Reasoning About Programs

Panagiotis Manolios
Northeastern University

April 20, 2017
Version: 104

Copyright ©2017 by Panagiotis Manolios

All rights reserved. We hereby grant permission for this publication to be used for personal or classroom use. No part of this publication may be stored in a retrieval system or transmitted in any form or by any means other personal or classroom use without the prior written permission of the author. Please contact the author for details.

Contents

1 Introduction **1**

I Programming 1

2 A Simple Functional Programming Language **3**

2.1 Constants 4
2.2 Booleans 5
2.3 Numbers 9
2.4 Other Atoms 11
2.5 Lists 11
2.6 Contract Violations 13
2.7 Termination 14
2.8 Beginner Mode 14
2.9 Contracts 17
2.10 Helpful Functions 17
2.11 Quote 19
2.12 Let 19
2.13 Testing 20
2.14 Data Definitions 22
2.15 Design Recipe 23
2.16 Program Mode 26
2.17 Dealing with Definition Failures 27

II Propositional Logic 31

3 Propositional Logic **33**

3.1 P = NP 37
3.2 The Power of Xor 37
3.3 Useful Equalities 38
3.4 Proof Techniques 41
3.5 Normal Forms and Complete Boolean Bases 42
3.6 Decision Procedures 43
3.7 Propositional Logic in ACL2s 44
3.8 Word Problems 45
3.9 The Declarative Approach to Design 46

III Equational Reasoning 49

4 Equational Reasoning 51
4.1 Testing Conjectures 60
4.2 Equational Reasoning with Complex Propositional Structure 62
4.3 The difference between theorems and context 64
4.4 How to prove theorems 65
4.5 Arithmetic 71

IV Definitions and Termination 75

5 Definitions and Termination 77
5.1 The Definitional Principle 77
5.2 Admissibility of common recursion schemes 82
5.3 Exercises 84
5.4 Complexity Analysis 88
5.5 Undecidability of the Halting Problem 88

V Induction 91

6 Induction 93
6.1 Induction Examples 96
6.2 Data-Function-Induction Trinity 99
6.3 The Importance of Termination 100
6.4 Generalization 101
6.5 Reasoning About Accumulator-Based Functions 101

VI Steering 107

7 Steering the ACL2 Sedan 109
7.1 Interacting with ACL2s 109
7.2 The Waterfall 109
7.3 Term Rewriting 110

VII ADTs 117

8 Abstract Data Types and Observational Equivalence 119
8.1 Abstract Data Types 119
8.2 Algebraic Data Types 121
8.3 Observational Equivalence 123
8.4 Queues 127
Introduction

1

Introduction

These lecture notes were developed for Logic and Computation, a freshman-level class taught at the College of Computer and Information Science of Northeastern University. Starting in Spring 2008, this is a class that all students in the college are required to take.

The goals of the Logic and Computation course are to provide an introduction to formal logic and its deep connections to computing. Logic is presented from a computational perspective using the ACL2 Sedan theorem proving system. The goal of the course is to introduce fundamental, foundational methods for modeling, designing, specifying and reasoning about computation. The topics covered include propositional logic, recursion, contracts, testing, induction, equational reasoning, termination analysis, term rewriting, and various proof techniques. We show how to use logic to formalize the syntax and semantics of the core ACL2s language, a simple LISP-based language with contracts. We then use the ACL2s language to formally reason about programs, to model systems at various levels of abstraction, to design and specify interfaces between systems and to reason about such composed systems. We also examine decision procedures for fragments of first-order logic and how such decision procedures can be used to analyze models of systems.

The students taking the Logic and Computation class have already taken a programming class in the previous semester, in Racket. The course starts by reviewing some basic programming concepts. The review is useful because at the freshman level students benefit from seeing multiple presentations of key concepts; this helps them to internalize these concepts. For example, in past semesters I have asked students to write very simple programs (such as a program to append two lists together) during the first week of classes and a surprisingly large number of students produce incorrect code.

During the programming review, we introduce the ACL2s language. This is the language we use throughout the semester and it is similar to Racket. The syntax and semantics of the core ACL2s language are presented in a mathematical way. We provide enough information so that students can determine what sequence of glyphs form a well-formed expression and how to formally evaluate well-formed expressions potentially containing user-defined functions with constants as arguments (this is always in a first-order setting). This is a pretty big jump in rigor for students and is advanced material for freshmen students, but they already have great intuitions about evaluation from their previous programming class. This intuition helps them understand the rigorous presentation of the syntax and semantics, which in turns helps strengthen their programming abilities.

The lecture notes are sparse. It would be great to add more exercises, but I have not done that yet. Over the course of many years, we have amassed a large collection of homework problems, so students see lots of exercises, and working through these exercises is a great way for them to absorb the material, but the exercises are not in the notes. You can think of the lecture notes as condensed notes for the course that are appropriate for someone who knows the material as a study guide. The notes can also be used as a starting point by

students, who should mark them up with clarifications as needed when they attend lectures. I advise students to read the lecture notes before class. This way, during class they can focus on the lecture instead of taking notes and they are better prepared to ask for clarifications.

When I started teaching the class, I used the ACL2 book, *Computer-Aided Reasoning, An Approach* by Kaufmann, Manolios and Moore. However, over the years I became convinced that using an untyped first-order logic was not the optimal way of introducing logic and computation to students because they come in with a typed view of the world. That's not to say they have seen type theory; they have not. But, they are surprised when a programming language allows them to subtract a string from a rational number. Therefore, with the help of my Ph.D. student Harsh Chamarthi, I have focused on adding type-like capabilities to ACL2s. Most notably, we added a new data definition framework to ACL2s that supports enumeration, union, product, record, map, (mutually) recursive and custom types, as well as limited forms of parametric polymorphism. We also introduced the `defunc` macro, which allows us to formally specify input and output contracts for functions. These contracts are very general, *e.g.*, we can specify that `/` is given two rationals as input, and that the second rational is not 0, we can specify that `zip` is given two lists of the same length as input and returns a list of the same length as output and so on. Contracts are also checked statically, so ACL2s will not accept a function definition unless it can prove that the function satisfies its contracts and that for every legal input and every possible computation, it is not possible during the evaluation of the function being defined to be in a state where some other function is poised to be evaluated on a value that violates its input contract. I have found that a significant fraction of erroneous programs written by students have contract violations in them, and one of the key things I emphasize is that when writing code, one needs to think carefully about the contracts of the functions used and why the arguments to every function call satisfy the function's contract. Contracts are the first step towards learning how to specify interfaces between systems. With the move to contracts, the ACL2 book became less and less appropriate, which led me to write these notes.

I have distributed these notes to the students in Logic and Computation for several years and they have found lots of typos and have made many suggestions for improvement. Thanks and keep the comments coming!

Part I

Programming

2

A Simple Functional Programming Language

In this chapter we introduce a simple functional programming language that forms the core of ACL2s. The language is a dialect of the Lisp programming language and is based on ACL2. In order to *reason* about programs, we first have to understand the *syntax* and *semantics* of the language we are using. The syntax of the language tells us what sequence of glyphs constitute well-formed expressions. The semantics of the language tells us what well-formed expressions (just *expressions* from now on) mean, *i.e.*, how to evaluate them. Our focus is on reasoning about programs, so the programming language we are going to use is designed to be simple, minimal, expressive, and easy to reason about.

What makes ACL2s particularly easy to reason about is the fact that it is a *functional* programming language. What this means is that every built-in function and in fact any ACL2s function a user can define satisfies the rule of Leibniz:

$$\text{If } x_1 = y_1 \text{ and } x_2 = y_2 \text{ and } \cdots \text{ and } x_n = y_n, \text{ then } (\texttt{f}\ x_1\ x_2 \cdots x_n) = (\texttt{f}\ y_1\ y_2 \cdots y_n)$$

Almost no other language satisfies this very strict condition, *e.g.*, in Java you can define a function `foo` of one argument that on input 0 can return 0, or 1, or any integer because it returns the number of times it was called. This is true for Scheme, LISP, C, C++, C#, OCaml, etc. The rule of Leibniz, as we will see later, is what allows us to reason about ACL2s in a way that mimics algebraic reasoning.

You interact with ACL2s via a Read-Eval-Print-Loop (REPL). For example, ACL2s presents you with a prompt indicating that it is ready to accept input.

```
ACL2S BB !>
```

You can now type in an expression, say

```
ACL2S BB !>12
```

ACL2s reads and evaluates the expression and prints the result

```
12
```

It then presents the prompt again, indicating that it is ready for another REPL interaction

```
ACL2S BB !>
```

We recommend that as you read these notes, you also have ACL2s installed and follow along in the "Bare Bones" mode. The "`BB`" in the prompt indicates that ACL2s is in the "Bare Bones" mode.

The ACL2s programming language allows us to design programs that manipulate objects from the ACL2s *universe*. The set of all objects in the universe will be denoted by All. All includes:

- Rationals: For example, $11, -7, 3/2, -14/15$.

- Symbols: For example, `x`, `var`, `lst`, `t`, `nil`.
- Booleans: There are two Booleans, `t`, denoting *true* and `nil`, denoting *false*.
- Conses: For example, `(1)`, `(1 2 3)`, `(cons 1 ())`, `(1 (1 2) 3)`.

The Rationals, Symbols, and Conses are disjoint. The Booleans `nil` and `t` are Symbols. Conses are Lists, but there is exactly one list, the empty list, which is not a cons. We will use `()` to denote the empty list, but this is really an abbreviation for the symbol `nil`.

The ACL2s language includes a basic core of built-in functions, which we will use to define new functions.

It turns out that expressions are just a subset of the ACL2s universe. Every expression is an object in the ACL2s universe, but not conversely. As we introduce the syntax of ACL2s, we will both identify what constitutes an expression and what these expressions mean as follows. If *expr* is an expression, then

$$[\![expr]\!]$$

will denote the semantics of *expr*, or what *expr* evaluates to when submitted to ACL2s at the REPL.

We will introduce the ACL2s programming language by first introducing the syntax and semantics of constants, then Booleans, then numbers, and then conses and lists.

2.1 Constants

All constants are expressions. The ACL2s Boolean constant denoting *true* is the symbol `t` and the constant denoting *false* is the symbol `nil`. These two constants are different and they evaluate to themselves.

$$[\![\texttt{t}]\!] = \texttt{t}$$
$$[\![\texttt{nil}]\!] = \texttt{nil}$$
$$\texttt{nil} \neq \texttt{t}$$

The numeric constants include the natural numbers:

$$0, 1, 2, \ldots$$

and the negative integers:

$$-1, -2, -3, \ldots$$

All integers evaluate to themselves, *e.g.*,

$$[\![3]\!] = 3$$
$$[\![-12]\!] = -12$$

The numeric constants also include the rationals:

$$1/2,\ -1/2,\ 1/3,\ -1/3,\ 3/2,\ -3/2,\ 2/3,\ -2/3, \ldots$$

We will describe the evaluation of rationals in Section 2.3.

2.2 Booleans

There are two built-in functions, `if` and `equal`.

When we introduce functions, we specify their *signature*. The signature of `if` is:

$$\texttt{if} : \textsf{Boolean} \times \textsf{All} \times \textsf{All} \rightarrow \textsf{All}$$

The signature of `if` tells us that `if` takes three arguments, where the first argument is a Boolean and the rest of the arguments are anything at all. It returns anything. So, the signature specifies not only the *arity* of the function (how many arguments it takes) but also its input and output contracts.

Examples of `if` expressions include the following.

```
(if t nil t)

(if nil 3 4)
```

All function applications in ACL2s are written in prefix form as shown above. For example, instead of $3 + 4$, in ACL2s we write `(+ 3 4)`. The `if` expressions above are elements of the ACL2s universe, *e.g.*, the first `if` expression is a list consisting of the symbols `if`, `t`, `nil`, and `t`, in that order.

Not every list starting with the symbol `if` is an expression, *e.g.*, the following are *not* expressions.

```
(if t nil)

(if 1 3 4)
```

The first list above does not satisfy the signature of `if`, which tells us that the function has an arity of three. The second list also does not satisfy the signature of `if`, which tells us that the input contract requires that the first argument is a Boolean. In general, a list is an expression if it satisfies the signature of a built-in or previously defined function.

The semantics of (`if` *test then else*) is as follows.

$$[\![\texttt{(if}\ \textit{test}\ \textit{then}\ \textit{else}\texttt{)}]\!] = [\![\textit{then}]\!],\ \text{when}\ [\![\textit{test}]\!] = \texttt{t}$$

$$[\![\texttt{(if}\ \textit{test}\ \textit{then}\ \textit{else}\texttt{)}]\!] = [\![\textit{else}]\!],\ \text{when}\ [\![\textit{test}]\!] = \texttt{nil}$$

For all ACL2s functions we consider, we specify the semantics of the functions only in the case that the signature of the function is satisfied, *i.e.*, only for expressions. If the input contract is violated, then we say that a contract violation has occurred and the function does not evaluate to anything; hence, it does not return a result. For example, as we have seen `(if 1 3 4)` is not an expression. If you try to evaluate it you will get an error message indicating that a contract violation has occurred.

Our first function, `if`, is an important and special function. In contrast to every other function, `if` is evaluated in a *lazy* way by ACL2s. Here is how evaluation works. To evaluate

$$[\![\texttt{(if}\ \textit{test}\ \textit{then}\ \textit{else}\texttt{)}]\!]$$

ACL2s performs the following steps.

1. First, ACL2s evaluates *test*, *i.e.*, it computes $[\![\textit{test}]\!]$.

2. If $[\![test]\!] =$ `t`, then ACL2s returns $[\![then]\!]$.

3. Otherwise, it returns $[\![else]\!]$.

Notice that *test* is always evaluated, but only one of *then* or *else* is evaluated. In contrast, for all other functions we define, ACL2s will evaluate them in a *strict* way by evaluating all of the arguments to the function and then applying the function to the evaluated results.

Examples of the evaluation of `if` expressions include the following:

$$[\![\texttt{(if t nil t)}]\!] = \texttt{nil}$$

$$[\![\texttt{(if nil 3 4)}]\!] = 4$$

Here is a more complex `if` expression.

```
(if (if t nil t) 1 2)
```

This may be confusing because it seems that the test of the `if` is a List, not a Boolean. However, notice that to evaluate an `if`, we evaluate the test first, *i.e.*:

$$[\![\texttt{(if t nil t)}]\!] = \texttt{nil}$$

Therefore,

$$[\![\texttt{(if (if t nil t) 1 2)}]\!] = [\![2]\!] = 2$$

The next function we consider is `equal`.

$$\texttt{equal} : \textsf{All} \times \textsf{All} \rightarrow \textsf{Boolean}$$

$[\![\texttt{(equal x y)}]\!]$ is `t` if $[\![x]\!] = [\![y]\!]$ and `nil` otherwise.

Notice that `equal` always evaluates to `t` or `nil`.

Here are some examples.

$$[\![\texttt{(equal 3 nil)}]\!] = \texttt{nil}$$

$$[\![\texttt{(equal 0 0)}]\!] = \texttt{t}$$

$$[\![\texttt{(equal (if t nil t) nil)}]\!] = \texttt{t}$$

That's it for the built-in Booleans constants and functions.

Let us now define some utility functions.

We start with `booleanp`, whose signature is as follows.

$$\textsf{All} \rightarrow \textsf{Boolean}$$

The name is the concatenation of the word "`boolean`" with the symbol "`p`." The "`p`" indicates that the function is a *predicate*, a function that returns `t` or `nil`. We will use this naming convention in ACL2s (most of the time). Other Lisp dialects indicate predicates using other symbols, *e.g.*, Scheme uses "`?`" (pronounced "huh") instead of "`p`."

