n-tuples of naturals is defined as follows: $<_1$ is $<$, the less than relation on the natural numbers. For $n > 1$, $\langle x_1, x_2, \ldots, x_n \rangle <_n \langle y_1, y_2, \ldots, y_n \rangle$ iff $x_1 <_1 y_1$ or $(x_1 = y_1$ and $\langle x_2, \ldots, x_n \rangle <_{n-1} \langle y_2, \ldots, y_n \rangle)$.*

Exercise 6.9 *Exhibit a bijection (*i.e., *a 1-1, onto map) between the ACL2 ordinals and the natural numbers.*

Exercise 6.10 *Exhibit a bijection between the ACL2 ordinals and ϵ_0 (*i.e., *the set-theoretic ordinals less than ϵ_0) which preserves order. (Note: You may wish to skip this exercise if the set-theoretic ordinals are new to you.)*

6.4 The Definitional Principle

The Definitional Principle allows you to add axioms defining new function symbols, under conditions that insure that the soundness of the logic is preserved. In fact, the addition of new definitions should not allow us to prove any new theorems about the existing function symbols. Logicians would say that the definitions provide *conservative extensions* of the given set of axioms.

It is not at first obvious that soundness can be imperiled by "bad definitions." If f is a function symbol mentioned nowhere else in the logic and *body* is a term, how can

(*f* `x`) = *body*

introduce unsoundness? One example of a "bad definition" is

(*f* `x`) = `y`.

If this were added as an axiom then we could prove (*f* `1`) = `t`, by instantiating the definition (using the substitution [`x` ◁ `1`; `y` ◁ `t`]). But we could also similarly prove (*f* `1`) = `nil`. Then, by the transitivity of equality, we could deduce that `t` = `nil`, contradicting the theorem that `t` and `nil` are distinct. To avoid this problem ACL2 prohibits "global variables," such as `y`, in definitions.

Another "bad definition" is

(*f* `x`) = (`+ 1` (*f* `x`)).

This "definition" contains no global variables. As a programmer you might recognize this definition as a nonterminating recursion. But as an axiom it causes unsoundness because it contradicts a theorem of arithmetic, namely, `i` $\neq$ (`+ 1 i`). Not every nonterminating recursive equation introduces unsoundness, but some do. Therefore, ACL2 prohibits nonterminating recursions. It can be shown [5] that if a recursive equation can be shown to terminate then there exists a function satisfying the equation.

The definitional principle is packaged so that new definitions are introduced with the **defun** form of Common Lisp. But such a form is "admissible" only if certain restrictions are met. If admissible, a new axiom is added. Here is the statement of the principle.

- *Definitional Principle*: The definition

  (**defun** f (x_1 ... x_n) *body*)

  is *admissible* provided:

  - f is a new function symbol, *i.e.*, there are no other axioms about it;
  - the x_i are distinct variable symbols;

- ◊ *body* is a term, possibly using f recursively as a function symbol, mentioning no variables freely other than the x_i; and
- ◊ certain "measure conjectures," described below, can be proved.

If admissible, the logical effect of the definition is to add a new axiom:

- ◊ *Definitional Axiom for f*:
  $(f\ x_1\ \ldots\ x_n) = body$.

The *measure conjectures*, if theorems, establish that a certain ordinal measure of the x_i decreases in each recursive call of f in *body*. In practice, the user of the definitional principle identifies the measure term, m, and supplies it, perhaps implicitly, with the proposed definition. The measure conjectures are defined syntactically. If there are k recursive calls of f in *body* then there are $k + 1$ measure conjectures. The first is `(e0-ordinalp` m`)`, which establishes that the measure yields an ordinal. Then, for each call of f, $(f\ a_1\ \ldots\ a_n)$, in *body*, there is a conjecture of the form `(implies (and` $\alpha_1\ \ldots\ \alpha_j$`) (e0-ord-<` m/σ m`))`, where the α_i are the expressions "ruling" the recursive call in question (*i.e.*, the α_i are the conditions under which the call is made; see the definition below) and σ is the substitution $[\ldots; x_i \lhd a_i; \ldots]$, mapping the formals, x_i, to the corresponding actuals, a_i, in the call in question. Each such conjecture establishes that the measure decreases in the given recursive call.

The set of *rulers*[4] of an occurrence of some subterm in a term x is defined as follows. If x is not an `if` expression, the set of rulers is empty. If x is `(if` $a\ b\ c$`)`, then the set of rulers is the set of rulers of the occurrence in a, b, or c (depending on whether the occurrence is in a, b, or c) together with a, if the occurrence is in b, and `(not` a`)`, if the occurrence is in c.

Consider the term `(if p (if (q (if x y z)) u v) w)`. Here is a table showing the rulers of the occurrences of certain variables.

Occurrence	Rulers
`x`	`p`
`y`	`p`
`z`	`p`
`u`	`p and (q (if x y z))`
`v`	`p and (not (q (if x y z)))`
`w`	`(not p)`

We do not consider `x` a ruler of `y`. Nor do we consider `(not x)` a ruler of `z`.[5]

[4] It is not crucial that you remember this definition when using the theorem prover. The reason is that when you define a recursive function, the theorem prover will print out the measure conjectures that it must prove, so for any given definition you will see what is required.

[5] The reason has to do with the induction scheme we prefer to generate from such definitions.

We now consider an example. The definition of `sum-list` (page 47) is admitted using the measure term `(acl2-count x)`. The measure conjectures are

```
(e0-ordinalp (acl2-count x)) , and
(implies (consp x)
         (e0-ord-< (acl2-count (cdr x)) (acl2-count x))) .
```

These conjectures are provable and hence `sum-list` is admitted, adding the following axiom.

♦ *Definitional Axiom for* `sum-list`:

```
(sum-list x) =
(if (endp x) 0 (+ (car x) (sum-list (cdr x))))
```

The measure must always return an ordinal, not just when the recursive branches are taken, and the measure must decrease on all possible inputs, not just those within the domain specified by a guard declaration (if any). Guards are extra-logical. They are not involved in either the admission of a definition or the resulting axiom. See guard.

Exercise 6.11 *What measure can be used to admit the following definition?*

```
(defun upto (i max)
  (if (and (integerp i)
           (integerp max)
           (<= i max))
      (+ 1 (upto (+ 1 i) max))
    0))
```

What is the value of `(upto 7 12)`*?*

Exercise 6.12 *What measure can be used to admit the following definition?*

```
(defun g (i j)
  (if (zp i)
      j
    (if (zp j)
        i
      (if (< i j)
          (g i (- j i))
        (g (- i j) j)))))
```

What is `(g 18 45)`*? What is* `(g 7 9)`*?*

Exercise 6.13 *What measure can be used to admit the following definition?*

```
(defun mlen (x y)
  (if (or (consp x) (consp y))
      (+ 1 (mlen (cdr x) (cdr y)))
    0))
```

What is `(mlen '(a b c) '(a b c d e))`*?*

Exercise 6.14 *What measure can be used to admit the following definition?*

```
(defun flen (x)
  (if (equal x nil)
      0
    (+ 1 (flen (cdr x)))))
```

What is `(flen '(a b c))`*? What is* `(flen '(a b c . 7))`*?*

Exercise 6.15 *What measure can be used to admit the following version of Ackermann's function?*

```
(defun ack (x y)
   (if (zp x)
       1
     (if (zp y)
         (if (equal x 1) 2 (+ x 2))
       (ack (ack (1- x) y) (1- y)))))
```

Exercise 6.16 *Which of the following are admissible? If inadmissible, would unsoundness result from adding the "defining equation" as an axiom?*

1. `(defun f (x) (f x))`
2. `(defun f (x) (not (f x)))`
3. `(defun f (x) (if (f x) t t))`
4. `(defun f (x) (if (f x) t nil))`
5. `(defun f (x) (if (f x) nil t))`
6. `(defun f (x) (if (zp x) 0 (f (- x 1))))`
7. `(defun f (x) (if (zp x) 0 (f (f (- x 1)))))`
8. `(defun f (x) (if (zp x) 0 (+ 1 (f (f (- x 1))))))`
9. `(defun f (x) (if (zp x) 0 (+ 2 (f (f (- x 1))))))`
10.
```
(defun f (x)
  (if (integerp x) (* x (f (+ x 1)) (f (- x 1))) 0))
```

6.5 The Induction Principle

Before we present ACL2's induction principle we discuss induction in simpler logical settings.

In traditional first-order logic, the induction principle is often stated as follows: "To prove $\forall n \phi(n)$ it is sufficient to prove (a) $\phi(0)$ and (b) $\forall n(\phi(n) \rightarrow \phi(n+1))$." Here, the variable n is understood to range only over the natural numbers, as it does when one is formalizing Peano arithmetic. The proof obligation labeled (a) is called the *base case*; proof obligation (b) is called the *induction step*. The hypothesis of the implication in (b) is called the *induction hypothesis* and its conclusion is called the *induction conclusion*. If free variables are understood to be implicitly universally quantified—the standard convention in most logics—then the principle can be stated still more simply as "$\phi(n)$ may be proved by proving (a) $\phi(0)$ and (b) $\phi(n) \rightarrow \phi(n+1)$." This is probably the most familiar formalization of the principle of mathematical induction. Note that a formalized induction principle is a syntactic rule that describes how to generate a finite number of formulas, in this case just two, so as to prove a given formula for an infinite number of cases.

Now consider a logic that includes formalizations of objects other than just the natural numbers. Then to prove $\phi(n)$ for all n, we would have to include another case, because we have to prove $\phi(n)$ when n is not a natural number. We might also alter (b) so that we explicitly assume n is a natural number. If the other kinds of objects are atomic in character, the additional case would look like another base case. But if the other kinds of objects are inductive in character, *e.g.*, negative integers, the additional case might be another induction step with its own induction hypothesis. This suggests one of the complications of the ACL2 induction principle: we must have a principle that allows us to prove something about all numbers (including the rationals and complex numbers), characters, strings, symbols, conses, and anything else that might be in the ACL2 universe.[6]

Rather than pursue the above scheme now, dealing with different types of objects, let us explore induction on the natural numbers a bit more.

Consider how the traditional induction principle handles a doubly quantified formula such as $\forall n \forall m\, \phi(n, m)$. The base case is $\forall m\, \phi(0, m)$, which might be written $\phi(0, m)$, if top-level universal quantifiers are dropped. The induction step is $\forall n\, [(\forall m\, \phi(n, m)) \rightarrow (\forall m\, \phi(n+1, m))]$. The outermost universal quantifier can be dropped. In addition, the universal quantifier in the conclusion can be lifted out and dropped, giving $(\forall m\, \phi(n, m)) \rightarrow \phi(n+1, m)$. Thus, while proving $\phi(n+1, m)$ we may assume $\forall m\, \phi(n, m)$. For example, if in the proof of $\phi(n+1, m)$ we need $\phi(n, m+1)$ we can obtain it by instantiation of the universally quantified induction hypothesis.

[6]The ACL2 universe is not closed. There are no axioms that say an object must have one of the five types listed.

Newcomers to logic sometimes mistakenly write the induction step as $\phi(n,m) \rightarrow \phi(n+1,m)$, perhaps because the universal quantifier in the hypothesis looks like it is at the top-level. Newcomers often then act as though the hypothesis were actually $\forall m\, \phi(n,m)$. That is, they instantiate the m in their hypothesis, *e.g.*, to obtain $\phi(n,m+1)$. This is a classic case of writing one thing and meaning another. In fact, it is unsound to prove $\phi(n,m) \rightarrow \phi(n+1,m)$ by instantiating the m in the hypothesis. For example, the technique can be used to prove the manifestly invalid $m = 2 \rightarrow 1 = 2$. A correct mechanized proof checker or theorem prover would not permit the instantiation of m here!

To make matters perhaps more confusing, since the newcomer's hypothesis $\phi(n,m)$ is weaker than what induction allows, $\forall m\, \phi(n,m)$, the newcomer's statement is stronger. Put another way, if the newcomer's statement can really be proved, then the actual induction step can also be proved. Sometimes we really do derive $\phi(n+1,m)$ from $\phi(n,m)$. But not by instantiating m!

Since it is sometimes necessary to instantiate m in the induction hypothesis, how does ACL2 deal with this, in the absence of explicit quantifiers? The answer is: appropriate instances of m must be chosen when the induction principle is applied. That is, the induction principle allows the construction of an induction step like $\phi(n,m+1) \rightarrow \phi(n+1,m)$. Indeed, since in the full first-order case we know we can assume the hypothesis for all m, ACL2's principle allows an arbitrary number of inductive instances, *e.g.*, $\phi(n,m) \wedge \phi(n,m+1) \wedge \phi(n,m^2) \rightarrow \phi(n+1,m)$.

Again, rather than pursue this elaboration further now, we return to the simple statement of the traditional scheme and consider another twist.

The classic induction hypothesis is "about" n and the induction conclusion is "about" $n+1$. This can be equivalently restated so that the induction hypothesis is about $n-1$ and the induction conclusion is about n, provided we add a new hypothesis requiring that $n \neq 0$. The result is the equivalent alternative induction principle: "$\phi(n)$ may be proved by proving (a) $\phi(0)$ and (b) $(n \neq 0 \wedge \phi(n-1)) \rightarrow \phi(n)$." The classic principle shows how to "construct" a bigger natural from n, while the alternate principle shows how to "decompose" n into a smaller natural.

In ACL2 we prefer the second style, primarily because it is easy to derive inductions in this style from function definitions that recursively decompose their arguments. This decomposition may involve other operations besides subtracting 1. For example, a recursively defined function f might compute its value on n by computing its value on both $n-1$ and $\lfloor n/2 \rfloor$. To prove some $\phi(n)$ involving this function, we might wish to assume two inductive hypotheses, rephrasing (b) as "$(n \neq 0 \wedge \phi(n-1) \wedge \phi(\lfloor n/2 \rfloor)) \rightarrow \phi(n)$" in strict analogy with the recursive definition of f.

Let us call $n-1$ and $\lfloor n/2 \rfloor$ "components" of n. Our induction principle should permit us to assume an inductive hypothesis about as many components of n as we wish, provided each is a smaller natural number than

n. Indeed, we might like "course of values" induction, also called "strong" induction, in which we get to assume an induction hypothesis about every natural number that is smaller than n. It is well known that strong induction is equivalent in logical power to classic induction; but sometimes strong induction is more convenient.

ACL2 does not support strong induction, but it does permit you to have an induction hypothesis about any fixed number of components, provided you can prove that each component is smaller than n. Before we describe the ACL2 Induction Principle, we digress to discuss well-founded induction, which is a generalization of strong induction. The ACL2 Induction Principle can be seen as a special case of well-founded induction, in which the induction hypothesis is made about a fixed finite number of components.

Recall that a well-founded structure is a pair $\langle S, \prec \rangle$ where S is a set and $\prec$ is a relation on S, *i.e.*, a subset of $S \times S$ such that there is no infinite sequence $\cdots \prec s_2 \prec s_1 \prec s_0$.

Exercise 6.17 *Show that $\prec$ is irreflexive,* i.e., *$\neg(s \prec s)$ for all $s \in S$.*

Exercise 6.18 *Show that $\langle S, \prec^+ \rangle$, is a well-founded structure, where $\prec^+$ is the transitive closure of $\prec$.*

Exercise 6.19 *Let T be a subset of S. An element m of T is a* minimal *element of T iff there is no element n of T such that $n \prec m$. Prove that a structure $\langle S, \prec \rangle$ is well-founded iff all non-empty subsets of S have a minimal element.*[7]

Mathematical induction works because the natural numbers (with the usual ordering) are well-founded: if some property fails to hold for all naturals, it fails for some minimal n, but holds for all smaller numbers, which is exactly what we prove does not happen. We can extend this idea to well-founded structures. The *principle of well-founded induction* states: Let $\langle W, \prec \rangle$ be well-founded and let P be a predicate on elements of W. Then:

$$\forall w P(w) \quad \text{iff} \quad \forall w[(\forall v[v \prec w \rightarrow P(v)]) \rightarrow P(w)]$$

Proof For the non-trivial direction, assume $\forall w[(\forall v[v \prec w \rightarrow P(v)]) \rightarrow P(w)]$, but $\neg(\forall w P(w))$, in which case the set $S = \{x : \neg P(x)\}$ is non-empty and therefore has a minimal element, say w. By the minimality of w, $\forall v[v \prec w \rightarrow P(v)]$, and hence using our assumption, $P(w)$, a contradiction. □

Exercise 6.20 *Show that mathematical induction is a special case of well-founded induction. (Instantiate W and $\prec$ appropriately.) What happened to the base case?*

[7] An informal proof is fine here. Those familiar with set theory will notice that they are using some form of the axiom of choice in their proof.

Exercise 6.21 *Show that course of values induction is a special case of well-founded induction. (Instantiate W and $\prec$ appropriately.)*

Here, finally, is a precise description of the induction principle for the ACL2 logic. As with the admission of recursive definitions, we formalize this in terms of the ordinals, substitutions that map the induction variables to their components, and measure conjectures establishing that these components are smaller in a suitable well-founded sense.

- *Induction Principle*: The formula ϕ may be derived by proving the formulas below,
  - ◊ *Base Case*:
    `(implies (and (not` q_1`) ... (not` q_k`)) `ϕ`)`
  - ◊ *Induction Step(s)*: For each $1 \leq i \leq k$,
    `(implies (and` q_i
    $\phi/\sigma_{i,1}$
    ...
    $\phi/\sigma_{i,h_i}$`)`
    ϕ`)`

  provided that for terms m, $q_1, \ldots, q_k$, and substitutions $\sigma_{i,j}$ ($1 \leq i \leq k$ and $1 \leq j \leq h_i$), the following measure conjectures are theorems.
  - ◊ `(EO-ORDINALP` m`)`
  - ◊ For each $1 \leq i \leq k$ and $1 \leq j \leq h_i$,
    `(IMPLIES` q_i `(EO-ORD-<` $m/\sigma_{i,j}$ m`))`

Each q_i determines an induction step. The $\sigma_{i,j}$, for a given i, determine the induction hypotheses for the q_i step. There is one substitution for each hypothesis. The q_i step has h_i induction hypotheses. The measure conjectures establish that the supplied measure, m, of the components is smaller than m of the induction variables. A single base case handles the situation in which all of the q_i are false. In practice this case may split into many subcases, depending on the q_i.

We now apply the principle to produce the induction hinted at in our previous remarks. Let q_1 be `(and (integerp n) (< 0 n))`. That is, we have one induction step for positive integers. Let q_2 be `(and (integerp n) (< n 0))`. That is, we have another induction step for negative integers. Let k be 2, so we have only two induction steps. In the q_1 case, let us have two induction hypotheses (so $h_1 = 2$). The first hypothesis is about $n - 1$ and the second is about $\lfloor n/2 \rfloor$. That is, $\sigma_{1,1}$ is [n ◁ `(- n 1)`] and $\sigma_{1,2}$ is [n ◁ `(floor n 2)`]. In the q_2 case, let us have one induction hypothesis (so

$h_2 = 1$) about $n + 1$. So $\sigma_{2,1}$ is [n ◁ (+ n 1)]. To justify these hypotheses we measure with (acl2-count n).

Therefore, the measure conjectures are as shown below.

```
(e0-ordinalp (acl2-count n))
(implies (and (integerp n) (< 0 n))
         (e0-ord-< (acl2-count (- n 1)) (acl2-count n)))
(implies (and (integerp n) (< 0 n))
         (e0-ord-< (acl2-count (floor n 2)) (acl2-count n)))
(implies (and (integerp n) (< n 0))
         (e0-ord-< (acl2-count (+ n 1)) (acl2-count n)))
```

The base case is defined by (and (not q_1) (not q_2)), which in this case can be simplified to the following formula.

Base Case:

```
(implies (or (not (integerp n))
             (equal n 0))
         (φ n))
```

This is propositionally equivalent to having two base cases, one for noninteger n and the other for 0.

The two induction steps are shown below.

Induction Step 1:

```
(implies (and (and (integerp n) (< 0 n))
              (φ (- n 1))
              (φ (floor n 2)))
         (φ n))
```

Induction Step 2:

```
(implies (and (and (integerp n) (< n 0))
              (φ (+ n 1)))
         (φ n))
```

A subtlety of the Induction Principle is that any variable not occurring freely in m may be mapped by $\sigma_{i,j}$ to an arbitrary term.

Exercise 6.22 *Use the Induction Principle to generate the following base case and induction step.*

Base Case:

```
(implies (zp x) (φ x y a))
```

Induction Step:

```
(implies (and (not (zp x))
              (φ (- x 1) y (+ y a)))
         (φ x y a))
```

Exercise 6.23 *Use the Induction Principle to generate a base case and an induction step that are equivalent to the following.*

Base Case 1:
```
(implies (zp x) (φ x))
```
Base Case 2:
```
(implies (equal x 1) (φ x))
```
Induction Step:
```
(implies (and (not (zp x))
              (not (equal x 1))
              (φ (- x 2)))
         (φ x))
```

Exercise 6.24 *Use the Induction Principle to generate a base case and some induction steps that are equivalent to the following.*

Base Case 1:
```
(implies (equal x 0)
         (φ x))
```
Base Case 2:
```
(implies (and (not (integerp x))
              (not (consp x)))
         (φ x))
```
Induction Step 1:
```
(implies (and (integerp x)
              (< x 0)
              (φ (+ x 1)))
         (φ x))
```
Induction Step 2:
```
(implies (and (integerp x)
              (< 0 x)
              (φ (- x 1)))
         (φ x))
```
Induction Step 3:
```
(implies (and (consp x)
              (φ (car x))
              (φ (cdr x)))
         (φ x))
```

Exercise 6.25 *Consider the following recursive function definition (more precisely, definition scheme). Characterize the relationship between the admission of such a definition and a certain instance of the Induction Principle.* Hint: *Write down the measure conjectures required for the admission of f. Instantiate the Induction Principle so that the m, k, h_k, and q_i of the Induction Principle are the corresponding choices of this definition. Let $\sigma_{i,j}$ of the Induction Principle be the substitution that maps each v_x to $\delta_{i,j,x}$.*

What is the base case of the induction? How many induction steps are there? Write down the first induction step. How many hypotheses does it have? What are they? What are the measure conjectures of the induction?

```
(defun f (v1 ... vn)
  (declare (xargs :measure m))
  (cond
   (q1 (list (f δ1,1,1 ... δ1,1,n)
             (f δ1,2,1 ... δ1,2,n)
             ...
             (f δ1,h1,1 ... δ1,h1,n)))
   ...
   (qk (list (f δk,1,1 ... δk,1,n)
             (f δk,2,1 ... δk,2,n)
             ...
             (f δk,hk,1 ... δk,h2,n)))
   (t nil)))
```

6.6 Additional Logical Constructs

This section describes some additional logical constructs that can be very useful at times.

6.6.1 Encapsulation

ACL2 provides an extension principle called *encapsulation* that permits the constrained introduction of undefined function symbols. Theorems must be proved to establish that the constraints are satisfiable. The following encapsulation introduces a constraint ϕ named *constraint* on a new function symbol f.

```
(encapsulate (((f x1 ... xn) => *))
  (local (defun f (x1 ... xn) body))
  (defthm constraint φ))
```

This `encapsulate` form is *admissible* provided its `defun` form is admissible and formula ϕ is a theorem of the logical world extended with the definition. Generally speaking, ϕ is a formula involving the function symbol f. If admissible, the encapsulation extends the theory by adding

- *Constraint Axiom for* f: ϕ.

Note that in the resulting extension, f is undefined but is known to satisfy ϕ. The admissibility requirements establish that there exists at least one

function satisfying the constraint ϕ, namely the "local witness" $(f\ x_1\ \ldots\ x_n) = body$.

Since the only axiom known about f is ϕ, it stands to reason that any theorem proved about f is actually valid for any function $\hat{f}$ that also has "property ϕ." To make this precise, define $\hat{\psi}$ to be the formula obtained from ψ by the replacement of function symbol f by $\hat{f}$.[8] Then a derived rule of inference, called *functional instantiation* [3], says that from any theorem θ one may infer the theorem $\hat{\theta}$ provided $\hat{\phi}$ is a theorem. That is, one may infer a theorem about $\hat{f}$ from an analogous theorem about f if $\hat{f}$ satisfies the constraint ϕ on f. See lemma-instance and constraint for more details about how functional instantiation works in ACL2.

The encapsulation facility is much more general than sketched here. See [26]. Together, encapsulation and functional instantiation provide a second order aspect (*i.e.*, they allow quantification over functions) to ACL2.

6.6.2 Additional Features and Remarks

We introduce some additional features of the logic that you may find useful. We do not give a full description here. The interested reader is encouraged to consult the online documentation and the companion book, *Computer-Aided Reasoning: ACL2 Case Studies* [22], which contains case studies (with exercises) that make use of such features.

- Logically speaking, multiple value vectors are just lists. (mv-let (v_0 ... v_n) *res body*) is logically equivalent to

```
(let ((v res))
  (let ((v0 (mv-nth 0 v))
        ...
        (vn (mv-nth n v)))
    body)),
```

  but syntactic restrictions are enforced so that execution can proceed without ever constructing the vectors as list objects. When proving theorems about multiple value functions you will often refer to the components of the answer vector using mv-nth.[9]

[8] $\hat{f}$ can be a function symbol or a lambda expression and must be of the same arity as f.

[9] Often in mechanical proofs, (mv-nth 0 v) is simplified to (car v), while (mv-nth 1 v) is not expanded. This is due to the heuristics for expanding the recursive function mv-nth, where 0 is the base case. This obscure note may help you formulate rewrite rules about the various components of the output vector of such a function.

- Single-threaded objects are provided. Such objects have the usual "copy-on-write" applicative semantics, *i.e.*, components of such objects may be modified by copying the superstructure of the object down to the affected component. This provides no improvement in the expressive power of the logic. But the use of single-threaded objects is syntactically restricted so that modifications may be implemented destructively via "pointer smashing." See [8] and the documentation for `stobj` and `defstobj`.

- It is possible to introduce an undefined function whose value is constrained to be some object satisfying a certain formula, provided such an object exists. This effectively extends the ACL2 logic to full first order logic. See `defchoose`.

- It is possible to introduce a function whose "body" is a universally (or existentially) quantified formula. This is done with `defun-sk` which is implemented using `defchoose`. For example, we formulate $\forall x[\exists y(x < y)]$ as follows.

```
(defun-sk not-biggest (x)
  (exists y (< x y)))

(defun-sk nothing-is-biggest ()
  (forall x (not-biggest x)))
```

  We can prove `(nothing-is-biggest)`, which represents the quantified formula above.

The logic is largely modeled after that of Nqthm [5, 7] and is the joint work of Bob Boyer, Matt Kaufmann, and J Moore. A precise description of the logic is given in [25].

7

Proof Examples

7.1 Simple Example Theorems

In this chapter we prove a few theorems to drive home the point that ACL2 is a mathematical logic rather than just a programming language. For use in our proofs, here are some of the theorems, axioms, and definitions mentioned above.

Name	Axiom, Definition, or Theorem
Thm 1	`t` $\neq$ `nil`
Ax 2	`x` = `nil` $\rightarrow$ `(if x y z)` = `z`
Ax 3	`x` $\neq$ `nil` $\rightarrow$ `(if x y z)` = `y`
Def `not`	`(not p)` = `(if p nil t)`
Ax 4	`(car (cons x y))` = `x`
Ax 5	`(cdr (cons x y))` = `y`
Ax 6	`(consp (cons x y))` = `t`
Thm 7	`(endp x)` = `(not (consp x))`

Lemma 1. `(endp (cons u v))` = `nil`.

Proof

`(endp (cons u v))`

= { 1. Thm 7 }

`(not (consp (cons u v)))`

= { 2. Ax 6 }

`(not t)`

= { 3. Def `not` }

`(if t nil t)`

= { 4. Thm 1 and Ax 3 }

`nil` □

The justification of step 1 means "by instantiation of Thm 7." We have instantiated the theorem, replacing `x` by `(cons u v)`, and written down the resulting equality as our first step in the proof. In step 2, the instantiation of Axiom 6 tells us that `(consp (cons u v))` is `t` so we may substitute

the latter for the former by substitution of equals for equals. In step 3, we replace `not` by its definition, appropriately instantiating it. In step 4, we use two axioms. An instance of Axiom 3 tells us that `(if t nil t)` is `nil`, provided `t` is not `nil` and Thm 1 establishes the hypothesis. Since equality is transitive, we have shown that `(endp (cons u v))` is equal to `nil`, as desired.

Instantiation, substitution of equals for equals, and the transitivity of equality are so common in our proofs that we rarely note their use. If you would like to see formal proofs in which these techniques are used or derived explicitly, see [37] or any other good textbook on mathematical logic. Technically, the proofs presented here are informal rather than formal. However they are sufficiently rigorous to make straightforward their translation to formal proofs.

We next define the list concatenation function, naming it `app` to avoid the clash with the existing function `append`.

```
(defun app (x y)
  (if (endp x)
      y
    (cons (car x)
          (app (cdr x) y))))
```

This is admitted to the logic because `app` is a new function symbol, `x` and `y` are distinct variable symbols, the body, above, is a term that mentions no variables other than `x` and `y`, and the measure conjectures below are theorems.

```
(e0-ordinalp (acl2-count x))
(implies (not (endp x))
         (e0-ord-< (acl2-count (cdr x))
                   (acl2-count x)))
```

We do not offer proofs of these theorems here, because they involve the ordinals and arithmetic and we are just getting started, but we discuss the proof of the second conjecture later.

Once `app` has been admitted, we have the following axiom.

Def `app`:

```
(app x y)
  =
(if (endp x)
    y
  (cons (car x)
        (app (cdr x) y)))
```

Lemma 2.

```
(app (cons a b) c) = (cons a (app b c)).
```

Proof

```
     (app (cons a b) c)
= { 1. Def app }
     (if (endp (cons a b))
         c
       (cons (car (cons a b))
             (app (cdr (cons a b)) c)))
= { 2. Lemma 1 }
     (if nil
         c
       (cons (car (cons a b))
             (app (cdr (cons a b)) c)))
= { 3. Ax 2 }
     (cons (car (cons a b))
           (app (cdr (cons a b)) c))
= { 4. Ax 4 }
     (cons a
           (app (cdr (cons a b)) c))
= { 5. Ax 5 }
     (cons a (app b c))  □
```

Because of the correspondence between terms and formulas we could have stated these lemmas using the function `equal` rather than the relation =. That is, we could have written

Lemma 1.
`(equal (endp (cons u v)) nil)`
Lemma 2.
`(equal (app (cons a b) c) (cons a (app b c)))`

7.2 Example Theorems with Induction

Now, moving a little faster, we prove that app is associative by induction.

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof We appeal to the Induction Principle:

ϕ	`(equal (app (app a b) c) (app a (app b c)))`
m	`(acl2-count a)`
q_1	`(consp a)`
k	1
$\sigma_{1,1}$	[`a` ◁ `(cdr a)`]
h_1	1

Scrutiny of the Induction Principle reveals that the measure theorems we are obliged to prove are those required for the admission of `app` with the variable symbol `a` used in place of `x`. Thus, according to the Induction Principle, it is sufficient to prove the following two cases.

Base Case.

```
(implies (not (consp a))
         (equal (app (app a b) c)
                (app a (app b c))))
```

Given Thm 7 and `app`, this simplifies to

```
(implies (not (consp a))
         (equal (app b c)
                (app b c)))
```

because `app` returns its second argument when its first is not a `consp`. But the concluding equality follows from the reflexivity of equality and the definition of `implies`.

Induction Step.

```
(implies (and (consp a)
              (equal (app (app (cdr a) b) c)    ; IH
                     (app (cdr a) (app b c))))
         (equal (app (app a b) c)
                (app a (app b c))))
```

Note that the induction hypothesis, IH, is just $\phi/\sigma_{1,1}$. Consider the left-hand side of the concluding equality:

`(app (app a b) c)`

= { Def `app`, `(consp a)` }

`(app (cons (car a) (app (cdr a) b)) c)`

= { Lemma 2 }

`(cons (car a) (app (app (cdr a) b) c))`

= { Induction Hypothesis }

`(cons (car a) (app (cdr a) (app b c)))`

= { Def `app`, `(consp a)` }

`(app a (app b c))` □

Exercise 7.1 *Try to mimic the above proof for the following instance of the associativity of* `app`.

```
(equal (app (app a b) a)
       (app a (app b a)))
```

Notice that the proof attempt fails. One often has to strengthen a theorem in order to apply induction.

The proof above does not illustrate any arithmetic reasoning. Let us turn to the previously mentioned

Measure Condition:

```
(implies (not (endp a))
         (e0-ord-< (acl2-count (cdr a))
                   (acl2-count a)))
```

Proof Using Thm 7, the definitions of `e0-ord-<` and `acl2-count`, and propositional calculus with equality, we can simplify the condition to

```
(implies (consp a)
         (< (acl2-count (cdr a))
            (+ 1
               (acl2-count (car a))
               (acl2-count (cdr a))))).
```

The concluding inequality is proved with facts about arithmetic and the the lemma that `acl2-count` returns a natural number. (This latter fact can be proved by induction.) □

The relevant facts about arithmetic allow us to "add" inequalities so as to deduce `(< (+ x y) (+ u v))` from `(< x u)` and `(<= y v)` (for example) and simplify arithmetic expressions. These facts are either axioms or additional lemmas we can prove.

Having proved that `app` is associative, the following corollary is trivial:

```
(equal (app (app (app a b) a) (app c a))
       (app a (app (app b a) (app c a))))
```

Proof Both sides of the above equality may be rewritten to

```
(app a (app b (app a (app c a))))
```

by the associativity of `app`. □

We exhibit this theorem simply to drive home the point that not every theorem about `app` is proved by induction!

7.3 Example Theorems About Trees

The following function flattens a binary tree.

```
(defun flatten (x)
  (cond ((atom x) (list x))
        (t (app (flatten (car x))
                (flatten (cdr x))))))
```

For example, `(flatten '((a . b) . (c . d)))` has the value `(a b c d)`. The function is admitted using the measure `(acl2-count x)`. The proof that `(acl2-count (car x))` is less than `(acl2-count x)` when `(consp x)` is analogous to the proof above for `(acl2-count (cdr x))`.

Here is an alternative algorithm for flattening a binary tree.

```
(defun flat (x)
  (cond ((atom x) (list x))
        ((atom (car x)) (cons (car x) (flat (cdr x))))
        (t (flat (cons (caar x)
                       (cons (cdar x) (cdr x)))))))
```

If you do not understand how this function "works," consider the following simple example.

```
 (flat '((a . (b . c)) . d))
=
 (flat (cons (cons 'a (cons 'b 'c)) 'd))
=
 (flat (cons 'a (cons (cons 'b 'c) 'd)))
=
 (cons 'a
       (flat (cons (cons 'b 'c) 'd)))
=
 (cons 'a
       (flat (cons 'b (cons 'c 'd))))
=
 (cons 'a
       (cons 'b (flat (cons 'c 'd))))
=
 (cons 'a
       (cons 'b
             (cons 'c (flat 'd))))
=
 (cons 'a
       (cons 'b
             (cons 'c
                   (cons 'd nil))))
=
 '(a b c d)
```

Of course, we can do this simple derivation only after admitting `flat`. How can we admit this definition? In the first recursive call of `flat`, the `acl2-count` of the argument is smaller. But in the second call, a little reflection will show that the `acl2-count` of the argument is unchanged. Hence, the following is a theorem.

```
(implies (and (consp x)
              (consp (car x)))
         (equal (acl2-count (cons (caar x)
                                  (cons (cdar x) (cdr x))))
                (acl2-count x)))
```

(Recall that `(atom x)` is equal to `(not (consp x))`.)

However, we observe that in the second call `(acl2-count (car x))` is smaller.

```
(implies (and (consp x)
              (consp (car x)))
         (< (acl2-count (car (cons (caar x)
                                   (cons (cdar x) (cdr x)))))
            (acl2-count (car x))))
```

This theorem follows trivially after simplifying the `(car (cons (caar x) ...))` to `(caar x)`.

Thus, we have the makings of a lexicographic argument for admitting `flat`: either the size of `x` decreases or the size of `x` stays fixed and the size of `(car x)` decreases, where size in measured with `acl2-count`.

In Exercise 6.8 (page 88)we saw that termination arguments based on lexicographic arguments can be formalized with the ordinals. Suppose `i1`, `i2`, `j1`, and `j2` are natural numbers. Let α and β be `(cons (+ 1 i1) i2)` and `(cons (+ 1 j1) j2)` respectively.[1] Then both `(e0-ordinalp` α`)` and `(e0-ordinalp` β`)` are easily proved from the definition of `e0-ordinalp`. Furthermore `(e0-ord-<` α β`)` is equivalent to

```
(or (< i1 j1) (and (equal i1 j1) (< i2 j2))).
```

Thus, we can admit `flat` with the following measure.

```
(cons (+ 1 (acl2-count x))
      (acl2-count (car x)))
```

This measure is an ordinal, according to `e0-ordinalp`, and decreases, according to `e0-ord-<`, on each recursive call in `flat`. We now prove the following theorem:

```
(equal (flat x) (flatten x))
```

Proof We induct, using the following instance of the Induction Principle.

[1] We increment the first component by 1 to insure that it is non-0, as required by `e0-ordinalp`.

ϕ	`(equal (flat x) (flatten x))`
m	`(cons (+ 1 (acl2-count x)) (acl2-count (car x)))`
q_1	`(and (not (atom x)) (atom (car x)))`
q_2	`(and (not (atom x)) (not (atom (car x))))`
k	2
$\sigma_{1,1}$	[x ◁ (cdr x)]
$\sigma_{2,1}$	[x ◁ (cons (caar x) (cons (cdar x) (cdr x)))]
h_1	1
h_2	1

Observe that the measure m used is just that used in the admission of `flat` and the measure conjectures required by the Induction Principle are the ones proved for `flat`. Since `(and (not` q_1`) (not` q_2`))` is equivalent to `(atom x)` (you should confirm this), the base case is:

Base Case.

```
(implies (atom x)
         (equal (flat x) (flatten x)))
```

This is trivial, since both sides of the conclusion are equal to `(list x)`.

Induction Step 1.

```
(implies (and (not (atom x))
              (atom (car x))
              (equal (flat (cdr x)) (flatten (cdr x))))
         (equal (flat x) (flatten x)))
```

We will rewrite the concluding equality to true. First, we expand the definitions of both `flat` and `flatten`, so that the conclusion becomes

```
(equal (cons (car x) (flat (cdr x)))
       (app (flatten (car x))
            (flatten (cdr x)))).
```

Since `(atom (car x))` is true, `(flatten (car x))` may be expanded to `(list (car x))`, producing

```
(equal (cons (car x) (flat (cdr x)))
       (app (list (car x))
            (flatten (cdr x)))).
```

We then expand the definition of `app` to produce

```
(equal (cons (car x) (flat (cdr x)))
       (cons (car x) (flatten (cdr x)))).
```

Finally, we appeal to our induction hypothesis, replacing `(flat (cdr x))` by `(flatten (cdr x))`, to produce the trivial identity

```
(equal (cons (car x) (flatten (cdr x)))
       (cons (car x) (flatten (cdr x)))).
```

This completes the proof of the first induction step.

Induction Step 2.

```
(implies (and (not (atom x))
              (not (atom (car x)))
              (equal (flat (cons (caar x)
                                 (cons (cdar x)
                                       (cdr x))))
                     (flatten (cons (caar x)
                                    (cons (cdar x)
                                          (cdr x))))))
         (equal (flat x) (flatten x)))
```

We first expand the definition of `flatten` twice in the equality hypothesis, rewriting the goal to the formula shown below.

```
(implies (and (not (atom x))
              (not (atom (car x)))
              (equal (flat (cons (caar x)
                                 (cons (cdar x)
                                       (cdr x))))
                     (app (flatten (caar x))
                          (app (flatten (cdar x))
                               (flatten (cdr x))))))
         (equal (flat x) (flatten x)))
```

Given that neither `x` nor `(car x)` is an atom, we can expand the `(flat x)` in the conclusion.

```
(implies (and (not (atom x))
              (not (atom (car x)))
              (equal (flat (cons (caar x)
                                 (cons (cdar x)
                                       (cdr x))))
                     (app (flatten (caar x))
                          (app (flatten (cdar x))
                               (flatten (cdr x))))))
         (equal (flat (cons (caar x)
                            (cons (cdar x)
                                  (cdr x))))
                (flatten x)))
```

Similarly, we can expand the `flatten`, twice, in the conclusion.

```
(implies (and (not (atom x))
              (not (atom (car x)))
              (equal (flat (cons (caar x)
                                 (cons (cdar x)
                                       (cdr x))))
                     (app (flatten (caar x))
                          (app (flatten (cdar x))
```

```
                                (flatten (cdr x))))))
        (equal (flat (cons (caar x)
                           (cons (cdar x)
                                 (cdr x))))
               (app (app (flatten (caar x))
                         (flatten (cdar x)))
                    (flatten (cdr x)))))
```

Observe that the `flat` expressions in the hypothesis and conclusion are identical. So we use our induction hypothesis by substituting the `app` expression of the hypothesis for the `flat` expression in the conclusion. It therefore suffices to prove the equality below.

```
(equal (app (flatten (caar x))
            (app (flatten (cdar x))
                 (flatten (cdr x))))
       (app (app (flatten (caar x))
                 (flatten (cdar x)))
            (flatten (cdr x))))
```

But this is just an instance of the previously proved associativity of `app`, so we are done. (An alternative proof would have been to use associativity earlier. Had we right-associated the `app` terms in the conclusion after expanding `(flatten x)` the induction hypothesis would have matched the conclusion perfectly.) □

The induction above is essentially "suggested" by the case analysis and recursion in `(flat x)`. Compare the induction done above to the definition of `flat`. Also, see Exercise 6.25.

7.4 Comments About the Proof Process

Discovering proofs and writing them down are two quite different activities. Often when we are looking for a proof we start with the conclusion of the induction step. We expand certain of the recursive functions there and try to massage the expanded conclusion into the form of the original conjecture. This often results in the realization that the formula "being proved" is not a theorem because of special cases that are not ruled out by the hypotheses. Alternatively, it may result in the realization that the formula being proved is not general enough to carry itself through induction. That is, one cannot massage the induction conclusion to look like the original conjecture because the conjecture was too specific. A new formula must then be found. A simple example of this is illustrated in Exercise 7.1 below and will occur over and over in your use of ACL2 in interesting applications.

We do not have much to say about how to find a suitably general conjecture. It can be one of the most creative kinds of mathematical thought. The key insight might come to you only after you have spent hours in

formula manipulation and given up in frustration to work on some non-intellectual task like riding your bike, mowing the lawn, or taking a shower. The subconscious is a powerful thing, especially after you have internalized the behavior of the symbols in your problem.

Once a suitable "proof" is found, a quick proof sketch can help identify the case structure (q_i) and instantiations ($\sigma_{i,j}$) of an inductive proof, as well as the key lemmas you need to massage the induction conclusion into the selected hypotheses. During this activity, we often use extremely informal notation. For example, we might simplify "in place," erasing or marking out subterms to write down their expansions without copying the entire formula. This activity can be done on the back of an envelope, on a napkin, or in the margins of a book. It generally solidifies your conviction that you have found a proof.

But years of teaching recursion and induction have shown that many mistakes remain in such proof sketches. Two very common ones are:

- The "induction hypotheses" used are not really instantiations of the formula being proved.
- Restrictive hypotheses added to make the original conjecture a theorem are not considered in the inductive step.

Here is a narrative description of how the second problem often crops up. You are trying to prove `(p x)`. Induction on `x` feels right and you begin to work out a proof of `(p x)`, given `(consp x)` and `(p (cdr x))`. The proof raises a case you had previously overlooked and so you add to your original conjecture a new hypothesis `(q x)` to rule it out. With `(q x)`, you complete your proof. You then check the base case, `(not (consp x))`, and find it trivial. You may feel that you have a proof but you may not!

The conjecture you are now working on is `(implies (q x) (p x))`. Your proof is by induction on `x`. The induction step is

```
(implies (and (consp x)
              (implies (q (cdr x)) (p (cdr x))))
         (implies (q x) (p x)))
```

This is propositionally equivalent to

```
(implies (and (consp x)
              (implies (q (cdr x)) (p (cdr x)))
              (q x))
         (p x))
```

You have a proof that `(consp x)`, `(q x)`, and `(p (cdr x))` imply `(p x)`. But you do not have the induction hypothesis `(p (cdr x))`. You have the induction hypothesis `(implies (q (cdr x)) (p (cdr x)))`. That is, you cannot assume `(p (cdr x))` is true unless you can establish `(q (cdr x))`.

When you modify your original conjecture to add a new hypothesis, you must consider how this hypothesis will interact with the inductive proof you

are trying to construct. In terms of the example above, most often you will find that you must prove that `(consp x)` and `(q x)` imply `(q (cdr x))`. A trivial example of this problem occurs in Exercise 7.4 below.

Because of problems like this we recommend: *After you have sketched out a "back of the envelope" proof, take the time to write it down carefully!* This is especially important for informal inductive proofs.

- Write down the theorem you are proving! You would be surprised how many times this step is omitted!
- Write down the instantiations of m, the q_i, and the $\sigma_{i,j}$ of the Induction Principle.
- Apply the $\sigma_{i,j}$ to the formula and write down the results. These are your induction hypotheses. You can avoid many mistakes simply by seeing them!
- Then show how to massage your induction conclusion so that you can use your hypotheses, with special attention to relieving the hypotheses of the hypotheses.
- Write down the lemmas you use in this manipulation and find proofs of them.

You should practice proofs of theorems like this until you are comfortable with them, especially with the induction principle and simplification by the use of definitions and previously proved theorems. We strongly recommend that you do the following exercises by hand rather than with the mechanized theorem prover, at least if you are not a mathematician already trained in the art of proof.

7.5 Exercises

Exercise 7.2 *Here is the definition of a list reverse function.*

```
(defun rev (x)
  (if (endp x)
      nil
    (app (rev (cdr x)) (list (car x)))))
```

A "true list" is a binary tree whose right-most branch terminates in `nil`. *True lists are recognized by the predicate defined as follows.*

```
(defun true-listp (x)
  (if (atom x)
      (equal x nil)
    (true-listp (cdr x))))
```

Prove

`(true-listp (rev x))`

or show a counterexample. If you show a counterexample, can you modify the conjecture to make it a theorem? If so, prove the modified conjecture.

Exercise 7.3 *Prove*

`(equal (app x nil) x)`

or show a counterexample. If you show a counterexample, can you modify the conjecture to make it a theorem? If so, prove the modified conjecture.

Exercise 7.4 *What is wrong with the following argument?*

`(implies (consp x) (equal (app x nil) x))`

Proof *Induct on* `x`. *The base case, defined by* `(not (consp x))`, *is a theorem because we know* `(consp x)`. *In the induction step we know* `(consp x)` *and* `(equal (app (cdr x) nil) (cdr x))`. *The induction conclusion is* `(equal (app x nil) x)`, *which expands to* `(equal (cons (car x) (app (cdr x) nil)) x)`. *By the induction hypothesis, we get* `(equal (cons (car x) (cdr x)) x)`, *which is true.* □

Exercise 7.5 *Prove*

`(equal (rev (app a b)) (app (rev a) (rev b)))`

or show a counterexample. If you show a counterexample, can you modify the conjecture to make it a theorem? If so, prove the modified conjecture.

Exercise 7.6 *Prove*

`(equal (rev (rev x)) x)`

or show a counterexample. If you show a counterexample, can you modify the conjecture to make it a theorem? If so, prove the modified conjecture.

Exercise 7.7 *The following function produces the mirror image of a tree.*

```
(defun swap-tree (x)
  (if (atom x)
      x
    (cons (swap-tree (cdr x))
          (swap-tree (car x)))))
```

Prove

`(equal (flatten (swap-tree x)) (rev (flatten x)))`

or show a counterexample. If you show a counterexample, can you modify the conjecture to make it a theorem? If so, prove the modified conjecture.

Exercise 7.8 *Here is another way to reverse a list.*

```
(defun rev1 (x a)
  (if (endp x)
      a
    (rev1 (cdr x) (cons (car x) a))))
```

Prove

```
(equal (rev1 x nil) (rev x))
```

or show a counterexample. If you show a counterexample, can you modify the conjecture to make it a theorem? If so, prove the modified conjecture.

Exercise 7.9 *Here is another way to flatten a tree, due to John McCarthy,*

```
(defun mc-flatten (x a)
  (if (atom x)
      (cons x a)
    (mc-flatten (car x)
                (mc-flatten (cdr x) a))))
```

Prove

```
(equal (mc-flatten x nil) (flatten x))
```

or show a counterexample. If you show a counterexample, can you modify the conjecture to make it a theorem? If so, prove the modified conjecture.

Part IV

Gaming

8

The Mechanical Theorem Prover

This part of the book is concerned with the mechanization of the logic. Our goal is to teach you how to use the theorem prover. We start, in this chapter, by sketching how the theorem prover works. Of course, knowing how something works—*e.g.*, an automobile, a programming language, a violin—is quite different from knowing how to use it effectively. So, in Chapter 9, we begin to explain how to use the theorem prover. Finally, in Chapter 10, we use the theorem prover to do many example proofs.

As you read this chapter you may begin to get the idea that the machine does everything for you. This is not true. A more accurate view is that the machine is a proof assistant that fills in the gaps in your "proofs." These gaps can be huge. When the system fails to follow your reasoning, you can use your knowledge of the mechanization to figure out what the system is missing. But you may find that the machine's inability to fill in the gap is because your "proof" was simply wrong. Indeed, you may even find that the formula you "proved" is not even a theorem!

You may come to think of the proof process as a game. The theorem is the "opponent." It will use all legal means to dodge your weapons and squirm free of your traps and fences. It can hide amid innocuous detail, shatter into a swarm of subproblems, or stand crystalline still and shimmering in front of you, daring you to find a chink in its armor. In recognition of this view of theorem proving we have named this part of the book "Gaming." You will be hard pressed to find a more challenging game.

8.1 A Sample Session

Here is how the theorem prover responds to the command to prove that app (defined on page 104) is associative. The user input consists of the first three lines of text following the ACL2 > prompt. Everything else was produced automatically. You should read this proof and compare it to the one on page 105.

```
ACL2 >(defthm associativity-of-app
        (equal (app (app a b) c)
               (app a (app b c))))
```

```
Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes
are suggested by this conjecture.  Subsumption reduces that
number to two.  However, one of these is flawed and so we
are left with one viable candidate.

We will induct according to a scheme suggested by (APP A B).
If we let  (:P A B C) denote *1 above then the induction
scheme we'll use is
(AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A) B C))
              (:P A B C))
     (IMPLIES (ENDP A) (:P A B C))).
This induction is justified by the same argument used to
admit APP, namely, the measure (ACL2-COUNT A) is decreasing
according to the relation E0-ORD-< (which is known to be
well-founded on the domain recognized by E0-ORDINALP).  When
applied to the goal at hand the above induction scheme produces
the following two nontautological subgoals.

Subgoal *1/2
(IMPLIES (AND (NOT (ENDP A))
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP A)
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But simplification reduces this to T, using the :definition
APP, the :rewrite rules CDR-CONS and CAR-CONS and primitive
type reasoning.

Subgoal *1/1
(IMPLIES (ENDP A)
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP A))
```

```
        (EQUAL (APP (APP A B) C)
               (APP A (APP B C)))).

But simplification reduces this to T, using the :definition
APP and primitive type reasoning.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM ASSOCIATIVITY-OF-APP ...)
Rules: ((:REWRITE CDR-CONS)
        (:REWRITE CAR-CONS)
        (:DEFINITION NOT)
        (:DEFINITION ENDP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:DEFINITION APP))
Warnings:  None
Time:  0.04 seconds (prove: 0.03, print: 0.00, other: 0.01)
 ASSOCIATIVITY-OF-APP
```

8.2 Organization of the Theorem Prover

As noted earlier and as depicted in Figure 8.1, the theorem prover takes input from both you and a data base, called the *logical world* or simply *world*. The world embodies a theorem proving strategy, developed by you and codified into *rules* that direct certain aspects of the theorem prover's behavior. When trying to prove a theorem, the theorem prover applies your strategy, possibly using hints you supply with the theorem, and prints its proof attempt. You have no interactive control over the system's behavior once it starts a proof attempt, except that you can interrupt it and abort the attempt. When the system succeeds, new rules, derived from the just-proved theorem, are added to the world according to directions supplied by you. When the system fails, you must inspect the proof attempt to see what went wrong.

Your main activity when using the theorem prover is designing your theorem proving strategy and expressing it as rules derived from theorems. There are over a dozen kinds of rules, each identified by a *rule class* name. The most common are rewrite rules, but other classes include type-prescription, linear, elim, and generalize rules. The basic command for telling the system to (try to) prove a theorem and, if successful, add rules to the data base is the `defthm` command.

(`defthm` *name formula*
  `:rule-classes` (*class*$_1$ ... *class*$_n$))

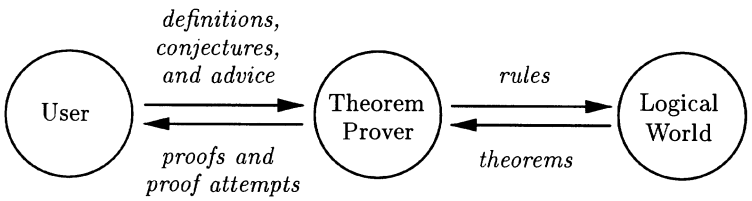

Figure 8.1: Data Flow in the Theorem Prover

The command directs the system to try to prove the given formula and, if successful, remember it under the name *name* and build it into the data base in each of the ways specified by the $class_i$. We will discuss many of the common rule classes in this chapter. But to find out details of the various rule classes, see `rule-classes`, and the documentation links under it.

Every rule has a *status* of either *enabled* or *disabled*. The theorem prover only uses enabled rules. So by changing the status of a rule or by specifying its status during a particular step of a particular proof with a "hint" (see `hints`), you can change the strategy embodied in the world. A set of rules can be collected together into a *theory* and the entire theory can be enabled or disabled. This allows a world to offer alternative strategies from which you may choose by enabling the appropriate theory. See `in-theory` for details. We ignore these issues during our description of the theorem prover but remind the reader that only enabled rules are relevant.

The theorem prover is organized as shown in Figure 8.2. At the center is a *pool* of formulas to be proved. Initially, your conjecture is the only formula in the pool. Surrounding the pool are six proof techniques. Although not depicted in the figure, each of the proof techniques uses rules in the logical world. To operate on a non-empty pool, a formula is drawn out and given to the first technique. Each technique is either applicable to the formula, in which case it reduces the formula to a set of n other formulas and deposits them into the pool, or else the technique is inapplicable to the formula, in which case it passes the formula to the next technique. In the case where n is 0, the formula is actually proved by the technique. The original conjecture is proved when there are no formulas left in the pool and the proof techniques have all halted. This organization is sometimes called "the waterfall" because in [5] it was described in those terms.[1]

The six proof techniques surrounding the pool are described at a high level in the successive sections below.

[1]Not discussed here is the modification to the waterfall to accommodate `force` and `case-split`.

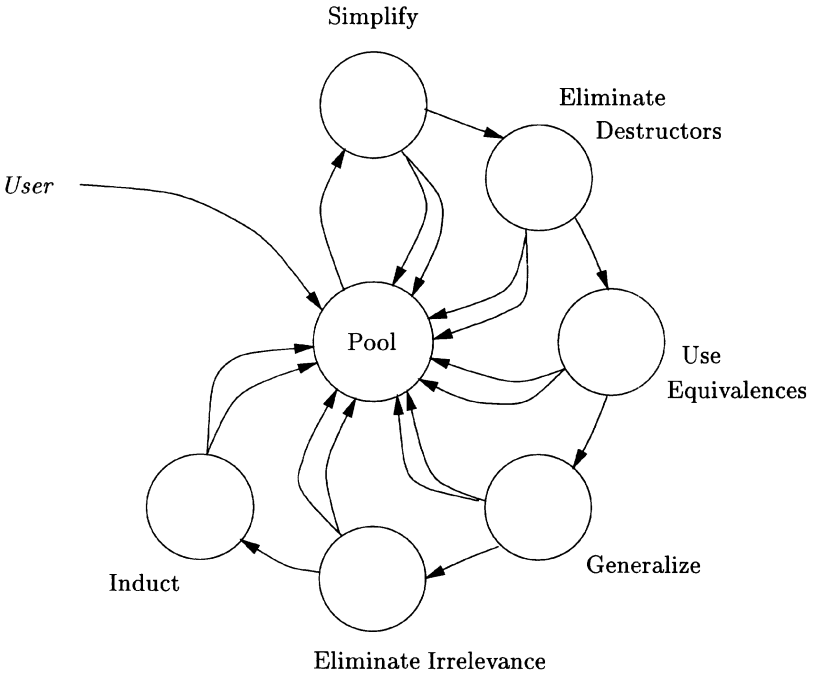

Figure 8.2: Organization of the Theorem Prover

The descriptions are tied together by a single classic Boyer-Moore example, the so-called `rev-rev` example. Consider the following recursive function definition:

```
(defun rev (x)
  (if (endp x)
      nil
    (app (rev (cdr x)) (list (car x)))))
```

This function reverses a list. For example, `(rev '(1 2 3))` evaluates to `(3 2 1)`. It has the property that if `a` is a true list, *i.e.*, one whose "final cdr" is `nil`, then `(rev (rev a))` is `a`. The hypothesis is necessary: `(rev (rev '(1 2 3 . 4)))` is `(1 2 3)`, not the original input.

The `rev-rev` formula we shall prove is

```
(implies (true-listp a)
         (equal (rev (rev a)) a)).
```

When this formula is put into the pool, it is drawn out and given to the simplification technique. That technique can do nothing with it and passes it to the next technique. In fact none of the first five techniques apply and the formula arrives at induction.

8.2.1 Induction

The key to a successful inductive argument is figuring out how to construct a proof of the formula from certain instances of the same formula. Those instances must be "smaller." The recursive functions in the formula provide suggestions for which instances to use: supply the instances obtained by expanding one or more of the recursive functions. Each recursive function decomposes its arguments in a well-founded way. Not only is this established when the function is admitted but the admission process identifies a set of measured arguments whose "size" is decreasing. Thus, an induction is suggested by each application of the function in which those measured arguments are occupied by distinct variable symbols: under the case split used in the function definition, provide inductive hypotheses corresponding to each recursive call (*e.g.*, replace each measured variable by the term used in its slot in each recursion). This suggested induction is justified by the same measure used to justify the function admission. See Exercise 6.25 for an exploration of the duality between recursion and the suggested induction.

Often, more than one set of arguments could be measured to justify a function definition. To each such set there corresponds an induction. But ACL2 only finds one justification at definition-time and hence, initially, each recursive function suggests just one induction. It is possible to prove an induction rule (see `induction`) so that a term suggests other inductions.

When a formula arrives at the induction technique, ACL2 computes all the inductions suggested by the terms in the formula. It then compares them, possibly combining several into one, and selects one regarded as most appropriate. It then prints the selected induction scheme, applies it to the formula at hand, uses simple propositional calculus to normalize the result, and puts each of the new formulas back into the pool. You can override its choice of induction by supplying an induction hint. See hints.

The "propositional calculus normalization" sometimes makes the instantiation of the induction scheme look different than the scheme itself. Suppose the induction step of the scheme is to assume test q and inductive instance p' to prove p. Suppose that the formula to be proved, p, is of the form $\alpha \rightarrow \beta$. Then the scheme would seem to call for the induction step $(q \wedge (\alpha' \rightarrow \beta')) \rightarrow (\alpha \rightarrow \beta)$. But the propositional normalization actually produces two formulas to prove: $(q \wedge \neg\alpha' \wedge \alpha) \rightarrow \beta$ and $(q \wedge \beta' \wedge \alpha) \rightarrow \beta$. The conjunction of these two is propositionally equivalent to the step required by the scheme.

The application of the induction technique to the `rev-rev` formula produces the following output. We have manually added the line numbers on the left.

```
 4. Name the formula above *1.
 5.
 6. Perhaps we can prove *1 by induction.  Two induction schemes
 7. are suggested by this conjecture.  These merge into one derived
 8. induction scheme.
 9.
10. We will induct according to a scheme suggested by (REV A).
11. If we let  (:P A) denote *1 above then the induction scheme
12. we'll use is
13. (AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A)))
14.               (:P A))
15.      (IMPLIES (ENDP A) (:P A))).
16. This induction is justified by the same argument used to
17. admit REV, namely, the measure (ACL2-COUNT A) is decreasing
18. according to the relation E0-ORD-< (which is known to be
19. well-founded on the domain recognized by E0-ORDINALP).  When
20. applied to the goal at hand the above induction scheme produces
21. the following three nontautological subgoals.
```

Line 4 is printed when a subgoal enters the induction mechanism. Lines 6–8 describe the candidate inductions. The candidates were suggested by `(true-listp a)` and `(rev a)`; but both make the same suggestion: induction on the cdr structure of `a`. Lines 13–15 give the induction scheme selected. This scheme calls for two formulas to be proved (the induction step, on the first two lines, and the base case, on the last).

Lines 16–19 explain why the induction is legal under the induction principle. Finally, lines 20–21 indicate how many goals are being put into the

pool. Note that three goals are added here, not two as might indicated by the induction scheme. Propositional normalization is responsible for the difference.

The three subgoals are not printed until they are removed from the pool for proof, but the goals and the names they are assigned are shown below.

```
 23. Subgoal *1/3
 24. (IMPLIES (AND (NOT (ENDP A))
 25.               (EQUAL (REV (REV (CDR A))) (CDR A))
 26.               (TRUE-LISTP A))
 27.          (EQUAL (REV (REV A)) A))

 95. Subgoal *1/2
 96. (IMPLIES (AND (NOT (ENDP A))
 97.               (NOT (TRUE-LISTP (CDR A)))
 98.               (TRUE-LISTP A))
 99.          (EQUAL (REV (REV A)) A))

103. Subgoal *1/1
104. (IMPLIES (AND (ENDP A) (TRUE-LISTP A))
105.          (EQUAL (REV (REV A)) A))
```

`Subgoal *1/1` is the base case. The other two, together, are the induction step. `Subgoal *1/3` is the "interesting" part of the induction step, in which one uses the conclusion of the induction hypothesis to prove the conclusion of the induction conclusion. `Subgoal *1/2` is a frequently overlooked case in which one must show that the hypothesis of the induction conclusion implies the hypothesis of the induction hypothesis. All three of these are put in the pool by induction. They are drawn out in the order listed above.

8.2.2 Simplification

Simplification is the heart of the theorem prover. We will discuss simplification in more detail later. The main things it does are:

- apply propositional calculus, equality, and rational linear arithmetic decision procedures,
- use type information and forward chaining rules to construct a "context" describing the assumptions governing each occurrence of each subterm,
- rewrite each subterm in the appropriate context, using definitions, conditional rewrite rules, and metafunctions,
- use propositional calculus normalization to convert the resulting formula to an equivalent set of formulas, reduce the set under subsumption, and deposit the surviving formulas back in the pool.

By "conditional rewrite rules" we mean rules that cause the system to replace certain terms by other terms, provided certain hypotheses can be established. For example, the axiom `(equal (car (cons x y)) x)` gives rise to a rewrite rule in the data base that directs the system to replace every term of the form `(car (cons` α β`))` by α. The axiom `(implies (not (consp x)) (equal (car x) nil))` gives rise to a rule that directs the system to replace `(car` α`)` by `nil`, if the system can prove that `(consp` α`)` is false. When you command the system to prove a theorem and to store it as a rewrite rule, the system generates such a rule. The generated rule is sensitive to the exact syntactic form of the theorem.

ACL2 supports congruence-based rewriting: it supports "substitution of equivalents," not just substitution of `equal`s. That is, rewrite rules can be generated from theorems that conclude not just with an `equal`-term but an *equiv*-term, where *equiv* is an arbitrary user-defined equivalence relation. You may define special equivalence relations and prove congruence rules permitting substitution of equivalents rewriting deep inside of terms. See Section 8.3.1 and see also and `equivalence` `congruence`.

The simplifier above is not guaranteed to produce formulas that are stable under simplification; repeated trips through the simplifier, via insertion into and extraction from the pool, are used to reach the final stable form (if any).

When `Subgoal *1/3`, above, arrives at the simplifier, it is simplified in two successive steps. The first merely expands `(ENDP A)` to `(NOT (CONSP A))` and removes the double `NOT`'s, naming the resulting formula `Subgoal *1/3'`. (When a formula is transformed to exactly one other formula, the new formula is given the same name as the old one with a prime appended at the end.) When `Subgoal *1/3'` is put into the pool, it is immediately extracted and simplified. That simplification expands the recursive functions `TRUE-LISTP` and `REV` to produce `Subgoal *1/3''`, which is put into the pool.

Line 23 below shows `Subgoal *1/3` as it is drawn out of the pool and given to the simplifier. The message printed by the first simplification is on line 29. The formula produced follows that. The second simplification's message and output formula start at line 37.

```
23. Subgoal *1/3
24. (IMPLIES (AND (NOT (ENDP A))
25.               (EQUAL (REV (REV (CDR A))) (CDR A))
26.               (TRUE-LISTP A))
27.          (EQUAL (REV (REV A)) A)).
28.
29. By the simple :definition ENDP we reduce the conjecture to
30.
31. Subgoal *1/3'
32. (IMPLIES (AND (CONSP A)
33.               (EQUAL (REV (REV (CDR A))) (CDR A))
```

```
34.               (TRUE-LISTP A))
35.          (EQUAL (REV (REV A)) A)).
36.
37. This simplifies, using the :definitions TRUE-LISTP and REV,
38. to
39.
40. Subgoal *1/3''
41. (IMPLIES (AND (CONSP A)
42.               (EQUAL (REV (REV (CDR A))) (CDR A))
43.               (TRUE-LISTP (CDR A)))
44.          (EQUAL (REV (APP (REV (CDR A)) (LIST (CAR A))))
45.                 A)).
```

The last subgoal is immediately extracted from the pool and given to the simplifier, but the simplifier does not change it. It is stable and is passed to the next proof technique.

8.2.3 Destructor Elimination

Destructor elimination is a way to get rid of certain function applications by expanding certain variables into terms that make explicit their construction. For example, suppose a formula mentions `(CAR A)` and `(CDR A)`. If `A` is not a cons, those expressions simplify to `NIL`. If `A` is a cons, we could, without loss of generality, replace `A` by `(CONS A1 A2)`, for new variable symbols `A1` and `A2`. Doing so would allow us to get rid of `(CAR A)` and `(CDR A)`, replacing them, respectively, with `A1` and `A2`.

Here is the output produced when destructor elimination is applied to `Subgoal *1/3''` above.

```
47. The destructor terms (CAR A) and (CDR A) can be eliminated
48. by using CAR-CDR-ELIM to replace A by (CONS A1 A2),
49. generalizing (CAR A) to A1 and (CDR A) to A2.  This produces
50. the following goal.
51.
52. Subgoal *1/3'''
53. (IMPLIES (AND (CONSP (CONS A1 A2))
54.               (EQUAL (REV (REV A2)) A2)
55.               (TRUE-LISTP A2))
56.          (EQUAL (REV (APP (REV A2) (LIST A1)))
57.                 (CONS A1 A2))).
```

Observe the hypothesis on line 53 produced by replacing `A` in `(CONSP A)`. This hypothesis is now manifest in the construction of `A`. When `Subgoal *1/3'''` is put into the pool, it is immediately drawn out and simplified. Simplification eliminates this redundant hypothesis and otherwise changes nothing. The simplified goal is named `Subgoal *1/3'4'`. (The system will

not produce a name with more than three primes, but you can think of `'4'` as four primes, `'5'` as five primes, and so on.)

The transformation above is justified logically by the

Axiom. `CAR-CDR-ELIM`:

```
(implies (consp x)
         (equal (cons (car x) (cdr x)) x)).
```

This axiom is an example of a more general form:

```
(implies (hyp x)
         (equal (constructor (dest1 x) ... (destn x))
                x)).
```

Such theorems can be stored as "destructor elimination" or elim rules. See `elim`. The (*dest*$_i$ *x*) are the *destructor* terms. When destructor elimination is applied to a formula containing an instance of some (*dest*$_i$ *x*) in which the variable *x* is bound to some variable *a*, the technique applies. It "splits" the formula into two cases according to whether (*hyp a*) is true and in the case where it is true, it replaces all of the *a*'s in the formula (except those inside *dest*$_i$ applications) by (*constructor* (*dest*$_1$ *a*) ... (*dest*$_n$ *a*)). Then it generalizes all the (*dest*$_i$ *a*) terms (including the ones just introduced) to distinct new variable symbols, $a_1, \ldots, a_n$. In generalizing it restricts the a_i using generalization rules discussed below. The resulting formulas are put in the pool.

Here is another example of a destructor elimination rule. Suppose `firstn` and `nthcdr` are defined so that the following is a theorem.

```
(implies (and (integerp n)
              (<= 0 n)
              (<= n (len x)))
         (equal (append (firstn n x) (nthcdr n x))
                x))
```

This is in the form of a destructor elimination rule. The destructor terms are `(firstn n x)` and `(nthcdr n x)`. The constructor is `append`. Suppose the destructor elimination technique were applied to the formula `(p (firstn i a) (nthcdr i a) i a)`, *i.e.*, to a formula involving one or more suitable instances of the destructor terms. Then destructor elimination would split the conjecture into two subgoals.

```
(implies (not (and (integerp i)
                   (<= 0 i)
                   (<= i (len a))))
         (p (firstn i a) (nthcdr i a) i a))
(implies (and (and (integerp i)
                   (<= 0 i)
                   (<= i (len (append u v)))))
              ...)
         (p u v i (append u v)))
```

The first of these subgoals handles the "pathological" case where the destructors are being "inappropriately" used. To prove that subgoal it would be best to have rules to reduce (firstn i a) and (nthcdr i a) to other expressions in this case.

The second of these subgoals handles the "normal" case. Note that here a has been replaced by (append u v), and (firstn i a) and (nthcdr i a) have been replaced, respectively, by u and v. The "..." in the hypotheses of the second subgoal stand for hypotheses about u and v that are derived by the generalization technique from rules about (firstn i a) and (nthcdr i a). Observe that the effect of the elimination rule here is to eliminate firstn and nthcdr in favor of append, *i.e.*, to trade "destructors" for "constructors." Whether this is a good move in the context of a proof really depends on which rules are in the data base. We often arrange strategies based on rewriting patterns of constructors.

A more sophisticated destructor elimination rule is shown below.

```
(implies (acl2-numberp x)
         (equal (+ (mod x y) (* y (floor x y))) x))
```

In this theorem, (mod x y) and (floor x y) are the destructor terms and the constructor is the lambda expression (lambda (d1 d2 y) (+ d1 (* y d2))). When this rule is available and the destructor elimination technique is presented with a formula containing, say (MOD I J) or (FLOOR I J), the technique splits on whether I is a number. In the affirmative case, destructor elimination replaces I by (+ R (* J Q)), (MOD I J) by R, and (FLOOR I J) by Q. Provided there are appropriate generalization lemmas available for mod and floor, this eliminates the destructors mod and floor in favor of the constructors + and * without loss of generality.

The destructor elimination technique, like the simplifier, actually supports substitution of equivalents for equivalents rather than just substitution of equals for equals. See page 139 and elim.

8.2.4 Use of Equivalences

The next step in the waterfall attempts the use of equalities appearing in the goal formula. If the formula contains the hypothesis (equal *lhs rhs*) and elsewhere in the formula there is an occurrence of *lhs*, then *rhs* is substituted for *lhs* in every such occurrence with one exception: if *lhs* occurs on one side of an equality, we only substitute into that side of the equality. This restricted substitution method is called *cross-fertilization*. If the equality hypothesis comes from an inductive argument, we throw it away after using it in this fashion. We treat equalities symmetrically and, when it is possible to substitute left-for-right and right-for-left, make a choice based on heuristics.

ACL2 actually supports a more general form of substitution involving equivalence relations. The use of equalities is generalized to the use of any equivalence relation, *equiv*, and substitution is correspondingly restricted to *equiv*-hittable occurrences. See page 139.

Returning to the `rev-rev` proof, recall that destructor elimination produced `Subgoal *1/3'''`, which was further simplified to `Subgoal *1/3'4'`. That subgoal cannot be further simplified and has no destructors in it. Thus, ACL2 tries to use the equivalences in it. Below is the subgoal, the message printed by equivalence substitution, and the formula produced and added to the pool.

```
62. Subgoal *1/3'4'
63. (IMPLIES (AND (EQUAL (REV (REV A2)) A2)
64.               (TRUE-LISTP A2))
65.          (EQUAL (REV (APP (REV A2) (LIST A1)))
66.                 (CONS A1 A2))).
67.
68. We now use the first hypothesis by cross-fertilizing
69. (REV (REV A2)) for A2 and throwing away the hypothesis.
70. This produces
71.
72. Subgoal *1/3'5'
73. (IMPLIES (TRUE-LISTP A2)
74.          (EQUAL (REV (APP (REV A2) (LIST A1)))
75.                 (CONS A1 (REV (REV A2))))).
```

Observe the rather peculiar substitution chosen: A variable was replaced by a non-variable term on only one side of an `equal` even though the variable occurred on both sides.[2] The fact that this proof techniques discards a hypothesis makes it even more interesting. It is possible that the input formula is a theorem but the output formula is not. This does not mean that ACL2 is unsound! It means ACL2 may adopt a goal that it cannot prove. If the output formula is a theorem, the input formula is too. That is, the output may be more general than the input. Because we are likely, at this point in the waterfall, to appeal eventually to induction, ACL2's heuristics have been designed to try to strengthen the goal.

`Subgoal *1/3'5'` is added to the pool, extracted, and given in turn to simplification, destructor elimination, and equivalence substitution. None of them apply and so it arrives at the fourth technique.

[2]The substitution done by this heuristic may "undo" a previous rewrite or, as here, may appear to use the equality "backwards" (*i.e.*, not in the left-to-right sense imposed on any rewrite rule that may later be generated from the equality). While the rewriter gives special significance to the left/right orientation of certain equalities when generating rewrite rules, equalities in the formula being proved are used symmetrically.

8.2.5 Generalization

The fourth proof technique explicitly attempts to generalize the goal. The basic heuristic is to find a subterm that appears in both the hypothesis and the conclusion, in two different hypotheses, or on opposite sides of an equivalence, and replace that subterm by a new variable symbol. Furthermore, if type information (see type-prescription) or generalization rules (see generalize) can be used to restrict the type of the new variable, then it is so restricted. The generalized formula is then added to the pool.