Here is how we define functions with contracts in ACL2s. The `check=` forms allow us to write down what we expect our function will return on various legal inputs.

```
(defunc booleanp (x)
  :input-contract ...
```

```
  :output-contract ...
  (if (equal x t)
      t
    (equal x nil)))
(check= (booleanp t) t)
(check= (booleanp nil) t)
(check= (booleanp 12) nil)
```

The contracts were deliberately elided. We will add them shortly, but first we discuss how to evaluate expressions involving `booleanp`.

How do we evaluate `(booleanp 3)`?

⟦`(booleanp 3)`⟧

= { Semantics of `booleanp` }

⟦`(if (equal 3 t) t (equal 3 nil))`⟧

= { Semantics of `equal`, ⟦`(equal 3 t)`⟧= `nil`, Semantics of `if` }

⟦`(equal 3 nil)`⟧

= { Semantics of `equal`, ⟦`(equal 3 nil)`⟧= `nil` }

`nil`

Above we have a sequence of expressions each of which is equivalent to the next expression in the sequence for the reason given in the hint enclosed in curly braces. For example the first equality holds because we expanded the definition of `booleanp`, replacing the formal parameter `x` with the actual argument 3.

The next thing is: what is the input contract for `booleanp`?

It is `t` because there are no constraints on the input to the function. All *recognizers* will have an input contract of `t`. A recognizer is a function that given any element of the ACL2s universe recognizes whether it belongs to a particular subset. In the case of `booleanp`, the subset being recognized is the set of Booleans {`t`, `nil`}.

What about the output contract? Since `booleanp` is a recognizer it returns a Boolean! We express this as follows:

```
(booleanp (booleanp x))
```

So, all together we have:

```
(defunc booleanp (x)
  :input-contract t
  :output-contract (booleanp (booleanp x))
  (if (equal x t)
      t
    (equal x nil)))
```

What does the contract mean? Well, let us consider the general case. Say that function f with parameters $x_1, \ldots, x_n$ has the input contract *ic* and the output contract *oc*, then what the contract means is that for any assignment of values from the ACL2s universe to the variables $x_1, \ldots, x_n$, the following formula is always true.

ic implies *oc*

Hence, the contract for `booleanp` means that for any element of the ACL2s universe, x,

$$\texttt{t} \text{ implies } \texttt{(booleanp (booleanp } x\texttt{))}$$

If we wanted to make the universal quantification and the implication explicit, we would write the following, where the domain of x is implicitly understood to be All.

$$\langle \forall x :: \texttt{t} \Rightarrow \texttt{(booleanp (booleanp } x\texttt{))} \rangle$$

Notice that by the relationship between $\Rightarrow$ (implication) and `if`, the above is equivalent to

$$\langle \forall x :: \texttt{(if t (booleanp (booleanp } x\texttt{)) t)} \rangle$$

By the semantics of `if`, we can further simplify this to

$$\langle \forall x :: \texttt{(booleanp (booleanp } x\texttt{))} \rangle$$

So, for any ACL2s element x, `booleanp` returns a `boolean`.
Let us continue with more basic definitions.

$$\texttt{and} : \text{Boolean} \times \text{Boolean} \to \text{Boolean}$$

```
(defunc and (a b)
  :input-contract (if (booleanp a) (booleanp b) nil)
  :output-contract (booleanp (and a b))
  (if a b nil))
```

$$\texttt{implies} : \text{Boolean} \times \text{Boolean} \to \text{Boolean}$$

```
(defunc implies (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (implies a b))
  (if a b t))
```

$$\texttt{or} : \text{Boolean} \times \text{Boolean} \to \text{Boolean}$$

```
(defunc or (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (or a b))
  (if a t b))
```

How do we evaluate the above? Simple:

$[\![$`(or t nil)`$]\!]$
$=$ { Definition of `or` }
$[\![$`(if t t nil)`$]\!]$
$=$ { Semantics of `if` }
If $[\![$`t`$]\!]$ = `nil` then $[\![$`nil`$]\!]$ else $[\![$`t`$]\!]$
$=$ { Constants evaluate to themselves }
If `t` = `nil` then `nil` else `t`
$=$ { `t` is not `nil` }
`t`

Exercise 2.1 *Define:* `not`, `iff`, `xor`, *and other Boolean functions.*

$$\texttt{not} : \textsf{Boolean} \rightarrow \textsf{Boolean}$$

```
(defunc not (a)
  :input-contract (booleanp a)
  :output-contract (booleanp (not a))
  (if a nil t))
```

$$\texttt{iff} : \textsf{Boolean} \times \textsf{Boolean} \rightarrow \textsf{Boolean}$$

```
(defunc iff (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (iff a b))
  (if a b (not b)))
```

$$\texttt{xor} : \textsf{Boolean} \times \textsf{Boolean} \rightarrow \textsf{Boolean}$$

```
(defunc xor (a b)
  :input-contract (and (booleanp a) (booleanp b))
  :output-contract (booleanp (xor a b))
  (if a (not b) b))
```

2.3 Numbers

We have the following built-in recognizers:

$$\texttt{integerp} : \textsf{All} \rightarrow \textsf{Boolean}$$

$$\texttt{rationalp} : \textsf{All} \rightarrow \textsf{Boolean}$$

Here is what they mean.
$[\![$`(integerp x)`$]\!]$ is `t` iff $[\![$x$]\!]$ is an integer.
$[\![$`(rationalp x)`$]\!]$ is `t` iff $[\![$x$]\!]$ is a rational.

Note that integers are rationals. This is just a statement of mathematical fact.

Notice also that ACL2s includes the *real* rationals and integers, not approximations or bounded numbers, as you might find in most other languages, including C and Java.

We also have the following functions.

$$\texttt{+} : \mathsf{Rational} \times \mathsf{Rational} \rightarrow \mathsf{Rational}$$

$$\texttt{*} : \mathsf{Rational} \times \mathsf{Rational} \rightarrow \mathsf{Rational}$$

$$\texttt{<} : \mathsf{Rational} \times \mathsf{Rational} \rightarrow \mathsf{Boolean}$$

$$\texttt{unary--} : \mathsf{Rational} \rightarrow \mathsf{Rational}$$

$$\texttt{unary-/} : \mathsf{Rational} \rightarrow \mathsf{Rational}$$

Wait, what about `(unary-/ 0)`? The contract really is:

$$\texttt{unary-/} : \mathsf{Rational} \setminus \{0\} \rightarrow \mathsf{Rational}$$

How do we express this kind of thing?

```
(defunc unary-/ (a)
  :input-contract (and (rationalp a) (not (equal a 0)))
  ...)
```

The semantics of the above functions should be clear (from elementary school). Here are some examples.

$$[\![\texttt{(+ 3/2 17/6)}]\!] = \texttt{13/3}$$

$$[\![\texttt{(* 3/2 17/6)}]\!] = \texttt{17/4}$$

$$[\![\texttt{(< 3/2 17/6)}]\!] = \texttt{t}$$

$$[\![\texttt{(unary-- -2/8)}]\!] = \texttt{1/4}$$

$$[\![\texttt{(unary-/ -2/8)}]\!] = \texttt{-4}$$

Exercise 2.2 *Define subtraction on rationals* `-` *and division on rationals* `/`*. Note that the second argument to* `/` *cannot be 0.*

Let's define some more functions, starting with a recognizer for positive integers.

$$\texttt{posp} : \mathsf{All} \rightarrow \mathsf{Boolean}$$

```
(defunc posp (a)
  :input-contract t
  :output-contract (booleanp (posp a))
  (if (integerp a)
      (< 0 a)
    nil))
```

What if we tried to define `posp` as follows?

```
(defunc posp (a)
  :input-contract t
```

```
:output-contract (booleanp (posp a))
(and (integerp a)
     (< 0 a)))
```

Well, notice that the contract for `<` is that we give it two rationals. How do we know that `a` is rational? What we would like to do is to test that `a` is an integer first, before testing that `(< 0 a)`, but the only way to do that is to use `if`. This is another reason why `if` is special. When checking the contracts of the `then` branch of an `if`, we can assume that the `test` is *true*; when checking the contracts of an `else` branch, we can assume that the `test` is *false*. No other ACL2s function gives us this capability. If we want to collect together assumptions in order to show that contracts are satisfied, we have to use `if`.

Exercise 2.3 *Define* `natp`*, a recognizer for natural numbers.*

We also have built-in

$$\texttt{numerator} : \textsf{Rational} \rightarrow \textsf{Integer}$$

$$\texttt{denominator} : \textsf{Rational} \rightarrow \textsf{Pos}$$

$[\![\texttt{(numerator a)}]\!]$ is the numerator of the number we get after simplifying $[\![\texttt{a}]\!]$.
$[\![\texttt{(denominator a)}]\!]$ is the denominator of the number we get after simplifying $[\![\texttt{a}]\!]$.
To simplify an integer `x`, we return `x`.
To simplify a number of the form `x/y`, where `x` is an integer and `y` a natural number, we divide both `x` and `y` by the $gcd(|\texttt{x}|, \texttt{y})$ to obtain `a/b`. If $\texttt{b} = 1$, we return `a`; otherwise we return `a/b`. Note that `b` (the denominator) is always positive.
Since rational numbers can be represented in many ways, ACL2s returns the simplest representation, *e.g.*,

$$[\![2/4]\!] = 1/2$$

$$[\![4/2]\!] = 2$$

$$[\![132/765]\!] = 44/255$$

2.4 Other Atoms

Symbols and numbers are *atoms*. The ACL2s universe includes other atoms, such as strings and characters. For example, `"hello"` is a string and `#\A` is a character. Strings and characters evaluate to themselves.

2.5 Lists

Lists allow us to create non-atomic data.
Our first built-in function is a recognizer for *conses*.

$$\texttt{consp} : \textsf{All} \rightarrow \textsf{Boolean}$$

Conses are non-empty lists and are comprised of a first element and the rest of the list. Here are the functions for accessing the first and rest of a cons.

$$\texttt{first} : \mathsf{Cons} \to \mathsf{All}$$

$$\texttt{rest} : \mathsf{Cons} \to \mathsf{All}$$

We now define `listp`, a recognizer for lists, as follows.

$$\texttt{listp} : \mathsf{All} \to \mathsf{Boolean}$$

```
(defunc listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
  (if (consp l)
      (listp (rest l))
    (equal l () )))
```

The last built-in function is:

$$\texttt{cons} : \mathsf{All} \times \mathsf{List} \to \mathsf{Cons}$$

The semantics of the built-in functions is given by the following rules. Notice that the second argument to `cons` can either be `()` or a cons.

$$[\![\texttt{(cons x ())}]\!] = \texttt{(}[\![\mathrm{x}]\!]\texttt{)}$$

$$[\![\texttt{(cons x y)}]\!] = \texttt{(}[\![\mathrm{x}]\!]\ \texttt{...)} \text{ where } [\![\mathrm{y}]\!] = \texttt{(...)}$$

$$[\![\texttt{(consp x)}]\!] = \texttt{t} \text{ iff } [\![\mathrm{x}]\!] \text{ is of the form } \texttt{(...)} \text{ but not } \texttt{()}.$$

Notice that since `consp` is a recognizer it returns a Boolean. So, if $[\![\mathrm{x}]\!]$ is an atom, then $[\![\texttt{(consp x)}]\!] = \texttt{nil}$.

Here are some examples.

$$[\![\texttt{(consp 3)}]\!] = \texttt{nil}$$

$$[\![\texttt{(consp (cons nil nil))}]\!] = \texttt{t}$$

$$[\![\texttt{(consp nil)}]\!] = \texttt{nil}$$

The semantics of `first` and `rest` is given with the following rules.

$$[\![\texttt{(first x)}]\!] = \texttt{a}, \text{where } [\![\mathrm{x}]\!] = \texttt{(a ...)} \text{ for some } \texttt{a}, \ \ldots$$

$$[\![\texttt{(rest x)}]\!] = \texttt{(...)}, \text{where } [\![\mathrm{x}]\!] = \texttt{(a ...)} \text{ for some } \texttt{a}, \ \ldots$$

Here are some examples.

$$[\![\texttt{(first (cons (if t 3 4) (cons 1 ())))}]\!] = 3$$

$$[\![\texttt{(first (rest (cons (if t 3 4) (cons 1 ()))))}]\!] = 1$$

$$[\![\texttt{(rest (cons (if t 3 4) (cons 1 (if t nil t))))}]\!] = \texttt{(cons 1 ())}$$

If you try evaluating `(rest (cons (if t 3 4) (cons 1 (if t () t))))` at the ACL2s command prompt, here is what ACL2s reports.

```
(1)
```

Since lists are so prevalent, ACL2s includes a special way of constructing them. Here is an example.

```
(list 1)
```

is just a shorthand for `(cons 1 ())`, *e.g.*, notice that asking ACL2s to evaluate

```
(equal (LIST 1) (cons 1 ()))
```

results in `t`. What is `list` really? (By the way notice that symbols in ACL2s, such as `list`, are case-insensitive.) It is not a function. Rather, it is a *macro*. There is a lot to say about macros, but for our purposes, all we need to know is that a macro gives us abbreviation power. In general

(list x_1 x_2 $\cdots$ x_n)

abbreviates (or is shorthand for)

(cons x_1 (cons x_2 $\cdots$ (cons x_n nil) $\cdots$))

2.6 Contract Violations

Consider

```
(unary-/ 0)
```

If you try evaluating this, you get an error because you violated the contract of `unary-/`. When a function is called on arguments that violate the input contract, we say that the function call resulted in an *input contract violation*. If such a contract violation occurs, then the function does not return anything.

Contract checking is more subtle than this, *e.g.*, consider the following definition.

```
(defunc foo (a)
  :input-contract (integerp a)
  :output-contract (booleanp (foo a))
  (if (posp a)
      (foo (- a 1))
    (rest a)))
```

ACL2s will not admit this function unless it can prove that every function call in the body of `foo` satisfies its contract, a process we call *body contract checking* and that `foo` satisfies its contract, a process we call *function contract checking*. These checks yield five body contract conjectures and one function contract conjecture.

Exercise 2.4 *Identify all the body contract checks and function contract checks that the definition of* `foo` *gives rise to. Which (if any) of these conjectures is always true? Which (if any) of these conjectures is sometimes false?*

Notice that contract checking happens even before the function is admitted. This is called "static" checking. Another option would have been to perform this check "dynamically." That is, all the contract checking above would be performed as the code is running.

2.7 Termination

All ACL2s function definitions have to terminate on all inputs that satisfy the input contract.

For example, consider the following "definition."

```
(defunc listp (a)
  :input-contract t
  :output-contract (booleanp (listp a))
  (if (consp a)
      (listp a)
    (equal a nil)))
```

ACL2s will not accept the above definition and will report that it could not prove termination.

Let's look at another example.

Define a function that given n, returns $0 + \cdots + n$.

Here is one possible definition:

```
;; sum-n: integer -> integer
;; Given integer n, return 0+1+2+...+n
(defunc sum-n (n)
  :input-contract (integerp n)
  :output-contract (integerp (sum-n n))
  (if (equal n 0)
      0
    (+ n (sum-n (- n 1)))))
(check= (sum-n 5) (+ 1 2 3 4 5))
(check= (sum-n 0) 0)
(check= (sum-n 3) 6)
```

Exercise 2.5 *The above function does not terminate. Why? Change only the input contract so that it does terminate. Next, change the output contract so that it gives us more information about the type of values* `sum-n` *returns.*

2.8 Beginner Mode

At this point, we transition from "Bare Bones" mode to "Beginner Mode" mode. If you are following along with ACL2s, switch to"Beginner" mode, which is indicated by "`B`" in the prompt.

Recall the issue we had when we defined `posp`? If not, review the notes. In Beginner Mode we will address this issue.

It turns out to be really useful if Boolean functions such as `and`, `or` and `not` are just abbreviations for `if`. This is actually the case in Beginner mode. The Boolean functions

are really *macros* that get expanded into `if` expressions. As was the case with the `list` macro, `and` and `or` accept an arbitrary number of arguments. For example,

1. `(and)` abbreviates `t`
2. `(and p)` abbreviates `p`
3. `(and p q)` abbreviates `(if p q nil)`
4. `(and p q r)` abbreviates `(if p (if q r nil) nil)`
5. `(or)` abbreviates `nil`
6. `(or p)` abbreviates `p`
7. `(or p q)` abbreviates `(if p p q)`

and so on. This makes writing contracts simpler.

Here is an example. Suppose we want to define a function that given an integer ≥ -5 checks to see if it is > 5.

Here is how one might define the function.

```
(defunc >5 (x)
  :input-contract (and (>= x -5) (integerp x))
  :output-contract (booleanp (>5 x))
  (> x 5))
```

ACL2s does not accept the definition. What's the problem? Well, ACL2s checks the contracts for the expressions in the input and output contracts of a function definition! This is called input contract checking and output contract checking.

What's the contract for `>=`? That it is given rationals, but we don't know that `x` is a rational!

We have to write our input contracts so that they accept anything as input.

Here's a second attempt.

```
(defunc >5 (x)
  :input-contract (and (integerp x) (>= x -5))
  :output-contract (booleanp (>5 x))
  (> x 5))
```

This works, in part because `and` is really an abbreviation for `if`.

When checking output contracts, we get to assume that the input contract holds. In the above definition, that means that we get to assume that `x` is an integer that is ≥ -5. This assumption allows contract checking to pass on `(> 5)`. Contract checking also passes on `booleanp`, as it has an input contract of `t`.

Next, we will define `even-natp`, a function that given a natural number determines whether it is even. We will start with a recursive definition and here are some tests.

```
(check= (even-natp 0) t)
(check= (even-natp 1) nil)
(check= (even-natp 22) t)
```

Here is the definition.

```
(defunc even-natp (x)
```

```
  :input-contract (natp x)
  :output-contract (booleanp (even-natp x))
  (if (equal x 0)
      t
    (not (even-natp (- x 1)))))
```

This is a data-driven definition. Check the function and body contracts.

The astute reader will have noticed that we could have written a non-recursive function, *e.g.*:

```
(defunc even-natp (x)
  :input-contract (natp x)
  :output-contract (booleanp (even-natp x))
  (natp (/ x 2)))
```

This is not a data-driven definition. It required more insight.

Next, let's define a version of the above function that works for integers. Again, we will write a recursive definition. Here are some tests.

```
(check= (even-integerp 0) t)
(check= (even-integerp 1) nil)
(check= (even-integerp -22) t)
```

The integer datatype can be characterized as follows.

$$Int : 0 \mid Int + 1 \mid Int - 1$$

So, a basic way of defining recursive functions over the integers is to have three cases, two of which are recursive, as per the data definition above.

```
(defunc even-integerp (x)
  :input-contract (integerp x)
  :output-contract (booleanp (even-integerp x))
  (cond ((equal x 0) t)
        ((< x 0) (not (even-integerp (+ x 1))))
        (t (not (even-integerp (- x 1))))))
```

The above function uses `cond`, a macro that expands into a nest of `if`s. In general,

```
(cond (c1 e1)
      (c2 e2)
      ...
      (cn en))
```

expands into

```
(if c1
    e1
  (if c2
      e2
    ...
    (if cn
        en
      nil)))
```

Notice the last nil! For the sake of code clarity, we will always make the last test of a `cond` (`cn` above) equal to `t`. That is, we will always make the trailing else case explicit.

2.9 Contracts

Even though contracts can be quite complicated, we will mostly restrict the use of contracts as indicated below.

```
(defunc f (x1 ... xn)
  :input-contract t | (R1 xi) | (and (R1 y1) ... (Rm ym))
  :output-contract (R (f x1 ... xn))
  ...)
```

The vertical bar `|` denotes a choice and our input contracts are of three possible forms, where the y_i are arguments to `f` without repetitions and R, R_i are recognizers.

Notice that if our contracts are of this form, then contract checking of the `:input` and `:output` contracts is easy. We are using recognizers everywhere, so we know that contract checking will succeed. For this reason, we often do not explicitly mention input and output contract checking.

2.10 Helpful Functions

In this section, we introduce several helpful functions. As you read this section, try to define the functions on your own. These functions are built-in to Beginner mode.

The `endp` function checks if a list is empty.

```
(defunc endp (l)
  :input-contract (listp l)
  :output-contract (booleanp  (endp l))
  (not (consp l)))
```

Notice that `endp` is not a recognizer because the input contract is not `t`.

Can we make the input-contract `t`? Yes. Consider the following function.

```
(defunc atom (l)
  :input-contract t
  :output-contract (booleanp (atom l))
  (not (consp l)))
```

The functions `atom` and `endp` have the same output contract and body. `Atom` is a recognizer (but for historical reasons does not end with a `p`). `Atom`, as the name implies, recognizes atoms. Why do we want two functions with the same body but different input contracts? Well, the idea is that we should only use `endp` when we are checking a list and using `endp` gives us more guarantees. If we make a mistake, then by using `endp`, we enable ACL2s to find the error for us. On the other hand, we should only use atom when in fact we might want to check non-lists.

The `len` function returns the length of a list. This is the simplest example of a data-driven definition. The idea is to define `len` using a template derived from the contract of its input variable, which is a list. A list is either empty or consists of the `first` element of the

list and the `rest` of the list. Therefore our template also has two cases and in the second case, we can assume that `len` returns the correct answer on the `rest` of the list, so all that is left is to add one to the answer.

```
(defunc len (l)
  ; Returns the length of list l.
  :input-contract (listp l)
  :output-contract (natp (len l))
  (if (endp l)
      0
    (+ 1 (len (rest l)))))
```

Notice that the function definition above has a comment describing the function. A semicolon (;) denotes the beginning of a comment and the comment lasts until the end of the line.

The `app` function appends two lists together. Let's try using a data-driven definition. There are two arguments. In cases where there are multiple arguments, we have to think about which of the arguments should control the recursion in `app`. It is simpler when only one argument is needed, so let's try it with the first argument. You might find it useful to either visualize how `app` should work, or to try it on some examples, or to develop a simple notation to experiment with your options. Here is what such a notation might look like. First, let us see what happens if we try recurring on the first argument of `app`. As was the case with the definition of `len`, we have two cases to consider: either the first argument is the empty list or it is a non-empty list. The first case is easy.

`app () Y` = `Y`

For the second case, we might come up with the following.

`app (cons a B) Y` = `aBY` = `cons a (app B Y)`

Notice that the first argument to `app` is a cons and we are using capital letters to denote lists and lower-case letters to denote elements. The `aBY` just indicates that in the answer first we want the element `a` and then the lists `B` and `Y`. How can we express that using a recursive call of `app` on the rest of the first argument? By consing `a` onto the list obtained by calling `app` on `B` and `Y`.

```
(defunc app (x y)
  ; App appends two lists together
  :input-contract (and (listp x) (listp y))
  :output-contract (listp (app x y))
  (if (endp x)
      y
    (cons (first x) (app (rest x) y))))
```

What if we try recurring on the second argument to `app`? Then the base case is easy.

`app X ()` = `X`

For the recursive case, we might come up with the following.

`app X (cons a B)` = `XaB` = `(app (?? X a) B)`

Notice that the second argument in the recursive call has to be in terms of `B`, the rest of the second argument to `app`. The `??` function should add `a` at the end of `X`. We may call such a function `snoc` since it is the symmetric version of `cons`. We do not have such a function, so if we want to recur on the second argument, we need to define it.

Exercise 2.6 *Define:* `snoc` *and a version of* `app` *that recurs on its second argument, as outlined above.*

Notice that data-driven definitions are guaranteed to terminate because in the recursive call the argument upon which the definition is based is "decreasing." We will make this precise later.

The `rev` function reverses a list.

```
(defunc rev (x)
  ; Rev reverses a list
  :input-contract (listp x)
  :output-contract (listp (rev x))
  (if (endp x)
      ()
    (app (rev (rest x)) (list (first x)))))
```

2.11 Quote

Even though not every expression is an object in the universe, it is the case that for every object in the universe, there is an expression that evaluates to it. An easy way to denote such an object is to use quote. For example `'(if 1)` denotes the two element list whose first element is the symbol `if` and whose second element is 1.

Certain atoms, including numbers, but also characters and strings evaluate to themselves, so we do not normally quote such atoms. However, when non-Boolean symbols are used in an expression we have to be careful because symbols are used as variables. For example, `x` in the body of `rev` (above) is a variable. If we really want to denote the symbol `r`, we have to write `'r`. Consider the difference between `(cons r l)` and `(cons 'r l)`. In the first expression we are consing the value corresponding to the variable `r` to `l`, but in the second, we are consing the symbol `r` to `l`.

2.12 Let

A `let` expression:

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

binds its local variables, the `v`i, in parallel, to the values of the `x`i, and evaluates `body` using that binding.

For example:

```
(let ((x '(a b c))
      (y '(c d)))
  (app (app x y) (app x y)))
```

evaluates to `(a b c c d a b c c d)`. This saves us having to type `'(a b c)` and `'(c d)` multiple times. Notice how the use of quotes also simplifies things, *e.g.*, instead of `(list 'a 'b 'c)` we can write `'(a b c)`.

Maybe we can avoid having to type `(app x y)` multiple times. What about?

```
(let ((x '(a b c))
      (y '(c d))
      (z (app x y)))
  (app (app x y) z))
```

This does not work. Why not? Because let binds in parallel, so `x` and `y` in the `z` binding are not yet bound.

What we really want is a binding form that binds sequentially. That is what `let*` does.

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

binds its local variables, the `v`*i*, sequentially, to the values of the `x`*i*, and evaluates `body` using that binding. So the following works.

```
(let* ((x '(a b c))
       (y '(c d))
       (z (app x y)))
  (app (app x y) z))
```

As does this further simplified expression.

```
(let* ((x '(a b c))
       (y '(c d))
       (z (app x y)))
  (app z z))
```

So, `let` and `let*` give us abbreviation power.

2.13 Testing

Instead of

```
(check= (app (list 1 2) (list)) (list 1 2))
```

we can write

```
(test? (equal (app (list x y) (list)) (list x y)))
```

The above means that we are claiming that for all elements of the ACL2s universe, x and y,

```
(app (list x y) (list)) is equal to (list x y)
```

If we only had access to constants, like 1 and 2, we would have to write out an infinite number of tests to say the same thing.

ACL2s also supports `thm` forms. If we use `thm` instead of `test?`, we are asking ACL2s to *prove* the property (not just test it). ACL2s will fail if it cannot find a proof (even if the property is true). We will not use `thm` until later, when we get to theorem proving because `thm` can fail even if the property holds. Sometimes even `test?` will report that it proved the property, but all that is required for `test?` to succeed is that no counterexample is found.

To summarize `test?`, tells ACL2s to test the property. ACL2s might be able to prove it, in which case we know that it is true. If it can't, it might find a counterexample, in which case we know it is false. If neither case holds, the form succeeds and that means ACL2s could not prove that the property is true and after testing it, it did not find a counterexample. `Test?` is highly customizable, *e.g.*, we can tell it how much testing to perform. To see more information, issue the following command on the ACL2s REPL.

```
:doc test?
```

Let's explore `test?` in more detail, using the functions `even-natp` and `even-integerp`, defined previously.

Here is a `test?` property that claims that `even-integerp` and `even-natp` agree on natural numbers.

```
(test? (implies (natp n)
                (equal (even-integerp n)
                       (even-natp n))))
```

This is a property of our code. This gives us way more power than `check=` because if the property is true, then that corresponds to an infinite number of checks.

`Test?` forms should be of the form

```
(test? (implies hyp concl))
```

where the hypothesis (or antecedent) `hyp` is of the form

```
(and (R1 x1) ... (Rn xn) ...)
```

and all the R*i*'s are recognizers and the x*i*s are variables appearing in the conclusion, `concl`. The second `...` can be some other, extra assumptions.

We have to perform contract checking on all the non-recognizers. The stuff after the recognizers must satisfy its contracts, assuming everything before it holds. The functions in the conclusion must satisfy their contracts assuming that the hypothesis holds.

Consider the `test?` above. To satisfy the input contract of `even-integerp`, the hypothesis must imply that `n` is an integer and to satisfy the input contract of `even-natp` the hypothesis must imply that `n` is a natural number, hence the `natp` check suffices for contract checking the property. What if we replace `natp` by `integerp`? Then contract checking fails because `even-natp` is only defined on natural number so we have no idea what it does with negative integers.

Now, consider a second `test?` form.

```
(test? (implies (and (integerp n)
                     (< n 0))
                (equal (even-integerp n)
                       (even-natp (* n -1)))))
```

Go over contract checking. Note that `<` is OK, because `n` is an integer and `even-natp` is OK because `n` is an integer less than 0, so `(* n -1)` is a natural number.

Notice that these two properties characterize `even-integerp` in terms of `even-natp`, so they show another way we could have defined `even-integerp`:

```
(defunc even-integerp (x)
  :input-contract (integerp x)
  :output-contract (booleanp (even-integerp x))
  (if (natp x)
      (even-natp x)
    (even-natp (* x -1))))
```

What if we write the following. Does contract checking succeed?

```
(test? (implies (and (natp n) (< 20/3 n))
                (equal (even-integerp n)
                       (even-natp n))))
```

Yes. The extra assumption `(< 20/3 n)` poses no problem because `20/3` and `n` are both rationals.

What about the following?

```
(test? (implies (< 20/3 n)
                (equal (even-integerp n)
                       (even-natp n))))
```

Contract checking fails and reveals an error in our property because `<` does not have its contracts satisfied and neither do the functions in the conclusion.

2.14 Data Definitions

ACL2s provides a powerful data definition framework that allows us to define new data types. New data types are created by combining primitive types using defdata type combinators.

The primitive types include `rational`, `nat`, `integer` and `pos` whose recognizers are `rationalp`, `natp`, `integerp` and `posp`, respectively. Notice the naming convention we use: we append the character "p" to typenames to obtain the name of their recognizer. In ACL2s, the type `all` includes everything in the universe, *i.e.*, every type is a subtype (subset) of `all`.

We introduce the ACL2s data definition framework via a sequence of examples.

Singleton types allow us to define types that contain only one object. For example:

```
(defdata one 1)
```

All data definitions give rise to a recognizer. The above data definition gives rise to the recognizer `onep`.

Enumerated types allow you to define finite types.

```
(defdata name (enum '(emmanuel angelina bill paul sofia)))
```

Range types allow you to define a range of numbers. The two examples below show how to define the rational interval [0..1] and the integers greater than 2^{64}.

```
(defdata probability (range rational (0 <= _ <= 1)))
(defdata big-nat (range integer ((* 1024 1024) < _)))
```

Notice that we need to provide a domain, which is either `integer` or `rational`, and the set of numbers is specified with inequalities using `<` and `<=`. One of the lower or upper bounds can be omitted, in which case the corresponding value is taken to be negative or positive infinity.

Product types allow us to define structured data. The example below defines a type consisting of list with exactly two strings.

```
(defdata fullname (list string string))
```

Records are product types, where the fields are named. For example, we could have defined `fullname` as follows.

```
(defdata fullname-rec (record (first . string)
                              (last . string)))
```

In addition to the recognizer `fullname-recp`, the above type definition gives rise to the constructor `fullname-rec` which takes two strings as arguments and constructs a record of type `fullname-rec`. The type definition also generates the accessors `fullname-rec-first` and `fullname-rec-last` that when applied to an object of type `fullname-rec` return the `first` and `last` fields, respectively.