Here is the contribution of generalization to the rev-rev proof.

```
72. Subgoal *1/3'5'
73. (IMPLIES (TRUE-LISTP A2)
74.          (EQUAL (REV (APP (REV A2) (LIST A1)))
75.                 (CONS A1 (REV (REV A2))))).
76.
77. We generalize this conjecture, replacing (REV A2) by RV.
78. This produces
79.
80. Subgoal *1/3'6'
81. (IMPLIES (TRUE-LISTP A2)
82.          (EQUAL (REV (APP RV (LIST A1)))
83.                 (CONS A1 (REV RV)))).
```

Observe that (REV A2) occurs on both sides of an equal. It is generalized to the new variable RV. This new subgoal is added to the pool. None of the proof techniques discussed so far are applicable to it and the prover arrives at the fifth technique. But before we discuss that technique we discuss the restrictions imposed by generalize rules.

In this example there are no applicable generalize rules. But suppose we had previously proved the theorem that REV preserves the length of its argument,

```
(equal (len (rev x)) (len x))
```

and stored it as a generalize rule. Then when (REV A2) is replaced by RV, the generalization heuristic would add the following additional hypothesis.

```
(EQUAL (LEN RV) (LEN A2))
```

What actually happens is that the applicable generalize rules are instantiated so as to contain the term being generalized (*e.g.*, x in the rule is replaced by A2 so the rule mentions (REV A2)); that instance of the rule is added as a hypothesis to the goal; finally, the target term, (REV A2), is replaced by a new variable, RV.

The same restrictions are imposed when destructor terms are eliminated by the introduction of new variable symbols.

8.2.6 Elimination of Irrelevance

The fifth proof technique is the last one before induction. It attempts to eliminate irrelevant hypotheses in the conjecture, by partitioning them into cliques according to the variables they mention. If it can find an isolated clique of hypotheses, then either the formula is a theorem because those hypotheses are collectively false, or else they are irrelevant. It uses type information (see page 145) in a trivial way to guess that a clique is not false.

This occurs when `Subgoal *1/3'6'`, above, arrives at the elimination of irrelevance. Here is the exchange:

```
80. Subgoal *1/3'6'
81. (IMPLIES (TRUE-LISTP A2)
82.          (EQUAL (REV (APP RV (LIST A1)))
83.                 (CONS A1 (REV RV)))).
84.
85. We suspect that the term (TRUE-LISTP A2) is irrelevant to
86. the truth of this conjecture and throw it out.  We will thus
87. try to prove
88.
89. Subgoal *1/3'7'
90. (EQUAL (REV (APP RV (LIST A1)))
91.        (CONS A1 (REV RV))).
```

Observe that the hypothesis, `(TRUE-LISTP A2)`, is irrelevant once `(REV A2)` is generalized to `RV`. `Subgoal *1/3'7'` is then put into the pool.

When it is drawn out and passed around, none of the first five proof techniques apply to it. Induction will be tried. The order in which formulas are drawn from the pool is irrelevant, since all must be proved. But the draw is so orchestrated that we do not try to prove a subgoal by induction until we have processed every subgoal produced by the last induction. For that reason, this subgoal `Subgoal *1/3'7'` is given a special name indicating that it is destined for induction and that it is the first such subgoal arising from the induction attempt on `*1`.

```
89. Subgoal *1/3'7'
90. (EQUAL (REV (APP RV (LIST A1)))
91.        (CONS A1 (REV RV))).
92.
93. Name the formula above *1.1.
```

Recall that we have been following the progress of the first subgoal produced by the induction on `*1`, namely `Subgoal *1/3`, the "interesting" induction step. The pool at this point contains two other formulas from that induction: `Subgoal *1/2` and `Subgoal *1/1`. Those two formulas are drawn out before induction is applied to `*1.1`. Both simplify to `t`.

8.2.7 The Rev-Rev Log

Here is the complete log of the `rev-rev` proof, starting with the user's input and annotated with line numbers. You should read it to acquaint yourself with its structure and remind yourself of the waterfall underlying this structure.

```
 1. ACL2 >(defthm rev-rev
 2.         (implies (true-listp a) (equal (rev (rev a)) a)))
 3.


 4. Name the formula above *1.
 5.
 6. Perhaps we can prove *1 by induction.  Two induction schemes
 7. are suggested by this conjecture.  These merge into one derived
 8. induction scheme.
 9.
10. We will induct according to a scheme suggested by (REV A).
11. If we let  (:P A) denote *1 above then the induction scheme
12. we'll use is
13. (AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A)))
14.               (:P A))
15.      (IMPLIES (ENDP A) (:P A))).
16. This induction is justified by the same argument used to
17. admit REV, namely, the measure (ACL2-COUNT A) is decreasing
18. according to the relation E0-ORD-< (which is known to be
19. well-founded on the domain recognized by E0-ORDINALP).  When
20. applied to the goal at hand the above induction scheme produces
21. the following three nontautological subgoals.
22.
23. Subgoal *1/3
24. (IMPLIES (AND (NOT (ENDP A))
25.               (EQUAL (REV (REV (CDR A))) (CDR A))
26.               (TRUE-LISTP A))
27.          (EQUAL (REV (REV A)) A)).
28.
29. By the simple :definition ENDP we reduce the conjecture to
30.
31. Subgoal *1/3'
32. (IMPLIES (AND (CONSP A)
33.               (EQUAL (REV (REV (CDR A))) (CDR A))
34.               (TRUE-LISTP A))
35.          (EQUAL (REV (REV A)) A)).
36.
37. This simplifies, using the :definitions TRUE-LISTP and REV,
38. to
39.
40. Subgoal *1/3''
41. (IMPLIES (AND (CONSP A)
```

```
42.               (EQUAL (REV (REV (CDR A))) (CDR A))
43.               (TRUE-LISTP (CDR A)))
44.          (EQUAL (REV (APP (REV (CDR A)) (LIST (CAR A))))
45.                 A)).
46.
47. The destructor terms (CAR A) and (CDR A) can be eliminated
48. by using CAR-CDR-ELIM to replace A by (CONS A1 A2),
49. generalizing (CAR A) to A1 and (CDR A) to A2.  This produces
50. the following goal.
51.
52. Subgoal *1/3'''
53. (IMPLIES (AND (CONSP (CONS A1 A2))
54.               (EQUAL (REV (REV A2)) A2)
55.               (TRUE-LISTP A2))
56.          (EQUAL (REV (APP (REV A2) (LIST A1)))
57.                 (CONS A1 A2))).
58.
59. This simplifies, using the :type-prescription rule REV and
60. primitive type reasoning, to
61.
62. Subgoal *1/3'4'
63. (IMPLIES (AND (EQUAL (REV (REV A2)) A2)
64.               (TRUE-LISTP A2))
65.          (EQUAL (REV (APP (REV A2) (LIST A1)))
66.                 (CONS A1 A2))).
67.
68. We now use the first hypothesis by cross-fertilizing
69. (REV (REV A2)) for A2 and throwing away the hypothesis.
70. This produces
71.
72. Subgoal *1/3'5'
73. (IMPLIES (TRUE-LISTP A2)
74.          (EQUAL (REV (APP (REV A2) (LIST A1)))
75.                 (CONS A1 (REV (REV A2))))).
76.
77. We generalize this conjecture, replacing (REV A2) by RV.
78. This produces
79.
80. Subgoal *1/3'6'
81. (IMPLIES (TRUE-LISTP A2)
82.          (EQUAL (REV (APP RV (LIST A1)))
83.                 (CONS A1 (REV RV)))).
84.
85. We suspect that the term (TRUE-LISTP A2) is irrelevant to
86. the truth of this conjecture and throw it out.  We will thus
87. try to prove
88.
89. Subgoal *1/3'7'
90. (EQUAL (REV (APP RV (LIST A1)))
```

```
 91.        (CONS A1 (REV RV))).
 92.
 93. Name the formula above *1.1.
 94.
 95. Subgoal *1/2
 96. (IMPLIES (AND (NOT (ENDP A))
 97.               (NOT (TRUE-LISTP (CDR A)))
 98.               (TRUE-LISTP A))
 99.          (EQUAL (REV (REV A)) A)).
100.
101. But we reduce the conjecture to T, by primitive type reasoning.
102.
103. Subgoal *1/1
104. (IMPLIES (AND (ENDP A) (TRUE-LISTP A))
105.          (EQUAL (REV (REV A)) A)).
106.
107. By the simple :definition ENDP we reduce the conjecture to
108.
109. Subgoal *1/1'
110. (IMPLIES (AND (NOT (CONSP A)) (TRUE-LISTP A))
111.          (EQUAL (REV (REV A)) A)).
112.
113. But simplification reduces this to T, using the :executable-
114. counterparts of REV, EQUAL and CONSP, primitive type reasoning
115. and the :definition TRUE-LISTP.
116.
117. So we now return to *1.1, which is
118.
119. (EQUAL (REV (APP RV (LIST A1)))
120.        (CONS A1 (REV RV))).
121.
122. Perhaps we can prove *1.1 by induction.  Two induction schemes
123. are suggested by this conjecture.  Subsumption reduces that
124. number to one.
125.
126. We will induct according to a scheme suggested by (REV RV).
127. If we let  (:P A1 RV) denote *1.1 above then the induction
128. scheme we'll use is
129. (AND (IMPLIES (AND (NOT (ENDP RV)) (:P A1 (CDR RV)))
130.                (:P A1 RV))
131.      (IMPLIES (ENDP RV) (:P A1 RV))).
132. This induction is justified by the same argument used to
133. admit REV, namely, the measure (ACL2-COUNT RV) is decreasing
134. according to the relation EO-ORD-< (which is known to be
135. well-founded on the domain recognized by EO-ORDINALP).  When
136. applied to the goal at hand the above induction scheme produces
137. the following two nontautological subgoals.
138.
139. Subgoal *1.1/2
```

```
140. (IMPLIES (AND (NOT (ENDP RV))
141.               (EQUAL (REV (APP (CDR RV) (LIST A1)))
142.                      (CONS A1 (REV (CDR RV)))))
143.          (EQUAL (REV (APP RV (LIST A1)))
144.                 (CONS A1 (REV RV)))).
145.
146. By the simple :definition ENDP we reduce the conjecture to
147.
148. Subgoal *1.1/2'
149. (IMPLIES (AND (CONSP RV)
150.               (EQUAL (REV (APP (CDR RV) (LIST A1)))
151.                      (CONS A1 (REV (CDR RV)))))
152.          (EQUAL (REV (APP RV (LIST A1)))
153.                 (CONS A1 (REV RV)))).
154.
155. But simplification reduces this to T, using the :definitions
156. REV and APP, primitive type reasoning and the :rewrite rules
157. CAR-CONS and CDR-CONS.
158.
159. Subgoal *1.1/1
160. (IMPLIES (ENDP RV)
161.          (EQUAL (REV (APP RV (LIST A1)))
162.                 (CONS A1 (REV RV)))).
163.
164. By the simple :definition ENDP we reduce the conjecture to
165.
166. Subgoal *1.1/1'
167. (IMPLIES (NOT (CONSP RV))
168.          (EQUAL (REV (APP RV (LIST A1)))
169.                 (CONS A1 (REV RV)))).
170.
171. But simplification reduces this to T, using the :definitions
172. REV and APP, primitive type reasoning, the :rewrite rules
173. CAR-CONS and CDR-CONS and the :executable-counterparts of
174. CONSP and REV.
175.
176. That completes the proofs of *1.1 and *1.
177.
178. Q.E.D.
179.
180. Summary
181. Form:  ( DEFTHM REV-REV ...)
182. Rules: ((:DEFINITION IMPLIES)
183.         (:ELIM CAR-CDR-ELIM)
184.         (:TYPE-PRESCRIPTION REV)
185.         (:DEFINITION TRUE-LISTP)
186.         (:EXECUTABLE-COUNTERPART EQUAL)
187.         (:DEFINITION NOT)
188.         (:DEFINITION ENDP)
```

```
189.          (:DEFINITION REV)
190.          (:EXECUTABLE-COUNTERPART CONSP)
191.          (:REWRITE CAR-CONS)
192.          (:EXECUTABLE-COUNTERPART REV)
193.          (:REWRITE CDR-CONS)
194.          (:FAKE-RUNE-FOR-TYPE-SET NIL)
195.          (:DEFINITION APP))
196. Warnings:  None
197. Time:  0.13 seconds (prove: 0.08, print: 0.03, other: 0.02)
198.
199. Proof succeeded.
200. ACL2 >
```

As we see from the proof above, the proof of `rev-rev` is not only inductive but it involves invention of an interesting lemma. The lemma was produced by simplifying the induction step of `rev-rev`, using destructor elimination, equality substitution, generalization, and elimination of irrelevance to arrive at:

```
 89. Subgoal *1/3'7'
 90. (EQUAL (REV (APP RV (LIST A1)))
 91.        (CONS A1 (REV RV))).
 92.
 93. Name the formula above *1.1.
```

Formula `*1.1` is a worthy fact. It says that if you append the singleton list containing `A1` to the right end of an arbitrary list, `RV`, and reverse the result, you get the same thing you would get if you consed `A1` onto the front of the reverse of `RV`. We do not know of a proof of `rev-rev` that does not make use of a comparable lemma about `rev` and `app`. Formula `*1.1` is proved by a second, automatically selected induction, starting on line 122.

8.3 Simplification Revisited

In this section we expand our description of the simplifier. On page 126 we described the simplifier as having four steps: decision procedures, establishing context, rewriting, and normalization and subsumption. Without doubt, rewriting is the most important aspect: successful use of the theorem prover requires successful control of the rewriter.

Without loss of generality, we assume the formula to which the simplifier is applied is of the form `(implies (and` $p_1 \ \ldots \ p_n$`)` q`)`. The p_i are the hypotheses and q is the conclusion.

Our presentation is organized as follows. First we discuss equivalence relations and congruence rules, since these are fundamental to several aspects of the simplifier. Then we will discuss each of the four steps (decision procedures, context, rewriting, and normalization and subsumption) in the order in which they occur.

8.3.1 Congruence-Based Reasoning

The most frequently used rule of inference in most proofs is the familiar idea of substitution of equals for equals. ACL2 supports a general form of substitution of equals for equals, based on the ideas of user-defined equivalence relations and congruence rules. The importance of user-defined equivalence and congruence rules is difficult to appreciate at first. They inherit their importance, in part, from the fact that ACL2 does not provide abstract data types. New kinds of objects must be represented in terms of existing ACL2 primitives, *e.g.*, sets are represented as lists. The operations on these objects are defined as functions on the existing data types, *e.g.*, the set operations are functions on lists that ignore duplication and order. Because such representations are often not unique, ACL2's equality predicate, `equal`, is too strong: it distinguishes objects that all the operations of the new type treat equivalently. Thus, one must define an equivalence relation on the new type and prove that the new operations respect this notion of equivalence. Once that is done, ACL2 can use that equivalence to substitute into nests of the new operations. Use of equivalence relations and congruence rules is fundamental to the simplifier as well as other proof techniques in the waterfall.

The need for generalized equivalence relations is easily seen by considering the representation of finite sets as linear lists. Thus, we might think of `(1 2 3)` and `(3 1 2 1)` as equivalent representations of the set $\{1, 2, 3\}$.

Suppose we define `(un a b)` to return (a representation of the) union of (the sets represented by) `a` and `b`. Thus, `(un '(1 2) '(3 4))` might be `(1 2 3 4)`. Is un commutative? Depending on how we actually define it, it may not be commutative. For example, `(un '(1 2) '(3 4))` might be `(1 2 3 4)` but `(un '(3 4) '(1 2))` might be `(3 4 1 2)`. These two objects are not `equal`. But we could define a relation, `(set-equal a b)`, that returns `t` or `nil` according to whether the set represented by `a` is the same as the set represented by `b`. Then we would have the theorem

```
(set-equal (un b a) (un a b)).
```

Suppose we use `mem` to test for "set membership," *i.e.*, `(mem e x)` is `t` or `nil` according to whether `e` is an element of `x`. Consider proving

```
(implies (mem e (un a b))
         (mem e (un b a))).
```

The "natural" proof is "use the commutativity of un to replace `(un b a)` by `(un a b)`." If asked to justify such a move, we might say "substitution of equals for equals." But the commutativity result we have for un is not an equality! What allows us to use it to replace `(un b a)` by `(un a b)` in our conjecture?

One might respond by observing that `set-equal` is an equivalence relation: it is symmetric, reflexive, and transitive. While true, that is not enough. For example, we cannot use commutativity to replace `(un b a)` by

`(un a b)` in `(equal (car (un a b)) (car (un b a)))` or else we could prove a non-theorem. It is important that `mem` respects `set-equal` in the sense that `(mem e x)` returns the same result as `(mem e y)` whenever `x` is `set-equal` to `y`. This is a *congruence rule* and might be phrased as

```
(implies (set-equal x y)
         (equal (mem e x) (mem e y))).
```

This congruence rule allows us to substitute `set-equals` for `set-equals` in the second argument of a `mem` expression, without changing the value of the `mem` expression. Given the congruence rule, we can use the commutativity rule—as stated in terms of `set-equal`—just as though it were an equality. That is, we can use it as a rewrite rule.

There is one final twist to discuss. The twist has to do with the fact that the congruence rule uses two difference senses of equality. In the rule above we see both `set-equal` and `equal`. In general, we might see any two equivalence relations here. For example, in Lisp the membership "predicate" is named `member`. But instead of returning `t` to indicate success, `(member e x)` returns the first tail of `x` that starts with `e`. This way `member` can be used to determine both whether and where `e` occurs in `x`. For this sense of membership, the congruence rule above does not hold. Let `x` be `'(1 2)` and `y` be `'(2 1)`. Let `e` be `1`. Then `(set-equal x y)` is true. But `(member e x)` returns `(1 2)` while `(member e y)` returns `(1)`. These two objects are not `equal`. But they are "propositionally equal" in the sense that they are `nil` or non-`nil` in unison. So we have another congruence rule.

```
(implies (set-equal x y)
         (iff (member e x) (member e y)))
```

This congruence rule allows the substitution of `set-equals` for `set-equals`, in the second argument of `member` expressions, while preserving `iff`. Put another way, if a `member` expression occurs in a position in which only its propositional value is important, then its second argument occurs in a position in which only its set value is important.

At the top of a conjecture, only the propositional value is important. As we explore the subterms occurring in the conjecture, we can use congruence rules to keep us appraised of which equivalence relations we must preserve to maintain top-level propositional equivalence.

A binary relation is known to be an equivalence relation if it has been proved Boolean, symmetric, reflexive, and transitive and those four facts have been marked as an `equivalence` rule. To prove and store the necessary rules, use the `defequiv` command. Some familiar relations that can be defined in ACL2 and proved to be equivalence relations are

- `(iff a b)`, expressing the idea that `a` and `b` are both `nil` or both non-`nil` (without requiring that they both be Boolean). For example, `t` is equivalent (modulo `iff`) to `'(1 2 3)`.

- `(equall a b)`, expressing the idea that two lists are equal except for the atom marking the end of the lists. Thus, `'(1 2 1 3)` and `'(1 2 1 3 . ABC)` are `equall`.
- `(perm a b)`, expressing the idea that lists `a` and `b` are permutations of one another. For example `'(a b c a c)` is a permutation of `'(a a b c c)`.
- `(alist-equal a b)`, expressing the idea that when `a` and `b` are regarded as association lists they assign the same value to every key. For example, `'((A . 1) (B . 2) (A . 3))` is `alist-equal` to `'((B . 2) (A . 1))`.
- `(set-equal a b)`, expressing the idea that lists `a` and `b` contain the same set of elements. For example, `'(a a b c c)` is `set-equal` to `'(b a c)`.

We say an occurrence of *lhs* in a formula is *equiv-hittable* if congruence rules are available to establish that the occurrence can be replaced by any equivalent (modulo *equiv*) term without changing the propositional value of the formula.

Congruence rules are derived from theorems of the form

```
(implies (equiv1 x y)
         (equiv2 (f ... x ...)
                 (f ... y ...)))
```

where $equiv_1$ and $equiv_2$ are known equivalence relations. To prove a congruence rule, use `defcong` and see also `congruence`. Such a congruence rule informs ACL2 that if `x` and `y` are equivalent (modulo $equiv_1$) then the result of replacing one by the other in the indicated argument position of f produces an equivalent term (modulo $equiv_2$). We sometimes characterize such a congruence by saying that it allows $equiv_1$ substitution (in the indicated argument position) into f while *preserving* $equiv_2$. One can thus justify a deep substitution by chaining together congruence rules, starting from a congruence rule that preserves `iff` (propositional equivalence). ACL2 can do such chaining, provided appropriate congruence rules are available for every relevant argument position of every relevant function symbol.

Generally speaking when you represent a new type of object (*e.g.*, sets as lists) you might consider introducing a corresponding equivalence relation. Then when you define the elementary operations on the "new" objects, *e.g.*, `mem` and `un`, you should consider proving the appropriate congruence rules. Here are the rules for `mem` and `un`.

```
(implies (set-equal x y)
         (iff (mem e x)
              (mem e y)))
```

```
(implies (set-equal x y)
         (set-equal (un x a)
                    (un y a)))
(implies (set-equal x y)
         (set-equal (un a x)
                    (un a y)))
```

The first says that propositional equivalence is preserved when set equivalence is preserved in the second argument of mem. The second says that set equivalence is preserved when set equivalence is preserved in the first argument of un. The third says that set equivalence is preserved when set equivalence is preserved in the second argument of un. These three rules may be conveniently expressed as shown below. The names of the variables are unimportant.

```
(defcong set-equal iff (mem e x) 2)
(defcong set-equal set-equal (un x a) 1)
(defcong set-equal set-equal (un a x) 2)
```

Defcong is defined as a macro that expands into a defthm form. This is a common use of macros. See defcong.

Now suppose that the rewriter encounters the term

(mem α
     (un (un β γ) δ))

while it is trying to preserve propositional equivalence. The three congruence rules tell the rewriter that it can replace β, γ, and δ (as well as the un-terms containing them) by set-equal terms. You should think of congruence rules as providing a road-map with which ACL2 can figure out the equivalence relations to preserve while rewriting given subterm occurrences.

Why bother? The knowledge that it is sufficient to preserve set-equal while rewriting, say, β, is only important if you have also proved rules that allow the rewriter to replace one term by a set-equal term. Here are some such rules.

```
(set-equal (un b a) (un a b))
(set-equal (un (un a b) c) (un a (un b c)))
(set-equal (un b nil) b)
```

These are just rewrite rules; they direct the system to replace the left-hand side by the right-hand side in set-equal-hittable contexts.

Congruence rules just permit these rules to be used.

Finally, it is possible that if you have several different equivalence relations, then some will refine others. For example, perm is a refinement of set-equal in the sense that if (perm a b) holds, then so does (set-equal a b). Thus, if an occurrence of a term is set-equal-hittable then it is also perm-hittable. That is, the system can use rules for perm and for set-equal when in a set-equal-hittable position. It is useful therefore to bring to the system's attention the refinement relations between your equivalence

relations. This is done by proving refinement rules; see defrefinement and refinement.

Having sketched the role of equivalence relations and congruence rules, we now return to the details of the simplification process. Recall that simplification proceeds in four steps: use of decision procedures, establishment of a context, rewriting, and normalization and subsumption.

8.3.2 Decision Procedures

When a formula is given to the simplifier three decision procedures are applied. The first is propositional calculus. The second is congruence closure, using equivalence relations. The third is rational linear arithmetic.

Propositional Calculus
The default propositional procedure is based on the normalization of `if` expressions: (a) propositional connectives are expanded in terms of `if`; (b) the `if` terms are distributed, so (*f* (`if` *a b c*)) becomes (`if` *a* (*f b*) (*f c*)) and (`if` (`if` *a b c*) *x y*) becomes (`if` *a* (`if` *b x y*) (`if` *c x y*)); and (c) the resulting tree is explored to determine whether every reachable tip is non-`nil`. The user may direct the system on a particular subgoal to use ordered binary decision diagrams [11, 32] instead. See bdd. Bdds are most effective on large propositional problems; we do not recommend using them until you are familiar with the rest of the system. ACL2 extends BDDs to cons trees, and involves term rewriting in their construction.

Congruence Closure
The congruence closure procedure uses the context to compute equivalence classes, chooses a canonical representative of every equivalence class (namely the lexicographically smallest term), and substitutes that representative for all members of the class into all function applications allowing it. For example, if (*equiv a b*) and (*equiv b c*) are known from the present context, then (*equiv a c*) is added to the context; and moreover, if *a* is lexicographically less than *b* and *c*, then *equiv*-hittable occurrences of *b* and *c* are replaced by *a*. This is done iteratively. See defequiv, defcong, and defrefinement to introduce an equivalence relation, congruence rule, or refinement.

Linear Arithmetic
The linear arithmetic procedure is a decision procedure for rational linear inequalities, *i.e.*, formulas made up of variables, constants, sums, differences, products of constants with variables, equalities, and inequalities. In this discussion we use the word "inequality" loosely to include (`equal x y`), since, when x and y are rational, that equality is equivalent to the conjunction of (`<= x y`) and (`<= y x`).

The procedure works by trying to derive a contradiction from the negation of the goal. The procedure organizes the inequalities from the conjecture into a *linear data base* and then combines them by cross-multiplication and addition so as to create new inequalities. The linear data base for a formula contains all the hypothesis inequalities and the negation of the conclusion inequality, if any. If a contradiction can be derived from this data base, the formula is a theorem of linear arithmetic. All function applications other than sums, differences, and products with constants are treated as variables.

For example, linear arithmetic can be used to prove the following.

```
(implies (and (< (* 3 a) (* 2 b))
              (<= (* 5 b) (+ (* 7 a) c)))
         (< a (* 2 c)))
```

The formula above would be proved the same way if `a`, `b`, and `c` were replaced by more complicated expressions.

Sometimes it is necessary to augment the linear data base with inequalities derived from theorems about other function symbols. For example, let `(pos e x)` be the position at which `e` occurs in the list `x` or the length of `x`, `(len x)`, if `e` does not occur. The following is a theorem but is not a consequence of linear arithmetic.

```
(implies (and (< 0 j)
              (< (* 2 (len a)) k))
         (< (pos e a) (+ k j)))
```

However if we add the hypothesis that `(<= (pos e a) (len a))`, the new formula is a consequence of linear arithmetic. You can bring such inequalities to the theorem prover's attention by proving linear rules.

A *linear rule* is a theorem that concludes with an inequality. If an instance of one of the terms mentioned as an addend in the inequality arises in the linear data base, the rule is instantiated so as to create that instance. If the hypotheses of the rule can be established, the instantiated inequality is added to the linear data base. The hypotheses are established by rewriting them, as described in the section on rewriting below. See **linear** for more details.

8.3.3 Context

If the formula is not proved by one of the foregoing procedures, the simplifier will rewrite each hypothesis and then the conclusion. Rewriting is done in a *context* that specifies what may be assumed true. When rewriting the conclusion, we assume all of the hypotheses. When rewriting a hypothesis, we assume the other hypotheses and the negation of the conclusion.

The context actually consists of two kinds of information, each of which is derived from the assumptions described above. One kind of information

is arithmetic in nature. The other is type theoretic. The former is derived from the arithmetic inequalities among the assumptions, and linear rules, as described above. The type theoretic information is discussed here.

Sometimes the construction of the context proves the theorem. For example, type information may establish that two hypotheses are contradictory. In this case, no rewriting is done and success is reported immediately.

Before we descend further into the derivation of this type theoretic information it should be noted that many successful users have only a vague idea of how this part of ACL2 works. This section will give you a fairly good model. But you should not be discouraged by its length: it is possible to use ACL2 successfully without pulling the levers described here.

A *type statement* is a claim that a term has a certain type. We might record the assumption `(orderedp a)` by making the type statement that "`(orderedp a)` has type non-`nil`." Similarly, we might record the assumption `(rationalp x)` with the statement that "`x` has type rational." What then do we mean by "type?" In the next section we explain what a "type" is and sketch how we deduce type information from assumptions.

However, the type deduction algorithm, called *primitive type reasoning* or *type set reasoning* in the theorem prover output, can be extended by two kinds of rules.

- Type-prescription rules allow you to inform the type algorithm of the type of the output produced by a function. A type-prescription rule about `orderedp` might assert that it is Boolean-valued. Then when we assume `(orderedp a)` we deduce that `(orderedp a)` has type `t` rather than the much larger type non-`nil`. See `type-prescription`.

- Compound-recognizer rules are applicable to Boolean-valued functions of one argument. These rules allow you to tell the system how to deduce type information about the argument. For example, a compound-recognizer rule might tell the type mechanism to deduce from `(orderedp a)` that `a` is a true list. See `compound-recognizer`.

We discuss these two types of rules after discussing types. Then we describe how we assemble our assumptions into a context for the rewriter.

Types

We partition the ACL2 universe into fourteen *primitive types*. A *type* is any union of these primitive types. The fourteen primitive types are: {`0`}, the positive integers, the positive ratios (*i.e.*, non-integer rationals), the negative integers, the negative ratios, the complex rationals, the characters, the strings, {`nil`}, {`t`}, the symbols other than `nil` and `t`, the proper conses (*i.e.*, conses that are true lists), the improper conses, and all others. Note that three primitive types contain only a single object: {`0`}, {`nil`}, and {`t`}

These primitive types are used to build familiar types. For example, the naturals are the union of {`0`} and the positive integers. The rationals

are the union of the first five primitive types listed. The ACL2 numbers are the union of the first six primitive types. The symbols are the union of {`nil`}, {`t`}, and the other symbols. The conses are the union of the proper and improper conses. The true lists are the union of {`nil`} and the proper conses. The type non-`nil` is the union of all the primitive types except {`nil`}.

The ACL2 user refers to the type of a term x by formulas about x composed from `0`, `<`, `integerp`, `rationalp`, `complex-rationalp`, `acl2-numberp`, `characterp`, `stringp`, `equal`, `null`, `nil`, `t`, `symbolp`, `consp`, `atom`, `listp`, and `true-listp`.[3] Such terms are called *type expressions* about x. In such expressions, x is called the *typed term*. For example, `i` is the typed term in the type expression `(and (integerp i) (< 0 i))`. A type expression about x can be turned into a unique statement of the form "x has type s." It is because of type expressions that the user does not necessarily need to understand the type system.

We can compute the intersection, union, and complement of types. The satisfiability of conjunctions, disjunctions, and negations of type expressions can be deduced by an obvious computation on the underlying types. For example, it is impossible for x to be both an integer and a symbol (because the intersection of the types is empty). But it is possible for x to be both an integer and a positive rational (indeed, such an x has type positive integer). Similarly, if x is a true list and a symbol, then x is `nil`.

When we say the context records type information about the assumptions we mean it records the strongest type expressions it can deduce from the assumptions.

Here is a simple example. Suppose we have the assumptions

1. `(integerp i)`
2. `(rationalp x)`
3. `(< 0 x)`
4. `(primep j)`
5. `(equal i x)`
6. `(equal a 'ABC)`
7. `(not (consp e))`

After processing the first, we know that `i` is an element of the integers, *i.e.*, a negative integer, zero, or a positive integer. After processing the next two we know that `x` is either a positive integer or a positive ratio. Upon processing assumption 4, in the absence of any information about `primep`, we know that `(primep j)` is non-`nil`. Upon processing assumption 5 we

[3] Of course, the user may use macros such as `>` that expand to calls of these functions.

know that (equal i x) has type t. In addition, we now know that i and x must have the same type, so we can intersect their types to create a new type for each: both i and x are positive integers. Upon processing assumption 6 we know that (equal a 'ABC) is t and we also know that a is a symbol other than nil and t. The type algorithm can make no other use of the information that a is ABC. After assumption 7 we know that e is in the union of all the primitive types except the two containing conses.

Type-Prescription Rules

Consider a function application, $(f\ a_1\ \ldots\ a_n)$, and a type expression, *expr*, about it. A type-prescription rule for f may be derived from a theorem of the form (implies (and $hyp_1\ \ldots\ hyp_n$) *expr*). When the type algorithm must deduce the type of a term, $(f\ a'_1\ \ldots\ a'_n)$, that is an instance of the typed term, it instantiates the rule accordingly and then attempts to determine, by type reasoning alone, that each instantiated hyp_i is true in the current context. If so, it deduces a type statement about $(f\ a'_1\ \ldots\ a'_n)$ from the instantiated type expression *expr* and conjoins (intersects) that type statement with the current context.

More than one type-prescription rule may be applicable and all are used and conjoined. See type-prescription for restrictions and details.

An example type-prescription rule is (true-listp (rev x)). It allows the type algorithm to deduce that the type of (rev (app a b)) is either nil or a proper cons. Another type-prescription rule is the following.

```
(implies (and (integerp i)
              (integerp j)
              (not (equal j 0)))
         (integerp (rem i j)))
```

It allows the type algorithm to deduce that the type of (rem a b) is the set of integers, provided, in the current context, the type of a is a subset of the integers and the type of b is a subset of the non-zero integers.

The theorem prover can deduce type-prescription rules at definition time. For example, it may be able to deduce that primep returns either t or nil. In that case the processing of assumption 4 above will tell us that (primep j) is t rather than merely non-nil.

Compound-Recognizer Rules

We frequently define application-specific "recognizers," *i.e.*, Boolean-valued functions of one argument that recognize certain kinds of objects. Examples are suggested by the terms (primep j) and (btreep a). Sometimes the truth or falsity of such expressions imply primitive type information about the argument. Perhaps when (primep j) is true it is known that j is a positive integer. Perhaps when (btreep a) is false it is known that a is a cons. (This would be the case, for example, if btreep were defined to return t on all atoms but put some restriction on conses.)

Suppose f is a unary Boolean function symbol, x is a variable symbol, and $expr_i$ is a type expression about x. A compound-recognizer rule may be derived from a theorem of one of the following forms.

- `(implies (`f x`)` $expr_1$`)`
- `(implies (not (`f x`))` $expr_1$`)`
- `(and (implies (`f x`)` $expr_1$`) (implies (not (`f x`))` $expr_2$`))`
- `(iff (`f x`)` $expr_1$`)`
- `(equal (`f x`)` $expr_1$`)`

When $(f\ a)$ is assumed true (or false) the rule may allow the deduction of the corresponding type information about a, depending on the parity of the assumption and available rules, in the obvious way. See `compound-recognizer` for restrictions and details.

A particularly useful compound recognizer rule in some applications is the one that represents the definition of what it is to be `booleanp`. The function is defined as follows.

```
(defun booleanp (x)
  (if (equal x t) t (equal x nil)))
```

Unless `booleanp` is disabled, the hypothesis `(booleanp` e`)` is expanded and splits the goal formula into two cases. But if `booleanp` is disabled, the fact that e is `t` or `nil` is hidden. However, by proving

```
(iff (booleanp x) (or (equal x t) (equal x nil)))
```

as a compound-recognizer rule and then disabling `booleanp`, the type information about e is made available without case splits.

Assembling the Context

To construct a context from some assumptions, we compute a linear data base from the inequalities among our assumptions, using linear rules as appropriate. We also compute type information from our assumptions, using type prescription and compound recognizer rules. We then elaborate the type information with forward chaining as described below.

A *forward chaining* rule can be constructed from virtually any theorem. A typical forward chaining theorem has the form

`(implies (and` p_1 ... p_n`)` q`)`.

We call p_1 the *trigger term*. The trigger term can actually be specified by the user and can be any term whatsoever. If an instance of the trigger term occurs in the current context and the corresponding instances of the p_i are all true in the current context, then the corresponding instance of q is represented as a type statement and added to the context. The process is iterated until no changes occur. Care is taken not to loop forever. See `forward-chaining` for restrictions and details.

8.3.4 Rewriting

Recall that if the simplifier cannot prove the formula with one of the decision procedures, then it rewrites each hypothesis and the conclusion, under a context derived for each as described above.

The simplifier sweeps across the formula calling the rewriter on each hypothesis and the conclusion in turn.[4] Though it hardly ever matters, the sweep is left to right and the rewritten hypotheses are used to construct the contexts for the unrewritten ones and the conclusion.

Here is a user-level description of how the rewriter works. The following description is not altogether accurate but is relatively simple and predicts the behavior of the rewriter in nearly all cases you will encounter.

If given a variable or a constant to rewrite, the rewriter returns it. Otherwise, it is dealing with a function application, $(f\ a_1\ \ldots\ a_n)$. In most cases it simply rewrites each argument, a_i, to get some a_i' and then "applies rewrite rules" to $(f\ a_1'\ \ldots\ a_n')$, as described below.

But a few functions are handled specially. If f is `if`, the test, a_1, is rewritten to a_1' and then a_2 and/or a_3 are rewritten, depending on type reasoning about whether a_1' is `nil`. If f is `equal` or a recognizer (such as `integerp`), type reasoning is tried after rewriting the arguments and before rewrite rules are applied to the call of f. Finally, if f is a lambda expression, (`lambda` $(v_1\ \ldots\ v_n)$ *body*), then rewriting is applied to *body* after binding each v_i to a_i'.

Now we explain how rewrite rules are applied to $(f\ a_1'\ \ldots\ a_n')$. We call this the *target term* and are actually interested in a given occurrence of that term in the formula being rewritten.

Associated with each function symbol f is a list of rewrite rules. The rules are all derived from axioms, definitions, and theorems, as described below, and are stored in reverse chronological order – the rule derived from the most recently proved theorem is the first one in the list. The rules are tried in turn and the first one that "fires" produces the result.

A rewrite rule for f may be derived from a theorem of the form

```
(implies (and hyp1 ... hypk)
         (equiv (f b1 ... bn)
                rhs))
```

where *equiv* is a known equivalence relation. Note that the definition of f is of this form, where $k = 0$ and *equiv* is `equal`. A theorem concluding with a term of the form (`not` $(p\ \ldots)$) is considered, for these purposes, to conclude with (`iff` $(p\ \ldots)$ `nil`). A theorem concluding with $(p\ \ldots)$, where p is not a known equivalence relation and not `not`, is considered to conclude with (`iff` $(p\ \ldots)$ `t`).

[4]Note the distinction between *simplification* and *rewriting* as we use the terms here. The former is a formula-level activity while the latter is a term-level activity. The former orchestrates the latter.

Such a rule causes the rewriter to replace instances of the *pattern*, (*f* b_1 ... b_n), with the corresponding instance of *rhs* under certain conditions as discussed below.

Suppose the target term occurs in an *equiv*-hittable position in the formula. Suppose in addition that it is possible to instantiate variables in the pattern so that the pattern matches the target. We will depict the instantiated rule as follows.

```
(implies (and hyp'1 ... hyp'k)
         (equiv (f a'1 ... a'n)
                rhs'))
```

To apply the instantiated rule the rewriter must establish its hypotheses. To do so, rewriting (and propositional calculus) is used recursively to establish each hypothesis in turn, in the order in which they appear in the rule. This is called *backchaining*. If all the hypotheses are established, the rewriter then recursively rewrites rhs' to get rhs''. Certain heuristic checks are done during the rewriting to prevent some loops. Finally, if certain heuristics approve of rhs'', we say the rule *fires* and the result is rhs''. This result replaces the target term.

Special Hypotheses

A few interesting special cases arise in the process of trying to establish the hypotheses.

The first special case is that hyp_i is an arithmetic inequality, *e.g.*, (< *u* *v*). In this case, the two arguments are rewritten, to u' and v', and then the linear arithmetic decision procedure is applied to (< u' v') using the linear data base in the context. During this process, new linear lemmas may be added temporarily to the data base, in support of the terms introduced in u' and v'. Rewrite rules are not applied to the hypothesis itself, (< u' v'), unless the linear procedure cannot decide it.

The second special case arises when the instantiated hypothesis hyp_i' contains *free variables*, that is, variables that do not occur in the pattern or in any previous hypothesis. A theorem illustrating this problem is

```
(implies (and (divides p q)
              (not (equal p 1))
              (not (equal p q)))
         (not (primep q))).
```

Interpreted as a rule, this means that we can rewrite `(primep q)` to `nil`, provided we can rewrite `(divides p q)` to `t`, `(equal p 1)` to `nil`, and `(equal p q)` to `nil`. When we apply this rule to the target `(primep a)`, we substitute *a* for `q` to make the pattern of the rule match the target. But note that we have not substituted anything for `p`. In this rule, `p` is a free variable. The partially instantiated hyp_1 is `(divides p a)`. Chances are this will not rewrite to true unless we substitute for `p` some term related to the formula we are trying to prove! So the system must guess a choice for

the free variables. It does this in a very weak way. It simply searches the type information in the current context, looking for a term that matches the partially instantiated hypothesis. That is, it tries to find terms that, when substituted for the free variables in hyp_i' produce a term that is explicitly assumed true in the current context. A hypothesis containing a free variable is not rewritten at all! If the system finds a way to instantiate the free variables of hyp_i', it instantiates subsequent hypotheses accordingly, and tries to establish them. It never backs up to consider other choices for the free variables.

The third special case is that the hypothesis is of one of three forms `(syntaxp` *hyp*`)`, `(force` *hyp*`)`, or `(case-split` *hyp*`)`. Hypotheses of the first form are not logical restrictions at all but pragmatic metatheoretic restrictions on when to use the rule. `Syntaxp` always returns `t`. But when the rewriter encounters such a hypothesis it evaluates the form inside the `syntaxp` to decide whether the rule should fire. See `syntaxp` for details. Hypotheses of the other two forms are logically restrictive. Both `force` and `case-split` are defined as the identity function, so, for example, `(force` *hyp*`)` is true if and only if *hyp* is true. But when the rewriter encounters a hypothesis marked with `force` or `case-split` it tries to establish it as above and if that fails it assumes *hyp* and goes on. It returns to prove *hyp* later. See `force` and `case-split` for details.

Heuristic Checks

Recall that the rewriter makes a few final checks before firing the rule. Two deserve mention here. If the rule is a function definition then firing is tantamount to "opening up" or "expanding" a call of the function. If the definition is recursive, care must be taken not to expand indefinitely. For example, something must prevent `(app a b)` from opening to introduce `(app (cdr a) b)` and that in turn opening to introduce `(app (cdr (cdr a)) b)`, and so on. The final heuristic check prevents that. The rewriter does not fire the rule if the rule is a recursive definition and the rewritten rhs, rhs'', fails certain tests. One test permitting firing is that the arguments to the rewritten recursive call already appear in the formula being proved by the simplifier. Another test permitting the firing is that the arguments be symbolically simpler. Occasionally these heuristics will disallow an expansion that is important to your proof. You must then explicitly direct the system to expand the function call. See `expand`.

The second noteworthy final check concerns rules like `(equal (f x y) (f y x))` that commute, or more generally permute, the arguments to a function. Care must be taken not to indefinitely permute the arguments using such a rule. The rewriter will not fire such a rule unless the rewritten right hand side occurs before the target term in a total ordering on ACL2 terms based on lexicographic comparisons. You may think of the system as using permutative rules only to swap arguments into alphabetical order. See `loop-stopper`.

Before we leave the rewriter a supremely important point must be made: Basically, *it just does what you tell it to do with your rewrite rules.* If you tell it to loop forever, by rewriting a to b, b to c, and c to a, then it will loop forever, or as long as the resources of time and memory allow.

8.3.5 Normalization and Subsumption

We now leave the rewriter and return to the level of simplification. The simplifier is working on some goal formula, (`implies` (`and` hyp_1 ... hyp_k) *concl*), by rewriting the parts, in turn. Let us assume it has just rewritten hyp_k.[5] Suppose the result is a term that involves some `if`-expressions. For simplicity, suppose the result is (`p` (`if` *a b c*)).

You might expect the simplifier to move on to *concl*, rewriting it in a context in which hyp_1, ..., (`p` (`if` *a b c*)) are assumed true. But that is not what it does.

Instead it first lifts the `if`-expressions out of the rewritten term and splits the problem into as many cases as there are paths through the `if`-expressions. In our simple case above there are two paths: a is true and (`p` b) is true, or a is false and (`p` c) is true. The simplifier then proceeds to rewrite *concl* under each such extension of the hypotheses. Thus, in the simple case above, *concl* is rewritten in two different contexts. In the first case, the context contains hyp_1, ..., a, and (`p` b). In the second case, it contains hyp_1, ..., (`not` a), and (`p` c). The process of simplifying a formula thus yields a set of formulas whose conjunction is equivalent to the original.

The simplifier tries to clean up the set of formulas by throwing out or combining certain of them. For example, if one formula is (`implies` p q) and another is (`implies` (`and` p r) q), then clearly we might as well just prove the former. Moreover, if one formula is (`implies` (`and` p r) q) and another is (`implies` (`and` p (`not` r)) q), then we might as well just prove (`implies` p q). This is called *subsumption/replacement.*

If the result of subsumption/replacement is a set containing a single formula that is identical to the input formula, then the simplifier does not apply and passes the formula on to destructor elimination.

If the result is the empty set of formulas, then the simplifier proved the input formula.

Otherwise, the simplifier deposits each of the formulas into the pool.

[5] The analogous processing is applied after the conclusion is rewritten, just as though the conclusion were negated and appeared as hypothesis $k + 1$ and `nil` was inserted as the new *concl*.

8.4 Comments

This completes our sketch of how the theorem prover works. Recall the remark made at the beginning of this chapter.

> As you read this chapter you may begin to get the idea that the machine does everything for you. This is not true. A more accurate view is that the machine is a proof assistant that fills in the gaps in your "proofs." These gaps can be huge. When the system fails to follow your reasoning, you can use your knowledge of the mechanization to figure out what the system is missing.

It is perhaps impressive that the theorem prover can prove `rev-rev` completely automatically. But the validity of the `rev-rev` formula is obvious to most programmers. Unfortunately, so is the "validity" of the same formula without the crucial `true-listp` hypothesis. The latter fact justifies the use of a mechanical theorem prover: you will probably find that you frequently believe in the validity of formulas that are not theorems! The former fact, that validity is often obvious to you, justifies the use of a theorem prover that tries to fill in the gaps in your arguments. In constructing complicated arguments in support of practical applications, you will toss off observations like `rev-rev` without giving them a second thought and sometimes the theorem prover, using all of its power, will be able to prove them.

9

How to Use the Theorem Prover

It is one thing to understand how a tool works. It is another to know how to use it to get a particular job done. This chapter begins to explain how to use the tool just described. Here are some key ideas to keep in mind.

- The theorem prover is automatic only in the sense that you cannot steer it once it begins a proof attempt. You can only interrupt it and abort.
- You are responsible for guiding it, usually by getting it to prove the necessary lemmas. Get used to thinking that it rarely proves anything substantial by itself.
- Never prove a lemma without also thinking of what kind of rules should be made from it. You *always* specify the kind of rules to produce from a lemma, even when you say nothing about rules. The command `(defthm` *name p*`)` means "prove formula *p*, give it the name *name*, and *make it a rewrite rule*." If you do not want a theorem to be turned into a rule use `(defthm` *name p* `:rule-classes nil)`.

Bear in mind that this chapter provides only some basic information for getting started. In particular, it does not describe a way to monitor the application of rewrite rules (see **break-rewrite**). However, the last section of this chapter does suggest an approach to finer-grained interaction with the system. Further assistance in using ACL2 may be found later in this part. We also recommend the tutorial [27], which shows the use of ACL2's predecessor (Pc-)Nqthm ([4]) on a non-trivial example in considerable detail.

9.1 The Method

There are many different styles among ACL2 users. Some users tend to exercise every feature of the system while others exercise as few as possible. We recommend and will teach a single high-level proof style in this book. As you become a more sophisticated user, you will discover and possibly adopt other features of ACL2.

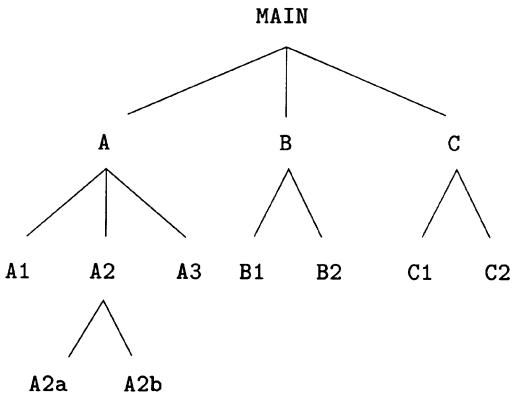

Figure 9.1: A Proof Tree

Prerequisite: To lead ACL2 to a proof you must know where the proof is. More generally, you must be able to prove theorems in this logic. Some find it helpful to practice hand proofs of simple ACL2 theorems. See the exercises in Chapters 6 and 7.1 and throughout the case studies in the companion book ([22]).

Imagine a full proof tree of some goal named `MAIN`, as depicted in Figure 9.1. `MAIN` is proved using the lemmas named `A`, `B`, and `C`, which themselves are proved with the lemmas indicated. To lead ACL2 to a proof of `MAIN`, you must ultimately prove every lemma in this tree. As a practical matter, you may not have worked out the proof of every lemma before you start to use ACL2. In fact, most users discover the structure of the proof tree by interacting with ACL2. Merely keeping track of the evolving tree, what has been proved and what remains to be proved is often daunting. We will describe one procedure, which will help you discover proofs that can be checked with ACL2. The goal of the procedure is to produce a sequence of `defthm` commands that lead ACL2 to a proof of the main theorem. The procedure will produce a postorder traversal of the tree. That is, using the procedure you will prove the lemmas of Figure 9.1 in the order: `A1`, `A2a`, `A2b`, `A2`, `A3`, `A`, `B1`, `B2`, `B`, `C1`, `C2`, `C`, `MAIN`. We note this only to make it obvious that there are many other styles one might follow. Furthermore, ACL2 contains proof structuring devices that allow you to structure the commands into the very tree shown, should you decide that is how you wish to present them. These devices are exploited, for example, in the top-down methodology presented by Kaufmann in his case study in the companion volume, [22]. But the fundamental problem is discovering the tree in the first place. ACL2 and this procedure can help you.

We use ACL2 in conjunction with a text editor. We prefer Emacs because we can run ACL2 as a process under Emacs and have the output piped into a buffer, called the *`*shell*` buffer*, that we can explore with Lisp-specific search and move commands. We generally prepare our input commands in a second buffer, called the *script buffer*.

When we are done, the script buffer will contain the postorder traversal of the proof tree for the main theorem. But during the project, the script is logically divided into two parts by an imaginary line we call the *barrier*. The part above the barrier consists of commands that have been carried out successfully by ACL2 in the `*shell*` buffer. The part below the barrier consists of commands that we intend to carry out. We sometimes refer to the first part as the *done* list and the second part as the *to-do* list.

Initially, the script buffer should contain the main theorem, p, written as a `defthm` command, *e.g.*, `(defthm main` p`)`. The barrier is at the top of the buffer, *i.e.*, the done list is empty and the to-do list contains just the main theorem.

Here is "The Method" often used to tackle a proof project.

1. Think about the proof of the first theorem in the to-do list. Structure the proof either as an induction followed by simplification or just simplification. Have the necessary lemmas been proved? That is, are the necessary lemmas in the done list already? If so, proceed to Step 2. Otherwise, add the necessary lemmas at the front of the to-do list and repeat Step 1.

2. Call the theorem prover on the first theorem in the to-do list and let the output stream into the `*shell*` buffer. Abort the proof if it runs more than a few seconds.

3. If the theorem prover succeeded, advance the barrier past the successful command and go to Step 1.

4. Otherwise, inspect the output of the failed proof attempt in the `*shell*` buffer, starting from the beginning, not the end. Basically you should look for the first place the proof attempt deviates from your imagined proof. Modify the script appropriately. We discuss this at length below. It usually means adding lemmas to the to-do list, just in front of the theorem just tried. It could mean adding hints to the current theorem. In any case, after the modifications go to Step 1. (We discuss a variant of Step 4 in Section 9.4.)

The most important part of The Method is the first word of Step 1. We use the term "The Method" partly in jest to poke gentle fun at the very idea that there is an algorithm for discovering proofs of deep and beautiful theorems. But The Method is a good starting point.

One might ask why we do not provide a user interface that supports The Method. No design we have ever contemplated was sufficiently flexible

to allow what really goes on in a major proof effort. We often evaluate small expressions typed directly into the `*shell*` to query the current world or test the behavior of defined functions. We often use Emacs to grab expressions from theorem prover output to help us create such tests or create appropriate lemmas for insertion into the script buffer. Failed proofs often reveal that some key goal is not a theorem. Modifications may then be necessary on both sides of the barrier and in the `*shell*`. Such considerations argue for an unrestricted interface used with discipline.

One might also criticize The Method as being too flat. A tree is being built but it is being encoded in a linear traversal. There is a good reason for this. Consider the proof tree of Figure 9.1. Often, lemmas `A`, `B`, and `C` involve the same collection of concepts. The lemmas necessary to prove `A`, *i.e.*, `A1`, `A2`, and `A3`, are often used in the proofs of `A`'s peers. The flat structure is often good because it allows sharing: all the lemmas necessary to prove a goal are available during the proof of that goal's subsequent peers. But there is a bad effect and you must look out for it: in large proofs the logical world gets so complicated you cannot control the simplifier.

For example, the strategy you adopt to prove one goal, *e.g.*, `A`, may conflict with the strategy you adopt for a subsequent one, *e.g.*, `B`. The most dramatic example of such a conflict is perhaps when the combined strategies simply loop indefinitely or cause catastrophic expansion. Perhaps your proof of `A` calls for `app`-nests to be right-associated but your proof of `B` calls for these nests to be left-associated. In successfully carrying out your plan for `A` you would introduce a rewrite rule, say `A1`, to right-associate `app`. Upon focusing your attention on `B`, you might prove `B1` to left-associate `app`. When the rewriter next encounters an `app` nest it will go into an infinite loop. Such loops will show up in The Method when rewriting fails to terminate. This often manifests itself by a stack overflow in the host Common Lisp. In some Lisps, such as GCL, a rewrite loop can manifest itself in a segmentation error (in which the Lisp process is aborted). A less dramatic example of conflicting strategies is when the rules for `A` simply transform the subgoals of `B` into forms that you find hard to recognize or reconcile with your intended proof.

The moral is: if you adopt conflicting strategies, it is best to be aware of it when you do it and localize the strategies to their intended goals. In small proofs (*e.g.*, those involving only dozens of rules) or in proofs whose structure you can keep clearly in mind, the simplest way to address this problem is to disable conflicting rules from prior goals before starting a new goal; this leaves those rules available should you ever need them again. See `in-theory`.

But in large proofs it is best, eventually, to use books (or encapsulation) to structure the proof, layering it appropriately, and isolating the proofs of independent peers. You might find, for example, that the lemmas at a given level in the tree, *e.g.*, lemmas `A1`, ..., `C2`, constitute a useful simplification strategy about a certain collection of concepts. You may choose to develop

a book containing all of those and then use the book to prove A, B, and C. You may also find that the proof of A requires additional "tactical" lemmas that conflict with those of B and so wish to isolate the two peers. Therefore, you might prove A in one book and B in another, importing the basic lemmas into each book and then augmenting the two worlds appropriately for their separate goals. The two books would then export only their main results, A and B, hiding the details of their proofs. C might be handled analogously. Then your proof of MAIN might end up as a very short script: three include-book commands bring in A, B, and C, and then the defthm for MAIN. See books for details. Encapsulation is an alternative to books that allows you to hide the proof of a given theorem without producing a corresponding file. See encapsulate.

We say "eventually" above because you should not be too rigid about introducing layers in a proof or else you may get lost in the layers! You must simply use good sense to structure your proofs.

A final criticism of The Method is that it is bottom-up. You find yourself proving the low-level lemmas for some sub-subgoal, like A2a, before you have seen A, B, and C mechanically assembled into MAIN. There is a reason for this. Very often—much more so than you might think—your formalization of the problem will be wrong, either in the sense that the defined concepts do not have properties you think they do or they are defined in an intractable way. By encouraging you to get your hands dirty and actually start using the definitions you have a chance to assess the whole plan in the back of your mind. It is possible to experiment in a top-down way, assuming formulas (and thus adding their rules to the world) and then using them to prove the main results. Sometimes this is useful. It often brings your attention to additional key lemmas omitted from your initial proof sketch. It more often highlights the need for many routine lemmas that will be discovered and proved in the natural course of events by The Method. See skip-proofs for details of how to assume that which must, ultimately, be proved, and remember that until it is proved you cannot be sure it is even valid! See also the case study by Kaufmann in the companion volume, [22], for a top-down methodology along these lines that has tool support, as well as Moore's case study in that volume for the use of a top-down macro.

When you are comfortable with ACL2 you will not use The Method as rigidly as it is described above, largely because it only produces flat proofs. You will probably use it by default to explore the problem, possibly adding unproved rules so the theorem prover can accompany you in your explorations. You will identify key layers or theories that need to be developed. You will recognize when it is important to do proofs in isolation. You will then use The Method, or your modification of it, to develop the appropriate scripts for each of the books you envision.

But ultimately you must learn how to build scripts that lead ACL2 to a given realistic goal from a reasonable but not quite perfect initial world. The Method is a good way to approach that problem.

9.2 Inspecting Failed Proofs

You will spend most of your time looking at ACL2 output, trying to figure out why ACL2 did not prove something that (a) you think is a theorem and (b) you think is now completely obvious given all the work ACL2 and you have already done. This section is intended to help you take advantage of that output.

In a nutshell, the most common activity is to focus on the first subgoal that ACL2 cannot simplify which does not ultimately get proved using other techniques. ACL2 provides a tool that automatically selects parts of the output on which you should probably focus; see `proof-tree`. However, in this section we focus directly on the linear output, in particular from a failed proof of the following theorem.

Theorem. `Main`

```
(equal (app (app a a) a)
       (app a (app a a)))
```

Note that the main theorem does not state that `app` is associative but states a weaker property. We try to prove this theorem in ACL2's initial world, right after defining `app`. In that world, the theorem that `app` is associative has not been proved.

The user who submits the following theorem is not following The Method because he or she could not have a proof in mind! Nevertheless, it is tempting to expect the theorem prover to do your thinking for you and this example should quickly disabuse you of that expectation!

```
 1. ACL2 >(defthm main
 2.         (equal (app (app a a) a)
 3.                (app a (app a a)))
 4.         :rule-classes nil)

 5.
 6. Name the formula above *1.
 7.
 8. Perhaps we can prove *1 by induction.  Three induction schemes
 9. are suggested by this conjecture.  Subsumption reduces that
10. number to one.
11.
12. We will induct according to a scheme suggested by
13. (APP A (APP A A)).  If we let  (:P A) denote *1 above then
14. the induction scheme we'll use is
15. (AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A)))
16.               (:P A))
17.      (IMPLIES (ENDP A) (:P A))).
18. This induction is justified by the same argument used to
19. admit APP, namely, the measure (ACL2-COUNT A) is decreasing
20. according to the relation E0-ORD-< (which is known to be
21. well-founded on the domain recognized by E0-ORDINALP).  When
```

```
22. applied to the goal at hand the above induction scheme produces
23. the following two nontautological subgoals.
24.
25. Subgoal *1/2
26. (IMPLIES (AND (NOT (ENDP A))
27.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
28.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
29.          (EQUAL (APP (APP A A) A)
30.                 (APP A (APP A A)))).
31.
32. By the simple :definition ENDP we reduce the conjecture to
33.
34. Subgoal *1/2'
35. (IMPLIES (AND (CONSP A)
36.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
37.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
38.          (EQUAL (APP (APP A A) A)
39.                 (APP A (APP A A)))).
40.
41. This simplifies, using the :definition APP, primitive type
42. reasoning and the :rewrite rules CDR-CONS and CAR-CONS, to
43.
44. Subgoal *1/2''
45. (IMPLIES (AND (CONSP A)
46.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
47.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
48.          (EQUAL (CONS (CAR A) (APP (APP (CDR A) A) A))
49.                 (APP A (CONS (CAR A) (APP (CDR A) A))))).
50.
51. This simplifies, using the :definition APP, primitive type
52. reasoning and the :rewrite rule CONS-EQUAL, to
53.
54. Subgoal *1/2'''
55. (IMPLIES (AND (CONSP A)
56.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
57.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
58.          (EQUAL (APP (APP (CDR A) A) A)
59.                 (APP (CDR A)
60.                      (CONS (CAR A) (APP (CDR A) A))))).
61.
62. The destructor terms (CAR A) and (CDR A) can be eliminated
63. by using CAR-CDR-ELIM to replace A by (CONS A1 A2),
64. generalizing (CAR A) to A1 and (CDR A) to A2.  This produces
65. the following goal.
66.
67. Subgoal *1/2'4'
68. (IMPLIES (AND (CONSP (CONS A1 A2))
69.                (EQUAL (APP (APP A2 A2) A2)
70.                       (APP A2 (APP A2 A2))))
```

```
          (EQUAL (APP (APP A2 (CONS A1 A2)) (CONS A1 A2))
                 (APP A2 (CONS A1 (APP A2 (CONS A1 A2))))))).

This simplifies, using primitive type reasoning, to

Subgoal *1/2'5'
(IMPLIES (EQUAL (APP (APP A2 A2) A2)
                (APP A2 (APP A2 A2)))
          (EQUAL (APP (APP A2 (CONS A1 A2)) (CONS A1 A2))
                 (APP A2 (CONS A1 (APP A2 (CONS A1 A2))))))).

We generalize this conjecture, replacing (APP A2 (CONS A1 A2))
by L and (APP A2 A2) by AP and restricting the type of the new
variable L to be that of the term it replaces, as established
by primitive type reasoning and APP.  This produces

Subgoal *1/2'6'
(IMPLIES (AND (CONSP L)
              (EQUAL (APP AP A2) (APP A2 AP)))
          (EQUAL (APP L (CONS A1 A2))
                 (APP A2 (CONS A1 L)))).

Name the formula above *1.1.

Subgoal *1/1
(IMPLIES (ENDP A)
          (EQUAL (APP (APP A A) A)
                 (APP A (APP A A)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP A))
          (EQUAL (APP (APP A A) A)
                 (APP A (APP A A)))).

But simplification reduces this to T, using the :definition
APP and primitive type reasoning.

So we now return to *1.1, which is

(IMPLIES (AND (CONSP L)
              (EQUAL (APP AP A2) (APP A2 AP)))
          (EQUAL (APP L (CONS A1 A2))
                 (APP A2 (CONS A1 L)))).

Perhaps we can prove *1.1 by induction.  Four induction schemes
are suggested by this conjecture.  Subsumption reduces that
```

```
... ⟨ 497 lines deleted ⟩

616. Subgoal *1.1.2.5/1'''
617. (CONSP A2).
618.
619. We suspect that this conjecture is not a theorem.  We might
620. as well be trying to prove
621.
622. Subgoal *1.1.2.5/1'4'
623. NIL.
624.
625. Obviously, the proof attempt has failed.
626.
627. Summary
628. Form:  ( DEFTHM MAIN ...)
629. Rules: ((:TYPE-PRESCRIPTION APP)
630.         (:ELIM CAR-CDR-ELIM)
631.         (:REWRITE CONS-EQUAL)
632.         (:REWRITE CDR-CONS)
633.         (:REWRITE CAR-CONS)
634.         (:FAKE-RUNE-FOR-TYPE-SET NIL)
635.         (:DEFINITION NOT)
636.         (:DEFINITION ENDP)
637.         (:DEFINITION APP))
638. Warnings:  None
639. Time:  0.65 seconds (prove: 0.43, print: 0.21, other: 0.01)
640.
641. ****** FAILED ******  See :DOC failure  ****** FAILED ******
```

On lines 1–4 we issue the **defthm** command to prove our main theorem. Lines 5–641 contain the theorem prover's response. The proof attempt fails and we have deleted much of it for brevity. But many points about the theorem prover can be made from this unsuccessful attempt to prove a simple theorem.

- It is not "smart enough" to prove even this simple theorem without help from the user!
- The output is produced in real time. It describes an ongoing proof attempt, not a proof.
- At 60 lines per page this failed attempt produced more than 10 pages of output.
- Unless you are piping the output into a scrollable window, text editor, or file, most of it is lost.
- From the summary at the bottom (line 639) we see that it takes only 0.65 seconds to produce this output, a rate of about 15 pages a

second. During this time the system does four successive inductions. You cannot read it fast enough to steer it.

- Do not spend much time reading the theorem prover's output unless it fails or runs for a "long time."
- A "long time" in this setting is several seconds.
- Ridiculously long subgoal names indicate an unsuccessful strategy.
- `Subgoal *1.1.2.5/1'4'` (line 622) is `nil` and the proof attempt stops. This does not mean the original formula was not a theorem! In fact, we know the original formula is a theorem. Failure simply means the system could not prove it. Sometimes the system's search strategy will lead it to try to prove subgoals that are manifestly false. When that happens, it fails.
- The system stops automatically in this example. But it does not always stop: it can "run forever" or exhaust physical resources on your machine. That is why the output is produced in real time and you should pay cursory attention to it. We often let it scroll by at full speed and abort when the subgoal numbering gets deep.

Suppose you were confronted with this failed proof attempt. Given The Method, the appropriate response is to begin to read the output *from the top*. Do not start reading the output at `Subgoal *1.1.2.5/1'4'`.

On line 6 the system gives the formula the temporary name `*1` and at line 8 begins an inductive proof. Recall the waterfall (page 123). The system only tries induction when all else fails. Is this a theorem to be proved by induction?

The induction message (lines 8–23) is one of two main *checkpoints* in the output and it should always raise a red flag when you see it. It is crucial that you not read past an induction argument until you are convinced that induction is the appropriate mathematical technique for the goal at hand. Perhaps the goal can be proved by appeal to other lemmas.

If you decide induction is plausible—most likely because you have an informal proof in mind—you must next determine whether the particular scheme chosen is an appropriate one. Read lines 15–17.

The system is inducting on the structure of `A`. The base case is line 17, when `A` is not a cons. The induction step is on lines 15–16. Assuming `A` is a cons and the formula holds for `(CDR A)`, the system will attempt to prove the formula.

A little experience with induction will teach you that this is probably not going to work. By "experience" we do not mean experience with the mechanization of induction in ACL2. We mean experience with the mathematical technique. The mathematically experienced user would stop reading upon seeing that the system was trying to prove this theorem by

induction. That is simply the wrong attack and everything else that follows is pointless. We will see why soon.

However, let us assume that we do not see anything wrong with this inductive attack and just continue reading the output.

`Subgoal *1/2` (lines 25–30) is the inductive step, printed with our particular formula rather than in schematic form. Following that are three successive simplifications, `Subgoal *1/2'`, `Subgoal *1/2''`, and `Subgoal *1/2'''`, each obtained from the former by opening up function definitions and applying a few axioms. The last is just a simplification of the first.

- Simplification is good. Skip past it, for now.
- The first message that mentions destructor elimination, use of equalities, cross-fertilization, generalization, irrelevant terms, or induction should raise a red flag. Simplification has done all it can. The formula just above that message is the most important checkpoint in the output. We call it the *simplification checkpoint*. The important thing about the formula at the simplification checkpoint is that it is stable under simplification.

The crucial message appears at line 62, when the system reports that it will try to eliminate destructors. The formula just before that, `Subgoal *1/2'''`, is as simple as the goal is going to get with simplification alone. You know that destructor elimination and whatever else the system will try will ultimately fail: you are reading a failed proof! So you must figure out how to prove `Subgoal *1/2'''` (lines 55–60).

Study the conjecture at the checkpoint. Ask yourself

- Is the formula even valid? Perhaps your "theorem" is not a theorem.
- If it is valid, why? Sketch a little proof.
- Which theorems are used in that little proof that are not in the logical world?
- If all the theorems you need are in the world, why were they not applied? There are three common answers:
  - The pattern of a key rule does not match what is in the formula being proved. Perhaps another rule fired, messing up the pattern you expected.
  - Some hypothesis of the rule cannot be established.
  - The rule is disabled.
- If there is a missing theorem, is it suspiciously like the one you are trying to prove? If so, perhaps the wrong induction was done or the formula you are trying to prove is too weak.

- On the other hand, if there is a missing theorem and it is different from the one you are trying to prove, then you have probably identified a key lemma. Most often, such lemmas are about new combinations of functions from the original theorem and the functions introduced by rewriting.

- Can you phrase the missing theorem as a rewrite rule so that the checkpoint goal simplifies further, ideally to true?

If answers come to you, repair the script accordingly and proceed with Step 1 of The Method.

In the example being discussed, no proof of `Subgoal *1/2'''` comes to mind. The only missing lemma that might come to mind is the associativity of `app`. But suppose we do not think of it. What else should we do? Since we know we are in an induction, we ought to figure out how to use the induction hypothesis (lines 56–57) to prove the induction conclusion (lines 58-60).

But there is no way to use the hypothesis in the conclusion. They do not match up. For instance, look at the left-hand side of each.

hypothesis: `(APP (APP (CDR A) (CDR A)) (CDR A))`
conclusion: `(APP (APP (CDR A) A) A)`

The first `(CDR A)` in the one matches the corresponding `(CDR A)` in the other. But the next two `(CDR A)`'s are mismatched with `A`'s. Things are even more mismatched on the right-hand side. In this situation you should

- contemplate whether there are rewrite rules that would allow the rewriter to transform the conclusion into (something involving) the hypothesis.

Again, none come to mind.

The offending `(CDR A)`'s came into the hypothesis from our choice of inductive instance. This suggests we should choose a different induction hypothesis.

Evidently, an appropriate induction hypothesis could be obtained if we could substitute `(CDR A)` for the *first* `A` in the conjecture and *leave the other two* `A's alone`. That is, the left-hand side of the "induction hypothesis" we wish we had is `(APP (APP (CDR A) A) A)`.

But that is not legal! Induction must uniformly replace the induction variable by something smaller. To get the instance we need, we must distinguish the first `A` from the other `A`'s on the left-hand side.

Therefore, we are proving the wrong theorem! We should be proving a theorem in which some of the `A`'s here are replaced by some other variable or variables.

The experienced user of the theorem prover would not read past the induction checkpoint in this example. Even the novice should not read past the simplification checkpoint. In fact, most of the time, the problem will be evident at the simplification checkpoint.

However, returning to our example, we briefly explain what the system does after the simplification checkpoint. In this case, it is not very enlightening. At line 62 it begins to eliminate the CARs and CDRs by renaming A to be a cons of two other variables. After a little simplification, it generalizes (line 82), producing `Subgoal *1/2'6'`. Nothing more can be done for that goal, so the system gives it the temporary name `*1.1` and will ultimately try to prove it by induction. Unfortunately, inspection of `*1.1` will reveal that it is not a theorem. None of this should be surprising since we know we are proving the wrong theorem! In fact, generalization often produces goals that are not theorems.

Meanwhile, the system had another top-level subgoal to prove: the base case of the induction, `Subgoal *1/1` (lines 95–98). The proof of the base case proceeds without difficulty and finishes by line 108.

So on line 110 the system attempts to prove the doomed `*1.1`. It tries induction. We have deleted almost 500 lines of output. The system tried two more inductions before finally producing a subgoal that was manifestly false.

Many readers are swamped by the output of ACL2. They feel obliged to read it all. That is a waste of your time. Read it until you understand why the proof attempt failed; better yet, read it until you understand why the system deviated from your intended proof. There was a remote chance when `main` was submitted that the system would somehow hit upon a generalization that it could prove. The whole exercise only costs 0.65 seconds so we see no harm in letting it run; cycles are free and it is virtually impossible, given human reaction time, to stop it more quickly. But do not read more than you need!

Now, recall what we have learned: we are trying to prove the wrong theorem. We need a formula like

```
(equal (app (app a a) a)
       (app a (app a a)))
```

except with the first `a` of the left-hand side distinguished from the next two. A candidate formula is

```
(equal (app (app a b) b)
       (app a (app a a)))
```

but of course that is not a theorem (it is easy to construct a counterexample exploiting the fact that `b` occurs on only one side). We need some `b`'s on the right and the "obvious" modification is

```
(equal (app (app a b) b)
       (app a (app b b)))
```

We cannot be more specific in our guidance or justification of these ideas. They are the result of mathematical insight gained simply by doing proofs and understanding what the function app does.

The formula above is provable by induction. The proof is enlightening because the problems raised in the last attempt are so completely solved by this strange little formula. However, you probably recognize that the associativity of app is a still more general formula that would do the job.

- It is almost always best to prove the most general theorems you can state conveniently.

Therefore, if we were following The Method we would type

```
(defthm associativity-of-app
  (equal (app (app a b) c)
         (app a (app b c))))
```

into the script buffer just in front of `main`. We would be cognizant of the fact that the rule generated from this theorem will right-associate all app nests as long as it is enabled.

Virtually every time we type a theorem that will become a rewrite rule we give thought to the termination of the rewriting scheme we are evolving.

- This is most often done by selecting a normal form for all the terms that arise in the problem and "orienting" every rule to drive the pattern (left-hand side of the conclusion) closer to the normal form.

- Furthermore, every subexpression in the pattern should be written in the selected normal form because the pattern is matched *after* the arguments of the target have been rewritten. If you are rewriting `(g x)` to `(h x)` then a rewrite rule with a pattern (left-hand side) of `(f (g x))` will never be used: the pattern will not be seen because the `g` in a potential target term will be rewritten to `h`!

- Sometimes no single normal form is adequate. In that case we choose an arbitrary normal form and *stick to it.* We do not just orient the rules randomly.

- When we need a lemma that relates terms not in normal form or that, when used as a rewrite rule, would drive a term out of normal form, we either specify `:rule-classes nil` or immediately disable it, so as not to introduce a loop in the rewriter. Such lemmas must be specified explicitly in subsequent `:use` hints (see `hints`) or enabled with attention paid to the conflicting rules.

This kind of thinking in connection with your rewrite rules is absolutely crucial to using The Method successfully. Time and time again the answer to vexing questions about why ACL2 failed to find a proof can be traced back to the failure to follow the four items of advice above.

Recall that ACL2 supports congruence-based reasoning, as described in Section 8.3.1.

- Most rewrite rules conclude with an `equal`, but do not forget that you can use `iff` and your own equivalence relations.

- If you intend to rewrite with equivalence relations, be sure to prove the necessary congruence rules.

Having added `associativity-of-app` to the script, it becomes the first theorem in the to-do list. We go to Step 1 of The Method: Think. The proof of the associativity of `app` is just a standard induction down the cdr of `a` followed by simplification using the definition and axioms. We admit that in simple cases like this it is reasonable to skip Step 1 and just throw the proposed theorem at the theorem prover. But if such a proof attempt does not succeed quickly then we revert to Step 1.

So we are ready for Step 2: command ACL2 to try. It produces a proof (not shown here) in 0.05 seconds.

- Skim successful proofs cursorily looking for the highlights: induction and simplification, or just simplification, as you expected.

Why read successful proofs? It can happen that the system finds a different proof than the one you are expecting. This could be a sign that something is seriously wrong, *e.g.*, a hypothesis is false because it was entered incorrectly. It could also be a sign that your model of the logical world is not accurate. You cannot afford *not* to understand the proof strategy you are constructing!

- If the system's proof involves an unexpected induction, it is a sign that a crucial lemma may be missing or inapplicable. While the system could fill in the gap this time, you might really need that lemma for some other theorem.

- If you expected an induction and there was none, then there are already sufficient lemmas in the world to prove this theorem. Their use is reported in the proof. Study them.