We can create list types using the `listof` combinator as follows.

```
(defdata loi (listof integer))
```

This defines the type consisting of lists of integers.

Union types allow us to take the union of existing types. Here is an example.

```
(defdata intstr (oneof integer string))
```

Recursive type expressions involve the `oneof` combinator and product combinations, where additionally there is a (recursive) reference to the type being defined. For example, here is another way of defining a list of integers.

```
(defdata loi (oneof nil (cons integer loi)))
```

The data definition framework has more advanced features, *e.g.*, it supports mutually-recursive types, recursive record types, map types, custom types, and so on. We will introduce such features as needed.

2.15 Design Recipe

Consider the following function definition.

```
(defunc foo (x y z)
  :input-contract (and (natp x) (natp y) (true-listp z))
  ...)
```

The input contract, which specifies the types of the inputs, dictates that you should have at least 8 tests because for each variable, there should be as many tests as there are cases

in the data definition of its type and all possible combinations of tests spanning multiple variables should be considered. That gives rise to $2 * 2 * 2 = 8$ tests. So you should have 8 tests of the form.

```
(check= testi ansi)
```

Tests and examples are very important. Use them to understand the specification, *e.g.*, by considering all cases. Visualize the computation; this helps you write the code. If you have difficulty writing code for the general case, use examples as a guide.

Most of the code we are going to look at is data driven: we will be recurring by counting down by 1 or by traversing a list. Take advantage of this as follows.

1. Identify the variable(s) that control the recursion.
2. Once you do that, write down the template consisting of the base cases and recursive cases.
3. If you get stuck, look at the examples and generalize.
4. After you write your program, evaluate it on the examples you wrote.

Contracts play a key role in how we think about function definitions. Make sure you understand the contracts for all the functions you use. Many programming errors students make are due to contract violations, so as you are developing your program check to see that every function call respects its contract.

Take advantage of `test?` to enhance the testing we get from `check=`. Use `test?` to write down properties that should be true of the functions you define. The more coverage your tests provide the better. When designing `test?` properties, make the properties implementation independent, *e.g.*, if a function is supposed to remove duplicates, but the exact order in which elements appear in the output is not specified, then write `test?` properties that do not assume a particular order. This makes it possible to go back and modify the function in the future without having `test?` forms fail. Notice that the same advice holds for `check=` forms. For example, instead of checking that the output is some particular list, check that it is a permutation of the list.

Efficiency is a significant issue when designing systems, but it is mostly an orthogonal issue. For example, if we want to develop a sorting algorithm, then the specification for a sorting algorithm is independent of the implementation. This separation of concerns allows us to design systems in a modular, robust way. In this course, we will not care that much about efficiency. It will come up and we will mention it, but our emphasis will be on simple definitions and specifications.

The templates that arise when defining functions over lists and natural numbers should be obvious, but here is a brief review. Recall the data definition for lists.

$$\mathit{List} : \texttt{()} \mid \texttt{(cons } \mathit{All} \ \mathit{List}\texttt{)}$$

We say that cons is a *constructor*. Now, when we define recursive functions, we use the *destructors* `first` and `rest` to destruct a cons into its constituent parts. Functions defined this way work because every time I apply a destructor I decrease the size of an element. What about `nat`? The idea is similar.

$$\mathit{Nat} : 0 \mid \mathit{Nat} + 1$$

So, `+ 1` is a constructor and the corresponding destructor used when we define recursive functions is `- 1`. What about `integer`?

$$Int : 0 \mid Int + 1 \mid Int - 1$$

So, `+ 1` and `- 1` are the constructors and the corresponding destructors used when we define recursive functions are `- 1` and `+ 1`, respectively.

We now discuss what templates user-defined datatypes that are recursive give rise to. Many of the datatypes we define are just lists of existing types. For example, we can define a list of rationals as follows.

```
(defdata lor (listof rational))
```

If we define a function that recurs on one of its arguments, which is a list of rationals, we just use the list template and can assume that if the list is non-empty then the first element is a rational and the rest of the list is a list of rationals.

If we have a more complex data definition, say:

```
(defdata PropEx (oneof boolean symbol
                       (list UnaryOp PropEx)
                       (list Binop PropEx PropEx)))
```

Then the template we wind up with is exactly what you would expect from the data definition.

```
(defunc foo (px ...)
  :input-contract (and (PropExp px) ...)
  :output-contract (... (foo px ...))
    (cond ((booleanp px) ...)
          ((symbolp px) ...)
          ((UnaryOpp (first px)) ... (foo (second px)) ...)
          (t ...(foo (second px)) ... (foo (third px)) ...)))
```

Notice that in the last case, there is no need to check `(BinOpp (first px))`, since it has to hold, hence the `t`. Also, for the recursive cases, we get to assume that `foo` works correctly on the destructed argument (`(second px)` and `(third px)`).

All of your functions where the recursion is governed by variables of type `propexp` should use the above template.

We now explore data definitions in a little more detail. Recall the data definition for lists.

$$List : \texttt{()} \mid \texttt{(cons } All\ \ List\texttt{)}$$

We say that cons is a *constructor*. Now this definition is recursive, *i.e.*, *List* is defined in terms of itself. Why does such a circular definition make sense?

The above is really a fixpoint definition of lists. This view allows us to do away with the circularity. Here's the idea. Start with

$$L_0 = \{\texttt{()}\}$$

and then create all conses of the form

```
(cons x l)
```

and repeat, *i.e.*, we can define

$$L_{i+1} = \{\texttt{()}\} \cup \{\texttt{(cons}\ x\ l\texttt{)} : x \in \mathit{All} \wedge l \in L_i\}$$

and now we define *List* to be the union of all the L_i.

$$\mathit{List} = \bigcup_{i \in \mathbb{N}} L_i$$

When we define recursive functions using the design recipe, we use the *destructors* `first` and `rest` to destruct a cons into its constituent parts. Functions defined this way make sense because they terminate: every time we apply a destructor we take an element in L_{i+1} (let $i+1$ be the smallest index such that the element is in L_{i+1} and notice that because first we check that the element is not the empty list, we know that the element is not in L_0) and get, at worst, an element in L_i. Therefore, after finitely many steps we have to reach the base case and therefore the function is guaranteed to terminate. In this way, we have shown how to remove the apparent circularity in the definition of lists.

Let's rephrase this to see why we used the term *fixpoint.* We start by defining the following function.

$$f(Y) = \{\texttt{()}\} \cup \{\texttt{(cons}\ x\ l\texttt{)} : x \in \mathit{All} \wedge l \in Y\}$$

A *fixpoint* for f is a set Z such that $f(Z) = Z$. Does f have a fixpoint? Yes. *List*! That is:

$$f(\mathit{List}) = \mathit{List}$$

How do we compute a fixpoint? Well, we take the limit of f as follows

$$\mathit{List} = \lim_{i \in \mathbb{N}} f^{i+1}(\emptyset)$$

where f^i is the i-fold composition of f so $f^0(X) = X$, $f^1(X) = f(X)$, $f^2(X) = f(f(X)), \ldots$. This is what is called the *least* fixpoint. Notice that $f^i(\emptyset) = L_i$.

2.16 Program Mode

Sometimes it is helpful to temporarily turn off theorem proving in ACL2s. Why would you want to? Well, say you want to quickly prototype your function definitions because ACL2s is complaining or you just want to use ACL2s without all the theorem proving turned on.

The answer is yes you can. Just put this one line right before the point you want to switch from ACL2s's normal mode, called logic mode, to program mode.

```
:program
```

ACL2s will still test contracts and will report a contract violation if it finds it. Think of this as free testing. ACL2s will not worry about termination or about proving any theorems at all (so body contracts and function contracts are not proved).

Once you're done exploring, you can undo past the `:program` command to go back to logic mode, or you can even switch back and forth with the following command

```
:logic
```

If you mix up modes like this, then you will not be able to define logic mode functions that depend on program mode functions.

2.17 Dealing with Definition Failures

While it's amazing that ACL2s can statically prove that your functions satisfy their contracts automatically (you'll see how amazing this is once we start proving theorems), unfortunately, there will be times when you give it a function definition that is logically fine, but, alas, ACL2s cannot prove that it is correct.

If this has ever happened to you, read on.

What do you do in such a situation?

Well, there are two options I want to show you. Let's go through them in turn.

The first option is to revert to "program mode" and turn off testing. In program mode and with testing off, ACL2s behaves the way most programming languages do: ACL2s does not try to prove or test any conjectures. Don't resort to using Racket or Lisp or whatever. Do this instead! For example, suppose you have the following definition.

```
(defunc ! (x)
  :input-contract (integerp x)
  :output-contract (integerp (! x))
  (if (equal x 0)
    1
    (* x (! (- x 1)))))
```

ACL2s complains about something (termination), so you can turn off testing and revert to program mode as follows. Note: it is important to turn off testing before you going to program mode.

```
(acl2s-defaults :set testing-enabled nil)
:program
```

Now, if you submit the function definition, ACL2s accepts it.

```
(defunc ! (x)
  :input-contract (integerp x)
  :output-contract (integerp (! x))
  (if (equal x 0)
    1
    (* x (! (- x 1)))))
```

Now you can test your code. For example:

```
(foo 10)
```

works as expected, but

```
(foo -1)
```

leads to a stack overflow (which indicates a termination problem).

Unfortunately, if you define a program mode function, then every new function that depends on it will also have to be a program mode function. My suggestion is that you do this to debug your code. If you can fix what the problem is great. If not, then it is better to use the next option if you can.

The first option was draconian. You turned off the power of the theorem prover completely.

The second option represents a more measured response. Instead of turning off the theorem prover, we tell it to try proving termination, etc., but if it fails, to continue anyway. In essence, we are asking ACL2s for its best effort.

```
(acl2s-defaults :set testing-enabled nil)
(set-defunc-termination-strictp nil)
(set-defunc-function-contract-strictp nil)
(set-defunc-body-contracts-strictp nil)
```

The first command above is as before: we turn off testing. You don't have to do that, but sometimes it helps (*e.g.*, if you defined a non-terminating function and we try to test it, you'll get a stack overflow).

The other commands tell ACL2s to not be strict with regards to termination, function contracts and body contracts. After ACL2s finishes processing your function definition, it gives you a little summary of what it was able to prove.

Let's see what happens with our above function definition. Here is what ACL2s outputs.

```
...
**The definition of ! was accepted in program mode!!
Function Name : !
Termination proven ------ [ ]
Main Contract proven ---- [ ]
Body Contracts proven --- [ ]
```

This means that neither termination, nor the main contract, nor the body contracts were proven.

If we fix the contract problem, *e.g.*, as follows.

```
(defunc ! (x)
  :input-contract (natp x)
  :output-contract (integerp (! x))
  (if (equal x 0)
    1
    (* x (! (- x 1)))))
```

Then ACL2s does prove everything and now the output looks as follows.

```
...
Function Name : !
Termination proven ------ [*]
Main Contract proven ---- [*]
Body Contracts proven --- [*]
```

So, the *'s tell you what parts of the function admission process was successful.

If you only need to do this for 1 function definition, you can revert back to the default settings with the following commands:

```
(acl2s-defaults :set testing-enabled t)
(set-defunc-termination-strictp t)
(set-defunc-function-contract-strictp t)
(set-defunc-body-contracts-strictp t)
```

So, you can go back and forth and you can selectively turn testing on and off.

If you get stuck on a homework problem, use the second option, but you can also use the first option if you really need to. If your code is correct, you will get full credit either way.

Part II

Propositional Logic

3

Propositional Logic

The study of logic was initiated by the ancient Greeks, who were concerned with analyzing the laws of reasoning. They wanted to fully understand what *conclusions* could be derived from a given set of *premises*. Logic was considered to be a part of philosophy for thousands of years. In fact, until the late 1800's, no significant progress was made in the field since the time of the ancient Greeks. But then, the field of modern mathematical logic was born and a stream of powerful, important, and surprising results were obtained. For example, to answer foundational questions about mathematics, logicians had to essentially create what later became the foundations of computer science. In this class, we'll explore some of the many connections between logic and computer science.

We'll start with propositional logic, a simple, but surprisingly powerful fragment of logic. Expressions in propositional logic can only have one of two values. We'll use T and F to denote the two values, but other choices are possible, *e.g.*, 1 and 0 are sometimes used.

The expressions of propositional logic include:

1. The *constant expressions true* and *false*: they always evaluate to T and F, respectively.

2. The *propositional atoms*, or more succinctly, *atoms*. We will use p, q, and r to denote propositional atoms. Atoms range over the values T and F.

Propositional expressions can be combined together with the propositional operators, which include the following.

The simplest operator is negation. Negation, $\neg$, is a *unary* operator, meaning that it is applied to a single expression. For example $\neg p$ is the negation of atom p. Since p (or any propositional expression) can only have one of two values, we can fully define the meaning of negation by specifying what it does to the value of p in these two cases. We do that with the aid of the following truth table.

p	$\neg p$
T	F
F	T

What the truth table tells us is that if we negate T we get F and if we negate F we get T.

Negation is the only unary propositional operator we are going to consider. Next we consider *binary* (2-argument) propositional operators, starting with *conjunction*, $\wedge$. The conjunction (and) of p and q is denoted $p \wedge q$ and its meaning is given by the following truth table.

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

Each row in a truth table corresponds to an *assignment*, one possible way of assigning values (T or F) to the atoms of a formula. The truth table allows us to explore all relevant assignments. If we have two atoms, there are 4 possibilities, but in general, if we have n atoms, there are 2^n possible assignments we have to consider.

In one sense, that's all there is to propositional logic, because every other operator we are going to consider can be expressed in terms of $\neg$ and $\wedge$, and almost every question we are going to consider can be answered by the construction of a truth table.

Next, we consider *disjunction*. The disjunction (or) of p and q is denoted $p \vee q$ and its meaning is given by the following truth table.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

In English usage, "p or q" often means p or q, but not both. Consider the mother who tells her child:

> You can have ice cream or a cookie.

The child is correct in assuming this means that she can have ice cream or a cookie, but not both.

As you can see from the truth table for disjunction, in logic "or" always means at least one.

We can write more complex formulas by using several operators. An example is $\neg p \vee \neg q$, where we use the convention that $\neg$ binds more tightly than any other operator, hence we can only parse the formula as $(\neg p) \vee (\neg q)$. We can construct truth tables for such expressions quite easily. First, determine how many distinct atoms there are. In this case there are two; that means we have four rows in our truth table. Next we create a column for each atom and for each operator. Finally, we fill in the truth table, using the truth tables that specify the meaning of the operators.

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

Next, we consider implication, $\Rightarrow$. This is called logical (or material) implication. In $p \Rightarrow q$, p is the antecedent and q is the consequent. Implication is often confusing to students because the way it is used in English is quite complicated and subtle. For example, consider the following sentences.

> If Obama invented the Internet, then the inhabitants of Boston are all dragons.

Is it true?
What about the following?

> If Obama was elected president, then the inhabitants Tokyo are all descendants of Godzilla.

Logically, only the first is true, but most English speakers will say that if there is no connection between the antecedent and consequent, then the implication is false.

Why is the first logically true? Because here is the truth table for implication.

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Here are two ways of remembering this truth table. First, $p \Rightarrow q$ is equivalent to $\neg p \vee q$. Second, $p \Rightarrow q$ is false only when p is T, but q is F. This is because you should think of $p \Rightarrow q$ as claiming that if p holds, so does q. That claim is true when p is F. The claim can only be invalidated if p holds, but q does not.

As a final example of the difference between logical implication (whose meaning is given by the above truth table) and implication as commonly used, consider a father telling his child:

> If you behave, I'll get you ice cream.

The child rightly expects to get ice cream if she behaves, but also expects to *not* get ice cream if she doesn't: there is an implied threat here.

The point is that the English language is subtle and open for interpretation. In order to avoid misunderstandings, mathematical fields, such as Computer Science, tend to use what is often called "mathematical English," a very constrained version of English, where the meaning of all operators is clear.

Above we said that $p \Rightarrow q$ is equivalent to $\neg p \vee q$. This is the first indication that we can often reduce propositional expressions to simpler forms. If by simpler we mean less operators, then which of the above is simpler?

Can we express the equivalence in propositional logic? Yes, using equality of Booleans, $\equiv$, as follows $(p \Rightarrow q) \equiv (\neg p \vee q)$.

Here is the truth table for $\equiv$.

p	q	$p \equiv q$
T	T	T
T	F	F
F	T	F
F	F	T

How would you simplify the following?

1. $p \wedge \neg p$

2. $p \vee \neg p$

3. $p \equiv p$

Here is one way.

1. $(p \wedge \neg p) \equiv \mathit{false}$

2. $(p \vee \neg p) \equiv \mathit{true}$

3. $(p \equiv p) \equiv \mathit{true}$

The final binary operator we will consider is $\oplus$, xor. There are two ways to think about xor. First, note that xor is e*x*clusive *or*, meaning that exactly one of its arguments is true. Second, note that xor is just the Boolean version of not equal. Here is the truth table for $\oplus$.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

To avoid using too many parentheses, from now on we will follow the convention: $\neg$ binds tightest, followed by $\{\wedge, \vee\}$, followed by $\Rightarrow$, followed by $\{\oplus, \equiv\}$. Hence, instead of

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

we can write

$$p \vee \neg q \Rightarrow r \oplus \neg r \Rightarrow q \wedge \neg p$$

We will also consider a *ternary* operator, *i.e.*, a operator with three arguments. The operator is *ite*, which stands for "if-then-else," and means just that: if the first argument holds, return the second (the then branch), else return the third (the else branch). Since there are three arguments, there are eight rows in the truth table.

p	q	r	$\mathit{ite}(p, q, r)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

Here are some very useful ways of characterizing propositional formulas. Start by constructing a truth table for the formula and look at the column of values obtained. We say that the formula is:

- *satisfiable* if there is at least one T

- *unsatisfiable* if it is not satisfiable, *i.e.*, all entries are F
- *falsifiable* if there is at least one F
- *valid* if it is not falsifiable, *i.e.*, all entries are T

We have seen examples of all of the above. For example, $p \wedge q$ is satisfiable, since the assignment that makes p and q both T results in $p \wedge q$ also being T. This example is also falsifiable, as evidenced by the assignment that makes p F and q T. An example of an unsatisfiable formula is $p \wedge \neg p$. If you construct the truth table for it, you will notice that every assignment makes it F (so it is falsifiable too). Finally, an example of a valid formula is $p \vee \neg p$.

Notice that if a formula is valid, then it is also satisfiable. In addition, if a formula is unsatisfiable, then it is also falsifiable.

Validity turns out to be really important. A valid formula, often also called a *theorem*, corresponds to a correct logical argument, an argument that is true regardless of the values of its atoms. For example $p \Rightarrow p$ is valid. No matter what p is, $p \Rightarrow p$ always holds.

3.1 P = NP

A natural question arises at this point: Is there an algorithm that given a propositional logic formula returns "yes" if it is satisfiable and "no" otherwise?

Here is an algorithm: construct the truth table. If we have the truth table, we can easily decide satisfiability, validity, unsatisfiability, and falsifiability.

The problem is that the algorithm is inefficient. The number of rows in the truth table is 2^n, where n is the number of atoms in our formula.

Can we do better? For example, recall that we had an inefficient recursive algorithm for `sum-n` (the function that given a natural number n, returns $\sum_{i=0}^{n} i$). The function required n additions, which is exponential in the number of bits needed to represent n ($\log n$ bits are needed). However, with a little math, we found an algorithm that was efficient because it only needed 3 arithmetic operations.

Is there an efficient algorithm for determining Boolean satisfiability? By efficient, we mean an algorithm that in the worst case runs in polynomial time. Gödel asked this question in a letter he wrote to von Neumann in 1956. No one knows the answer, although this is one of the most studied questions in computer science. In fact, most of the people who have thought about this problem believe that no polynomial time algorithm for Boolean satisfiability exists.

3.2 The Power of Xor

Let us take a short detour, I'll call "the power of xor."

Suppose that you work for a secret government agency and you want to communicate with your counterparts in Europe. You want the ability to send messages to each other using the Internet, but you know that other spy agencies are going to be able to read the messages as they travel from here to Europe.

How do you solve the problem?

Well, one way is to have a shared secret: a long sequence of F's and T's (0's and 1's if you prefer), in say a code book that only you and your counterparts have. Now, all messages are really just sequences of bits, which we can think of as sequences of F's and T's, so you take your original message m and xor it, bit by bit, with your secret s. That gives rise to coded message c, where $c \equiv m \oplus s$. Notice that here we are applying $\equiv$ and $\oplus$ to sequences of Boolean values, often called *bit-vectors*.

Anyone can read c, but they will have no idea what the original message was, since s effectively scrambled it. In fact, with no knowledge of s, an eavesdropper can extract no information about the contents of m from c, except for the length of the message, which can be partially hidden by padding the message with extra bits.

But, how will your counterparts in Europe decode the message? Notice that some propositional reasoning shows that $m = c \oplus s$, so armed with your shared secret, they can determine what the message is.

This is one of the most basic encryption methods. It provides extremely strong security but is difficult to use because it requires sharing a secret. While sharing a key might be feasible for government agencies, it is *not* feasible for you and all the companies you buy things from on the Internet.

The shared secret should be a random sequence of bits and once bits of the secret key are used, they should never be used again. Why? This method is called the one-time pad method.

Exercise 3.1 *Show that this scheme is secure. Here's how. Show that for any coded message c of length l, if an adversary only knows c (but not m and not s), then for* any *m (of length l), there exists a secret s (of length l) such that $c = m \oplus s$.*

Exercise 3.2 *If s, the secret key, is not a random sequence, why is this a bad idea? For example, what if s is all 0's or all 1's?*

Exercise 3.3 *If you keep reusing s, the secret key, why is this a bad idea?*

Is this a reasonable way to exchange information? Well you have probably seen movies with the "red telephone" that connects the Pentagon with the Kremlin. While a red telephone was never actually used, there *was* a system in place to allow Washington to directly and securely communicate with Moscow. The original system used encrypted teletype messages based on one-time pads. The countries exchanged keys at their embassies.

3.3 Useful Equalities

Here are some simple equalities involving the constant *true*.

1. $p \vee true \equiv true$
2. $p \wedge true \equiv p$
3. $p \Rightarrow true \equiv true$
4. $true \Rightarrow p \equiv p$
5. $p \equiv true \equiv p$

6. $p \oplus true \equiv \neg p$

Here are some simple equalities involving the constant *false*.

1. $p \vee false \equiv p$
2. $p \wedge false \equiv false$
3. $p \Rightarrow false \equiv \neg p$
4. $false \Rightarrow p \equiv true$
5. $p \equiv false \equiv \neg p$
6. $p \oplus false \equiv p$

Why do we have separate entries for $p \Rightarrow false$ and $false \Rightarrow p$, above, but not for both $p \vee false$ and $false \vee p$? Because $\vee$ is commutative. Here are some equalities involving commutativity.

1. $p \vee q \equiv q \vee p$
2. $p \wedge q \equiv q \wedge p$
3. $p \equiv q \equiv q \equiv p$
4. $p \oplus q \equiv q \oplus p$

What about $\Rightarrow$. Is it commutative? Is $p \Rightarrow q \equiv q \Rightarrow p$ valid? No. By the way, the right-hand side of the previous equality is called the *converse*: it is obtained by swapping the antecedent and consequent.

A related notion is the *inverse*. The inverse of $p \Rightarrow q$ is $\neg p \Rightarrow \neg q$. Note that the inverse and converse of an implication are equivalent.

Even though a conditional is not equivalent to its inverse, it is equivalent to its *contrapositive*:

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

The contrapositive is obtained by negating the antecedent and consequent and then swapping them.

While we're discussing implication, a very useful equality involving implication is:

$$(p \Rightarrow q) \equiv (\neg p \vee q)$$

Also, we often want to replace $\equiv$ by $\Rightarrow$, which is possible due to the following equality:

$$(p \equiv q) \equiv [(p \Rightarrow q) \wedge (q \Rightarrow p)]$$

Here are more equalities.

1. $\neg\neg p \equiv p$
2. $\neg true \equiv false$
3. $\neg false \equiv true$

4. $p \wedge p \equiv p$
5. $p \vee p \equiv p$
6. $p \Rightarrow p \equiv \mathit{true}$
7. $p \equiv p \equiv \mathit{true}$
8. $p \oplus p \equiv \mathit{false}$
9. $p \wedge \neg p \equiv \mathit{false}$
10. $p \vee \neg p \equiv \mathit{true}$
11. $p \Rightarrow \neg p \equiv \neg p$
12. $\neg p \Rightarrow p \equiv p$
13. $p \equiv \neg p \equiv \mathit{false}$
14. $p \oplus \neg p \equiv \mathit{true}$

Here's one set of equalities you have probably already seen: DeMorgan's Laws.

1. $\neg(p \wedge q) \equiv \neg p \vee \neg q$
2. $\neg(p \vee q) \equiv \neg p \wedge \neg q$

Here's another property: associativity.

1. $((p \vee q) \vee r) \equiv (p \vee (q \vee r))$
2. $((p \wedge q) \wedge r) \equiv (p \wedge (q \wedge r))$
3. $((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$
4. $((p \oplus q) \oplus r) \equiv (p \oplus (q \oplus r))$

We also have distributivity:

1. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
2. $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

We also have transitivity:

1. $[(p \Rightarrow q) \wedge (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$
2. $[(p \equiv q) \wedge (q \equiv r)] \Rightarrow (p \equiv r)$

Here are some equalities involving $\equiv$ and $\oplus$.

1. $[(p \oplus q) \wedge (q \oplus r)] \Rightarrow (p \equiv r)$
2. $[\neg(p \equiv q) \equiv (p \oplus q)]$
3. $[\neg(p \oplus q) \equiv (p \equiv q)]$

1. $[(p \Rightarrow q) \wedge (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$
2. $[(p \equiv q) \wedge (q \equiv r)] \Rightarrow (p \equiv r)$

Last, but not least, we have absorption:

1. $p \wedge (p \vee q) \equiv p$
2. $p \vee (p \wedge q) \equiv p$

Let's consider absorption more carefully. Here is a simple calculation:

Proof

$p \wedge (p \vee q)$

$\equiv$ { Distribute $\wedge$ over $\vee$ }

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

$\equiv$ { $(p \wedge p) \equiv p$ }

$p \vee (p \wedge q)$ □

The above proof shows that $p \wedge (p \vee q) \equiv p \vee (p \wedge q)$, so if we show that $p \wedge (p \vee q) \equiv p$, we will have also shown that $p \vee (p \wedge q) \equiv p$.

3.4 Proof Techniques

Let's try to show that $p \wedge (p \vee q) \equiv p$. We will do this using a proof technique called case analysis.

Case analysis: If f is a formula and p is an atom, then f is valid iff both $f|_{((p\ true))}$ and $f|_{((p\ false))}$ are valid. By $f|_{((p\ x))}$ we mean, substitute x for p in f.

Proof

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

$\equiv$ { Case analysis }

$true \wedge (true \vee q) \equiv true$ and $false \wedge (false \vee q) \equiv false$

$\equiv$ { Basic Boolean equalities }

$true \equiv true$ and $false \equiv false$

$\equiv$ { Basic Boolean equalities }

$true$ □

Another useful proof technique is *instantiation*: If f is a valid formula, then so is $f|_\sigma$, where σ is a *substitution*, a list of the form:

$$((atom_1\ formula_1) \cdots (atom_n\ formula_n)),$$

where the atoms are "target atoms" and the formulas are their images. The application of this substitution to a formula uniformly replaces every free occurrence of a target atom by its image.

Here is an example of applying a substitution. $(a \vee \neg(a \wedge b))|_{((a\ (p \vee q))(b\ a))} = ((p \vee q) \vee \neg((p \vee q) \wedge a))$.

Here is an application of instantiation. $a \vee \neg a$ is valid, so therefore, so is $(a \vee \neg a)|_{((a\ (p \vee (q \wedge r))))} = (p \vee (q \wedge r)) \vee \neg(p \vee (q \wedge r))$.

3.5 Normal Forms and Complete Boolean Bases

We have seen several propositional operators, but do we have any assurance that the operators are complete? By complete we mean that the propositional operators we have can be used to represent any Boolean function.

How do we prove completeness?

Consider some arbitrary Boolean function f over the atoms $x_1, \ldots, x_n$. The domain of f has 2^n elements, so we can represent the function using a truth table with 2^n rows. Now the question becomes: can we represent this truth table using the operators we already introduced?

Here is the idea of how we can do that. Take the disjunction of all the assignments that make f true. The assignments that make f true are just the rows in the truth table for which f is T. Each such assignment can be represented by a *conjunctive clause*, a conjunction of *literals*, atoms or their negations. So, we can represent each of these assignments. Now to represent the function, we just take the disjunction of all the conjunctive clauses.

Consider what happens if we try to represent $a \oplus b$ in this way. There are two assignments that make $a \oplus b$ true and they can be represented by the conjunctive clauses $a \wedge \neg b$ and $\neg a \wedge b$, so $a \oplus b$ can be represented as the disjunction of these two conjunctive clauses: $(a \wedge \neg b) \vee (\neg a \wedge b)$.

Notice that we only need $\neg, \vee$, and $\wedge$ to represent any Boolean function!

The formula we created above was a disjunction of *conjunctive clauses*. Formulas of this type are said to be in *disjunctive normal form* (DNF). If each conjunctive clause includes all the atoms in the formula, then we say that the formula is in *full disjunctive normal form.* Another type of normal form is *conjunctive normal form* (CNF): each formula is a conjunction of *disjunctive clauses*, where a disjunctive clause is a disjunction of literals. Disjunctive clauses are also just called *clauses.* If each clause includes all the atoms in the formula, then we say that the formula is in *full conjunctive normal form.*

Can you come up with a way of representing an arbitrary truth table in full CNF? (Hint: Consider what we did for DNF.)

Any formula can be put in DNF or CNF. In fact, the input format for modern SAT solvers is CNF, so if you want to check the satisfiability of a Boolean formula using a SAT solver, you have to transform the formula so that it is in CNF.

Exercise 3.4 *You are given a Boolean formula and your job is to put it in CNF and DNF. How efficiently can this be done?*

Hint: Consider formulas of the form

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$$

$$(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \cdots \wedge (x_n \vee y_n)$$

Back to completeness. We saw that these three operators are already complete: $\neg, \vee, \wedge$. Can we do better? Can we get rid of some of them?

We can do better because $\vee$ can be represented using only $\wedge, \neg$. Similarly $\wedge$ can be represented using only $\vee, \neg$. How?

Exercise 3.5 *Can we do better yet? No. $\neg$ is not complete; neither is $\wedge$; neither is $\vee$. Prove it.*

Next, think about this claim: you can represent all the Boolean operators using just *ite* (and the constants *false*, *true*). If we can represent $\neg$ and $\vee$, then as the previous discussion shows, we're done.

$$\neg p \equiv ite(p, false, true)$$

$$p \vee q \equiv ite(p, true, q)$$

Exercise 3.6 *Represent the rest of the operators using ite.*

Exercise 3.7 *There are 16 binary Boolean operators. Are any of them complete? If so, exhibit such a operator and prove that it is complete. If not, prove that none of the operators is complete.*

3.6 Decision Procedures

When a formula is not valid, it is falsifiable, so there exists an assignment that makes it false. Such an assignment is often called a *counterexample* and can be very useful for debugging purposes. Since ACL2s is a programming language, we can use it to write our own decision procedure that provides counterexamples to validity.