- If the system's proof was entirely by rewriting, you might not need this lemma as a rewrite rule. Earlier rules may always do the rewriting for you.

- If rewriting participated in the proof before the first induction or other techniques were used, you might want to contemplate whether there is a conflict between the rewrite rules used and the rewrite rule (if any) introduced by this lemma. Evidently, existing rules transform this one. Will its pattern ever match anything if those existing rules fire? Perhaps now that you have proved this lemma you can disable the earlier rules.

Having proved associativity-of-app we follow The Method by advancing the barrier and repeating Step 1: we submit the next command, which is main again. We expect it to succeed by simplification. This time we get:

```
 1. ACL2 >(defthm main
 2.         (equal (app (app a a) a)
 3.                (app a (app a a)))
 4.         :rule-classes nil)

 5.
 6. By the simple :rewrite rule ASSOCIATIVITY-OF-APP we reduce
 7. the conjecture to
 8.
 9. Goal'
10. (EQUAL (APP A (APP A A))
11.        (APP A (APP A A))).
12.
13. But we reduce the conjecture to T, by primitive type reasoning.
14.
15. Q.E.D.
16.
17. Summary
18. Form:  ( DEFTHM MAIN ...)
19. Rules: ((:REWRITE ASSOCIATIVITY-OF-APP)
20.         (:FAKE-RUNE-FOR-TYPE-SET NIL))
21. Warnings:  None
22. Time:  0.02 seconds (prove: 0.00, print: 0.00, other: 0.02)
23. MAIN
```

9.3 Another Example

Here is another illustration of The Method, with more focus on inspecting output to find missing theorems. Suppose we have added the definitions of app and rev and have proved

```
(defthm rev-rev
  (implies (true-listp a)
           (equal (rev (rev a)) a))).
```

Note that rev-rev is a conditional rewrite rule. Our goal is to prove

```
(defthm main
  (equal (rev x)
         (if (endp x)
             nil
           (if (endp (cdr x))
               (list (car x))
```

```
              (cons (car (rev (cdr x)))
                    (rev
                     (cons (car x)
                           (rev (cdr (rev (cdr x))))))))))
  :rule-classes nil).
```

This is an ugly-looking theorem! But it expresses a surprising fact about `rev`. Indeed, as first observed to one of us by Rod Burstall, you could use this equality as a definition of reverse: it is a way to define how to reverse a list without using the auxiliary function **append**. Here we will not try to admit this as a definition or add it as an alternative definition (see definition) but just focus on proving it as a theorem about the `rev` we have already defined. We make the theorem have `:rule-classes nil` because we do not want `(rev x)` to be rewritten this way.[1]

Step 1 is to sketch a proof. If either of the two conditions tested in the `if`-nest is true, the formula is easy to prove. Otherwise, we have to look at the messy `(cons (car (rev (cdr x))) (rev (cons ...)))` expression. We know that the theorem prover will simplify the `(rev (cons ...))` by expanding the definition of `rev`. Rather than work out the expansion by hand, we just call the theorem prover on the goal above, expecting (correctly) that the system will fail to prove the theorem. This is an important and common use of the system and its verbose output: submit formulas merely to see how your current rules simplify them. Just do not be misled into letting the system dictate your strategy.

When this formula is submitted, the system runs for three seconds. We ignore all but the first few lines of output. The first simplification checkpoint is:

```
Subgoal 3''
(IMPLIES (AND (CONSP X)
              (CONSP (REV (CDR X)))
              (CONSP (CDR X)))
         (EQUAL (APP (CDR (REV (CDR X))) (LIST (CAR X)))
                (APP (REV (REV (CDR (REV (CDR X)))))
                     (LIST (CAR X))))).
```

Two things should catch your eye. The first is `(CONSP (REV ...))`. The system does not know that `rev` returns a cons precisely when its argument is a cons. In the goal above, it helps little to know this, since it just removes a true hypothesis. But we can anticipate that the negation of this hypothesis will be considered later (indeed, it characterizes `Subgoal 2`). So we might as well "teach" the theorem prover how to simplify this question. We add to our to-do list the following theorem.

[1] Readers interested in logic might consider whether it is even possible to admit such a function under the ACL2 principle of definition. See the case study by Moore in [22].

```
(defthm consp-rev
  (equal (consp (rev x))
         (consp x)))
```

This is a rewrite rule that replaces any instance of `(consp (rev x))` by the corresponding instance of `(consp x)`. This lemma will be proved by induction followed by simplification. If we consider its proof, then we anticipate needing a lemma about `(consp (app ...))`, but expect the system will generate it. We return to this point below.

Meanwhile, the second thing that should catch your attention in the subgoal above is the `(REV (REV (CDR (REV ...))))` term. How did this term escape being hit by `rev-rev`? The only possible answer is that the theorem prover could not establish the hypothesis, in this case, `(true-listp (CDR (REV ...)))`. We could prove that directly, but a better strategy is to break it up into two observations, which we expect the system to chain together to get the needed conclusion.

```
(defthm true-listp-rev
  (true-listp (rev x)))
(defthm true-listp-cdr
  (implies (true-listp x)
           (true-listp (cdr x))))
```

The first will be proved by induction followed by simplification (with a lemma about `true-listp` and `app`) and the second by simplification.

It is not a good idea to add too many rewrite rules at once, without studying their interaction, so we now proceed with Step 1 again, for the three new lemmas. We have already considered their proofs—or perhaps we have not, but our intuition is that the proofs are likely to be easy for the theorem prover to find—and so take Step 2 and submit the commands. All three succeed, as expected.

So now we submit `main` again, simply to see the effect of our new rules on our checkpoint. In fact, the system proves `main` by simplification. Looking more closely at `Subgoal 3''`, above, we might note that once `rev-rev` is applied, the concluding equality is an identity. The other two subgoals deal with the special cases when `x` is empty or a singleton.

More can be said of the three lemmas we added. First, we anticipated needing some lemmas above but expected ACL2 to bridge the gap. That can be a frustrating strategy. It is better to be disciplined about adding the rules you think the system will need. To prove `consp-rev`, a disciplined user might first prove

```
(defthm consp-app
  (iff (consp (app a b))
       (or (consp a) (consp b))))
```

The question `(consp (app ...))` will come up in a proof about `(consp (rev ...))` because `rev` expands to `app`. The theorem prover happens

to generates a sufficient lemma about app to prove consp-rev. But that lemma is not added to the data base—only *you* add things to the data base. The next time (consp (app ...)) comes up, you might not be so lucky! It is better to arrange for it to simplify away.

The form of both consp-app and consp-rev is surprising to many new users. Weaker theorems that come to mind (for the app case) are as follows.

```
(implies (consp a) (consp (app a b)))
(implies (consp b) (consp (app a b)))
(implies (not (consp (app a b))) (not (consp a)))
(implies (not (consp (app a b))) (not (consp b)))
```

No single one of these four rules captures the logical content of consp-app. Even then, consp-app is pragmatically stronger because it eliminates the term (consp (app a b)) and introduces a case split. The four rules above allow ACL2 to settle certain consp questions, but only if they are raised by other rules. In addition, having all four rules would be very inefficient because they all cause backchaining.

- Use unconditional rewrite rules when possible.

The true-listp-rev theorem is another example where the disciplined user would have first proved a lemma about true-listp and app before expecting to prove something about true-listp and rev. The lemma we would choose is

```
(defthm true-listp-app
  (equal (true-listp (app a b))
         (true-listp b)))
```

to simply eliminate the question without backchaining.

Finally, consider true-listp-cdr. It too causes back chaining. Logically speaking, we could strengthen it to an unconditional rewrite rule.

```
(defthm true-listp-cdr
  (equal (true-listp (cdr x))
         (or (atom x) (true-listp x))))
```

But pragmatically, we do not want this rule because it rewrites in the opposite direction of the definition of true-listp. That is, the definition replaces (true-listp x) by something involving (true-listp (cdr x)); the strengthened rule above "undoes" that expansion. Introducing the strengthened rule above would cause the rewriter to loop indefinitely. We could have, alternatively, used the strengthened rule and disabled the definition of true-listp. But we tend to leave recursive functions enabled so that they unwind properly in inductive proofs.

9.4 Finer-Grained Interaction: The "Proof-Checker"

We have seen that using The Method may involve the inspection of ACL2 output from a failed proof attempt. Such inspection often leads the user to discover useful rewrite rules to prove, problems with the theorem prover's choice of induction scheme, or even modifications to make in the statement of the alleged theorem.

In this section we introduce an interactive utility, the *proof-checker*. This tool can help you discover why a proof attempt failed, by providing a number of ways to guide a new proof attempt interactively. We illustrate the proof-checker by using the example of the preceding section. See `proof-checker` for details. An important aspect of the proof-checker is that it can free you from much of the need to inspect the theorem prover's output stream. In fact we elide some prover output in the example presented below.

First we make a critical point. Although the proof-checker can free you from the need to anticipate necessary lemmas, the downside is that you can be lulled into the false notion that you no longer need to pay attention to Step 1 of The Method: *Think*. Experience can provide a sense for when the conjecture is somehow sufficiently simple that one can rely on guidance provided by a proof-checker session. We say more on this issue in remarks at the end of this section.

We begin the interactive proof session by using `verify`, which gives us the prompt "`->:`". All user input to the proof-checker is shown below on lines starting with this prompt.

```
ACL2 !>(verify
        (equal (rev x)
               (if (endp x)
                   nil
                 (if (endp (cdr x))
                     (list (car x))
                   (cons (car (rev (cdr x)))
                         (rev
                          (cons (car x)
                                (rev (cdr (rev (cdr x))))))))))))
->:
```

Although our goal is to let the proof-checker be an active assistant, we think just a little about how to proceed. Do we want to start the proof with induction, or should we simplify first? The prover starts with simplification, of course, but often it seems clear that the argument is fundamentally an inductive one and one issues the proof-checker `induct` command first. See `acl2-pc::induct`.[2] We are welcome to follow Step 1 of The Method and

[2] In some documentation, in particular the Emacs Info version, the proof-checker commands are prefixed by "`acl2-pc||`" rather than "`acl2-pc::`". All proof-checker commands are documented in the section `proof-checker-commands`.

work out the proof outline before we start, which tells us whether to start with induction or simplification. In this case, however, we use our intuition and start with simplification, expecting it to bring to light some missing rewrite rules that may be useful.

Hence, we begin by issuing the command **bash**, which calls the simplifier. All goals that would normally be put into the pool (see Section 8.2) are instead presented to the user as new goals to be proved.

```
->: bash
***** Now entering the theorem prover *****

[[[theorem prover output omitted here]]]

Creating two new goals:  (MAIN . 1) and (MAIN . 2).

The proof of the current goal, MAIN, has been completed.
However, the following subgoals remain to be proved:
  (MAIN . 1) and (MAIN . 2).
Now proving (MAIN . 1).
->:
```

What has happened? The proof-checker maintains a stack of named goals. This stack initially contains a single goal called **MAIN** which is the formula to be proved, *i.e.*, the argument of **verify**. The message above indicates that **MAIN** has been removed from the stack but new goals **(MAIN . 1)** and **(MAIN . 2)** have been pushed onto the stack. The top goal on the stack is the one being proved, in this case, the goal named **(MAIN . 1)**.

We can inspect all the goals (see acl2-pc::print-all-goals), but instead we start by looking at the current goal using the command **th** (which is mnemonic for "theorem"). Notice that a goal consists of hypotheses and a conclusion.

```
->: th
*** Top-level hypotheses:
1. (CONSP X)
2. (CONSP (REV (CDR X)))
3. (CONSP (CDR X))

The current subterm is:
(EQUAL (APP (CDR (REV (CDR X))) (LIST (CAR X)))
       (APP (REV (REV (CDR (REV (CDR X)))))
            (LIST (CAR X))))
->:
```

We now inspect this goal as we might have inspected a checkpoint from a failed proof in the preceding section. We notice a term in the conclusion of the form **(REV (REV ...))**. Of course, we are surprised to see such a term, since we vaguely recall having already proved a rule for it.

Before finding that rule, we first "move" to the subterm in question. The astute reader may have noticed the labeling of the conclusion above: "The current subterm." In fact the proof-checker maintains a pointer to a subterm of each goal's conclusion, which points initially to the entire conclusion (as above). But the proof-checker command `dv` ("dive") allows us to move that pointer. The command `(dv 2 1)` moves the pointer from the top of the conclusion to the second argument of the `EQUAL` and then to the first argument of that (second) `APP`.

```
->: (dv 2 1) ;; or, type 2 and then 1
->: p ;; print the current subterm
(REV (REV (CDR (REV (CDR X)))))
->: p-top ;; show entire conclusion, highlighting current subterm
(EQUAL (APP (CDR (REV (CDR X))) (LIST (CAR X)))
       (APP (*** (REV (REV (CDR (REV (CDR X)))))
                 ***)
            (LIST (CAR X))))
->:
```

Now that we are pointing to the `(REV (REV ...))` term, we can explore the issue of rewriting this term.

```
->: sr ;; or, show-rewrites
1. REV-REV
  New term: (CDR (REV (CDR X)))
  Hypotheses: ((TRUE-LISTP (CDR (REV (CDR X)))))
->:
```

Evidently the rule `REV-REV` does apply, as we might expect. Let us direct the proof-checker to apply this rewrite rule.

```
->: p
(REV (REV (CDR (REV (CDR X)))))
->: r ; or, rewrite; or, (rewrite 1); or, (rewrite rev-rev)
Rewriting with REV-REV.

Creating one new goal:  ((MAIN . 1) . 1).
->: p
(CDR (REV (CDR X)))
->:
```

Aha! The rewrite succeeded in simplifying the `(REV (REV ...))` term, but it created a new goal.

```
->: goals ;; display the names on the goal stack

(MAIN . 1)
((MAIN . 1) . 1)
(MAIN . 2)
->:
```

Let us see if the current goal can be simplified, preferably to `t`. Note that we move the pointer to the top of the conclusion before calling `bash` once again.

```
->: top ;; move to the top of the conclusion
->: bash
***** Now entering the theorem prover *****

[Note:  A hint was supplied for our processing of the goal
above.  Thanks!]

But we reduce the conjecture to T, by primitive type reasoning.

Q.E.D.


The proof of the current goal, (MAIN . 1), has been completed.
However, the following subgoals remain to be proved:
  ((MAIN . 1) . 1).
Now proving ((MAIN . 1) . 1).
->:
```

So far, so good; but we have not yet discovered any useful rewrite rules to prove. The new goal, obtained from the hypothesis of the rule `rev-rev` applied above, is about to suggest such a rule.

```
->: th
*** Top-level hypotheses:
1. (CONSP X)
2. (CONSP (REV (CDR X)))
3. (CONSP (CDR X))

The current subterm is:
(TRUE-LISTP (CDR (REV (CDR X))))
->:
```

If we attempt to use the `bash` command to simplify (or prove) this goal, the prover is stumped.

We have been led somewhat more directly than in the preceding section to the need for rules `true-listp-rev` and `true-listp-cdr`. We temporarily exit the proof-checker in order to prove these rules (which are shown on page 172).

```
->: exit ;; return to ACL2 top-level
Exiting....
 NIL
ACL2 !>
```

After proving the two rewrite rules mentioned above, we re-enter the proof-checker session and attempt to prove the goal above. Notice that with no arguments, `verify` re-enters the previous proof-checker session.

```
ACL2 !>(verify)
->: bash
***** Now entering the theorem prover *****

[Note:  A hint was supplied for our processing of the
goal above.  Thanks!]

But simplification reduces this to T, using the :rewrite
rules TRUE-LISTP-CDR and TRUE-LISTP-REV.

Q.E.D.


The proof of the current goal, ((MAIN . 1) . 1), has
been completed, as have all of its subgoals.
Now proving (MAIN . 2).
->: goals

(MAIN . 2)
->:
```

There is one goal remaining.

```
->: th
*** Top-level hypotheses:
1. (CONSP X)
2. (NOT (CONSP (REV (CDR X))))

The current subterm is:
(NOT (CONSP (CDR X)))
->:
```

The second hypothesis contains a term that could be simplified with an appropriate rewrite rule. We have thus been led by the proof-checker to discovery of the rule `consp-rev`, which was discovered with somewhat less guidance in the preceding section (see page 172). Again we exit the proof-checker in order to prove this rule and subsequently re-enter the proof-checker using `(verify)`. Then we may issue the `bash` command as before. This time we rather arbitrarily use `prove`, which invokes the full power of the prover and either proves the goal completely or causes no change to the proof-checker state.

```
->: prove
***** Now entering the theorem prover *****
```

```
But we reduce the conjecture to T, by the simple :rewrite
rule CONSP-REV.

Q.E.D.

*!*!*!*!*!*!* All goals have been proved! *!*!*!*!*!*!*
You may wish to exit.
->:
```

The proof has succeeded! We have the option of creating a **defthm** event at this point, which will have associated :**instructions** recording the commands we gave in the interactive session. However, we prefer to avoid these low-level :**instructions** in order to increase the likelihood that a proof will replay if modifications are made to our proof script. We exit the proof-checker loop and the proof now succeeds automatically.

```
->: exit
Exiting....
 NIL
ACL2 !>(defthm main
         (equal (rev x)
                (if (endp x)
                    nil
                  (if (endp (cdr x))
                      (list (car x))
                    (cons (car (rev (cdr x)))
                          (rev
                           (cons (car x)
                                 (rev (cdr (rev (cdr x))))))))))
         :rule-classes nil)

But simplification reduces this to T, using the :definitions
APP, ENDP and REV, primitive type reasoning and the :rewrite
rules CAR-CONS, CDR-CONS, CONSP-REV, REV-REV, TRUE-LISTP-
CDR and TRUE-LISTP-REV.

Q.E.D.

Summary
Form:  ( DEFTHM MAIN ...)
Rules: ((:DEFINITION APP)
        (:DEFINITION ENDP)
        (:DEFINITION REV)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS)
        (:REWRITE CONSP-REV)
        (:REWRITE REV-REV)
        (:REWRITE TRUE-LISTP-CDR)
```

```
        (:REWRITE TRUE-LISTP-REV))
Warnings:  None
Time:  0.06 seconds (prove: 0.02, print: 0.03, other: 0.01)
 MAIN
ACL2 !>
```

The proof-checker provides many other commands; see `proof-checker-commands`. The advanced user has the option of extending the proof-checker's power by defining compound commands, called *macro commands*, that can use non-trivial control structures. More importantly, among the basic capabilities not illustrated above are:

- undo (`undo` and `restore`);
- choose the goal to consider next (`cg`, `change-goal`);
- substitute equals for equals (=), or more generally, using equivalence relations (`equiv`);
- use induction (`induct`) to replace the current goal by goals for base and induction steps;
- consider cases (`casesplit`, `claim`, `split`);
- use binary decision diagrams for (primarily) propositional reasoning (`bdd`);
- manipulate the hypotheses (`contradict`, `demote`, `promote`, `drop`);
- invoke steps of the waterfall (`elim`, `generalize`; see also `use`);
- set the current theory (`in-theory`);
- expand function calls (`expand`, `x`, `x-dumb`);
- simplify the current subterm (`s`, `s-prop`, `sl`);
- use forward chaining (`forwardchain`); and
- save sessions (`ex`, `save`; see also `retrieve`).

As was done here, the proof checker is often used to find rules that will ultimately be used in an "automatic" proof. This mixing of the two proof engines can be extremely helpful: use the proof checker to explore proofs but record your strategies as general rules for the theorem prover. We often use the proof checker on unproved subgoals extracted from failed proofs. No sign of this appears in the final list of theorems.

Notice that the worked example in this section did not lead us to create the lemma `consp-app` of the preceding section (see page 172). The preceding section emphasized *thinking* about proof strategies, which for example could lead one to discover the lemma `consp-app`. The assistance given by

the proof-checker frees the user from some of that thinking, but as a result we did not find the lemma `consp-app`, a lemma that might turn out to be useful in later proofs even though it was not needed for this one. There is clearly a trade-off here between thinking and letting the system do some of the work. Individual style and experience will govern the extent to which one uses the proof-checker (or theorem prover, for that matter) to avoid some thinking, or uses thinking to avoid some potential interaction entirely. Experience suggests that new users tend to err on the side of doing insufficient thinking, which is why we have stressed Step 1 of The Method throughout most of this chapter. However, proof-checker interaction can provide a balance in the trade-off as one gains experience with ACL2.

10

Theorem Prover Examples

This chapter contains several examples and their solutions. Each section will start with an English description of a problem or will contain phrases such as "Why?" and "Prove the following." When you reach such a phrase, we recommend that you stop and work out a solution before reading further. Usually this will require that you define functions, translate informal correctness criteria into ACL2, and perhaps prove (on paper) the main theorem. Once you have a pencil and paper proof, think about how to decompose the proof into ACL2 rules and use the theorem prover to check your proof; you can then compare your results with ours.

We will take the "brain-dead" approach to using ACL2, *i.e.*, we will not think about how to structure our proofs, but rather, we will blindly try to prove theorems and will then react to ACL2's responses. This approach works fine for small examples and gives us the opportunity to stumble onto (and discuss) important issues.

Each section contains a set of related problems whose solutions are developed independently of the other sections in a new ACL2 session, *i.e.*, in a `ground-zero` theory (see `theories`). You are encouraged to evaluate the functions we define and to experiment with other approaches to solving the exercises. The examples will consist of proving two versions of the factorial function equivalent, proving a theorem about *, proving insertion sort correct, proving some theorems about functions that manipulate trees, proving an adder and multiplier correct, and proving a compiler for a stack-based machine correct. As a point of reference, an ACL2 expert can solve all of the problems in this chapter in about half a day.

To make the presentation more concise, we will present only the relevant parts of the output produced by ACL2 (prefaced by a line number). Therefore, it may help to run ACL2 while going through our solutions.

10.1 Factorial

Define the factorial function; define a tail recursive version; prove the two are "equivalent."

Since functions in ACL2 are total, we have to define the factorial function on any possible input. The standard way of dealing with this is to

M. Kaufmann et al., *Computer-Aided Reasoning*

coerce any value outside the intended domain to a base element, which in this case is 0. Notice the use of zp to test for 0 (and non-naturals); see zp and zero-test-idioms.

```
(defun fact (x)
  (if (zp x)
      1
    (* x (fact (- x 1)))))
```

For the tail recursive version, we introduce a variable that contains a partial product.

```
(defun tfact (x p)
  (if (zp x)
      p
    (tfact (- x 1) (* x p))))
```

We can test the functions to see if they seem to be the same (tfact is given an initial partial product of 1).

```
ACL2 >(fact 10)
3628800
ACL2 >(tfact 10 1)
3628800
```

If we try to evaluate fact on large numbers, we will get an error (which can look as follows in GCL).

```
Error: Invocation history stack overflow.
```

This occurs because the underlying Common Lisp places limits on the sizes of its stacks. One way to alleviate this problem is to compile the functions (see comp), which we can do as follows.

```
:comp (fact tfact)
```

Because tfact is tail-recursive, the recursion is replaced by iteration. This greatly eases the restriction on computation imposed by the size of the invocation stack and makes it possible to execute the function on large numbers.

```
;; Note: Tail-recursive call of TFACT was replaced by iteration.
```

To prove that fact and tfact are "equivalent," we will prove the following theorem.

```
(defthm fact=tfact
  (equal (tfact x 1) (fact x)))
```

Submitting fact=tfact to ACL2 produces the following. We only show the simplification checkpoint (see 165).

```
42. Subgoal *1/2''
43. (IMPLIES (AND (INTEGERP X)
44.               (< 0 X)
45.               (EQUAL (TFACT (+ -1 X) 1)
46.                      (FACT (+ -1 X))))
47.          (EQUAL (TFACT (+ -1 X) X)
48.                 (* X (FACT (+ -1 X))))).
```

The same goal may be obtained without perusing ACL2 output by instead entering the proof-checker with

```
(verify
  (equal (tfact x 1) (fact x)))
```

and issuing the command `(then induct bash)`. See page 174. We use ACL2 without the proof-checker in the present chapter.

That the proof attempt failed will come as no surprise to readers experienced in constructing proofs by induction because they will notice that the above theorem needs to be strengthened: opening `tfact` will change the second argument from a constant to `x`, and no induction hypothesis will match this new term (one cannot substitute for a constant). This is what happens on `Subgoal *1/2''`, above.

We strengthen the theorem so that instead of a `1` in `fact=tfact`, we have a variable.

```
(defthm fact=tfact-lemma
  (equal (tfact x p)
         (* p (fact x))))
```

ACL2 fails to prove `fact=tfact-lemma`. If we look at the simplification checkpoint, we see:

```
67. Subgoal *1/2'4'
68. (IMPLIES (AND (INTEGERP X) (< 0 X))
69.          (EQUAL (* X P (FACT (+ -1 X)))
70.                 (* P X (FACT (+ -1 X))))).
```

What lemma is suggested by the failed proof attempt? Notice that the conclusion of `Subgoal *1/2'4'` has the form `(equal (* A B C) (* B A C))`; this is a "simple" theorem about multiplication, *i.e.*, it is a theorem we expect elementary school students to know. Since such theorems are very useful to have around, ACL2 experts have written several arithmetic books. Some of these books, *e.g.*, `top-with-meta`, are part of the ACL2 source code distribution, but are not loaded into the initial data base. If we load `top-with-meta`, then ACL2 "knows" the above theorem. We use include-book to load `top-with-meta` (which probably resides in a different directory in your filesystem).

```
(include-book "/local/acl2/books/arithmetic/top-with-meta")
```

We will discuss how to prove the above theorem without the use of books in the next section.

If we then submit `fact=tfact-lemma` to ACL2, we get:

```
60. Subgoal *1/1.2'
61. (IMPLIES (NOT (INTEGERP X))
62.          (ACL2-NUMBERP P)).
63.
64. We suspect that this conjecture is not a theorem.  We might
65. as well be trying to prove
66.
67. Subgoal *1/1.2''
68. NIL.
69.
70. Obviously, the proof attempt has failed.
```

This output tells us exactly why `fact=tfact-lemma` is not a theorem. `Subgoal *1/1.2'` suggests that `fact=tfact-lemma` is false if `x` is not an integer and `p` is not a number. In fact, if `x` is not an integer, then `tfact` will return `p`, which can be anything, but `(* p (fact x))` is always an ACL2 number (*i.e.*, `(acl2-numberp (* x y))` is a theorem). We have made the classic mistake of strengthening the theorem to the point where the resulting term is not true. Our advice on using induction is: simplify and strengthen as much as possible, but no more. We try the following.

```
(defthm fact=tfact-lemma-for-acl2-numberp
  (implies (acl2-numberp p)
           (equal (tfact x p)
                  (* p (fact x)))))
```

ACL2 proves this lemma, which is stronger than `fact=tfact`, hence, ACL2 can easily prove the main result.

```
1. (defthm fact=tfact
2.   (equal (tfact x 1) (fact x)))
3.
4. But simplification reduces this to T, using the :type-prescription
5. rule FACT, the :definition FIX, the :rewrite rules FACT=TFACT-
6. LEMMA-FOR-ACL2-NUMBERP and UNICITY-OF-1 and primitive type
7. reasoning.
8.
9. Q.E.D.
```

There are other ways of defining a tail recursive version of `fact`, *e.g.*,

```
(defun tfact2 (x p)
  (if (zp x)
      (fix p)
    (tfact2 (- x 1) (* x p)))).
```

With this definition, we can prove the following theorem, which does not have a hypothesis:

```
(thm (equal (tfact2 x p)
            (* p (fact x)))).
```

10.2 Associative and Commutative Functions

In the previous section we were confronted with the following failed proof attempt:

```
67. Subgoal *1/2'4'
68. (IMPLIES (AND (INTEGERP X) (< 0 X))
69.          (EQUAL (* X P (FACT (+ -1 X)))
70.                 (* P X (FACT (+ -1 X))))).
```

We used the `top-with-meta` book to bypass the problem; in this section we will prove the required theorem without the use of books. It turns out that what we do is applicable to any associative and commutative function.

You might think that `*` is a function symbol, but if you type `:pe *` (see pe and history), you see that `*` is really a macro. Since macros are just syntactic sugar, *i.e.*, they are expanded into expressions, you may wonder why ACL2 insists on printing `(BINARY-* X (BINARY-* P (FACT (+ -1 X))))` as `(* X P (FACT (+ -1 X)))`. The answer is that internally, ACL2 manipulates the translated version of the term, but as a service to the user of the theorem prover, it "pretty prints" the term. Certain other macros are also treated this way, *e.g.*, `+` and `append`. See also `macro-aliases-table`.

Why does ACL2 get stuck on `Subgoal *1/2'4'`? This situation is one that you may encounter with any associative and commutative function. We start by asking: what rewrite rules do I need in order to put a function that is commutative and associative into canonical form?

What about `(equal (* x y) (* y x))`? Strictly speaking this is a bad rewrite rule because if applicable, it will keep on rewriting a term forever. However, since such rules are often useful, ACL2 has heuristics that prevent such rules from looping. See `loop-stopper`. Roughly, the heuristics allow such a rule to fire only if the resulting term is lexicographically "smaller" than the original term, *e.g.*, `(* a b)` is smaller than `(* b a)`, but not the other way around.

To deal with associativity we use `(equal (* (* x y) z) (* x (* y z)))`, which pushes parentheses to the right. Are these two rules enough to put terms into canonical form? It depends on what we mean by canonical form, but what will happen if ACL2 is given the above two theorems and asked to prove `(equal (* y (* x z)) (* x (* y z)))`? Notice that neither of the two rules apply, hence the two rules are not enough to prove

this example. Let us prove the above theorem on paper.

Proof

```
    (* y (* x z))
= { Associativity of * }
    (* (* y x) z)
= { Commutativity of * }
    (* (* x y) z)
= { Associativity of * }
    (* x (* y z)) □
```

The proof only requires the associativity and commutativity of *, but the rewrite rules we get from commutativity and associativity do not by themselves put terms into a canonical form. We also need the following theorem.

```
(defthm commutativity-of-*-2
  (equal (* y (* x z))
         (* x (* y z))))
```

Are these three rules enough? Prove `commutativity-of-*-2`.

ACL2 cannot prove `commutativity-of-*-2` without assistance. This should not be surprising since it was not able to prove `Subgoal *1/2'4'`. Does ACL2 "know" that * is associative and commutative? One way to determine what ACL2 knows is to scan its source file `axioms.lisp`, a file that describes the theory of ACL2 by enumerating the axioms and definitions. Another way is to use the history command :pl by typing the following.

```
:pl binary-*
```

ACL2 will print out all of the rules whose top function symbol is `binary-*`; this includes `associativity-of-*` and `commutativity-of-*`.

```
62. Rune:        (:REWRITE COMMUTATIVITY-OF-*)
63. Status:      Enabled
64. Lhs:         (* X Y)
65. Rhs:         (* Y X)
66. Hyps:        T
67. Equiv:       EQUAL
68. Outside-in:  NIL
69. Subclass:    BACKCHAIN
70. Loop-stopper: ((X Y BINARY-*))
71.
72. Rune:        (:REWRITE ASSOCIATIVITY-OF-*)
73. Status:      Enabled
74. Lhs:         (* (* X Y) Z)
75. Rhs:         (* X Y Z)
76. Hyps:        T
```

```
77. Equiv:      EQUAL
78. Outside-in: NIL
79. Subclass:   ABBREVIATION
```

While our hand proof of `commutativity-of-*-2` only requires the associativity and commutativity of `*`, the theorem prover does not prove `commutativity-of-*-2` even with the rewrite rules `associativity-of-*` and `commutativity-of-*` present, because the rewrite rules are applied in one direction. We have to force ACL2 to reproduce our hand proof. We do this below by using a hint to force ACL2 to take the first and last steps of our hand proof.

```
(defthm commutativity-of-*-2
  (equal (* y (* x z))
         (* x (* y z)))
  :hints (("Goal"
           :use ((:instance associativity-of-* (y x) (x y))
                 (:instance associativity-of-*))
           :in-theory (disable associativity-of-*))))
```

The `:hints` argument to `defthm` allows us to give hints to the theorem prover. Each hint is attached to the name of some goal, with `"Goal"` being the name of the top-level conjecture. For more details, see hints.

The `:use` hint instantiates the named theorems and adds each as a hypothesis to the goal in question. This is a simple but subtle way to use previously proved theorems. Write down the goal produced and find a proof of it.

The `:in-theory` hint allows us to change the status of rules. Notice that we disable `associativity-of-*`. If we keep it enabled, it removes the instantiated hypotheses just added, by rewriting them away (to `t`).

We can mimic the above proof to get a set of rewrite rules that produce canonical terms for any associative and commutative function. We start in a ground-zero theory by using `encapsulate` to define `ac-fun`, a constrained function about which we know only that it is associative and commutative.

```
(encapsulate
 ((ac-fun (x y) t))
 (local (defun ac-fun (x y) (declare (ignore x y))
          nil))
 (defthm associativity-of-ac-fun
   (equal (ac-fun (ac-fun x y) z)
          (ac-fun x (ac-fun y z))))
 (defthm commutativity-of-ac-fun
   (equal (ac-fun x y)
          (ac-fun y x))))
```

We have the same problems with ac-fun that we had with *, namely that ac-fun terms are not rewritten into a canonical form. For example, the following theorem is not proved.

```
(thm
 (equal
  (ac-fun (ac-fun f (ac-fun c d)) (ac-fun (ac-fun c b) a))
  (ac-fun (ac-fun (ac-fun a c) b) (ac-fun c (ac-fun d f)))))
```

Therefore, we prove commutativity-2-of-ac-fun as follows.

```
(defthm commutativity-2-of-ac-fun
  (equal (ac-fun y (ac-fun x z))
         (ac-fun x (ac-fun y z)))
  :hints (("Goal"
           :in-theory (disable associativity-of-ac-fun)
           :use ((:instance associativity-of-ac-fun)
                 (:instance associativity-of-ac-fun
                            (x y) (y x))))))
```

Ac-fun terms are now rewritten into a canonical form, hence, ACL2 can easily prove the previous thm. Note that it is not the case that equivalent terms containing only ac-fun (and equal) are rewritten to t (that is a complicated issue), as the following example shows.

```
(thm
 (implies (equal (ac-fun a d) (ac-fun a e))
          (equal (ac-fun a (ac-fun c d))
                 (ac-fun a (ac-fun c e)))))
```

We can use functional instantiation (see lemma-instance) to show that any associative and commutative function also satisfies commutativity-2-of-ac-fun, as follows.

```
(defthm commutativity-2-of-*
  (equal (* y (* x z))
         (* x (* y z)))
  :hints (("Goal"
           :by (:functional-instance
                commutativity-2-of-ac-fun
                (ac-fun binary-*)))))
```

ACL2 generates and establishes the constraints required, namely, that * is associative and commutative.

Notice that for the :functional-instance hint, we had to associate ac-fun with binary-* (instead of *, which is a macro). Another way to prove the above theorem, which bypasses this problem and highlights a very powerful feature of ACL2, is to use a pseudo-lambda expression (see lemma-instance) as follows.

```
(defthm commutativity-2-of-*
  (equal (* y (* x z))
         (* x (* y z)))
  :hints (("Goal"
           :by (:functional-instance
                commutativity-2-of-ac-fun
                (ac-fun (lambda (x y) (* x y)))))))
```

Since there are many functions that are associative and commutative, it may help to have a macro that automates this process. The macro will have one argument, the name of a function, and will generate the appropriate `defthm` form, using functional instantiation. Write this macro.

We must name the `defthm` produced by the macro. If the name of function is *op*, then the name of our `defthm` will be `commutativity-2-of-`*op*. We define `make-name` to make it convenient to create symbols. See the documentation for `intern-in-package-of-symbol`, `concatenate`, and other unfamiliar functions that appear below.

```
(defun make-name (prefix name)
  (intern-in-package-of-symbol
   (concatenate 'string
                (symbol-name prefix)
                "-"
                (symbol-name name))
   prefix))
```

The macro can now be defined as follows.

```
(defmacro commutativity-2 (op)
  `(defthm ,(make-name 'commutativity-2-of op)
     (equal (,op y (,op x z))
            (,op x (,op y z)))
     :hints (("Goal"
              :by (:functional-instance
                   commutativity-2-of-ac-fun
                   (ac-fun (lambda (x y) (,op x y))))))))
```

We can use the above macro on `*` as follows.

```
(commutativity-2 *)
```

Another solution to this problem, which we recommend you look at, can be found in the file `cowles/acl2-asg.lisp` in the `book/` directory of the ACL2 distribution.

10.3 Insertion Sort

Define an insertion sort on integers and prove it correct.

We define `insert`, a function that inserts a number into a list, and we use it to define `insertion-sort`.

```
(defun insert (a x)
  (cond ((atom x) (list a))
        ((<= a (car x)) (cons a x))
        (t (cons (car x) (insert a (cdr x))))))

(defun insertion-sort (x)
  (cond ((atom x) nil)
        (t (insert (car x) (insertion-sort (cdr x))))))
```

What does it mean for this function to be correct? At the least, the function must return an ordered list. We define a predicate to recognize ordered lists.

```
(defun orderedp (x)
  (cond ((atom (cdr x)) t)
        (t (and (<= (car x) (cadr x))
                (orderedp (cdr x))))))
```

We use `orderedp` to state a correctness condition.

```
(defthm insertion-sort-is-ordered
  (orderedp (insertion-sort x)))
```

The first simplification checkpoint in the proof attempt is

```
36. Subgoal *1/2''
37. (IMPLIES (AND (CONSP X)
38.               (ORDEREDP (INSERTION-SORT (CDR X))))
39.          (ORDEREDP (INSERT (CAR X)
40.                            (INSERTION-SORT (CDR X))))).
```

Not surprisingly we need to know that `insert` preserves `orderedp`. We therefore prove the following (which ACL2 guesses if we let it).

```
(defthm insert-ordered
  (implies (orderedp x)
           (orderedp (insert a x))))
```

ACL2 now proves `insertion-sort-is-sorted`.