Exercise 3.8 *Write a decision procedure for validity in ACL2s.*

While we are on the topic of decision procedures, it is worth pointing that we characterized formulas as satisfiable, unsatisfiable, valid, or falsifiable. Let's say we have a decision procedure for one of these four characterizations. Then, we can, rather trivially, get a decision procedure for any of the other characterizations.

Why?

Well, consider the following.

Proof

Unsat f

$\equiv$ { By the definition of sat, unsat }

not (Sat f)

$\equiv$ { By definition of Sat, Valid }

Valid $\neg f$

$\equiv$ { By definition of valid, falsifiable }

not (Falsifiable $\neg f$) $\square$

How do we use these equalities to obtain a decision procedure for either of unsat, sat, valid, falsifiable, given a decision procedure for the other?

Well, let's consider an example. Say we want a decision procedure for validity given a decision procedure for satisfiability.

Valid f

$\equiv$ { not (Sat f) $\equiv$ $Valid \neg f$, by above }

not (Sat $\neg f$)

What justifies this step? Propositional reasoning and instantiation.

Let p denote "(Sat f)" and q denote "(Valid $\neg f$)." The above equations tell us $\neg p \equiv q$, so $p \equiv \neg q$.

If more explanation is required, note that $(\neg p \equiv q) \equiv (p \equiv \neg q)$ is valid. That is, you can transfer the negation of one argument of an equality to the other.

Make sure you can do this for all 12 combinations of starting with a decision procedure for sat, unsat, valid, falsifiable, and finding decision procedures for the other three characterizations.

There are two interesting things to notice here.

First, we took advantage of the following equality:

$$(\neg p \equiv q) \equiv (p \equiv \neg q)$$

There are lots of equalities like this that you should know about, so study the provided list.

Second, we saw that it was useful to extract the propositional skeleton from an argument. We'll look at examples of doing that. Initially this will involve word problems, but later it will involve reasoning about programs.

3.7 Propositional Logic in ACL2s

This class is about logic from a computational point of view and our vehicle for exploring computation is ACL2s. ACL2s has *ite*: it is just `if`!

Remember that ACL2s is in the business of proving theorems. Since propositional logic is used everywhere, it would be great if we could use ACL2s to reason about propositional logic. In fact, we can.

Consider trying to prove that a propositional formula is valid. We would do that now, by constructing a truth table. We can also just ask ACL2s. For example, to check whether the following is valid

$$(p \Rightarrow q) \equiv (\neg p \vee q)$$

We can ask ACL2s the following query

```
(thm (implies (and (booleanp p) (booleanp q))
              (iff (implies p q) (or (not p) q))))
```

Try it in Beginner mode.

In fact, if a propositional formula is valid (that is, it is a theorem) then ACL2s will definitely prove it. We say that ACL2s is a decision procedure for propositional validity. A *decision procedure* for propositional validity is a program that given a formula can decide whether or not it is valid. We saw that ACL2s indicates that it has determined that a formula is valid with "Q.E.D." [1]

What if you give ACL2s a formula that is not valid? Try it with an example, say:

$$(p \oplus q) \equiv (p \vee q)$$

We can ask ACL2s the following query

```
(thm (implies (and (booleanp p) (booleanp q))
              (iff (xor p q) (or p q))))
```

As you can see, ACL2s also can provide counterexamples to false conjectures.

[1] Q.E.D. is abbreviation for "quod erat demonstrandum," Latin for "that which was to be demonstrated."

3.8 Word Problems

Next, we consider how to formalize word problems using propositional logic.

Consider formalizing and analyzing the following.

> Tom likes Jane if and only if Jane likes Tom. Jane likes Bill. Therefore, Tom does not like Jane.

Here's the kind of answer I expect you to give.

> Let p denote "Tom likes Jane"; let q denote "Jane likes Tom"; let r denote "Jane likes Bill."
>
> The first sentence can then be formalized as $p \equiv q$.
>
> We denote the second sentence by r.
>
> The third sentence contains the claim we are to analyze, which can be formalized as $((p \equiv q) \wedge r) \Rightarrow \neg p$.
>
> This is not a valid claim. A truth table shows that the claim is violated by the assignment that makes p, q, and r *true*. This makes sense because r (that Jane likes Bill) does not rule out q (that "Jane likes Tom"), but q requires p (that "Tom likes Jane").

Consider another example.

> A grade will be given if and only if the test is taken. The test has been taken. Was a grade given?

Anything of the form "a iff b" is formalized as $a \equiv b$. The problem now becomes easy to analyze.

> John is going to the party if Mary goes. Mary is not going. Therefore, John isn't going either.

How do we formalize "a if b"? Simple: $b \Rightarrow a$. Finish the analysis.

> John is going to the party only if Mary goes. Mary is not going. Therefore, John isn't going either.

How do we formalize "only if"? A simple way to remember this is that "if" is one direction of "if and only if" and "only if" is the other direction. Thus, "a only if b" is formalized as $a \Rightarrow b$.

Try this one.

> John is going to the party only if Mary goes. Mary is going. Therefore, John is going too.

One more.

> Paul is not going to sleep unless he finishes the carrot hunt on Final Fantasy XII. Paul went to sleep. Therefore, he finished the carrot hunt on Final Fantasy XII.

How do we formalize "a unless b"? It is $\neg b \Rightarrow a$. Why? Because "a unless b" says that a has to be *true*, except when (unless) b is *true*, so when b is *true*, a can be anything. The only assignment that violates "a unless b" is when a is *false* and b is *false*. So, notice that "a unless b" is equivalent to "a or b".

One more example of unless.

> You will not get into NEU unless you apply.

is the same as

> You will not get into NEU if you do not apply.

which is the same as

> You will not get into NEU or you will apply.

So, the hard part here is formalizing the problem. After that, even ACL2s can figure out if the argument is valid.

3.9 The Declarative Approach to Design

Most of the design paradigms you have seen so far require you to describe how to solve problems algorithmically. This is true for functional, applicative, and object-oriented programming paradigms. A rather radically different approach to design is to use the declarative paradigm. The idea is to specify *what* we want, not *how* to achieve it. A satisfiability solver is then used to find a solution to the constraints.

3.9.1 Avionics Example

Let us consider an example from the avionics domain.

We have a set of *cabinets* $C = \{C_1, C_2, \ldots, C_{20}\}$. Cabinets are physical locations on an airplane that provide network access, battery power, memory, CPUs, and other resources.

We also have a set of *avionics applications* $A = \{A_1, A_2, \ldots, A_{500}\}$. The avionics applications are software programs and include applications such as navigation, control, collision detection, and collision avoidance.

Our job is to map each application to one cabinet subject to a large number of constraints.

Instead of us figuring out how to achieve this mapping, by using the declarative approach we will instead just specify the constraints we have on the mapping and we will let the declarative system we are using figure out a solution for us. This allows us to operate at a much higher level of abstraction than is the case with functional, imperative, or object-oriented approaches.

To make the idea concrete, we consider one simple example of a constraint: applications A_1, A_2, and A_3 have to be separated. What this means is that no pair of them can reside on the same cabinet. Here is how we might express this constraint in the declarative language CoBaSA. Assume that we have defined `A`, the array of 500 applications and `C`, the array of 20 cabinets.

```
Map AC A C
For_all cab in C {AC(1,cab) implies ((not AC(2,cab)) and (not AC(3,cab)))}
For_all cab in C {AC(2,cab) implies (not AC(3,cab))}
```

The first line tells us that `AC` is a *map*, a function from `A` to `C`. When we define a map, we get access to *indicator variables*. Such variables are Boolean variables of the form `AC(app,cab)`, where `AC(app,cab)` = *true* iff `AC(app)` = `cab`, *i.e.*, map `AC` applied to application `app` returns `cab`.

3.9.2 Solving Declarative Constraints

In this section, we will get a glimpse into the process by which the three lines of CoBaSA constraints above get turned into a formula in propositional logic that is then given to a SAT solver.

We start by writing the constraints above using standard mathematical notation, starting with the second constraint.

$$\langle \forall c \in C :: AC_1^c \implies \neg AC_2^c \wedge \neg AC_3^c \rangle$$

The $\forall$ symbol is a *universal quantifier*. It states that for all cabinets ($c \in C$) if application A_1 gets mapped to the cabinet (the indicator variable AC_1^c) then neither application A_2 nor application A_3 get mapped to the same cabinet. We can actually rewrite this using propositional logic as follows:

$$\bigwedge_{c \in C} AC_1^c \implies \neg AC_2^c \wedge \neg AC_3^c$$

So, universal quantification can be thought of as conjunction. Now, if we expand this out, we wind up with the following:

$$\begin{gathered}
(AC_1^1 \implies \neg AC_2^1 \wedge \neg AC_3^1) \wedge \\
(AC_1^2 \implies \neg AC_2^2 \wedge \neg AC_3^2) \wedge \\
\dots \\
(AC_1^{20} \implies \neg AC_2^{20} \wedge \neg AC_3^{20})
\end{gathered}$$

So, we are almost at the point where we can give this to a SAT solver. One issue, however, is that most SAT solvers require their input to be in CNF (Conjunctive Normal Form). What we have is a conjunction, but the conjuncts are not clauses. Here is how to turn them into clauses. We show how to do this for the first conjunct only, since the rest follow the same pattern. The first conjunct above gets turned into the following two clauses.

$$\neg AC_1^1 \vee \neg AC_2^1$$

$$\neg AC_1^1 \vee \neg AC_3^1$$

Notice that these two clauses are semantically equivalent to the conjunct they correspond to.

There is one more issue to deal with before we can use a SAT solver: SAT solvers tend to require DIMACS file format. In the DIMACS format, variables are represented by positive

integers, negated variables by negative integers, and each clause is a list of integers that ends with a 0 and a newline. So, the first thing to do is to come up with a mapping from indicator variables into the positive integers so that no two indicator variables get mapped to the same number. Here is one way of doing that.

$$var(AC_a^c) = 20(a-1) + c$$

With this mapping, the translation of the 2nd CoBaSA constraint to DIMACS format gives us the following 40 disjuncts:

```
-1 -21 0
-1 -41 0
-2 -22 0
-2 -42 0
...
-20 -40 0
-20 -60 0
```

The third CoBaSA constraint gets translated in a similar way. What about the first constraint?

Well, here is one way of thinking of the `map` constraint using logic.

$$\langle \forall a \in A :: \langle \exists c \in C :: AC_a^c \rangle \rangle$$

This says that for every application a, there exists some cabinet c (this is the meaning of the existential quantifier $\exists$), such that the indicator variable AC_a^c holds (is *true*). Notice that we could have said that there exists a *unique* cabinet c such that the indicator variable AC_a^c holds. This would have been a more faithful translation, but it turns out that if more than one indicator variable is *true*, then all that means is that we have a choice as to where to place a, so we prefer to have fewer constraints and not insist on uniqueness. Now, just as universal quantification can be thought of as conjunction, existential quantification can be thought of a disjunction, so we can rewrite the above constraint using only propositional operators as:

$$\bigwedge_{a \in A} \bigvee_{c \in C} AC_a^c$$

We proceed as previously by expanding this out with the goal of generating a CNF formula.

$$AC_1^1 \vee AC_1^2 \vee AC_1^3 \cdots \vee AC_1^{20}$$

$$AC_2^1 \vee AC_2^2 \vee AC_2^3 \cdots \vee AC_2^{20}$$

$$...$$

$$AC_{500}^1 \vee AC_{500}^2 \vee AC_{500}^3 \cdots \vee AC_{500}^{20}$$

Finally, we apply *var* to transform the indicator variables into numbers in order to obtain the DIMACS version of the above formula.

```
1 2 3 ... 20 0
21 22 23 ... 40 0
...
9980 9981 9982 ... 10000 0
```

Part III

Equational Reasoning

4

Equational Reasoning

We just finished studying propositional logic, so let's start by considering the following question:

> Why do we need more than propositional logic?

After all, we were able to do a lot with propositional logic, including declarative design, security and digital logic.

What we are really after, however, is reasoning about programs, and while propositional logic will play an important role, we need more powerful logics.

To see why, let's simplify things for a moment and consider conjectures involving numbers and arithmetic operations.

Consider the conjecture:

Conjecture 1 $a + b = ba$

What does it mean for this conjecture to be true or false?

Well, there is a source of ambiguity here. If a, b are constants (like 1, 2, etc.) then we can just evaluate the equality and determine if it is true or not.

However, a and b are variables. This is similar to the propositional formulas we saw, *e.g.*,

$$p \wedge q \;\Rightarrow\; p \vee q$$

Recall that p and q are atoms, and the above formula is valid. What that means is that no matter what value p and q have, the above formula is true. Another way of saying this is that the above formula evaluates to true under all assignments.

In Conjecture 1, a and b range over a different domain than the Booleans, let's say they range over the rationals.

So, what we really mean when we say that conjecture 1 is valid (or true) is that for any rational number a and any rational number b, $a + b = ba$. Another way of saying this is that the above formula evaluates to true under all assignments. Notice the similarity with the Boolean case.

Is Conjecture 1 a valid formula?

No. We can come up with a counterexample.

Is Conjecture 1 unsatisfiable?

No. It is both satisfiable and falsifiable. Again, this is exactly the kind of characterization we used to classify Boolean formulas. How many assignments falsify the conjecture? How many assignments satisfy the conjecture?

What about the following conjecture?

Conjecture 2 $a + b = b + a$

Can we come up with a counterexample?
No.
Is the formula valid?
Yes.

How do we prove that a conjecture is valid in the case of propositional logic? We use a truth table, with a row per possible assignment, to show that no counterexample exists. A counterexample is an assignment that evaluates to false.

Can we do something similar here?

Yes, but the number of assignments is unfortunately infinite. That means, we can never completely fill in a table of assignments.

We need a radically new idea here.

We want something that allows us to do a finite amount of work and from that to deduce that there are no counterexamples in the infinite table, were we even able to construct it.

Let's look at how we might do this, but in the context of programs. First we start with the definition of `alen`, a length function whose domain is `All`.

```
(defunc alen (l)
  :input-contract t
  :output-contract (natp (alen l))
  (if (atom l)
      0
    (+ 1 (alen (rest l)))))
```

Consider the following conjecture:

Conjecture 3 `(equal (alen (list x)) (alen x))`

Is this conjecture true (valid) or false (falsifiable)?

What does it mean for Conjecture 3 to be true? That no matter what object of the ACL2s universe `x` is, the above equality holds.

Conjecture 3 is false. Why?

Suppose that `x` = `1`, then the conjecture evaluates to `nil`, *i.e.*,

$$[\![\texttt{(equal (alen (list 1)) (alen 1))}]\!] = [\![\texttt{(equal 1 0)}]\!] = \texttt{nil}$$

So, finding a counterexample is "easy." All we have to do is to find an assignment under which the conjecture evaluates to `nil`. This is just like the Boolean logic case.

Is Conjecture 3 unsatisfiable? No. Again, all we have to do is find one satisfying assignment, *e.g.*, `x` = `(1)`. How many assignments falsify the conjecture? How many assignments satisfy the conjecture?

What about:

Conjecture 4
```
(equal (alen (cons x (list z)))
       (alen (cons y (list z))))
```

Here we can't find a counterexample.