But this is not enough. For example, if `insertion-sort` always returned `nil`, then we would be able to prove the above theorem even though we would not consider the function correct. We have to show that `insertion-sort` returns a permutation of its input. First, we define the notion of a permutation.

```
(defun in (a b)
  (cond ((atom b) nil)
        ((equal a (car b)) t)
        (t (in a (cdr b)))))
```

```
(defun del (a x)
  (cond ((atom x) nil)
        ((equal a (car x)) (cdr x))
        (t (cons (car x) (del a (cdr x))))))

(defun perm (x y)
  (cond ((atom x) (atom y))
        (t (and (in (car x) y)
                (perm (cdr x) (del (car x) y))))))
```

We now prove that `insertion-sort` returns a permutation of its input.

```
(defthm insertion-sort-is-perm
  (perm (insertion-sort x) x))
```

The first simplification checkpoint is:

```
36. Subgoal *1/2''
37. (IMPLIES (AND (CONSP X)
38.               (PERM (INSERTION-SORT (CDR X)) (CDR X)))
39.          (PERM (INSERT (CAR X)
40.                        (INSERTION-SORT (CDR X)))
41.                X)).
```

Although ACL2 completes the proof on its own, we take it as a challenge to save ACL2 from attempting subsidiary induction arguments. The goal above suggests the need to show that `insert` preserves `perm`. If we formulate a rewrite rule that looks like this goal, then it is not clear that ACL2 will be able to prove it—the `car` and `cdr` may somehow get in the way of induction—and more seriously, we wonder a bit if the rule may somehow loop when it is used later. An example of such a looping rule is discussed below (page 197). We can let the prover's next goal guide us in finding a suitable rule.

```
43. The destructor terms (CAR X) and (CDR X) can be eliminated
44. by using CAR-CDR-ELIM to replace X by (CONS X1 X2), generalizing
45. (CAR X) to X1 and (CDR X) to X2.  This produces the following
46. goal.
47.
48. Subgoal *1/2'''
49. (IMPLIES (AND (CONSP (CONS X1 X2))
50.               (PERM (INSERTION-SORT X2) X2))
51.          (PERM (INSERT X1 (INSERTION-SORT X2))
52.                (CONS X1 X2))).
```

We can derive the following theorem by generalizing the goal above. In fact ACL2 comes up with this exact generalization (using different variable names), but it is rare that ACL2's generalization heuristics are so on-target.

```
(defthm insert-perm-cons
  (implies (perm x y)
           (perm (insert a x) (cons a y))))
```

ACL2 can now prove `insertion-sort-is-perm` without any induction other than the one at the top level.

10.4 Tree Manipulation

The function `flatten` (page 49) returns a list of the tips of a tree.

```
(defun flatten (x)
  (cond ((atom x) (list x))
        (t (append (flatten (car x)) (flatten (cdr x))))))
```

In Exercise 7.9, the following, more efficient, function (why is it more efficient?) is introduced.

```
(defun mc-flatten (x a)
  (cond ((atom x) (cons x a))
        (t (mc-flatten (car x)
                       (mc-flatten (cdr x) a)))))
```

We will give a solution to Exercise 7.9, by proving that the above functions are equivalent.

```
(defthm flatten-mc-flatten
  (equal (mc-flatten x nil)
         (flatten x)))
```

As we saw with the factorial example, this theorem should raise some flags: do we need to prove a stronger theorem? Since opening the definition of `mc-flatten` will replace the above `nil` with a term involving the variable `x`, the answer is yes. We therefore attempt to come up with a rule for rewriting the term `(mc-flatten x y)` in terms of `flatten`. After a little thought, we try:

```
(defthm flatten-mc-flatten-lemma
  (equal (mc-flatten x a)
         (append (flatten x) a)).
```

ACL2 proves the `flatten-mc-flatten-lemma`, but has to prove the associativity of `append` as a separate induction. Since this is a useful theorem to have around, we prove it.

```
(defthm associativity-of-append
  (equal (append (append x y) z)
         (append x (append y z)))).
```

ACL2 can now prove `flatten-mc-flatten`, but it uses induction. Why? What other fact is required so that the theorem prover can prove `flatten-mc-flatten` entirely by simplification? We leave you to contemplate this and move on to another tree processing function.

Admit the following function.

```
(defun gopher (x)
  (if (or (atom x)
          (atom (car x)))
      x
    (gopher (cons (caar x) (cons (cdar x) (cdr x))))))
```

ACL2 does not admit `gopher` because `acl2-count` does not decrease on the recursive call. Note that `gopher` recurs exactly the way `flat` does on one of its recursive calls (see page 108), hence, we use the `acl2-count` of the `car` as the measure.

```
(defun gopher (x)
  (declare (xargs :measure (acl2-count (car x))))
  (if (or (atom x)
          (atom (car x)))
      x
    (gopher (cons (caar x) (cons (cdar x) (cdr x))))))
```

The following function uses `gopher` to determine if two trees have the same fringe; admit it.

```
(defun samefringe (x y)
  (if (or (atom x)
          (atom y))
      (equal x y)
    (and (equal (car (gopher x))
                (car (gopher y)))
         (samefringe (cdr (gopher x))
                     (cdr (gopher y))))))
```

If we look at the first simplification checkpoint during admission, we see:

```
26. Goal'
27. (IMPLIES (AND (CONSP X)
28.               (CONSP Y)
29.               (EQUAL (CAR (GOPHER X))
30.                      (CAR (GOPHER Y))))
31.          (< (ACL2-COUNT (CDR (GOPHER X)))
32.             (ACL2-COUNT X))).
33.
34. Name the formula above *1.
```

The suggested rewrite rule is shown below.

```
(defthm gopher-acl2-count-cdr
  (implies (consp x)
           (< (acl2-count (cdr (gopher x)))
              (acl2-count x))))
```

ACL2 can prove the above and can subsequently admit `samefringe`.

Many experienced users would make the rule above a linear rule. As such it would cause the linear arithmetic procedure to add the indicated inequality to the linear data base whenever some inequality mentioned (an instance of) `(acl2-count (cdr (gopher x)))` and the hypothesis `(consp x)` can be proved. If stored as a rewrite rule, as above, the rule is quite restricted in its applicability: it conditionally rewrites the indicated <-expression to `t`. This is all we need in the current situation.

Prove that `samefringe` is correct.

We choose the following theorem.

```
(defthm correctness-of-samefringe
  (equal (samefringe x y)
         (equal (flatten x)
                (flatten y))))
```

ACL2 cannot prove this theorem with its current rules. Looking at the first simplification checkpoint, we see:

```
70. Subgoal *1/3'
71. (IMPLIES (AND (CONSP X)
72.               (CONSP Y)
73.               (NOT (EQUAL (CAR (GOPHER X))
74.                           (CAR (GOPHER Y)))))
75.          (NOT (EQUAL (FLATTEN X) (FLATTEN Y)))).
76.
77. Name the formula above *1.1.
```

This suggests the following rewrite rule.

```
(defthm car-gopher-car-flatten
  (implies (consp x)
           (equal (car (gopher x))
                  (car (flatten x)))))
```

It turns out that, for this simple example, it does not make a difference which way we orient the rewrite rule. In general, you should rewrite more complicated terms into simpler onesand you should design rewrite rules that massage terms into canonical forms.

We try proving `correctness-of-samefringe` again. Once again, ACL2 cannot prove the theorem and once again, we check the first simplification checkpoint:

```
102. Subgoal *1/2.2'
103. (IMPLIES (AND (CONSP X)
104.                (CONSP Y)
105.                (EQUAL (CAR (FLATTEN X))
106.                       (CAR (FLATTEN Y)))
107.                (EQUAL (FLATTEN (CDR (GOPHER X)))
108.                       (FLATTEN (CDR (GOPHER Y))))
109.                (SAMEFRINGE (CDR (GOPHER X))
110.                            (CDR (GOPHER Y))))
111.           (EQUAL (FLATTEN X) (FLATTEN Y))).
```

We decide to try moving the `cdr` to the left (outside) of the `flatten` in lines 107 and 108; this suggests that we prove the following theorem.

```
(defthm cdr-flatten-gopher
  (implies (consp x)
           (equal (flatten (cdr (gopher x)))
                  (cdr (flatten (gopher x))))))
```

ACL2 proves the above theorem, but when we try to prove the theorem `correctness-of-samefringe`, we get the following peculiar error. (This is a GCL error; if you are using a different Common Lisp, you may get a different error message.)

```
90. Error: Value stack overflow.
91. Fast links are on: do (si::use-fast-links nil) for debugging
92. Error signalled by ACL2_*1*_ACL2::DEFTHM-FN.
93. Broken at COND.  Type :H for Help.
94. ACL2>>
```

In GCL it is even possible to get a segmentation error that aborts the Lisp process.

What is going on? This is something that happens to everyone (although it happens much more frequently to beginners): the rewriter has entered an infinite loop. Often, by inspection one can figure out what combination of rewrite rules is leading to an infinite loop, but if not, one can use `brr` to examine the rewriter. See `break-rewrite` for the details, but briefly, with `brr` one can `monitor` rewrite rules: when a monitored rule is tried, the rewriter will enter an interactive break, where you can inspect the context (there are many things you can do, see `brr-commands`). In many situations, the following trick is all that is needed. After a stack overflow, reenter the top-level ACL2 loop[1], type `:brr t` and try proving the theorem again. This will lead to another stack overflow, but now, enter raw Lisp and type `(cw-gstack *deep-gstack* state)`. This will print out the rewrite stack, which usually makes it clear why the rewriter is looping. After doing this, we observe the following loop.

[1] In GCL it is best to enter raw Lisp first and execute `(si::use-fast-links nil)` to prevent a stack overflow from manifesting itself as a segmentation error.

```
31. 9. Attempting to apply (:REWRITE CDR-FLATTEN-GOPHER) to
32.      (FLATTEN (CDR (GOPHER X)))
33. 10. Rewriting (to simplify) the rhs of the conclusion,
34.      (CDR (FLATTEN (GOPHER X))),
35.    under the substitution
36.      X : X
37. 11. Rewriting (to simplify) the first argument,
38.      (FLATTEN (GOPHER X)),
39.    under the substitution
40.      X : X
41. 12. Attempting to apply (:DEFINITION FLATTEN) to
42.      (FLATTEN (GOPHER X))
43. 13. Rewriting (to simplify) the rewritten body,
44.      (BINARY-APPEND (FLATTEN (CAR #))
45.                     (FLATTEN (CDR #))),
46. 14. Rewriting (to simplify) the second argument,
47.      (FLATTEN (CDR (GOPHER X))),
48. 15. Attempting to apply (:REWRITE CDR-FLATTEN-GOPHER) to
49.      (FLATTEN (CDR (GOPHER X)))
```

What is the rewriter really doing here?
Given the term

```
1. (flatten (cdr (gopher x))),
```

ACL2 uses the rewrite rule `cdr-flatten-gopher` to rewrite it to

```
2. (cdr (flatten (gopher x))).
```

ACL2 then applies the definition of `flatten` to get

```
3. (cdr (append (flatten (car (gopher x)))
                (flatten (cdr (gopher x))))).
```

ACL2 then tries to simplify the second argument to **append**, but this is the same term we started with, so we managed to pump 1, our original term, to 3, a bigger term that contains 1 as a subterm.

Having identified the loop, we can introduce a new rewrite rule that prevents the loop from occurring or we can throw away the offending rewrite rule. Let us try the first approach. The following rule comes to mind.

```
(defthm flatten-gopher
  (equal (flatten (gopher x))
         (flatten x)))
```

Notice that since this rule was added to the theorem prover's rules after the definition of `flatten`, it will be used before the definition of `flatten` and will therefore keep the above loop from occurring.

ACL2 proves `correctness-of-samefringe`, the main theorem (by performing seven inductions, but since this is the last theorem we want to prove in this section, we do not investigate further).

It is instructive to try the second approach to stopping the loop. We have to replace `cdr-flatten-gopher` by a rewrite rule that does not loop. We can combine the above two rewrite rules into the following rewrite rule.

```
(defthm cdr-flatten-gopher
  (implies (consp x)
           (equal (flatten (cdr (gopher x)))
                  (cdr (flatten x)))))
```

The observant reader may have thought of this rewrite rule previously, when we chose `cdr-flatten-gopher` instead. ACL2 can now prove the main theorem, `correctness-of-samefringe`.

10.5 Binary Adder and Multiplier

In this section we will define and prove correct a binary adder and multiplier.

10.5.1 Binary Adder

Define a binary adder and prove that it adds.

We use `t` and `nil` to represent 1 and 0, respectively, and will represent binary numbers as lists of `t`'s and `nil`'s. Our plan is to define a serial adder. A serial adder works by adding two binary numbers bit by bit. Hence, it can be built out of a full adder using recursion, with an algorithm very similar to the one taught to children for adding base-10 numbers. (If you are not familiar with computer arithmetic, consult a book on computer architecture, *e.g.*, [18].)

In order to define a full adder, we define the following Boolean-valued functions.

```
(defun band (p q) (if p (if q t nil) nil))
(defun bor  (p q) (if p t (if q t nil)))
(defun bxor (p q) (if p (if q nil t) (if q t nil)))
(defun bmaj (p q c)
  (bor (band p q)
       (bor (band p c)
            (band q c))))
```

A full adder has three inputs (bits) and returns the sum of its inputs as two bits: a sum and a carry. We define a full adder as follows.

```
(defun full-adder (p q c)
  (mv (bxor p (bxor q c))
      (bmaj p q c)))
```

Recall that the serial adder will take as input two lists of Booleans and a carry-in bit. If we were designing hardware, we would probably know something about the binary numbers given as input to the adder, *e.g.*, their length, or that they have the same length. However, ACL2 functions have to be total, hence, we have to decide what to do even with inputs that are not of the same length. We decide to design an adder that correctly adds any pair of binary numbers, even if they are not the same length (we are thinking ahead here, because when we define a multiplier, it will be useful to have such an adder). We have to decide if the first bit of a list is the high-order bit or the low-order bit. It is more convenient to make the first bit the low-order bit: otherwise, we would have to align the numbers before adding them. Our definition is as follows.

```
(defun serial-adder (x y c)
  (if (and (endp x) (endp y))
      (list c)
    (mv-let (sum cout)
            (full-adder (car x) (car y) c)
            (cons sum (serial-adder (cdr x) (cdr y) cout)))))
```

ACL2 does not admit the above function because it cannot prove termination. If we scan through the output produced, we see:

```
 8. For the admission of SERIAL-ADDER we will use the relation
 9. EO-ORD-< (which is known to be well-founded on the domain
10. recognized by EO-ORDINALP) and the measure (ACL2-COUNT X).
```

If we think about why `serial-adder` terminates, we realize that termination also depends on `y`, *e.g.*, if the length of `y` is greater than the length of `x`, then `serial-adder` may take more than `(acl2-count x)` steps. In this case, the heuristics used by ACL2 to guess a measure do not work and we are forced to give ACL2 a measure explicitly.

```
(defun serial-adder (x y c)
  (declare (xargs :measure (+ (len x) (len y))))
  ...)
```

There are many other measures that work. Two examples are `(max (len x) (len y))` and `(+ (acl2-count x) (acl2-count y))`.

Execute the adder on a few examples. What does it mean for the adder to be correct? Well, it means that it adds! More specifically, the binary number returned is the sum of the binary numbers given as input plus the initial carry-in. In order to write this formally, we have to define a function that transforms a binary number into a number.

```
(defun n (v)
  (cond ((endp v) 0)
        ((car v) (+ 1 (* 2 (n (cdr v)))))
        (t (* 2 (n (cdr v))))))
```

We state correctness as follows.

```
(defthm serial-adder-correct
  (equal (n (serial-adder x y c))
         (+ (n x) (n y) (if c 1 0))))
```

ACL2 does not prove this theorem. Scanning through the output produced by ACL2 until we reach the simplification checkpoint, we see the following:

```
1310. Subgoal *1/1.4'5'
1311. (EQUAL (* 2 (N (SERIAL-ADDER X2 NIL T)))
1312.        (+ 1 1 (* 2 (N X2)))).
1313.
1314. Name the formula above *1.1.
```

If we scan ahead to the next checkpoint, we see:

```
1378. Subgoal *1/1.3'6'
1379. (EQUAL (+ 1 (* 2 (N (SERIAL-ADDER X2 NIL NIL))))
1380.        (+ 1 (* 2 (N X2)))).
1381.
1382. Name the formula above *1.2.
```

`*1.3` is the same as `*1.2`; finally we have:

```
1490. Subgoal *1/1.1'5'
1491. (EQUAL (* 2 (N (SERIAL-ADDER X2 NIL NIL)))
1492.        (* 2 (N X2))).
1493.
1494. Name the formula above *1.4.
```

The appropriate thing to do now is to determine which lemmas are suggested by `*1.1`, `*1.2`, and `*1.4`. Notice that `*1.4` has the form `(equal (* 2 a) (* 2 b))`; this suggests that the multiplication is not important and that the appropriate lemma is the simpler `(equal a b)`. Notice that the suggested lemma will allow ACL2 to discharge `*1.2` and `*1.3`. (As an aside, we point out that sometimes, on very large problems or with rules that match too often or take too much time[2], it is wiser to prove an unsimplified lemma because it is less applicable and therefore the theorem prover is faster.) After a similar analysis of `*1.1`, we decide to prove the following two lemmas.

```
(defthm serial-adder-correct-nil-nil
  (equal (n (serial-adder x nil nil))
         (n x)))
```

[2]Statistics supplied by `accumulated-persistence` can be used to identify rules that blow up ACL2's search space.

```
(defthm serial-adder-correct-nil-t
  (equal (n (serial-adder x nil t))
         (+ 1 (n x))))
```

We really need to prove the above lemmas in the order indicated because ACL2 cannot directly prove `serial-adder-correct-nil-t`. Why? ACL2 can now prove `serial-adder-correct`.

10.5.2 Binary Multiplier

Define a binary multiplier and prove that it multiplies.

Multiplication can be performed by repeatedly shifting and adding. That is, to multiply y by x, we initialize p, a partial sum, to 0 and iterate over x, one bit at a time from the low-order bit to the high order bit. If the current bit is 0, we multiply y by 2, otherwise we add y to p and multiply y by 2. Note that multiplying a binary number by 2 is the same as inserting a 0 low order bit (*i.e.*, shifting). A multiplier in ACL2 is defined as follows.

```
(defun multiplier (x y p)
  (if (endp x)
      p
    (multiplier (cdr x)
                (cons nil y)
                (if (car x)
                    (serial-adder y p nil)
                  p))))
```

The multiplier is correct if it multiplies. More specifically, the binary number returned is the product of the binary numbers given as input plus the initial partial sum. Here is a formal statement.

```
(defthm multiplier-correct
  (equal (n (multiplier x y p))
         (+ (* (n x) (n y)) (n p))))
```

ACL2 tries to prove this theorem, but after a second or so, it is clear from the output streaming by that many inductions will be necessary for this proof attempt to succeed (ACL2 actually runs for a long time and eventually runs out of memory). We therefore interrupt ACL2 and scan through the output until we reach the first simplification checkpoint, where we see the following:

```
77. Subgoal *1/3''
78. (IMPLIES (AND (CONSP X)
79.               (NOT (CAR X))
80.               (EQUAL (N (MULTIPLIER (CDR X) (CONS NIL Y) P))
81.                      (+ (N P) (* (N (CDR X)) 2 (N Y)))))
```

```
82.          (EQUAL (+ (N P) (* (N (CDR X)) 2 (N Y)))
83.                 (+ (N P) (* (N Y) 2 (N (CDR X))))))).
```

Notice that the conclusion of `Subgoal *1/3''` has the form `(equal (+` a `(* x y z)) (+` a `(* z y x)))`; this suggests that the addition is not important and that we need a lemma that allows us to conclude `(equal (* x y z) (* z y x))`. But, we have already seen such a lemma, namely `commutativity-of-*-2`. Once we get ACL2 to prove `commutativity-of--*-2`, `multiplier-correct` follows.

10.5.3 Miscellaneous

We can generate a gate-level design of an adder and multiplier for fixed length binary numbers by unrolling the recursive definitions the appropriate number of times. This idea was used to generate a netlist description of a formally verified chip [19]. Note that if we unroll the serial adder, we get a ripple carry adder. A formal netlist description language, similar to that described in Hunt's case study in the companion volume [22], has been used to describe and verify a chip which was then fabricated [20].

If one is interested in proving more complicated fixed-length hardware modules correct, then BDDs can be useful. BDDs [12, 32] are data structures used for the simplification of Boolean expressions. They have been found to work well in practice, especially with hardware. In ACL2, BDDs are generalized: they can represent not only Boolean values, but arbitrary ACL2 terms, and they are integrated with rewriting. The ACL2 distribution contains, in the directory `books/bdd`, examples highlighting the use of BDDs, *e.g.*, there are specifications of a simple ripple-carry ALU and a tree-structured propagate-generate ALU, as well as proofs—employing BDDs—of their equivalence.

10.6 Compiler for Stack Machine

We will define a simple stack-based machine and a compiler that given an expression generates code for the machine. Of course we will prove that our compiler is correct.

We want the stack machine to have instructions with names such as `push` and `pop`. Since we cannot define functions with such names in the `ACL2` package (these symbols are pre-defined in Common Lisp), we will define a new package as follows.

```
(defpkg "compile"
  (set-difference-eq
   (union-eq *acl2-exports*
       (union-eq '(acl2-numberp len)
```

```
          *common-lisp-symbols-from-main-lisp-package*))
  '(pop push top compile step eval)))
```

The constant `*acl2-exports*` is a list of symbols that is convenient to import into other packages. It includes many commonly used symbols—such as `defun`, `defthm`, `iff`, and so on—that you would otherwise have to prefix by `"ACL2::"`. `Union-eq` is a function that returns the set union of two lists; `set-difference-eq` is a function that takes the set difference of two lists. The above `defpkg` defines the new symbol package `compile` and imports exactly the symbols we want. We select this as the current package.

```
(in-package "compile")
```

10.6.1 Expressions

Our expressions are built out of symbols, numbers, and the functions `inc`, `sq`, `+`, and `*`. Notice that binary functions in expressions are written in infix notation.

```
(defun exprp (exp)
  (cond
   ((atom exp)
    (or (symbolp exp) (acl2-numberp exp)))
   ((equal (len exp) 2)
    (and (or (equal (car exp) 'inc)
             (equal (car exp) 'sq))
         (exprp (cadr exp))))
   (t
    (and (equal (len exp) 3)
         (or (equal (cadr exp) '+)
             (equal (cadr exp) '*))
         (exprp (car exp))
         (exprp (caddr exp))))))
```

Define the semantics of expressions: `inc` increments by one, `sq` squares, `+` adds, and `*` multiplies.

Since expressions can contain symbols, they have a value in the context of an *environment*, an alist (see page 31) that relates symbols to numbers. Here is the function to look up the value of a symbol in an environment.

```
(defun lookup (var alist)
  (cond ((endp alist)
         0)  ; default
        ((equal var (car (car alist)))
         (cdr (car alist)))
        (t (lookup var (cdr alist)))))
```

Alists are often used for representing environments, memories, functions, and related concepts. Note that you can update the value of a variable by consing a cons to the beginning of the alist, because `lookup` finds the first cons matching a variable. Another nice property of the alist representation is that the alist only has to contain the variables you are interested in; all other variables have a default value (in our case, 0). This can be very useful, *e.g.*, suppose you want to model a memory assigning values to 64-bit wide addresses.

The following function evaluates an expression in an environment.

```
(defun eval (exp alist)
  (cond
   ((atom exp)
    (cond ((symbolp exp) (lookup exp alist))
          (t exp)))
   ((equal (len exp) 2)
    (cond ((equal (car exp) 'inc)
           (+ 1 (eval (cadr exp) alist)))
          (t  ; 'sq
           (* (eval (cadr exp) alist)
              (eval (cadr exp) alist)))))
   (t  ; (equal (len exp) 3)
    (cond ((equal (cadr exp) '+)
           (+ (eval (car exp) alist)
              (eval (caddr exp) alist)))
          (t  ; *
             (* (eval (car exp) alist)
                (eval (caddr exp) alist)))))))
```

Evaluate `eval` on several examples.

10.6.2 Stack Machine

The stack machine will have six instructions: `pushv` pushes the value of a variable on the stack, `pushc` pushes a constant on the stack, `dup` duplicates the top of the stack, `add` adds the top two elements of the stack, `mul` multiplies the top two elements of the stack, and anything else acts as a skip. Define a function that given an instruction, an environment, and a stack, steps the machine for a single step and returns the new stack.

We start by defining some simple stack manipulation functions.

```
(defun pop (stk) (cdr stk))
(defun top (stk) (if (consp stk) (car stk) 0))
(defun push (val stk) (cons val stk))
```

We define step, the function that steps the machine for a single step as follows.

```
(defun step (ins alist stk)
  (let ((op (car ins)))
    (case op
      (pushv (push (lookup (cadr ins) alist) stk))
      (pushc (push (cadr ins) stk))
      (dup   (push (top stk) stk))
      (add   (push (+ (top (pop stk)) (top stk))
                   (pop (pop stk))))
      (mul   (push (* (top (pop stk)) (top stk))
                   (pop (pop stk))))
      (t stk))))
```

Define a function that given a program (a list of instructions), an environment, and a stack runs the program to completion. The function should return the final stack.

```
(defun run (program alist stk)
  (cond ((endp program) stk)
        ((run (cdr program)
              alist
              (step (car program) alist stk)))))
```

The machine that we are defining does not allow any looping, but in general—if we were defining a more complicated machine—we cannot admit a function such as run, because there are programs that never terminate. In such a situation, we would instead define a similar function that takes an extra argument indicating how many times to step the machine.

For an example of a more complex state machine in ACL2, see the article by Greve, Wilding and Hardin in [22]. In addition, for a comprehensive description of how to define such machines in ACL2 and how to configure the rewriter to facilitate proofs about their programs, see [6].

10.6.3 Compiler

Write a compiler that takes expressions and returns programs (for the stack machine) that evaluate the expressions.

```
(defun compile (exp)
  (cond
   ((atom exp)
    (cond ((symbolp exp)
           (list (list 'pushv exp)))
          (t (list (list 'pushc exp)))))
   ((equal (len exp) 2)
```

```
(cond ((equal (car exp) 'inc)
       (append (compile (cadr exp)) '((pushc 1) (add))))
      (t (append (compile (cadr exp)) '((dup) (mul))))))
(t (cond ((equal (cadr exp) '+)
          (append (compile (car exp))
                  (compile (caddr exp))
                  '((add))))
         (t (append (compile (car exp))
                    (compile (caddr exp))
                    '((mul)))))))
```

Prove that `compile` compiles.

Our notion of correctness is the following.

```
(defthm compile-is-correct
  (implies (exprp exp)
           (equal (top (run (compile exp) alist stk))
                  (eval exp alist))))
```

`Compile-is-correct` says that the value of expression `exp` in environment `alist` is equivalent to the top of the stack of a machine running the program produced by compiling `exp`, in environment `alist`. Notice that the initial value of the stack seems to play no important role in the above condition; all we care about is the top of the final stack.

Following our "brain-dead" approach to proving theorems, we try to prove this theorem and look at the first simplification checkpoint, where we see:

```
107. Subgoal *1/6.7
108. (IMPLIES (AND (CONSP EXP)
109.               (CONSP (CDR EXP))
110.               (NOT (CONSP (RUN (COMPILE (CAR EXP)) ALIST STK)))
111.               (EQUAL 0 (EVAL (CAR EXP) ALIST))
112.               (NOT (CONSP (RUN (COMPILE (CADDR EXP))
113.                                ALIST STK)))
114.               (EQUAL 0 (EVAL (CADDR EXP) ALIST))
115.               (EQUAL (+ 1 1 (LEN (CDDR EXP))) 3)
116.               (EQUAL (CADR EXP) '*)
117.               (EXPRP (CAR EXP))
118.               (EXPRP (CADDR EXP))
119.               (CONSP (RUN (APPEND (COMPILE (CAR EXP))
120.                                   (COMPILE (CADDR EXP))
121.                                   '((MUL)))
122.                           ALIST STK)))
123.          (EQUAL (CAR (RUN (APPEND (COMPILE (CAR EXP))
124.                                   (COMPILE (CADDR EXP))
125.                                   '((MUL)))
126.                           ALIST STK))
127.                 0)).
```

There are a few terms of the form (run (append x y) a s) present above. It is almost always the case when proving machines correct that we need a theorem about the composition of programs which says you can run a program composed of two parts by first running the first and then the second. We prove the following.

```
(defthm composition
  (equal (run (append prg1 prg2) alist stk)
         (run prg2 alist (run prg1 alist stk))))
```

We try to prove the main result again. Looking at the output produced by ACL2 induces panic and we instead decide to a step back and look at the big picture. What approach is ACL2 taking to this problem?

```
32. Perhaps we can prove *1 by induction.  Three induction schemes
33. are suggested by this conjecture.  Subsumption reduces that
34. number to two.  These merge into one derived induction scheme.
35.
36. We will induct according to a scheme suggested by
37. (EVAL EXP ALIST).  If we let  (:P ALIST EXP STK) denote *1
38. above then the induction scheme we'll use is
39. (AND (IMPLIES (AND (NOT (ATOM EXP))
40.                    (NOT (EQUAL (LEN EXP) 2))
41.                    (NOT (EQUAL (CADR EXP) '+))
42.                    (:P ALIST (CAR EXP) STK)
43.                    (:P ALIST (CADDR EXP) STK))
44.               (:P ALIST EXP STK))
```

If we look at the first conjunct (lines 39–44) of the induction scheme, we notice that ACL2 is trying to prove (:P ALIST EXP STK)) from (:P ALIST (CAR EXP) STK) and (:P ALIST (CADDR EXP) STK)) (in the case where we are multiplying). Is this reasonable? What happens when we run a compiled program? Try an example.

We will manipulate a term that matches the first conjunct of the induction scheme and will try to rewrite it until the appropriate induction is apparent. Below, we abbreviate compile, append, and stk by c, app, and s, respectively (this allows us to focus on the structure of the term).

```
    (top (run (c '(x * y)) a s))
= { Definition of com }
    (top (run (app (c 'x) (c 'y) '((mul))) a s))
= { Composition }
    (top (run (app (c 'y) '((mul))) a (run (c 'x) a s)))
= { Composition }
    (top (run '((mul)) a (run (c 'y) a (run (c 'x) a s))))
```

We have rewritten the term we started with above in terms of `x` and `y`, the subexpressions of `(x * y)`. We did this because induction allows us to assume what we want to prove for smaller instances of the problem, *e.g.*, we can assume that `(top (run (c 'x) a s))` is `(eval 'x a)` and that `(top (run (c 'y) a (run (c 'x) a s)))` is `(eval 'y a)`. At this point, it should clear that our induction hypotheses are too weak because in order to prove that the machine multiplies on the above example, not only do we need to know what is on top of the stack after we run the code produced by the compiler for `y`, but we also need to know that right below that is `(eval 'x a)`. We are now in the familiar situation where we have to strengthen a theorem in order to apply induction. As we saw, we want our theorem to tell us not only what happens to the top of the stack, but what happens to the rest of the stack. The following comes to mind.

```
(defthm compile-is-correct-general
  (implies (exprp exp)
           (equal (run (compile exp) alist stk)
                  (cons (eval exp alist) stk))))
```

Will this work? Let us rework the above example.

```
  (run (c '(x * y)) a s)
```

= { Definition of `com` }

```
  (run (app (c 'x) (c 'y) '((mul))) a s)
```

= { Composition }

```
  (run (app (c 'y) '((mul))) a (run (c 'x) a s))
```

= { Composition }

```
  (run '((mul)) a (run (c 'y) a (run (c 'x) a s)))
```

We can assume the following inductive hypotheses.

1. `(run (c 'x) a s)` is `(cons (eval 'x a) s)` and
2. `(run (c 'y) a (run (c 'x) a s))` is `(cons (eval 'y a) (cons (eval 'x a) s))`.

We can symbolically run the program to see that the `mul` instruction will pop the values of `x` and `y` off the stack and will push their product, therefore it seems that we can prove this theorem by induction. ACL2, however, does not prove the theorem.

```
12. We will induct according to a scheme suggested by
13. (EVAL EXP ALIST).  If we let  (:P ALIST EXP STK) denote *1
14. above then the induction scheme we'll use is
15. (AND (IMPLIES (AND (NOT (ATOM EXP))
16.                    (NOT (EQUAL (LEN EXP) 2))
17.                    (NOT (EQUAL (CADR EXP) '+))
```

```
18.                    (:P ALIST (CAR EXP) STK)
19.                    (:P ALIST (CADDR EXP) STK))
20.              (:P ALIST EXP STK))
```

Inspection of the first conjunct of the induction scheme shows why the proof attempt fails. Above, when we considered this part of the induction, we assumed induction hypothesis 2, whose stack (`(cons (eval 'x a) s)`) differs from `s`, the stack of the induction conclusion. The first conjunct of the induction scheme generated by ACL2, however, mentions the stack of the induction conclusion. Before continuing, let us make sure that the substitution we are suggesting does not violate the induction principle. The measure used to justify this induction, `(acl2-count exp)` (which is the measure used to admit `eval`), does not mention `stk`, therefore, we can substitute anything for `stk` (as long as we substitute something smaller for `exp`). We have to tell ACL2 what the right induction scheme is. This is done by defining a function that recurs according to the scheme and giving ACL2 a hint (see hints) to use the induction scheme suggested by this function. Define such a function.

```
(defun compiler-induct (exp alist stk)
  (cond
   ((atom exp) stk)
   ((equal (len exp) 2)
    (compiler-induct (cadr exp) alist stk))
   (t  ; Any binary function may be used in place of append below
    (append (compiler-induct (car exp) alist stk)
            (compiler-induct (caddr exp)
                             alist
                             (cons (eval (car exp) alist)
                                   stk))))))
```

We give ACL2 the appropriate hint as follows.

```
(defthm compile-is-correct-general
  (implies (exprp exp)
           (equal (run (compile exp) alist stk)
                  (cons (eval exp alist) stk)))
  :hints (("Goal"
           :induct (compiler-induct exp alist stk))))
```

ACL2 can prove this theorem and then `compile-is-correct` follows.

11

Theorem Prover Exercises

This chapter contains exercises of varying degrees of difficulty. The exercises will allow you to gain experience in using the theorem prover. We cannot over-stress the importance of doing the exercises. It is one thing to understand how the theorem prover works and another to be a competent user. We remind you that solutions to all of the exercises are on the Web (see the link to this book's page on the ACL2 home page). We suggest that you do the exercises without consulting our solutions, but that once you are done, we recommend that you compare your solutions to ours.

Please do not be discouraged if some of these exercises take considerable thought and time. After all, we did warn that you will be hard pressed to find a more challenging game. With practice you will win the game with increasing frequency.

11.1 Starters

The exercises in this first section are generally simpler than those in the sections that follow. We suggest that you use them to get acquainted with the ACL2 theorem prover.

Exercise 11.1 *Are the following functions admissible? If not, why not? If so, admit them.*

```
(defun f (x)
  (if (endp x)
      0
    (1+ (f (cdr x)))))
(defun f (x)
  (if (null x)
      0
    (1+ (f (cdr x)))))
```

Exercise 11.2 *Recall the definitions of* `flatten` *and* `swap-tree` *(pages 49 and 115).*

- *Use ACL2 to prove the following theorem, or an appropriately-fixed theorem, from page 115:*

```
(equal (flatten (swap-tree x)) (rev (flatten x)))
```

- *Admit the function* `flat` *defined on page 108. Prove that* `(flat x)` *is equal to* `(flatten x)` *(a fact proved by hand on page 109).*

Exercise 11.3 *Prove the following. (*Hint: *You may find it helpful to use* `:pe` *to view the definitions of* `append` *and its supporting functions.)*

```
(defthm reverse-reverse
  (implies (true-listp x)
           (equal (reverse (reverse x))
                  x)))
```

Exercise 11.4 *Prove that* `(rev x)` *(see page 124) is equal to* `(reverse x)` *if* `x` *is not a string.*

11.2 Sorting

In this section, you will be asked to prove several sorting algorithms correct. What does it mean for a sorting algorithm to be correct? Correctness is captured by the following two conditions.

1. The output is ordered.
2. The output is a permutation of the input.

In Chapter 10, we proved the correctness of insertion sort and in the process defined both the notion of a permutation, `perm`, and of an ordered list, `orderedp` (on pages 193 and 192, respectively). That proof was relatively easy; however, the proofs in this section are going to require more work. We start by proving that `perm` is an equivalence relation. Recall that in the discussion of congruence-based reasoning (page 139), we saw that the theorem prover can use equivalence relations the way it uses `equal`, in the right contexts. The macro `defequiv` can be used to prove that a relation is an equivalence relation.

Exercise 11.5 *Create a certified book (see* `certify-book`*) that starts with definitions including the definition of* `perm`*, concludes with* `(defequiv perm)`*, and has any number of* `local` *events in between. We suggest proceeding as follows.*

- *Prove* `(perm x x)`.

- *Prove* `(implies (perm x y) (perm y x))`.
- *Prove* `(implies (and (perm x y) (perm y z)) (perm x z)))`.
- *Use* `:trans1` *to print out the immediate expansion (see page 37 of* `(defequiv perm)`.
- *Prove* `(defequiv perm)`.

We will use the fact that `perm` is an equivalence relation by proving some congruence rules. We use the macro `defcong` to prove congruence rules.

Exercise 11.6 *Use* `:trans1` *to print out the immediate expansion of the following.*
`(defcong perm perm (append x y) 1)`

Exercise 11.7 *Open a new book in which to put your solutions to the following.*

- *Prove* `(defcong perm perm (append x y) 1)`.
- *Prove* `(defcong perm perm (append x y) 2)`.

An interesting sorting algorithm is quicksort. The idea is to break a list in two, where the elements of the first list are those that are less than some pivot element and the second list contains the remaining elements. These two lists are then dealt with recursively. Our treatment of this problem ignores the important fact that this processing can be done *in situ*.

Exercise 11.8 *Define the function* `less` *that takes two arguments,* `x` *and* `lst`*, and returns the elements of* `lst` *that are less than* `x` *(in the sense of* `<`*).*

Exercise 11.9 *Define the function* `notless` *that takes two arguments,* `x` *and* `lst`*, and returns the elements of* `lst` *that are not less than* `x` *(in the sense of* `<`*).*

Given the above definitions, we define the function `qsort` as follows.

```
(defun qsort (x)
  (cond ((atom x) nil)
        (t (append (qsort (less (car x) (cdr x)))
                   (list (car x))
                   (qsort (notless (car x) (cdr x)))))))
```

Exercise 11.10 *Prove* `(perm (qsort x) x)`.

Exercise 11.11 *Define the Boolean valued function* `lessp` *that takes two arguments,* `x` *and* `lst`*, and returns* `t` *iff every element of* `lst` *is less than* `x` *(in the sense of* `<`*).*

Exercise 11.12 *Define the Boolean valued function* `notlessp` *that takes two arguments,* `x` *and* `lst`*, and returns* `t` *iff every element of* `lst` *is not less than* `x` *(in the sense of* `<`*).*

Exercise 11.13

- *Prove* `(defcong perm equal (lessp x lst) 2)`.
- *Prove* `(defcong perm equal (notlessp x lst) 2)`.

Exercise 11.14 *Prove* `(orderedp (qsort lst))`.

Exercise 4.15, on page 62, asked that you define the function `mergesort`. Before continuing, make sure that you have done the exercise.

Exercise 11.15 *Prove* `(orderedp (mergesort lst))`.

Exercise 11.16 *Prove* `(perm (mergesort lst) lst)`.

11.3 Compressed Lists

In this section, you will be asked to prove theorems about a function that *compresses* lists, *i.e.*, a function that removes adjacent duplicates.

Exercise 11.17 *Define a function to compress a list. Given a list of elements,* `compress` *returns the list with all adjacent duplicates removed,* e.g., `(compress '(x x x y z y x y y))` *is equal to* `'(x y z y x y)`.

Exercise 11.18 *Prove the following.*

```
(equal (compress (compress x)) (compress x))
```

Exercise 11.19 *Prove the following.*

```
(equal (compress (append (compress x) y))
       (compress (append x y))
```

Exercise 11.20 *Recall the recognizer* `orderedp` *for ordered lists, defined on page 192. An exercise on page 58 asked for the definition of a recognizer* `no-dupls-p` *for duplicate-free lists. Formulate and prove a theorem stating that the application of* `compress` *to an ordered list is duplicate-free. You may need an additional hypothesis.*

Exercise 11.21 *Define* `same-compress`*, a Boolean function of two arguments that returns* `t` *iff* `compress` *applied to one of the arguments is* `equal` *to* `compress` *applied to the other.*

Exercise 11.22 *Prove* `(defequiv same-compress)`.

Exercise 11.23 *Prove the following.*

```
(defcong same-compress same-compress (append x y) 2).
```

Exercise 11.24 *Prove the following.*

```
(defcong same-compress same-compress (append x y) 1).
```

Exercise 11.25 *Prove the following.*

```
(equal (rev (compress x))
       (compress (rev x)))
```

The function `rev` *reverses a list and was defined on page 124.*

11.4 Summations

In this section, we present a sequence of equations containing summations and ask that you formalize the summations in ACL2 and determine whether or not they hold. Note that $\sum_{i=1}^{n} f(i) = f(1) + \cdots + f(n)$. Try doing these exercises in the `ground-zero` theory, *i.e.*, in a new ACL2 session. Afterwards, do the exercises once more, but use one of the arithmetic books that comes with the ACL2 distribution; `top-with-meta` is one such book.

Exercise 11.26 *Formalize the following in ACL2.*

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

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.27 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} (3i^2 - 3i + 1) = n^3$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.28 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.29 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n}(2i)^2 = \frac{2n(n+1)(2n+1)}{3}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.30 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.31 *Formalize the following in ACL2.*

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

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.32 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n}(4i-1) = n(2n+1)$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.33 *Formalize the following in ACL2.*

$$\left(\sum_{i=1}^{n} i\right)^2 = \sum_{i=1}^{n} i^3$$

Is it a theorem? If so, prove it; if not, give a counterexample.

11.5 Tautology Checking

In this section, we define a notion of `if`-expressionand prove the correctness of a simple tautology checker for such expressions.

The following function recognizes expressions whose top function symbol is `if`.

```
(defun ifp (x)
  (and (consp x)
       (equal (car x) 'if)))
```

The following functions extract the test, the true branch, and the false branch of an `if`.

```
(defun test (x)
  (second x))

(defun tb (x)
  (third x))

(defun fb (x)
  (fourth x))
```

An expression whose only function symbol is `if` is called an `if`-expression. For example, `(if (if a b c) d e)` is an `if`-expression, but `(if (foo (if a b c)) d e)` is not.

Exercise 11.34 *Define the function* `if-exprp` *to recognize* `if`*-expressions.*

An `if`-expression is *normalized* if no `if` subexpression contains an `if` in its test. Here is a naive attempt at normalizing such expressions.

```
(defun if-n (x)
  (if (ifp x)
      (let ((test (test x))
            (tb (tb x))
            (fb (fb x)))
        (if (ifp test)
            (if-n (list 'if (test test)
                        (list 'if (tb test) tb fb)
                        (list 'if (fb test) tb fb)))
          (list 'if test (if-n tb) (if-n fb))))
    x))
```

Exercise 11.35 *Add the above definition in* :program *mode and execute it on several examples,* e.g., *try* `(if-n '(if (if (if a b c) d e) e b))`.

Notice that

`(if-n '(if (if` α β γ`)` δ ϵ`))`

expands to

`(if-n '(if` α `(if` β δ ϵ`) (if` γ δ ϵ`))).`

Hence, the termination argument requires some thought.

Exercise 11.36 *Admit* `if-n` *in* :logic *mode.*

Exercise 11.37 *Admit* `if-n` *with a natural-number-valued measure function. (We suspect that the measure function you used for the previous exercise returned ordinals past the naturals.)*

Exercise 11.38 *Define* `(peval x a)` *to determine the value of an* `if`*-expression under the alist* `a`*. For example, if* `x` *is* `(if (if t c b) c b)` *and* `a` *is* `((c . t) (b . nil))`*, then* `(peval x a)` *is* `t`*. Notice that* `t` *and* `nil` *retain their status Boolean constants.*

Exercise 11.39 *Define* `(tautp x)` *to recognize* tautologies*:* `if`*-expressions that evaluate to* `t` *under all alists. (*Hint*: Consider normalizing the* `if`*-expressionand then exploring all paths through it.)*

Exercise 11.40 *Prove that* `tautp` *is sound: when* `tautp` *returns* `t`*, its argument evaluates to non-*`nil` *under every alist.*

Exercise 11.41 *Prove that* `tautp` *is complete: when* `tautp` *returns* `nil`*, there is some alist under which the* `if`*-expression evaluates to* `nil`*.*

11.6 Encapsulation

In this section, we will use encapsulation and functional instantiation to prove the equivalence of two functions that apply an associative and commutative function to a list of objects. We will use similar techniques to prove an important theorem about permutations.

Exercise 11.42 *Use* `encapsulate` *to introduce the function* `ac` *which is constrained to be associative and commutative. (See also Section 10.2, page 187.)*

The following function applies `ac` to a list of elements.

```
(defun map-ac (lst)
  (cond ((endp lst) nil)
        ((endp (cdr lst)) (car lst))
        (t (ac (car lst) (map-ac (cdr lst))))))
```

`Map-act`, defined below, is similar to `map-ac`, but it is tail recursive.

```
(defun map-act-aux (lst a)
  (cond ((endp lst) a)
        (t (map-act-aux (cdr lst) (ac (car lst) a)))))

(defun map-act (lst)
  (cond ((endp lst) nil)
        (t (map-act-aux (cdr lst) (car lst)))))
```

Notice that we used a helper function to define `map-act`.

Exercise 11.43 *Prove* `(equal (map-act lst) (map-ac lst))`*.* (Hint*: You may find the macro* `commutativity-2`*, defined on page 191, useful.)*

Consider the following function that returns the maximum of two numbers.

```
(defun maxm (a b)
  (if (< a b)
      (fix b)
    (fix a)))
```

Exercise 11.44 *Prove that* `maxm` *is associative and commutative.*

Exercise 11.45 *Define the functions* `map-maxm` *and* `map-maxmt` *to apply* `maxm` *to a list.* `Map-maxm` *and* `map-maxmt` *correspond to* `map-ac` *and* `map-act`, *respectively.*

Exercise 11.46 *Use functional instantiation (see* `lemma-instance`*) in order to prove the following.*

```
(equal (map-maxmt lst) (map-maxm lst))
```

Notice that we can use the above approach to prove that the tail-recursive version of any function applying an associative and commutative function to a list is equal to the simpler version of the function. Rewriting complicated functions into simpler functions and reasoning about the simpler functions is an example of *compositional reasoning.* Such decomposition replaces problems with manageable pieces.

11.7 Permutation Revisited

Recall the function of `mergesort` (page 62). In this section we revisit the proof that `mergesort` returns a permutation of its input, which you were asked to prove in Exercise 11.16. But this time we are less interested in `mergesort` *per se* than in developing a "new" way to prove theorems about `perm`.

Exercise 11.47 *Define the function* `how-many` *so that* `(how-many e x)` *determines how many times* `e` *occurs as an element of the list* `x`.

Exercise 11.48 *Prove the following.*

```
(equal (how-many e (mergesort lst))
       (how-many e lst))
```

Many people would consider the theorem in Exercise 11.48 to be equivalent to `(perm (mergesort lst) lst)`. Indeed, in a suitable logic permitting ACL2 terms and quantification,

```
 (perm (mergesort lst) lst)
↔
 (∀ e [(how-many e (mergesort lst)) = (how-many e lst)] )
```

is a theorem.

Exercise 11.49 *The universal quantifier in the theorem above is crucial. The similar-looking formula*

```
(iff (perm (mergesort lst) lst)
     (equal (how-many e (mergesort lst))
            (how-many e lst)))
```

is not a theorem. Construct a counterexample.

Exercise 11.50 *Find a general way to use the theorem proved in Exercise 11.48 to prove* `(perm (mergesort lst) lst)`. *If you wish, you may add* `true-listp` *hypotheses to make the problem easier.*

You may not want to do Exercise 11.50 now. But you may someday find yourself struggling to prove theorems about `perm` and you should remember this: it is possible to convert permutation problems into `how-many` problems, which are often easier to solve. Contrast your solutions of Exercises 11.16 and 11.48.

We offer several solutions to Exercise 11.50 on the Web page. In one, we constrain two constants, `(alpha)` and `(beta)`, to have the property `(equal (how-many e (alpha)) (how-many e (beta)))` and then prove `(perm (alpha) (beta))`. In our proof, we define `(bounded-quantifierp x a b)` to check, for each element `e` in `x`, that `(how-many e a)` is equal to `(how-many e b)`. We then relate `bounded-quantifierp` to `perm`.

The next exercise is valuable even if you do not tackle Exercise 11.50 now. It will teach you something very important about functional instantiation.

Exercise 11.51 *How can the theorem* `(perm (alpha) (beta))`, *above, be used to prove* `(perm (mergesort lst) lst)`? *To work on this problem, first constrain* `(alpha)` *and* `(beta)` *as described above. Then pretend you proved the* `perm` *theorem by executing the following.*

```
(skip-proofs
 (defthm perm-alpha-beta
   (perm (alpha) (beta))))
```

Now prove the theorem `(perm (mergesort lst) lst)` *by functional instantiation. (*Hint: *This is easy once you see the power of functional instantiation.)*

In another solution to Exercise 11.50 we use a common "trick" in dealing with quantification in this setting: we define a function that exhibits a "bad guy," *i.e.*, a function that finds an `e` that occurs a different number of times in `a` than in `b` when `(perm a b)` is false.

11.8 The Extractor Problem

The following function builds a list containing the first n natural numbers, in reverse order.

```
(defun nats (n)
  (if (zp n)
      nil
    (cons (- n 1) (nats (- n 1)))))
```

The following function builds a list by recurring on map and consing the n^{th} element of lst to the result, where n is the car of map.

```
(defun xtr (map lst)
  (if (endp map)
      nil
    (cons (nth (car map) lst)
          (xtr (cdr map) lst))))
```

Exercise 11.52 *Prove* `(equal (xtr (nats (len x)) x) (rev x))`.

11.9 Finite Set Theory

We have seen that ACL2 has built-in functions to manipulate sets. Examples of such functions are member and subsetp. These functions assume a *flat* representation of sets. For example, `(subsetp '(1 2) '(2 1))` is t, but `(subsetp '((1 2)) '((2 1)))` is nil. The first expression corresponds to $\{1,2\} \subseteq \{2,1\}$, which in set theory is true, but the second expression can be viewed as corresponding to $\{\{1,2\}\} \subseteq \{\{2,1\}\}$, which in set theory is also true. Below, you will be asked to define *general* finite set theory functions, *i.e.*, ACL2 functions that do not assume sets are flat; for example, they consider `((1 2))` to be a subset of `((2 1))`. This exercise may be harder than it seems. Part of the problem is getting the definitions right and we are intentionally leaving some ambiguity in the next exercise so that you can explore various possibilities.

Exercise 11.53 *Define the functions* in, =<, *and* == *that correspond to set membership, subset, and set equality, respectively. These functions should be general set theory functions, as discussed above. (*Hint: *You may find it useful to use* mutual-recursion.*)*

Exercise 11.54 *Test your functions above on the following examples.*

1. `(== '((1 2)) '((2 1)))`
2. `(=< '((2 1) (1 2)) '((2 1)))`

3. `(in '((1)) '((2 (1)) (1 2)))`

4. `(== '(() (1 2 1) (2 1) x) '((1 2) x ()))`

5. `(== 'x 1)`

6. `(=< 'x 1)`

Exercise 11.55 *Prove* `(=< X X)`.

Exercise 11.56 *Prove the following.*
`(implies (and (=< X Y) (=< Y Z)) (=< X Z))`

Exercise 11.57 *Prove* `(defequiv ==)`.

A

Using the ACL2 System

A.1 Introduction

This appendix tells you the basics you need to know in order to use the ACL2 system. After some preliminaries, we explain the fundamental read-eval-print nature of interaction with ACL2. We then describe how to give commands that update the state of the system, *e.g.*, with definitions and theorems. We conclude by describing the structure of the documentation, illustrating its use with an example and presenting an abbreviated outline.

A.1.1 Getting Started

ACL2 is publicly available on the Web at `http://www.cs.utexas.edu/-users/moore/acl2/`. You are encouraged to explore this page, which has links to introductory material, to useful email addresses including an ACL2 mailing list, to research papers, and to the system itself along with instructions on how to obtain and install it.

A.1.2 Conventions

ACL2 has extensive documentation which is organized by topics. When a topic occurs in this book, recall that we may call attention to its status as a documentation topic by underlining it in typewriter font. For example, documentation is itself such a topic.

This appendix is not intended to require any prior knowledge of the programming language underlying ACL2. It suffices for now to understand that ACL2 syntax uses *prefix notation*: for function calls the operator appears before the operands, all within parentheses. For example, the sum of the numbers 5 and 7 is written `(+ 5 7)`.

In the following sample transcript, only the expression `(+ 5 7)` is typed in by the user. The rest is printed by the system, including the ACL2 prompt.

```
ACL2 !>(+ 5 7)
```

```
12
ACL2 !>
```

We will follow this convention throughout this appendix: user input immediately follows prompts, and everything else in the display is printed by the system.

A.2 The Read-Eval-Print Loop

This section describes the basics of interacting with ACL2. See also ld.

A.2.1 Entering ACL2

When you follow the installation instructions, you will create an executable image. In some Lisps, you will then need to invoke the command (lp) ("loop") in order to enter the ACL2 read-eval-print loop that is described below. Here, for example, is how one starts up ACL2 using Allegro Common Lisp, assuming that the ACL2 executable is called acl2. In GCL and perhaps other Lisps, you will immediately see the ACL2 prompt, ACL2 !>, in which case you should skip (lp).

```
% acl2
Allegro CL Enterprise Edition 5.0 [SPARC] (8/29/98 12:15)
 [[ additional output omitted ]]
ACL2(1): (lp)

ACL2 Version 2.5.  Level 1.  Cbd "/user/smith/".
Type :help for help.

ACL2 !>
```

A.2.2 Read, Eval, and Print

Here again is the example presented above.

```
ACL2 !>(+ 5 7)
12
ACL2 !>
```

This example illustrates interaction with ACL2, namely, using a *read-eval-print* loop.

- **Read:** The system reads the user's input expression, (+ 5 7).
- **Eval:** The system evaluates this input, adding 5 and 7.

- **Print:** The system prints out the result returned by the evaluation, 12.

Numerous operations are built into ACL2 besides the addition operator (+) shown above. In particular, the operator thm directs ACL2 to attempt the proof of a theorem. The transcript below shows a use of thm that causes ACL2 to prove the commutativity of addition: the sum of x and y, (+ x y), is equal to the sum of y and x, (+ y x). Again, only the expression following the prompt was supplied by the user.

```
ACL2 !>(thm (equal (+ x y) (+ y x)))

But simplification reduces this to T, using linear arithmetic
and primitive type reasoning.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: ((:FAKE-RUNE-FOR-LINEAR NIL)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.02 seconds (prove: 0.00, print: 0.01, other: 0.01)

Proof succeeded.
ACL2 !>
```

We postpone discussion of the theorem prover, but the example above provides some idea of how to interact with the system. Notice that this example illustrates that printing can go on during evaluation. It also shows that the *print* phase of the read-eval-print loop is skipped for thm, as it is for a few other built-in operations.

A.2.3 Exiting ACL2

The most direct way to quit ACL2, including the underlying Lisp, may be to issue the keyword command :good-bye. (Keyword commands in general are discussed in Section A.3.2.)

```
ACL2 !>:good-bye
dork.cs.utexas.edu%
```

The :good-bye command works when the underlying Lisp is GCL or (starting with ACL2 Version 2.5) Allegro CL. It may not work with other Lisps. How else can you exit an ACL2 session?

You can probably quit ACL2, including the underlying Lisp, entirely from inside the ACL2 loop by typing control-d one or more times. In

Emacs, use the combination `control-c control-d` instead of `control-d`. In MacIntosh Common Lisp (MCL), use `command-q`.

But here is a more reliable approach than the one of the preceding paragraph. You can always leave the ACL2 loop by typing `:q`, as follows.

```
ACL2 !>:q

Exiting the ACL2 read-eval-print loop.  To re-enter, execute
(LP).
ACL2>
```

You could alternatively evaluate `(value :q)`. In either case, you are left at the Lisp prompt, in what we sometimes refer to as *raw Lisp*. Each Lisp provides its own quit command to execute in raw Lisp, including `(user::bye)` in GCL, `(excl::exit)` in Allegro Common Lisp, `(user::quit)` in CMU Common Lisp, and `(ccl::quit)` in MCL.

A.2.4 Dealing with Interrupts and Breaks

In many Common Lisps, an ACL2 session can be interrupted by typing `control-c` (in Emacs, two `control-c` characters). In MCL, use `command-,` (the comma character while holding down the command key). When the interrupt is seen, you are left in a *break* loop in raw Lisp. Certain input errors (as illustrated below) can also leave you in `breaks`.

Your Lisp provides a way to abort the break and return to the top-level. In some Lisps, this will take you all the way out to raw Lisp and you must type `(lp)` to get back into the ACL2 command loop. In other Lisps, it will take you back into the ACL2 command loop. You can often get from the break loop back to the ACL2 read-eval-print loop by typing the token '`#.`' followed by a carriage return, which implements the command `(abort!)`. Here is an illustration of the user being accidentally thrown into a break loop because of a typo (the comma), followed by recovery using '`#.`'. The break prompt of your Lisp may resemble the ACL2 prompt; pay attention to the prompt.

```
ACL2 !>(+ 3,245 7)

Error: Illegal comma encountered by READ.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by COND.
Broken at COND.  Type :H for Help.
ACL2>>#.
Abort to ACL2 top-level

ACL2 Version 2.5.  Level 1.  Cbd "/user/smith/".
Type :help for help.

ACL2 !>
```

Occasionally, '#.' may not work, in particular when there are stack overflows.[1] In that case, follow these steps.

1. Return from the break using :q for GCL, :reset for Allegro, and q for CMU Lisp. This should put you in the ACL2 loop.
2. Exit the ACL2 loop using :q.
3. Re-enter the ACL2 loop using (lp).

Note: We include Steps 2 and 3 because we have seen cases in which the underlying Lisp has had insufficient stack for further processing unless one first returns to the top level of raw Lisp.

A.3 Managing ACL2 Sessions

The example using thm above shows that ACL2 comes with a built-in logical data base that is sufficiently rich to allow ACL2 to prove that addition is commutative. However, the utility of ACL2 derives largely from the ability to extend that logical data base, which we call a *logical world* or world, with new definitions and theorems. When the user's input causes the logical world to be changed, that input is called a command. A single command can generate several events that extend the logical world.

The following *command* defines two functions: one named add3 that adds 3 to its input, x, and one named sub3 that subtracts 3 from its input, x. These two definition (defun) events will be used in examples below. This is a contrived example, since one rarely uses the sequencing operator progn in ACL2, but it illustrates the distinction between the notions of *command* (user input that creates at least one world-changing event) and *event* (a form that updates the ACL2 world when successfully evaluated).

```
(progn
  (defun add3 (x)
    (+ x 3))
  (defun sub3 (x)
    (- x 3))
  )
```

Once this command has been submitted to ACL2, the functions defined can be tested, for example as follows.

```
ACL2 !>(add3 9)
12
ACL2 !>(sub3 12)
9
ACL2 !>
```

[1] For Lisp experts: The problem is that Lisp can reset the variable *readtable*.

A.3.1 Viewing the ACL2 Logical World

The ACL2 user's goal is generally the extension of the built-in logical world, by defining functions and by proving theorems that express desired properties of those functions. Logical worlds are extended using commands such as the one shown above. In this section we give an idea of how to obtain views of the current logical world.

The simplest view of the current logical world is obtained using :pbt ("print back through"), which shows the current command history. For example, after submitting the form above, we can use :pbt as follows.

```
ACL2 !>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1:x(PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
ACL2 !>
```

Now let us extend the history by proving a theorem, saying that add3 and sub3 are inverses.

```
(defthm add3-sub3-inverses
  (implies (acl2-numberp x)
           (and (equal (add3 (sub3 x)) x)
                (equal (sub3 (add3 x)) x))))
```

ACL2 proves this theorem. We can see that the logical world has been appropriately extended, as follows.

```
ACL2 !>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1  (PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
          2:x(DEFTHM ADD3-SUB3-INVERSES ...)
ACL2 !>
```

The documentation for history provides ways to get more details about the current logical world. For example, :pcb ("print command block") gives an outline of a specified command, and :pe prints an event.

```
ACL2 !>:pcb 1
          1  (PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
 L             (DEFUN ADD3 (X) ...)
 L             (DEFUN SUB3 (X) ...)
ACL2 !>:pe add3
          1  (PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
             \
>L            (DEFUN ADD3 (X) (+ X 3))
ACL2 !>
```

The character L near the left margin indicates that function add3 was defined in logic mode, which we discuss in Section A.3.4 below.

A.3.2 Keyword Commands

A *keyword* is a word starting with a colon character (:).[2] A *keyword command* is a keyword followed by the number of expressions expected by that keyword. Above we saw this example of a keyword command.

```
:pbt 0
```

Quite a few keyword commands are provided for querying the logical world, some of them illustrated above. Most of them are documented under history, though an additional one is :args, which gives useful information about a given function symbol.

The full story on keyword commands is available in the documentation under keyword-commands.

A.3.3 Undoing

We have illustrated the basic ACL2 activity of extending the logical world. Let us view the logical world as suggested by the output from :pbt, namely as a command stack $\langle c_n, c_{n-1}, \ldots, c_0 \rangle$. You will probably want to pop commands from this stack on occasion. The keyword command :u ("undo") removes the most recent (top) command from this stack. More generally, the keyword command :ubt k ("undo back through") pops all commands back through (and including) c_k. In either case, if the undoing was done by mistake, the keyword command :oops may be used in order to reverse its effect. The following transcript illustrates the use of these keyword commands.

```
ACL2 !>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1  (PROGN (DEFUN ADD3 # ...)
                   (DEFUN SUB3 # ...))
          2:x(DEFTHM ADD3-SUB3-INVERSES ...)
ACL2 !>:u
          1:x(PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
ACL2 !>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1:x(PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
```

[2] Technically, a keyword is a *symbol* in a *package* called the KEYWORD package, but that need not concern us here.

```
ACL2 !>:oops

Installing the requested world.  Note that functions being
re-defined during this procedure will not have compiled
definitions, even if they had compiled definitions before
the last :ubt or :u.

          2:x(DEFTHM ADD3-SUB3-INVERSES ...)
ACL2 !>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1  (PROGN (DEFUN ADD3 # ...)
                   (DEFUN SUB3 # ...))
          2:x(DEFTHM ADD3-SUB3-INVERSES ...)
ACL2 !>:ubt 1
          0:x(EXIT-BOOT-STRAP-MODE)
ACL2 !>:pbt 0
          0:x(EXIT-BOOT-STRAP-MODE)
ACL2 !>
```

A.3.4 Program and Logic Modes

The default defun-mode for interacting with ACL2 is called logic mode. It allows the user to present definitions that add corresponding axioms to the ACL2 logical world. For example, the definition of add3 above adds the equality of (add3 x) with (+ x 3). Logic mode makes the ACL2 proof engine available, but it imposes proof obligations for recursive definitions, as described in Chapter 6. A second mode, program mode, is provided in which one can prototype functions without worrying about proof obligations. See also default-defun-mode.

Let us try our examples by first entering program mode using the keyword command :program. Notice the resulting change in the prompt.

```
ACL2 !>:pbt 0
          0:x(EXIT-BOOT-STRAP-MODE)
ACL2 !>:program
ACL2 p!>(progn
          (defun add3 (x)
            (+ x 3))
          (defun sub3 (x)
            (- x 3))
          )

Summary
Form:  ( DEFUN ADD3 ...)
Rules: NIL
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, other: 0.01)
```

```
Summary
Form:  ( DEFUN SUB3 ...)
Rules: NIL
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, other: 0.01)
 SUB3
ACL2 p!>(add3 9)
12
ACL2 p!>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1  (PROGRAM)
          2:x(PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
ACL2 p!>(defthm add3-sub3-inverses
          (implies (acl2-numberp x)
                   (and (equal (add3 (sub3 x)) x)
                        (equal (sub3 (add3 x)) x))))

ACL2 Observation in TOP-LEVEL:  DEFTHM events are skipped
when the default-defun-mode is :PROGRAM.
 NIL
ACL2 p!>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1  (PROGRAM)
          2:x(PROGN (DEFUN ADD3 # ...)
                    (DEFUN SUB3 # ...))
ACL2 p!>:logic
ACL2 !>
```

The transcript above illustrates that the proof engine is turned off in program mode. So we have switched back to logic mode. Again, notice the change in the prompt. However, if we try to prove a theorem about add3 and sub3, we get an error.

```
ACL2 !>(defthm add3-sub3-inverses
          (implies (acl2-numberp x)
                   (and (equal (add3 (sub3 x)) x)
                        (equal (sub3 (add3 x)) x))))


ACL2 Error in ( DEFTHM ADD3-SUB3-INVERSES ...):  Function
symbols of mode :program are not allowed in the present
context.  Yet, the function symbols SUB3 and ADD3 occur
in the translation of the form


(IMPLIES (ACL2-NUMBERP X)
         (AND (EQUAL (ADD3 (SUB3 X)) X)
```

```
              (EQUAL (SUB3 (ADD3 X)) X))),
```

which is

```
(IMPLIES (ACL2-NUMBERP X)
         (IF (EQUAL (ADD3 (SUB3 X)) X)
             (EQUAL (SUB3 (ADD3 X)) X)
             'NIL)).
```

```
Summary
Form:  ( DEFTHM ADD3-SUB3-INVERSES ...)
Rules: NIL
Warnings:  None
Time:  0.05 seconds (prove: 0.00, print: 0.00, other: 0.05)

******** FAILED ********  See :DOC failure  ******** FAILED ********
ACL2 !>:pbt 0
          0  (EXIT-BOOT-STRAP-MODE)
          1  (PROGRAM)
          2  (PROGN (DEFUN ADD3 # ...)
                   (DEFUN SUB3 # ...))
          3:x(LOGIC)
ACL2 !>
```

The problem is clearly indicated using the keyword command :pcb, which uses the character 'P' to indicate that our functions were defined in program mode.

```
ACL2 !>:pcb 2
          2  (PROGN (DEFUN ADD3 # ...)
                   (DEFUN SUB3 # ...))
 P             (DEFUN ADD3 (X) ...)
 P             (DEFUN SUB3 (X) ...)
ACL2 !>
```

The reader interested in how to remedy this situation is invited to see the documentation for verify-termination.

Thus, the notion of *logical world* introduced above has a slightly misleading name. There is no logical axiom added when a function definition is made in :program mode, but the "logical" world is extended in order to support execution of that function.

A.3.5 Mechanics of ACL2 Interaction

Most successful ACL2 users run ACL2 inside an Emacs editor [38], specifically, in a shell running in a buffer that we refer to below as the **shell** *buffer*. The ACL2 user typically develops an ACL2 input file interactively by iterating through the following steps.

1. *Edit* commands in an *script buffer* that is connected to the input file under development.
2. *Submit* commands by copying commands from that script buffer into the `*shell*` buffer, and hitting a carriage return.
3. *Scroll* through the `*shell*` buffer to examine ACL2 output, especially if the command fails in any sense.

It is a good idea to avoid editing directly in the `*shell*` buffer. The above method both allows you to keep a record of what you have done by saving the script buffer on disk, and avoids problems that can arise when editing in a `*shell*` buffer. In particular, if a carriage return is hit in the middle of an input form, then the partial form is submitted; hence, an attempt to edit the submitted form will most likely lead to undesirable consequences. Specific suggestions for how to develop ACL2 input files are given in Chapter 9.

A.4 Using the Documentation

In this section we outline the structure and use of the ACL2 documentation. We remind the reader that topics in the User's Manual may be underlined in typewriter font. In particular, we recommend the topic acl2-tutorial, which has subtopics providing an introduction, tidbits, tips, and tutorial-examples.

A.4.1 Where to Find the Documentation

Below, all file names are relative to the ACL2 distribution (*i.e.*, directory `acl2-sources/` on your computer).

A good starting point for newcomers to ACL2 is the home page for ACL2, which may be found in the file `doc/HTML/acl2-doc.html` (in the ACL2 directory) or on the Web at `http://www.cs.utexas.edu/users/moore/acl2/`. Among the links on the home page are the following ones that lead to documentation.

- The User's Manual

- Two short tours of ACL2
- Hyper-Card

The User's Manual contains comprehensive documentation organized as a structured collection of topics. We recommend that beginners start with the tours, followed by the topic `acl2-tutorial` in the User's Manual. The Hyper-Card provides a concise, organized collection of links into the rest of the documentation.

A.4.2 The User's Manual

The ACL2 User's Manual is available in several formats, accessible as follows.

- HTML (for Web browsers):
  `doc/HTML/acl2-doc-major-topics.html`
- Emacs info:
  `doc/EMACS/acl2-doc-emacs.info`
- Hardcopy (over 800 pages; use `gunzip` to obtain Postscript):
  `doc/TEX/acl2-book.ps.gz`

The documentation can also be made available inside ACL2 sessions, but since the other (hypertext) formats are probably more useful, the ACL2 executable is built by default with such documentation omitted in order to save space.[3]

A tour through a fairly large collection of documentation topics may be found in Appendix B.

A.4.3 Documentation Example: Hints

Consider the following problem. You have submitted a theorem to ACL2, but the proof fails. However, you believe that ACL2 might find a proof if it can somehow use an instance of a theorem you have already proved. So, you pursue the following steps.

1. You look in the documentation for `defthm`, where you see a way to supply hints to the prover, together with a link to the documentation topic `hints`.

[3]The `makefile` target `large` may be used to build the executable image if in-session documentation is desired.

2. You check the documentation topic hints and see a reference to :use hints. There is not any detail there on how to use an instance of a previously-proved theorem, but you see a link to lemma-instance.

3. You follow that link and see an example illustrating what to do in the current case. You read the entire topic and learn that your hint can even tell the prover to use a theorem that is not yet proved, after first proving that additional theorem.

A.4.4 Outline of the User's Manual

Below we provide an abbreviated outline of the User's Manual. The reader may find it useful to take a quick top-down look through the outline.

```
* acl2-tutorial:: tutorial introduction to ACL2
  - introduction:: introduction to ACL2
  - startup:: How to start using ACL2; the ACL2 command loop
  - tidbits:: some basic hints for using ACL2
  - tips:: some hints for using the ACL2 prover
     A. ACL2 Basics
     B. Strategies for creating events
     C. Dealing with failed proofs
     D. Performance tips
     E. Miscellaneous tips and knowledge
     F. Some things you DON'T need to know
  - tutorial-examples:: examples of ACL2 usage
* bdd:: ordered binary decision diagrams with rewriting
* books:: files of ACL2 event forms
  - book-example:: how to create, certify, and use a simple book
  - certify-book:: how to produce a certificate for a book
  ...
* break-rewrite:: the read-eval-print loop entered to monitor rewrite rules
* documentation:: functions that display documentation at the terminal
  - args:: args, guard, type, constraint, etc., of a function symbol
  - doc-string:: formatted documentation strings
  - markup:: the markup language for ACL2 documentation strings
  - nqthm-to-acl2:: ACL2 analogues of Nqthm functions and commands
  ...
* events:: functions that extend the logic
  - defabbrev:: a convenient form of macro definition for simple expansions
  - defaxiom:: add an axiom
  - defchoose:: define a Skolem (witnessing) function
  - defcong:: prove that one equivalence relation preserves another in a given
   argument position of a given function
  - defconst:: define a constant
  - defequiv:: prove that a function is an equivalence relation
  - defevaluator:: introduce an evaluator function
  - deflabel:: build a landmark and/or add a documentation topic
  - defmacro:: define a macro
  - defpkg:: define a new symbol package
  - defrefinement:: prove that equiv1 refines equiv2
  - defstobj:: define a new single-threaded object
  - defstub:: stub-out a function symbol
  - deftheory:: define a theory (to enable or disable a set of rules)
  - defthm:: prove and name a theorem
  - defun:: define a function symbol
  - defun-sk:: define a function whose body has an outermost quantifier
  - encapsulate:: constrain some functions and/or hide some events
  - in-theory:: designate "current" theory (enabling its rules)
  - include-book:: load the events in a file
  - local:: hiding an event in an encapsulation or book
```

```
  - logic:: to set the default defun-mode to :logic
  - mutual-recursion:: define some mutually recursive functions
  - program:: to set the default defun-mode to :program
  - set-bogus-mutual-recursion-ok:: allow unnecessary "mutual recursion"
  - set-compile-fns:: have each function compiled as you go along.
  - set-ignore-ok:: allow unused formals and locals
  - set-inhibit-warnings:: control warnings
  - set-measure-function:: set the default measure function symbol
  - set-state-ok:: allow the use of STATE as a formal parameter
  - set-verify-guards-eagerness:: the eagerness with which guard verification is tried.
  - table:: user-managed tables
  - verify-guards:: verify the guards of a function
  - verify-termination:: convert a function from :program mode to :logic mode
  ...
* history:: functions that display or change history
  - oops:: undo a :u or :ubt
  - pbt:: print the commands back through a command descriptor
  - pc:: print the command described by a command descriptor
  - pcb:: print the command block described by a command descriptor
  - pcb!:: print in full the command block described by a command descriptor
  - pe:: print the event named by a logical name
  - pe!:: print all the events named by a logical name
  - pl:: print the rules whose top function symbol is the given name
  - u:: undo last command, without a query
  - ubt:: undo the commands back through a command descriptor
  - ubt!:: undo commands, without a query or an error
  ...
* miscellaneous:: a miscellany of documented functions and concepts
                  (often cited in more accessible documentation)
  - abort!:: to return to the top-level of ACL2's command loop
  - accumulated-persistence:: to get statistics on which runes are being tried
  - acknowledgments:: some contributors to the well-being of ACL2
  - acl2-count:: a commonly used measure for justifying recursion
  - arrays:: an introduction to ACL2 arrays.
  - breaks:: Common Lisp breaks
  - case-split:: like force but immediately splits the top-level goal on the hypothesis
  - command:: forms you type at the top-level, but...
  - computed-hints:: computing advice to the theorem proving process
  - current-package:: the package used for reading and printing
  - default-defun-mode:: the default defun-mode of defun'd functions
  - defun-mode:: determines whether a function definition is a logical act
  - defun-mode-caveat:: functions with defun-mode of :program considered unsound
  - disable-forcing:: to disallow forced case splits
  - disabledp:: determine whether a given name or rune is disabled
  - e0-ord-<:: the well-founded less-than relation on ordinals up to epsilon-0
  - e0-ordinalp:: a recognizer for the ordinals up to epsilon-0
  - embedded-event-form:: forms that may be embedded in other events
  - enable-forcing:: to allow forced case splits
  - escape-to-common-lisp:: escaping to Common Lisp
  - executable-counterpart:: a rule for computing the value of a function
  - failed-forcing:: how to deal with a proof failure in a forcing round
  - failure:: how to deal with a proof failure
  - force:: identity function used to force a hypothesis
  - forcing-round:: a section of a proof dealing with forced assumptions
  - guard:: restricting the domain of a function
  - hide:: hide a term from the rewriter
  - hints:: advice to the theorem proving process
  - i-am-here:: a convenient marker for use with rebuild
  - immediate-force-modep:: when executable counterpart is enabled,
   forced hypotheses are attacked immediately
  - in-package:: select current package
  - invisible-fns-alist:: functions that are invisible to the loop-stopper algorithm
  - keyword-commands:: how keyword commands are processed
  - lemma-instance:: an object denoting an instance of a theorem
  - lp:: the Common Lisp entry to ACL2
  - otf-flg:: pushing all the initial subgoals
  - prompt:: the prompt printed by ld
```

- redef:: a common way to set ld-redefinition-action
- redundant-events:: allowing a name to be introduced "twice"
- simple:: :definition and :rewrite rules used in preprocessing
- state:: the von Neumannesque ACL2 state object
- syntaxp:: to attach a heuristic filter on a :rewrite rule
- term:: the three senses of well-formed ACL2 expressions or formulas
- term-order:: the ordering relation on terms used by ACL2
- using-computed-hints:: how to use computed hints
- world:: ACL2 property lists and the ACL2 logical data base
- xargs:: giving hints to defun

...