How can we go about proving that Conjecture 4 is valid?

```
    (alen (cons x (list z)))
= { Definition of alen, instantiation }
    (if (atom (cons x (list z))) 0 (+ 1 (alen (rest (cons x (list z))))))
= { first-rest axioms }
    (if (atom (cons x (list z))) 0 (+ 1 (alen (list z))))
= { Definition of atom }
    (if (not (consp (cons x (list z)))) 0 (+ 1 (alen (list z))))
= { Definition of not }
    (if (if (consp (cons x (list z))) nil t) 0 (+ 1 (alen (list z))))
= { consp axioms }
    (if (if t nil t) 0 (+ 1 (alen (list z))))
= { if axioms }
    (if nil 0 (+ 1 (alen (list z))))
= { if axioms }
    (+ 1 (alen (list z)))
```

What we have shown so far is:

$$\texttt{(alen (cons x (list z)))} = \texttt{(+ 1 (alen (list z)))} \tag{4.1}$$

which we will feel free to write as

$$\texttt{(alen (cons x (list z)))} = 1 + \texttt{(alen (list z))} \tag{4.2}$$

because it should be clear how to go from (4.1) to (4.2) and because we have been trained to use infix for arithmetic operators since elementary school.

You should be able to continue the proof to show that

$$\texttt{(alen (cons x (list z)))} = 2 \tag{4.3}$$

Finish the proof.

Once the proof is done, we have shown that (4.3) is valid. Any validity that we establish via proof is called a *theorem*, so (4.3) is a theorem. To reason about built-in functions such as `consp`, `if`, and `equal` we use *axioms*, which you can think of as built-in theorems providing the semantics of the built-in functions. Every time we define a function that ACL2s admits, we also get a *definitional axiom*, which, for now, you can think of as an axiom stating that the function is equal to its body (but more on this soon). We can then reason from these basic axioms (which are also theorems) using what is called *first order logic.* First order logic includes propositional reasoning, but extends it significantly. We will introduce as much of first order reasoning as needed.

Back to our proof. We are not done with the proof of Conjecture 4, but there are at least two reasonable ways to proceed.

First, we might say:

If we simplify the RHS (Right Hand Side), we get

```
  (alen (cons y (list z)))
```

= { Definition of `len`, instantiation }

...

= { `if` axioms }

`(+ 1 (alen (list z)))`

= { Definition of `len`, instantiation }

...

= { Arithmetic }

2

So, the LHS (Left Hand Side) and RHS are equal.

What we realized is that the same steps that we used to simplify the LHS can be used in a symmetric way to simplify the RHS. In this class we will avoid proofs involving "...". Here's a better way to make the argument:

First, note that (4.3) is a theorem. By instantiating (4.3) with the substitution `((x y))`, we get:

`(alen (cons y (list z)))` = 2 (4.4)

Putting (4.3) and (4.4) together, we have

`(alen (cons x (list z)))` = `(alen (cons y (list z)))`

So, Conjecture 4 is a theorem.

We already saw that instantiation can be used in propositional logic. Its use is indispensable when reasoning about ACL2s programs!

This example highlights the new tool we have that allows us to reason about programs: proof. The game we will be playing is to construct proofs of conjectures involving some of the basic functions we have already defined (*e.g.*, `len`, `app`, and `rev`). We will focus on these simple functions because their simplicity allows us to focus exclusively on how to prove theorems without the added complexity of having to understand what conjectures mean.

Once we prove that a conjecture is valid, we say that the conjecture is a *theorem*. We are then free to use that theorem in proving other theorems. This is similar to what happens when we program: we define functions and then we use them to define other functions (*e.g.*, we define `rev` using `app`).

What's new here?

Well, we are beyond the realm of the propositional. We have variables ranging over the ACL2s universe, equality, and functions.

Let's look at equality. To simplify notation, we tend to write expressions involving `equal` using = instead. This is similar to what we did with arithmetic. For example, instead of writing the more technically correct

`(equal (alen (cons x z)) (alen (cons y z)))`

we usually write the more familiar

`(alen (cons x z))` = `(alen (cons y z))`

We feel free to go back and forth without justification.

If we want to be pedantic, here is how `equal` and = are related.

- $x = y \;\Rightarrow\;$ `(equal x y)` $=$ `t`
- $x \neq y \;\Rightarrow\;$ `(equal x y)` $=$ `nil`

When we use $=$ or $\neq$ in expressions, they bind more tightly than any of the propositional operators.

How can we reason about equality? We will use just two properties of equality. First, equality is what is called an *equivalence relation*, *i.e.*, it satisfies the following properties.

- *Reflexivity*: $x = x$
- *Symmetry*: $x = y \;\Rightarrow\; y = x$
- *Transitivity*: $x = y \wedge y = z \;\Rightarrow\; x = z$

That $=$ is an equivalence relation is what allows us to chain together the sequence of equalities in the proof of Conjecture 4 above to conclude that `(alen (cons x (list z)))` $= 2$.

The second property of equality we will use is the *Equality Axiom Schema for Functions*: For every function symbol `f` of arity n we have the axiom

$$(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \;\Rightarrow\; (\texttt{f}\; x_1 \cdots x_n) = (\texttt{f}\; y_1 \cdots y_n)$$

To reason about constants, we can use evaluation, *e.g.*, all of the following are theorems.

$$\texttt{t} \neq \texttt{nil}$$

$$\texttt{()} = \texttt{nil}$$

$$1 \neq 2$$

$$2/4 = 4/8$$

$$\texttt{(cons 1 ())} = \texttt{(list 1)}$$

To reason about built-in functions, such as `cons`, `first`, and `rest`, we have axioms for each of these functions that are derived from their semantics.

$$\texttt{(first (cons x y))} = \texttt{x}$$

$$\texttt{(rest (cons x y))} = \texttt{y}$$

$$\texttt{(consp (cons x y))} = \texttt{t}$$

$$x = \texttt{nil} \Rightarrow \texttt{(if x y z)} = \texttt{z}$$

$$x \neq \texttt{nil} \Rightarrow \texttt{(if x y z)} = \texttt{y}$$

What about instantiation? It is a rule of inference:

Instantiation: Derive $\varphi|_\sigma$ from φ. That is, if φ is a theorem and σ is a substitution, then by instantiation, $\varphi|_\sigma$ is a theorem.

For example, since this is a theorem `(equal (first (cons x y)) x)` we can derive `(equal (first (cons (foo x) (bar z))) (foo x))`.

More carefully, a substitution is just a list of the form:

$$\texttt{((}var_1\ term_1\texttt{)}\ \ldots\ \texttt{(}var_n\ term_n\texttt{))}$$

where the var_i are *target variables* and the $term_i$ are their *images*. We require that the var_i are distinct. The application of this substitution to a formula uniformly replaces every free occurrence of a target variable by its image.

What does it mean to say that the following is a theorem?

`(alen (cons x (list z)))` = `(alen (cons y (list z)))`

That no matter what you replace `x`, `y`, and `z` with from the ACL2s universe, the LHS and RHS evaluate to the same thing.

Let's try to prove another conjecture.

Conjecture 5 `(app (cons x y) z)` = `(cons x (app y z))`

```
  (app (cons x y) z)
= { Definition of app, instantiation }
  (if (endp (cons x y)) z (cons (first (cons x y)) (app (rest (cons x y)) z)))
= { Definition of endp, axioms for consp }
  (if nil z (cons (first (cons x y)) (app (rest (cons x y)) z)))
= { Axioms for if }
  (cons (first (cons x y)) (app (rest (cons x y)) z))
= { Axioms for first, rest }
  (cons x (app y z))
```

Unfortunately, the above "proof" has a problem. Unlike `alen`, which is defined for the whole ACL2s universe, `app` is only defined for lists.

Recall the definitions:

```
(defunc listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
  (if (consp l)
      (listp (rest l))
    (equal l ())))

(defunc endp (l)
  :input-contract (listp l)
  :output-contract (booleanp  (endp l))
  (not (consp l)))

(defunc app (a b)
  :input-contract (and (listp a) (listp b))
  :output-contract (listp (app a b))
  (if (endp a)
      b
    (cons (first a) (app (rest a) b))))
```

The definition of functions such as `app` give rise to *definitional axioms*. Here is the definitional axiom that `app` gives rise to:

`(listp a)` $\wedge$ `(listp b)`

```
⇒
(app a b)
=
(if (endp a)
    b
  (cons (first a) (app (rest a) b)))
```

In general, every time we successfully admit a function, we get two theorems of the form

$$ic \;\Rightarrow\; \texttt{(f } x_1 ... x_n\texttt{)} = body$$

$$ic \;\Rightarrow\; oc$$

where *ic* is the input contract for `f`, and where *oc* is the output contract for `f`. We will be very precise about what "successfully admit" means, but, for now, take this to mean that ACL2s accepts your function definition. Recall that this involves proving termination, proving the function contracts, and proving the body contracts.

So, we can't expand the definition of `app` in the proof of Conjecture 5, unless we know:

`(listp (cons x y))` ∧ `(listp z)`

which is equivalent to:

`(listp y)` ∧ `(listp z)`

So, what we really proved was:

Theorem 4.1 `(listp y)` ∧ `(listp z)` ⇒ `(app (cons x y) z)` = `(cons x (app y z))`

When we write out proofs, we will not explicitly mention input contracts when using a function definition because the understanding is that every time we use a definitional axiom to expand a function, we have to check that we satisfy the input contract, so we don't need to remind the reader of our proof that we did something we all understand always needs doing.

It is often the case that when we think about conjectures that we expect to be valid, we often forget to carefully specify the hypotheses under which they are valid. These hypotheses depend on the input contracts of the functions mentioned in the conjectures, so get into the habit of looking at conjectures and making sure that they have the needed hypotheses. *Contract checking* is the process of checking that a conjecture has all the hypotheses required by the contracts of the functions appearing in the conjecture. *Contract completion* is the process of adding the missing hypotheses (if any) identified during contract checking. Contract checking and completion is similar to what you do when you write functions: you check the body contracts of the functions you define and if you are calling the functions on arguments of the wrong type, then you modify your code appropriately. In the case of function definitions, as we have seen, it is often the case that if the function definition is wrong, there is also a contract violation. Similarly, if a conjecture is not valid, it is often the case that there is a contract violation.

Let's look at another example:

Conjecture 6 `(endp x)` ⇒ `(app (app x y) z)` = `(app x (app y z))`

Can I prove this? Check the contracts of the conjecture.

Contract checking and completion gives rise to:

Conjecture 7 `(listp x)` $\wedge$ `(listp y)` $\wedge$ `(listp z)` $\wedge$ `(endp x)` $\Rightarrow$ `(app (app x y) z)` = `(app x (app y z))`

By the way, notice all of the hypotheses. Notice the Boolean structure. This is why we studied Boolean logic first! Almost everything we will prove will include an implication.

Notice that in ACL2s, we would technically write:

```
(implies (and (listp x)
              (listp y)
              (listp z)
              (endp x))
         (equal (app (app x y) z)
                (app x (app y z))))
```

The first thing to do when proving theorems is to take the Boolean structure into account by writing the conjecture in the form:

$$hyp_1 \wedge hyp_2 \wedge \cdots \wedge hyp_n \;\Rightarrow\; conc$$

where we have as many *hyp*s as possible. We will call the set of top-level hypotheses (*i.e.*, $\{hyp_1, hyp_2, \ldots, hyp_n\}$) our *context*.

Our context for Conjecture 7 is:

C1. `(listp x)`

C2. `(listp y)`

C3. `(listp z)`

C4. `(endp x)`

We then look at our context and see what obvious things our context implies. The obvious thing here is that C1 and C4 imply that `x` must be `nil`, so we add to our context the following:

C5. `x` = `nil` { C1, C4 }

Notice that any new facts we add must come with a justification. We will use the convention that all elements of our context will be given a label of the form Ci, where i is a positive integer.

The next thing we do is to start with the LHS of the conclusion and to try and reduce it to the RHS, using our proof format. If we need to refer to the context in one of proof step justifications, say Context 5, we write C5.

```
  (app (app x y) z)
= {  Def of app, C5, Def of endp, if axioms  }
  (app y z)
= {  Def of app, C5, Def of endp, if axioms  }
  (app x (app y z))
```

Notice that we took bigger steps than before. Before we might have written:

```
    (app (app x y) z)
= { Def of app }
    (app (if (endp x) y (cons (first x) (app (rest x) y))) z)
= { C5 }
    (app (if (endp nil) y (cons (first nil) (app (rest nil) y))) z)
= { Def of endp }
    (app (if t y (cons (first nil) (app (rest nil) y))) z)
= { If axioms }
    (app y z)
...
```

So, the above four steps were compressed into one step. Why? Because many of the steps we take involve expanding the definition of a function. Function definitions tend to have a top-level `if` or `cond` and as a general rule we will not expand the definition of such a function unless we can determine which case of the top-level `if`-structure will be true. If we just blindly expand function definitions, we'll wind up with a sequence of increasingly complicated terms that don't get us anywhere. So, if we know which case of the top-level `if` is true, then why go to the trouble of writing out the whole body of the function? Why not just write out that one case? Well, that's why we allow ourselves to expand definitions as in the first proof of Conjecture 7.

One other comment about the first proof of Conjecture 7. Students often have no difficulty with the first step, but have difficulty with the second step. The second step requires one to see that the simple term:

```
(app y z)
```

can be transformed into the RHS

```
(app x (app y z))
```

This may seem like a strange thing to do because students are used to thinking about computation as unfolding over time. So, if `x` is `nil` then of course the following holds.

```
    (app (app x y) z)
= { Def of app, ... }
    (app y z)
```

Because when we compute `(app x y)` we get `y`.

What students initially have difficulty with is seeing that you can reverse the flow of time and everything still works. For example the following is true,

```
    (app y z)
= { Def of app, ... }
    (app (app x y) z)
```

Because starting with `(app y z)` we can run time in reverse to get `(app x (app y z))` (recall `x` is `nil`). In fact, this is "obvious" from the equality (=) axioms that tell us that equality is an equivalence relation (reflexive, symmetric, and transitive). The symmetry

axiom tells us that we can view computation as moving forward in time or backward. It just doesn't make a difference.

As an aside, it turns out that in physics, we can't reverse time and so this symmetry we have with computation is not a symmetry we have in our universe. One reason why we can't reverse time in physics is that the second law of thermodynamics precludes it. The second law of thermodynamics implies that entropy increases over time. There is an even more fundamental reason why time is not reversible. This second reason has to do with the fundamental laws of physics at the quantum level, whereas the second law of thermodynamics is thought to be a result of the initial conditions of our universe. The second reason is that in our current understanding of the universe, there are very small violations of time reversibility exhibited by subatomic particles. The extent of the violations is not fully understood and probably has something to do with the imbalance of matter and antimatter in the visible universe. There is almost no antimatter in the visible universe and one of the big open problems in physics is trying to understand why that is the case.

4.1 Testing Conjectures

Recall that since Conjecture 7 is a theorem, whatever we replace the free variables with, the conjecture will evaluate to `t`. A convenient way of checking the conjecture using ACL2s is to use `let`, as follows:

```
(let ((x nil)
      (y nil)
      (z nil))
  (implies (and (endp x)
                (listp x)
                (listp y)
                (listp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

An even more convenient method is to use ACL2s to test the conjecture. Here is how:

```
(test?
  (implies (and (endp x)
                (listp x)
                (listp y)
                (listp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

There are three possible outcomes.

1. ACL2s proves that the conjecture is a theorem.
2. ACL2s finds a counterexample, *i.e.*, the conjecture is falsifiable.
3. None of the above hold, *i.e.*, the conjecture satisfies all of the tests ACL2s tries.