* other:: other commonly used top-level functions
  - atsign:: (@) get the value of a global variable in state
  - assign:: assign to a global variable in state
  - comp:: compile some ACL2 functions
  - good-bye:: quit entirely out of Lisp
  - ld:: the ACL2 read-eval-print loop, file loader, and command processor
  - props:: print the ACL2 properties on a symbol
  - q:: quit ACL2 (type :q) -- reenter with (lp)
  - rebuild:: a convenient way to reconstruct your old state
  - set-guard-checking:: control checking guards during execution of top-level forms
  - set-inhibit-output-lst:: control output
  - skip-proofs:: skip proofs for an event -- a quick way to introduce unsoundness
  - thm:: prove a theorem
  - trans:: print the macroexpansion of a form
  - trans1:: print the one-step macroexpansion of a form

  ...

* programming:: built-in ACL2 functions -- this section lists over 250
  function symbols, most of which are Common Lisp functions. Some are
  ACL2-specific.
  - acl2-numberp:: recognizer for numbers
  - acl2-user:: a package the ACL2 user may prefer
  - case:: conditional based on if-then-else using eql
  - case-match:: pattern matching or destructuring
  - cond:: conditional based on if-then-else
  - compilation:: compiling ACL2 functions
  - declare:: declarations
  - illegal:: cause a hard error
  - io:: input/output facilities in ACL2
  - irrelevant-formals:: formals that are used but only insignificantly
  - mv:: returning multiple values
  - mv-let:: calling multi-valued ACL2 functions
  - pprogn:: evaluate a sequence of forms that return state
  - type-spec:: type specifiers in declarations
  - zero-test-idioms:: how to test for 0

  ....

* proof-checker:: support for low-level interaction
  - verify:: enter the interactive proof checker

  ...

* proof-tree:: proof tree displays
* release-notes:: pointers to what has changed
* rule-classes:: adding rules to the data base
  - built-in-clauses:: to build a clause into the simplifier
  - compound-recognizer:: make a rule used by the typing mechanism
  - congruence:: the relations to maintain while simplifying arguments
  - definition:: make a rule that acts like a function definition
  - elim:: make a destructor elimination rule
  - equivalence:: mark a relation as an equivalence relation
  - forward-chaining:: make a rule to forward chain when a certain trigger arises
  - generalize:: make a rule to restrict generalizations
  - induction:: make a rule that suggests a certain induction
  - linear:: make some arithmetic inequality rules
  - linear-alias:: make a rule to extend the applicability of linear arithmetic
  - meta:: make a :meta rule (a hand-written simplifier)
  - refinement:: record that one equivalence relation refines another
  - rewrite:: make some :rewrite rules (possibly conditional ones)
  - type-prescription:: make a rule that specifies the type of a term

```
  - type-set-inverter:: exhibit a new decoding for an ACL2 type-set
  - well-founded-relation:: show that a relation is well-founded on a set
* stobj:: single-threaded objects or "von Neumann bottlenecks"
* theories:: sets of runes to enable/disable in concert
* index:: An item for each documented ACL2 item.
```

B

Additional Features

One reason ACL2 can be applied to practical problems is that a lot of engineering effort has gone into the system so that users with different needs can tailor the system, without changing the underlying logic. This appendix summarizes a number of aspects of ACL2 that enhance its usability, pointing to relevant online documentation for details.[1] Readers new to ACL2 may wish to look at this appendix to get a sense of the flexibility of ACL2. Even those who have used ACL2 may find this appendix valuable for bringing some useful features to light.

For example, function definitions can be compiled, which can be important for some practical applications. Some users are acutely aware of the compiler and others never use it. Rather than explain how to use the compiler and associated features we will (in Section B.5.1) just note a few of the features and name the relevant documentation topics. The reader who knows in advance that `compilation` will be important can chase down those documentation topics to learn the details. The larger group of readers who have not even considered the question of compiling logical definitions can just note that such a concept exists and is supported by various ACL2 features explained in the documentation.

We do not expect you to remember much of what you are about to read. We hope you will remember that many such features exist, that ACL2 has a lot of "odd corners" not described in this book, and that they are documented online. That way, if you start to use ACL2 and find yourself wishing you could do something unusual (but perfectly sensible for the application in mind), you will know that it might already be available and can search the documentation for the idea.

We recommend visiting the `tidbits` and `tips` pages. These pages overlap with the present appendix but may well evolve over time to include yet more material not covered below. We also suggest that the reader review the other appendix, which contains basic information on the use of ACL2. The documentation pages will continue to evolve as the ACL2 system evolves to contain more capabilities. This appendix describes features implemented in ACL2 Version 2.5.

[1] The ACL2 online documentation can be surfed on the Web, from the ACL2 home page, `http://www.cs.utexas.edu/users/moore/acl2`, without downloading the system. In this book we underline topics that can be found in that `documentation`.

We begin below by describing approaches to high-level interaction with ACL2. Sections B.2 and B.3 then turn to topics connected with the use of rules: first rewrite rules, then other kinds of rules. We then turn to execution issues, first considering IO and the ACL2 `state` in Section B.4 and then turning to efficiency considerations in Section B.5. After describing a few remaining topics in Section B.6 we conclude with a brief description of some common problems and corresponding solutions in Section B.7.

B.1 Proof/Session Management

Here we give a hodgepodge of hints for successful interaction with ACL2 at the `command` level.

A discussion of the ACL2 read-eval-print loop may be found in `ld` (which stands for "load," a name already claimed by Common Lisp). The meaning of the prompt is explained in `prompt`. ACL2's *keyword command convention*, whereby, *e.g.*, `:pe car-cons` is treated like `(pe 'car-cons)`, is described in `keyword-commands`.

B.1.1 Structuring Mechanisms

It is often useful to structure proof development efforts into `books`. Books are read into the database by `include-book`. Events in a book that are not of interest outside the book may be declared `local`. Local events are not exported from the book by `include-book`.

B.1.2 Top-Down Proof

ACL2 lets you defer proofs by using `defaxiom` and `skip-proofs`. `Defaxiom` has the advantage that it is sound, in the sense that the system looks for these events and intends to guarantee that all theorems that you prove are logical consequences of the definitions and `defaxioms`. Use `defaxiom` when you do not expect the formula to be a theorem in the current `history`. Use `skip-proofs` when you are merely deferring a proof, but be careful not to trust ACL2 when you are using `skip-proofs` because it is taking your word that the proof obligations can be met.

See also `ld-skip-proofsp` and `ld`.

The companion volume [22] says more about top-down proof methods. There, Moore's case study shows how to define a simple macro, `top-down`, to structure proofs in a top-down style. Kaufmann's case study presents a more elaborate top-down style that uses `books`, `local`, `encapsulate`, and `skip-proofs` in combination to structure proof development efforts in a modular manner that supports both proof presentation and robustness of proof replay.

B.1.3 Starting a Session

Some ACL2 users find it helpful to load (using ld) an input file when ACL2 starts up. ACL2 looks for an optional file acl2-customization.lisp when first entering the read-eval-print loop. See acl2-customization.

How does one load an initial part of a book? One way is to use include-book, then flatten it into individual commands using :puff, and finally undo back through the desired starting point using :ubt or :ubt!. The following more direct approaches work even if the input file is not a book, *e.g.*, contains a form that is not an embedded-event-form. Rebuild lets you load forms, without proof, while specifying the last form to be loaded. The same effect can be achieved by placing the form (i-am-here) just after the last form that you have previously gotten ACL2 to accept, and then using ld with keyword :ld-skip-proofsp set to t.

B.1.4 Redefinition and Undoing

ACL2 keeps track of the commands you have executed and provides a variety of ways you can inspect and alter them. See history. For example, to undo the last command, use :u. To undo back through a given point, use :ubt. To undo a recent undo, use :oops.

Suppose you want to change the definition of a previously-defined function. The basic technique for doing so is to undo all commands back through the one that defined the function, using :ubt (or :ubt!). There is no way to undo a command or event that is not the most recent one. Thus, you may have to redo the subsequent events. The most basic way is with ld, which may be given a list of commands instead of a file name.

ACL2 provides an alternative approach to redefinition, using redef; see also ld-redefinition-action for more details. There is some risk in this approach, because the "theorems" already proved about a function may no longer be theorems after the redefinition. Hence, books may not contain :redef or related commands; see embedded-event-form. The function redefined-names will tell you which functions have been redefined.

Redefinition may be applied to most events, not just definitions. Notice that a second definition or theorem that is identical to the first is not considered a redefinition; see redundant-events.

B.1.5 Proof Trees

See proof-tree for how to use Emacs to obtain a concise view of the stream of proof output. ACL2 provides tools to use this tree to locate key parts of the output when attempting to debug failed proof attempts.

B.1.6 Modes and Switches

ACL2 provides the following means for modifying its default behavior.

- `:set-bogus-mutual-recursion-ok` allows `mutual-recursion` to be used even when one of the functions being defined does not call any of the others
- `:set-compile-fns` allows `compilation` of new definitions
- `:set-ignore-ok` allows unused formals and locals without explicit (`declare` (`ignore` ...))
- `:set-inhibit-warnings` controls warnings
- `:set-invisible-fns-alist` controls application of permutative rewrite rules (see `loop-stopper` and see Section B.2.9)
- `:set-irrelevant-formals-ok` allows definitions to have irrelevant formals
- `:set-measure-function` sets the default measure function
- `:set-state-ok` allows the use of `state` as a formal parameter
- `:set-verify-guards-eagerness` determines when to try `guard` verification
- `:set-well-founded-relation` sets the default well-founded relation (see `well-founded-relation`)
- `add-macro-alias` and `remove-macro-alias` control the association of function names with macro names
- `:set-guard-checking` controls the checking of `guards` during execution of top-level forms
- `:set-inhibit-output-lst` controls output

B.2 Working with the Rewriter

See page 149 for the basics of rewriting in ACL2.

The title of this section says it all: learn to work with the rewriter, as opposed to against it. Here are a few important tips:

- Decide upon a rewrite strategy: classify the terms that arise in your work according to their simplicity and rewrite "complicated" terms to "simple" ones. That is, put complicated terms on the left-hand side of your rules and simpler ones on the right-hand side.

- Invent appropriate equivalence relations: do not limit yourself to replacement of equals for equals. Be sure to prove the appropriate congruence rules.
- Normalize the left-hand sides of your rules: the left-hand sides are used for pattern matching, so if they contain terms that are simplified by other rules they will seldom be seen.
- Backchain to simpler terms: try to arrange your rules so that the hypotheses are simpler than the left-hand side of the conclusion.

B.2.1 Free Variables

The ACL2 rewriter is severely hampered when the hypothesis of a rewrite rule contains a variable that does not appear on the left-hand side of the conclusion. We discussed this on page 150 using the example theorem below.

```
(implies (and (divides p q)
              (not (equal p 1))
              (not (equal p q)))
         (not (primep q)))
```

The rewrite rule generated from this theorem rewrites (`primep` *q*) to `nil`. But to apply the rule we must first find a suitable instance, *p*, of `p`, such that (`divides` *p q*) is non-`nil`. We must then establish the other hypotheses, by rewriting. As discussed on page 150, the free variable `p` triggers a search, through the rewriter's context, for an assumption of precisely the form indicated by the first hypothesis containing it. The system issues a "free variable" warning when such a rewrite rule is stored. The warning indicates which hypothesis will determine the choice.

The hypotheses containing free variables should be listed first, since the application of the rule will fail unless appropriate choices are found. For example, the following rule is poorly stated.

```
(implies (and (graphp g)
              (nodep a)
              (nodep b)
              (path-from-to p a b g))
         (path-from-to (find-path a b g) a b g))
```

The rule is tried when a term of the form (`path-from-to` (`find-path` *a b g*) *a b g*) is rewritten. Note the free variable `p`. Because of the way the rule is phrased, the system first rewrites (`graphp` *g*), (`nodep` *a*), and (`nodep` *b*), before it tries to find a suitable instance of `p`. It would be better if the hypothesis containing `p` were listed first. You may use the `:corollary` option in `rule-classes` to rearrange a formula for purposes of rule generation.

When more than one hypothesis mentions the free variable, as in the `primep` example, you must choose which you want to list first. We generally choose the most unusual hypothesis or the one linking the free variable to the most other variables, as illustrated above. It is usually counterproductive to let a common hypothesis, *e.g.*, `(integerp i)`, determine the choice of a free variable because there are usually many such i in the context, but the system tries just one.

See also the `:restrict` hint described in hints.

Free variable considerations apply to linear and forward-chaining rules as well.

B.2.2 Conditional vs. Unconditional Rules

It is generally preferable to prove rewrite rules that have as few hypotheses as possible, in order to save the rewriter the trouble of proving those hypotheses. Consider the following conditional rule.

```
(defthm true-listp-append
  (implies (true-listp y)
           (true-listp (append x y))))
```

A preferable rule is the following.

```
(defthm true-listp-append
  (equal (true-listp (append x y))
         (true-listp y)))
```

It is easier to "debug" a proof involving unconditional rules because the theorem prover's output tells you more. For example, consider two databases, one containing the conditional rule and the other containing the unconditional rule. Consider attempting to prove `(true-listp (append a (mogrify b)))` with each database, but assume that the system cannot prove `(true-listp (mogrify b))` by rewriting alone. With the conditional database, `true-listp-append` is tried but the hypothesis cannot be relieved. The attempted use of the rule is not reported. The goal term is not rewritten and the system's other proof techniques are tried. But in the unconditional database, `true-listp-append` is tried and rewrites the goal to `(true-listp (mogrify b))`. The system then tries to prove that. The database may not lead the system to that proof, but at least the failed proof attempt identifies the key lemma for you.

B.2.3 Debugging Your Rules

When you have proved all the rules (you think) you need for a proof, but ACL2 fails to find the proof, it is often difficult to figure out what went wrong.

See break-rewrite and its subtopics to learn how to monitor the rewriter. A related tool is accumulated-persistence, which provides statistics that can help locate rules that are slowing down the rewriter.

A different approach to debugging a proof is to use an interactive proof utility. See verify and proof-checker. This tool allows you to construct a proof manually, *e.g.*, by applying specified rewrite rules to specified subterms. With this tool it often becomes obvious why an automatic proof is failing, *e.g.*, the next rule in your imagined proof does not match or some hypothesis is not provable. See Section 9.4.

B.2.4 Combinatoric Explosions

Sometimes the rewriter "blows up." The first symptom is apparently endless garbage collection or stack overflow. Follow the advice for finding infinite loops, page 197. A second symptom is that the theorem prover splits the goal conjecture into a huge number of cases. Often the problem can be traced back to the system's heuristics for handling "non-recursive" functions, i.e., functions which are not recursively defined. Such functions are generally just opened up. Consider disabling these symbols during the proof in question. Often, the proof will then quickly fail, but the formulas produced by the simplifier will contain function compositions that suggest appropriate rewrite rules.

B.2.5 Case Splitting and Forcing

There are times when you believe that one or more hypotheses of a certain rewrite rule should always be true in a given context, or if not, then the goal at hand should be provable in the case where that hypothesis is false. Corresponding capabilities are given by force and case-split. Related topics include disable-forcing, enable-forcing, failed-forcing, and immediate-force-modep.

B.2.6 Avoiding the Rewriter

The function hide, which is logically the identity function, prevents the rewriter from looking inside its argument. Printing of such terms can be controlled using eviscerate-hide-terms.

See also the documentation under hints for other methods for avoiding rewriting, notably :in-theory and :hands-off.

B.2.7 Executable Counterparts

When a function, f, is applied to constants, as in (`f 1 'x`), its logical value is computed by running the *executable counterpart* of f. If you do not want this to happen, disable (`:executable-counterpart` f). See `executable-counterpart`.

Disabling the executable counterpart of f is different from disabling f. The latter prevents the definitional axiom of f from being used to expand applications of f symbolically. Even when f is disabled, its values on constants will be computed by its executable counterpart unless that, too, is disabled.

This means that to disable a function, f, of no arguments you *must* disable both f and its executable counterpart, since all the arguments are constant in every application. The executable counterpart may be designated by a layer of parentheses, so to disable both f and its executable counterpart you may write the theory expression (`disable` f (f)).

B.2.8 Theories

ACL2 allows you to control the rewriter by specifying `theories`, which represent sets of rules. These are defined using *theory expressions*, which can be used globally, to set the current theory (see `in-theory` and `deftheory`) as well as locally, using `:in-theory` `hints`. Theory expressions can be built in a number of ways, including the following.

- (`current-theory :here`) returns the set of currently-enabled rules, or replace `:here` by any event name (see `logical-name`) to get the set of rules enabled immediately after that event was executed.
- (`union-theories theory1 theory2`) returns the rules that belong to at least one of the indicated theories.
- (`set-difference-theories theory1 theory2`) returns the rules in `theory1` that are not in `theory2`.

See also `disable` and `enable` for short-cuts.

B.2.9 Permutative Rules

Consider the following built-in rewrite rule.

```
ACL2 !>:pe commutativity-of-+
      -489  (DEFAXIOM COMMUTATIVITY-OF-+
                      (EQUAL (+ X Y) (+ Y X)))
ACL2 !>
```

This rule might appear to be one that would put you into an infinite loop, continually swapping the arguments of +. However, ACL2 has heuristics that tend to prevent such problems by doing such rewrites only when the resulting term is appropriately "smaller"; see loop-stopper and term-order. The user can affect this notion of "smaller"; see invisible-fns-alist.

B.2.10 Infinite Loops

The preceding section describes how ACL2 avoids potential infinite loops in the rewriter. However, it is very easy for the user to create collections of rewrite rules that do cause infinite loops in the rewriter, which are often manifested by stack overflows and segmentation errors in Lisp. See page 197 or the documentation for break-rewrite for how we track down these loops. See also theory-invariant for a way to avoid simultaneous enabling of conflicting rules.

Sometimes loops can be avoided if the rewriter is told to take into account the syntax of the actual term being rewritten, not just the pattern from the left-hand side of its conclusion. Syntaxp may be used for this purpose. For example, consider the following rewrite rule, which may be important if yet another rule can simplify expressions of the form (g (norm x)).

```
(defthm g-norm
  (implies (syntaxp (not (and (consp x)
                               (equal (car x) 'norm))))
           (equal (g x) (g (norm x)))))
```

Logically speaking, this theorem is equivalent to (equal (g x) (g (norm x))) because syntaxp returns t. Without the syntaxp hypothesis, the rule would loop, replacing (g x) by (g (norm x)) and then by (g (norm (norm x))), and so on. The syntaxp hypothesis prevents this by restricting the application of the rule to those x that are not of the form (norm ...).

B.3 Rule Classes

By default, ACL2 creates rewrite rules from defthm events; see rewrite. Rewrite rules are appropriate in the great majority of cases. But there are occasions where it is preferable to create other kinds of rules. Our goal here is to summarize some ideas for when various types of rules may be useful. Full details may be found in the individual documentation topics under rule-classes.

After rewrite rules, type-prescription rules are perhaps the most common; see type-prescription. These rules are useful when the theorem

prescribes an ACL2 type for a given term; see Section 8.3.3 and documentation for type-spec and compound-recognizer. For example, the following rule is built into ACL2.

```
(defthm consp-assoc
  (implies (alistp l)
           (or (consp (assoc name l))
               (equal (assoc name l) nil)))
  :rule-classes :type-prescription)
```

This rule is probably preferable to any rewrite rule one might try to create. Consider for example the following rewrite rule.

```
(defthm consp-assoc-rewrite
  (implies (and (alistp l)
                (not (equal (assoc name l) nil)))
           (consp (assoc name l))))
```

Unlike the above type-prescription rule, this rule will be considered every time that the rewriter encounters a term whose top function symbol is consp, which could slow down the prover.

Another common class of rules is the class of forward-chaining rules; see forward-chaining. These rules are particularly useful for propagating type-like information even when the "types" are not primitive ACL2 types. For example, the following rule is built into ACL2.

```
(defthm alistp-forward-to-true-listp
  (implies (alistp x)
           (true-listp x))
  :rule-classes :forward-chaining)
```

Whenever an instance of the term `(alistp x)` is known in a top-level context, this rule will add the corresponding instance of the term `(true-listp x)` to the context.

If a lemma does not suggest any rule classes, then it can be stored using `:rule-classes nil` in anticipation of a possible subsequent `:use` hint (see hints).

B.4 The ACL2 State and IO

ACL2 allows the user to perform traditionally non-applicative operations by using a single-threaded notion of state. When the variable `state` is used as the formal parameter of a function, the variable is assumed to refer to ACL2's global state. Syntactic restrictions are enforced on the use of `state`. `State` is an example of an ACL2 single-threaded object. Users can define single-threaded objects (or "stobjs") with defstobj.

Among the fields of `state` is a *global table* that allows one to program with global variables. In particular, the macro `assign` writes a global value and the macro `@` (see `@`[2]) reads a global value.

See also `pprogn` and `er-progn` for handy ways to perform sequential operations on `state`.

The ACL2 state also has fields that are used to implement IO. The most convenient functions provided for writing output are `fms`, `fmt`, and `fmt1`, which support a number of *tilde directives*. For example, the tilde directive `~xc` says to pretty-print the value associated with character `#\c`. Notice that in the following example, `fms` returns a new state.

```
ACL2 !>(fms "Argument bound to character 0: ~x0~%~
             Argument bound to character a: ~xa~%"
            (list (cons #\0 17)
                  (cons #\a (make-list 30)))
            (standard-co state) ; standard char out
            state ; ACL2 state object
            nil   ; evisceration tuple
            )
Argument bound to character 0: 17
Argument bound to character a:
(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
     NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
     NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
<state>
```

See `io` for a discussion of input utilities and other output utilities.

There is a way to do output without using state. Semantically, this output is a no-op; characters just magically appear in a *comment window*. In the following example we use the function `prog2$`, which ignores its first argument and returns its second argument. The macro `cw` prints like `fms`, but without state and with positional bindings for the tilde directives.

```
ACL2 !>(prog2$
        (cw "Argument bound to character 0: ~x0~%~
             Argument bound to character 1: ~x1~%"
            17 23)
        (+ 3 4))
Argument bound to character 0: 17
Argument bound to character 1: 23
7
```

A much more complex but general mechanism for bypassing `state` is described in the documentation for `wormhole`.

[2] In some versions of the hypertext documentation `@` is listed under the name `atsign` because of the markup language used.

B.5 Efficient Execution

There are several things the ACL2 user can do to speed up execution. The most important is `compilation`, but there are other techniques described below as well.

B.5.1 Compilation

You can arrange for ACL2 to compile each function as you define it by using `set-compile-fns`. This approach works well with underlying Lisps that have fast compilers, such as Allegro CL. However, with slower compilers such as GCL, it may be preferable to use `:comp` to compile all the functions of interest at once.

Another approach to `compilation` may be to certify `books`, which compiles them by default. See `certify-book`. Starting with Version 2.5 you can insure that you are loading compiled functions as follows.

```
(include-book "my-book" :load-compiled-file :comp)
```

The compilation done for books only affects the efficiency of functions whose guards have been verified. See Section B.5.3.

B.5.2 Declarations

Some underlying Lisps, in particular GCL, can benefit greatly from type declarations. See `declare` and `type-spec`. See also the case study by Greve, Wilding, and Hardin in the companion volume [22].

In many Lisps, *e.g.*, GCL, a common source of inefficiency is undeclared arithmetic. Most languages represent integers as machine words (on which arithmetic operations are implemented with the corresponding machine instructions) and use syntactic typing to distinguish words representing integers from words representing pointers or other data. By default, Lisp uses pointers to represent all data. Most Lisp implementations allocate memory in which to store arbitrarily large integers and pass around pointers to these numbers as "integers." Arithmetic therefore involves memory references and runtime type checking and can be quite slow. This can be avoided by declaring and proving that expressions produce values of limited size. To declare that the variable `x` holds an integer between -32768 and 32767 (inclusive), `declare` `x` to be of `type (integer -32768 32767)` or, equivalently, `(signed-byte 16)`. See `type-spec` for other such type specifiers. See also `the`.

B.5.3 Verifying Guards

As noted on page 51, Common Lisp functions are partial: many are defined only on subsets of the universe of Lisp objects. For example, the value of `(cdr 7)` is undefined in Common Lisp, but is `nil` in ACL2. To insure that ACL2 functions evaluate according to the axioms, a special evaluator is used—one that reflects ACL2's completion axioms.

ACL2's guards provide a means of declaring the intended domain of a function. Furthermore, `verify-guards` allows you to try to prove that the guards are "right," in the sense that the guard on a function's input implies the guards of its body. Since the guards on ACL2 primitives are those of Common Lisp, guard verification insures that Common Lisp will evaluate the function in accordance with the axioms, provided the input is known to satisfy the guard. See `:set-verify-guards-eagerness` to determine when guard verification is tried.

Every time a function, f, is defined in ACL2, a function of the same name with the same body is defined in the underlying Common Lisp. We call this the "raw Lisp" version of f. The raw Lisp version of f is used to evaluate a call of f only if the guard of f has been verified (see `verify-guards`) and the arguments to that call satisfy the guard. The "special evaluator" is used otherwise. The "special evaluator" is implemented by defining f in another package. For historic reasons, we call this symbol "`*1*`f" (pronounced "star one star f"). The body of `*1*`f is a version of the body of f, in which guards are checked at runtime and the completion values are used as necessary. Thus, for every defined function there are two Common Lisp definitions of the function, the raw Lisp version and the `*1*` version.

When a function is compiled with `:comp`, both the raw Lisp version and the `*1*` version are compiled. But when a function is compiled by `certify-book`, or when a compiled book is loaded by `include-book`, only the raw Lisp version is compiled. If execution efficiency is important and your functions are in books but have not had their guards verified, use `:comp t` after including the books.

See `guards-and-evaluation` for a thorough discussion of the relationship between guards and execution.

B.5.4 Efficient Code

The raw Lisp version of a function can be made more efficient by the appropriate choice of Lisp primitives to do certain operations. Note that these issues arise only in definitions whose guards are (to be) verified. If guards are not verified, the raw Lisp definitions are never executed.

ACL2 provides most of the Common Lisp primitives for equality. Although one can always test equality in ACL2 using `equal`, there are more

efficient equality tests to use in definitions whose guards are (to be) verified.

- (eq x y): x or y is a symbol.
- (int= x y): x and y are both integers
- (eql x y): x or y is a symbol, number, or character.
- (= x y): x and y are numbers
- (equal x y): always permissible

Many list-manipulation functions have counterparts based on eq, eql, and equal. For example, see assoc-eq, assoc, and assoc-equal, respectively.

There are a number of tests to use against nil. A good one to use is endp. For example, consider the following built-in definition of the function member-equal.

```
(defun member-equal (x lst)
  (declare (xargs :guard (true-listp lst)))
  (cond ((endp lst) nil)
        ((equal x (car lst)) lst)
        (t (member-equal x (cdr lst)))))
```

Although atom could be used instead of endp above, endp is a bit more efficient, and will be acceptable for guard verification if its argument is a true list (see true-listp).

B.5.5 Arrays and Single-Threaded Objects

The basic data structure in ACL2 is the cons pair, which in turn provides useful data structures such as association lists. However, consing is relatively slow, and the linear-time access provided by association lists may not be desirable.

ACL2 provides arrays for constant-time access. Still, each array has a corresponding association list, so arrays may not be appropriate for applications with intensive updating. ACL2 has single-threaded objects that can give both constant-time access and constant-time update; see stobj and defstobj.

B.5.6 Constants

Execution efficiency can benefit from the use of defconst rather than defun in order to define constants. Consider for example:

```
(defun large-n () (expt 2 64))
```

The above approach causes `(expt 2 64)` to be computed every time the term `(large-n)` is evaluated. The following approach causes `(expt 2 64)` to be computed only once.

```
(defconst *large-n* (expt 2 64))
```

Macros can also be used for efficiency. If you define

```
(defmacro large-n () (expt 2 64))
```

then every occurrence of `(large-n)` in subsequent definitions will be expanded to the actual value of `(expt 2 64)`. The price will be paid at compilation time rather than at runtime.

B.6 Other Topics

B.6.1 Packages

It can be handy to work in a package other than the ACL2 package, in order to avoid name clashes. See defpkg, current-package, in-package, and acl2-user.

B.6.2 Documentation Strings

We encourage users to document their ACL2 events. See doc-string, defdoc, and deflabel.

B.6.3 Quantification

ACL2 provides little support for reasoning about quantifiers ("there exists" and "for all"), but it does support quantifiers in definitions using defun-sk. The basic mechanism underlying the defun-sk macro is the defchoose event, which produces functions that pick "witnessing" values nonconstructively.

B.6.4 Books

A number of books are supplied with the ACL2 distribution. ACL2 users are encouraged to save effort by taking advantage of the definitions and proved theorems in these books, by using include-book. File README in the books/ subdirectory of the ACL2 distribution gives summaries of these books as well as instructions for certifying them.

B.6.5 Tables

ACL2 provides a notion of `table` through which the user can store arbitrary information. Built-in tables are used to implement most of the modes and switches presented in Section B.1.6.

B.7 Troubleshooting Guide

- *How can I quit an ACL2 session?* This topic is covered in detail in Section A.2.3 of the other appendix.
- *I have given a* `:use` *hint that seems to disappear and hence to be of no use. What's wrong?* If a lemma has been stored as a rewrite rule, and that rule is not disabled, then it is likely that the rewriter will in essence replace that lemma with `t` when you attempt to use it. A solution is to add to such `hints` an `:in-theory` hint that disables the lemma (see `disable`). ACL2 will print a warning in the event summary when an enabled rewrite rule is specified in a `:use` hint.
- *I'm having trouble verifying guards. What should I do?* Unless you are pushing for the most efficiency you can gain, consider omitting the guards or using `(set-verify-guards-eagerness 0)`.
- *ACL2 does not seem to know very much. It is unable to do very basic reasoning, for example, leaving terms like* `(+ -1 1 x)` *in goals that it cannot simplify further.* Consider including a book from the distribution (see Section B.6.4). For example, the book `books/arithmetic/top-with-meta` includes a useful set of proved rules about arithmetic.
- *I'm having trouble getting the prover to use particular facts.* See `hints`. If you want hints to be computed dynamically depending on the shape of the goal, see `computed-hints` and the collection of topics starting with `using-computed-hints`.
- *I want a rule enabled for* `Subgoal *1/2.1'` *but not for unrelated goals.* See `hints` and `goal-spec`.
- *Sometimes when the prover sees a goal to prove by induction, it decides to start over and use induction to prove the original goal. How can I stop that behavior?* See `otf-flg`.
- *How can I tell whether a rule is disabled?* See `disabledp`.
- *I'd like to see how a form is macroexpanded.* See `trans1` and `trans`.
- *Some languages have nice features for pattern matching in definitions. Does ACL2 provide anything like that?* See `case-match`.

- *Does ACL2 provide any way of raising an exception?* See `illegal` and `hard-error`.
- *Sometimes I have trouble admitting functions that recur on an argument until it hits zero.* See `zero-test-idioms`.
- *I have a theorem that should be provable by applying a few rewrite rules and a large amount of case splitting. The case splitting seems to swamp ACL2. What can help with that?* See `bdd`.
- *What should I do if I submit a command and the system is showing no response?* If you submitted a keyword command, did you give enough arguments? Otherwise, did you supply enough closing parentheses?
- *How can I control which rules are used for a proof attempt?* Theories are used for this purpose. See Section B.2.8.
- *How do I avoid infinite loops in the rewriter?* Attempt to construct rewrite rules that do not loop, but if that fails, see Section B.2.10.
- *I get segmentation errors or stack overflows on some proofs.* You probably have a loop in your rewrite rules. See Section B.2.10.

Bibliography

1. W. R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.
2. W. R. Bevier, W. A. Hunt, Jr., J S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.
3. R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
4. R. S. Boyer, M. Kaufmann, and J S. Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 5(2):27–62, 1995.
5. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
6. R. S. Boyer and J S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176. MIT Press, 1996.
7. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, second edition, 1997.
8. R. S. Boyer and J S. Moore. Single-threaded objects in ACL2, 1999. See URL `http://www.cs.utexas.edu/users/moore/publications/-acl2-papers.html#Foundations`.
9. R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.
10. B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, 1996.
11. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
12. R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 1992.
13. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques.* Boca Raton, FL, April 1995.

14. K. Devlin. *The Joy of Sets: Fundamentals of Contemporary Set Theory.* Springer-Verlag, 2nd edition, 1992.

15. R. L. Goodstein. *Recursive Number Theory.* North-Holland Publishing Company, Amsterdam, 1964.

16. M. Gordon and T. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

17. P. R. Halmos. *Naive Set Theory.* Van Nostrand, 1960.

18. J. P. Hayes. *Computer Architecture and Organization.* McGraw-Hill Publishing Company, 1988.

19. W. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

20. W. Hunt, Jr. and B. Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, 1992.

21. W. Hunt, Jr. and B. Brock. The `DUAL-EVAL` hardware description language and its use in the formal specification and verification of the FM9001 microprocessor. *Formal Methods in Systems Design*, 11:71–105, 1997.

22. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, 2000.

23. M. Kaufmann and J S. Moore. Design goals of ACL2. Technical Report 101, Computational Logic, Inc., 1994. See URL `http://www.cs.utexas.edu/-users/moore/publications/acl2-papers.html#Overviews`.

24. M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23:203–213, April 1997.

25. M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997. See URL `http://www.cs.utexas.edu/users/moore/-publications/acl2-papers.html#Foundations`.

26. M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 2000. To appear.

27. M. Kaufmann and P. Pecchiari. Interaction with the Boyer-Moore theorem prover: A tutorial study using the arithmetic-geometric mean theorem. *Journal of Automated Reasoning*, 16(1–2):181–222, 1996.

28. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine (part I). *CACM*, 3(4):184–195, 1960.

29. W. McCune. Otter 3.0 reference manual and guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994. See URL `http://-www.mcs.anl.gov/AR/otter/`.

30. D. Moon. MacLISP reference manual, revision 0. Technical report, MIT Project MAC, Camridge, MA, April 1974.

31. D. Moon, R. Stallman, and D. Weinreb. LISP machine manual, fifth edition. Technical report, MIT Artificial Intelligence Laboratory, Cambridge, MA, January 1983.

32. J S. Moore. Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning*, 12(1):33–45, 1994.

33. J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5$_K$86 floating-point division program. *IEEE Trans. Comp.*, 47(9):913–926, September 1998. See URL `http://www.cs.utexas.edu/users/moore/-publications/acl2-papers.html#Floating-Point-Arithmetic`.

34. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag, June 1992.

35. D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.

36. N. Shankar. *Metamathematics, Machines, and Godel's Proof.* Cambridge University Press, 1994.

37. J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Ma., 1967.

38. R. Stallman. *GNU Emacs Manual.* Free Software Foundation, 1987.

39. G. L. Steele, Jr. *Common Lisp The Language.* Digital Press, Burlington, MA, 1984.

40. G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, Burlington, MA, 1990. See URL `http://www.cs.cmu.edu/afs/-cs.cmu.edu/project/ai-repository/ai/html/cltl/clm/clm.html`.

41. W. Teitelman. InterLISP reference manual, third revision. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1978.

42. G. J. Wirsching. *The Dynamical System Generated by the 3n+1 Function*, volume 1681 of *Lecture Notes in Mathematics.* Springer-Verlag, 1998.

Index

Underlined words are the names of links in the online documentation. From the ACL2 home page, `http://www.cs.utexas.edu/users/moore/acl2`, select the link to the User's Manual and then the link to the Index of all documented topics.