Consider another example. Consider the claim that `app2`, below, is equivalent to `app`.

```
(defunc app2 (x y)
  :input-contract (and (listp x) (listp y))
  :output-contract (listp (app2 x y))
  (if (endp y)
      x
    (cons (first y) (app2 x (rest y)))))
```

The claim is false, but `app2` works fine on many tests, *e.g.*,

```
(check= (app2 '(1 2) '(1 2)) '(1 2 1 2))
(check= (app2 nil nil) nil)
(check= (app2 nil '(1 2 3)) '(1 2 3))
(check= (app2 '(1 2 3) nil) '(1 2 3))
```

Here is how we can use ACL2s to test the conjecture that `app2` is equivalent to our definition.

```
(test?
  (implies (and (listp x) (listp y))
           (equal (app2 x y)
                  (app x y))))
```

ACL2s gives us counterexamples. It also shows us cases in which the conjecture is true.

Now, suppose that the specification for `app2` only stated that `(app2 x y)` must return a list that contains all of the elements in `x` and all of the elements in `y`, where order doesn't matter, but repetitions do. The definition of `app2` above satisfies the specification. In addition, there are different functions that satisfy the specification. How can we write tests that are independent of the implementation? We cannot write simple `check=`'s because there are exponentially many correct answers that `app2` could return. We can't test that `app2` is equal to our solution for the same reason. But, we can write conjectures that capture the specification and ACL2s can be used to test these conjectures.

Here is one way of doing this. We test that every element in `(app2 x y)` is also an element of `(app x y)` and conversely.

```
; check that if a is in app2, it is in app
(test?
  (implies (and (listp x)
                (listp y)
                (in a (app2 x y)))
           (in a (app x y))))

; check that if a is in app, it is in app2
(test?
  (implies (and (listp x)
                (listp y)
                (in a (app x y)))
           (in a (app2 x y))))
```

Exercise 4.1 *Unfortunately, we can define* `app2` *in a way that does not satisfy the specification, but does satisfy the above* `test?`*'s. Exhibit such a definition and check that it passes the above tests. A better solution is to test that* `app2` *is a permutation of* `app`*. Define a function that checks if its arguments are permutations of one another and use this to test both your faulty definition of* `app2` *and the definition given above.*

You can control how much testing ACL2s does. The default number of tests depends on the mode, but you can set it to whatever number you want, *e.g.*, here is how to instruct ACL2s to run 1,000 tests.

```
(acl2s-defaults :set num-trials 1000)
```

To summarize, ACL2s provides `test?`, a powerful facility for automatically testing programs. Instead of having to manually write tests, ACL2s generates as many tests as requested automatically. The other major advantage is that we do not have to specify exactly what functions have to do. In the `app2` example above, we did not have to say what `app2` returns; instead, we specified the properties we expect `app2` to satisfy. The advantage is that we *decouple* the testing of `app2` from the development of `app2`. In fact, even if we change the implementation of `app2`, the tests can remain the same.

4.2 Equational Reasoning with Complex Propositional Structure

Many of the conjectures we will examine have rich propositional structure. We now examine how to reason about such conjectures.

Conjecture 8

```
(consp x)
⇒
 [[(listp (rest x)) ∧ (listp y) ∧ (listp z)
   ⇒
   (app (app (rest x) y) z) = (app (rest x) (app y z))]
  ⇒
  [(listp x) ∧ (listp y) ∧ (listp z)
   ⇒
   (app (app x y) z) = (app x (app y z))]]
```

The above conjecture has the form

$$A \Rightarrow [B \Rightarrow C]$$

where

A is `(consp x)`

B is `[(listp (rest x)) ∧ (listp y) ∧ (listp z)`
`⇒ (app (app (rest x) y) z) = (app (rest x) (app y z))]`

C is `[(listp x) ∧ (listp y) ∧ (listp z)`
`⇒ (app (app x y) z) = (app x (app y z))]`

What we are doing here is identifying some of the propositional structure of Conjecture 8. Here's why. It turns out that

$$A \Rightarrow [B \Rightarrow C] \equiv [A \wedge B] \Rightarrow C \tag{4.5}$$

This propositional equality is one we will use over and over. We will use it to rewrite Conjecture 8 so that the context has as many conjunctions as possible. After applying (4.5) to Conjecture 8, we get:

```
[(consp x) ∧
 [(listp (rest x)) ∧ (listp y) ∧ (listp z)
  ⇒
  (app (app (rest x) y) z) = (app (rest x) (app y z))]
⇒
[(listp x) ∧ (listp y) ∧ (listp z)
   ⇒
   (app (app x y) z) = (app x (app y z))]]
```

Applying (4.5) again and rearranging conjuncts gives us:

Conjecture 9

```
[(consp x) ∧
 (listp x) ∧
 (listp y) ∧
 (listp z) ∧
 [(listp (rest x)) ∧ (listp y) ∧ (listp z)
  ⇒
  (app (app (rest x) y) z) = (app (rest x) (app y z))]]
⇒
  (app (app x y) z) = (app x (app y z))]]
```

Now, we can extract the context. Doing so gives us:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp y)`

C4. `(listp z)`

C5. `[(listp (rest x)) ∧ (listp y) ∧ (listp z)] ⇒`
`[(app (app (rest x) y) z) = (app (rest x) (app y z))]`

Notice that we *cannot* use (4.5) on C5 to add the hypotheses of C5 to our context. Why?

We will be confronted with implications in our context (like C5) over and over. Usually what we will need is the consequent of the implication, but we can only use the consequent if we can also establish the antecedent, so we will try to do that. Here's how:

C6. `(listp (rest x))` { C1, C2, Def `listp` }

C7. `(app (app (rest x) y) z)` = `(app (rest x) (app y z))` $\{C6, C3, C4, C5, MP\}$

So, notice what we did. First we added C6 to our context. How did we get C6? Well, we know `(listp x)` (C2) and `(consp x)` (C1) so if we use the definitional axiom of `listp`, we get C6: `(listp (rest x))`.

Now, we have extended our context to include the antecedent of C5, so by propositional logic (Modus Ponens, abbreviated MP), we get that the conclusion also holds, *i.e.*, C7.

Recall that Modus Ponens tells us that if the following two formulas hold

$$A \Rightarrow B$$

$$A$$

Then so does the formula

$$B$$

We are now ready to prove the theorem. We start with the LHS of the equality in the conclusion of Conjecture 9.

```
   (app (app x y) z)
= { Def of app, C1, C2, C3 }
   (app (cons (first x) (app (rest x) y)) z)
= { Theorem 4.1 }
   (cons (first x) (app (app (rest x) y) z))
= { C7 }
   (cons (first x) (app (rest x) (app y z)))
= { Def of app, C1, C2, C3, C4 }
   (app x (app y z))
```

4.3 The difference between theorems and context

It is very important to understand the difference between a formula that is a theorem and one that appears in a context. A formula that appears in a context cannot be instantiated. It can only be used as is, in the proof attempt for the conjecture from which it was extracted. This is a major difference. Our contexts will never include theorems we already know. Theorems we already know are independent of any conjecture we are trying to prove and therefore do not belong in a context. A context is always formula specific.

Here is an example that shows why instantiation of context formulas leads to unsoundness. Here is a "proof" of

$$\mathtt{x} = 1 \Rightarrow 0 = 1 \tag{4.6}$$

Context:

C1. `x` = 1

Proof 0
= { Instantiate C1 with `((x 0))` }
1

So, now we have a "proof" of (4.6), but using (4.6) we can get:

$$\texttt{nil} \tag{4.7}$$

How?
Instantiate (4.6) with `((x 1))`, use Propositional logic, and Arithmetic.
Now we have a proof for any conjecture we want, *e.g.*,

$$\varphi \tag{4.8}$$

How?
Well, `nil` (false) implies anything, so this is a theorem

$$\texttt{nil} \ \Rightarrow \ \varphi$$

Now, φ follows using (4.7) and Modus Ponens.

The point is that a context is *completely* different from a theorem. The context of (4.6) does not tell us that for all `x`, `x` = 1. It just tells us that `x` = 1 in the context of conjecture (4.6). Contexts are just a mechanism for extracting propositional structure from a conjecture, which in turn allows us to focus on the important part of a proof and to minimize the writing we have to do.

4.4 How to prove theorems

When presented with a conjecture, make sure that you check contracts, as shown above.

If the contracts checking succeeds, make sure you understand what the conjecture is saying.

Once you do, see if you can find a counterexample.

If you can't find a counterexample, try to prove that the conjecture is a theorem.

One often iterates over the last two steps.

During the proof process, you have available to you all the theorems we have proved so far. This includes all of the axioms (`first-rest` axioms, `if` axioms, ...), all the definitional axioms (def of `app`, `len`, ...), all the contract theorems (contracts of `app`, `len`, ...). These theorems can be used at any time in any proof and can be instantiated using any substitution. They are a great weapon that will help you prove theorems, so make sure you understand the set of already proven theorems.

There are also local facts extracted from the conjecture under consideration. Recall that the first step is to try and rewrite the conjecture into the form:

$$[C_1 \wedge C_2 \wedge \ldots \wedge C_n] \ \Rightarrow \ \text{RHS}$$

where we try to make RHS as simple as possible. $C_1, \ldots, C_n$ are going to be the first n components of our context. Formulas in the context are specific to the conjecture under consideration. They are completely different from theorems (as per the above discussion).

A good amount of manipulation of the conjecture may be required to extract the maximal context, but it is well worth it.

The next step is to see what other facts the $C_1, \ldots, C_n$ imply. For example, if the current context is:

Context:

C1. `(endp x)`

C2. `(listp x)`

then we would add

C3. `x` = `nil` { C1, C2 }

This will happen a lot. Another case that will happen a lot is:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp (rest x))` $\Rightarrow \varphi$

then we would add

C4. `(listp (rest x))` { C1, C2, Def `listp` }

C5. φ { C4, C3, MP }

where MP is modus Ponens.

As was the case when we studied propositional logic, we have "word problems," something we now consider:

Conjecture 10 $x \leq xy$ *if* $y \geq 1$

Discussion: What does the above conjecture mean, anyway?

It means that for any values of x, and y, if $y \geq 1$ then $x \leq xy$.

Really? Any values? What if x and y are functions or strings or ...? Usually the domain is implicit, *i.e.*, "clear from context."

We will be using ACL2s, and we can't appeal to "context." This is a good thing!

Notice also that we can use ACL2s, a programming language, to make mathematical statements. Of course! Programming languages are mathematical objects and you reason about programs the way you reason about the natural numbers, the reals, sets, etc.: you prove theorems.

In ACL2s, we have to be precise about the conditions under which we expect the conjecture to hold. The conjecture can be formalized in ACL2s as follows:

```
(thm
 (implies (and (rationalp x)
               (rationalp y)
               (>= y 1))
          (<= x (* x y))))
```

In standard mathematical notation it is:

$$\langle \forall x, y \in Q :: y \geq 1 \;\Rightarrow\; x \leq xy \rangle$$

Is the above conjecture true?

Well, when given a conjecture, we can try one of two things:

1. Try to falsify it.
2. Try to prove it is correct.

How do we falsify a conjecture?

Simple exhibit a counterexample.

Remember that in the design recipe, we construct examples and tests. You should do the same thing with conjectures. That is, we can test that the conjecture is true on examples. Here are some:

1. $x = 0, y = 0$
2. $x = 12, y = 1/3$
3. $x = 9, y = 3/2$

Any others?

How do we test this in ACL2s? Put the conjecture in the body of a let.

```
(let ((x 0)
      (y 0))
  (implies (and (rationalp x)
                (rationalp y)
                (>= y 1))
           (<= x (* x y))))
```

We are using a programming language, so we can do better. We can write a program to test the conjecture on a large number of cases. How many cases are there? We can use a random number generator to "randomly" sample from the domain. We'll see how to do that in ACL2s.

```
(test?
  (implies (and (rationalp x)
                (rationalp y)
                (>= y 1))
           (<= x (* x y))))
```

If all of the tests pass, then we can try to prove that the conjecture is a theorem.

What would a "proof" of the above conjecture look like?

Most proofs are informal and it takes a long time for students to understand what constitutes an informal proof. This happens by osmosis over time.

In our case, we have a simple rule: it's a proof if ACL2s says it is.

```
(thm
 (implies (and (rationalp x)
               (rationalp y)
               (>= y 1))
```

```
        (<= x (* x y))))
```

Of course, this isn't a theorem.

Let's consider another example:

Conjecture 11 *$x(y+z) = xy + xz$*

How do we write this in ACL2s?

```
(thm (implies (and (rationalp x)
                   (rationalp y)
                   (rationalp z))
              (equal (* x (+ y z))
                     (+ (* x y) (* x z)))))
```

Is the above conjecture true?

Well, we can try to falsify it.

```
(let ((x 0)
      (y 0)
      (z 0))
  (equal (* x (+ y z))
         (+ (* x y) (* x z))))
```

We can try many examples. We can automatically generate random examples.

```
(test?
 (implies (and (rationalp x)
               (rationalp y)
               (rationalp z))
          (equal (* x (+ y z))
                 (+ (* x y) (* x z)))))
```

When do we give up falsifying this?

Can we just try all the possibilities? If we had infinite time. Do we? Maybe (ask a physicist), but, as a practical matter, we currently don't.

Maybe we should consider a proof. Can we prove the above?

One answer might be: "of course, multiplication distributes over addition."

In ACL2s, the conjecture turns out to be true

```
(thm (implies (and (rationalp x)
                   (rationalp y)
                   (rationalp z))
              (equal (* x (+ y z))
                     (+ (* x y) (* x z)))))
```

This is pretty amazing because a proof gives us a finite way of running an infinite number of examples. That's the power of logic and mathematics.

When ACL2s proves this theorem, is it thinking?

> The question of whether Machines Can Think ... is about as relevant as the question of whether Submarines Can Swim.
>
> Edsger W. Dijkstra: EWD898, 1984

See the EWD archives at the University of Texas at Austin.

Here is another example

```
(defunc app (a b)
  :input-contract (and (listp a) (listp b))
  :output-contract (listp (app a b))
  (if (endp a)
      b
    (cons (first a) (app (rest a) b))))

(defunc rev (x)
  :input-contract (listp x)
  :output-contract (listp (rev x))
  (if (endp x)
      nil
    (app (rev (rest x)) (list (first x)))))

(defunc in (a X)
  :input-contract (listp x)
  :output-contract (booleanp (in a X))
  (if (endp x)
      nil
    (or (equal a (first X))
        (in a (rest X)))))

(defunc del (a X)
  :input-contract (listp x)
  :output-contract (listp (del a X))
  (cond ((endp x) nil)
        ((equal a (first x)) (rest x))
        (t (cons (first x) (del a (rest x))))))
```

Conjecture 12

```
(listp x)
⇒
(in a x)   ⇒   (not (in a (del a x)))
```

Using induction (something we will describe later), the above conjecture leads to the following proof obligation:

```
(and (implies (endp x)
              (implies (listp x)
                       (implies (in a x)
                                (not (in a (del a x))))))
     (implies (and (consp x)
                   (equal a (first x)))
              (implies (listp x)
                       (implies (in a x)
```

```
                                (not (in a (del a x))))))
     (implies (and (consp x)
                   (not (equal a (first x)))
                   (implies (listp (rest x))
                            (implies (in a (rest x))
                                     (not (in a (del a (rest x)))))))
              (implies (listp x)
                       (implies (in a x)
                                (not (in a (del a x)))))))
```

Is this true? If so, give a proof. Is it false? If so, exhibit a counterexample.

Try this before reading further.

Conjecture 12 is false, *e.g.*, consider

```
(let ((x '(1 1))
      (a 1))
  ...)
```

where ... in the above let is Conjecture 12.

What about the following conjecture?

Conjecture 13

```
(listp x)
⇒
(in a x)   ⇒   (in a (app x y))
```

Which by induction leads to the following proof obligation:

```
(and (implies (endp x)
              (implies (listp x)
                       (implies (in a x)
                                (in a (app x y)))))
     (implies (and (consp x)
                   (equal a (first x)))
              (implies (listp x)
                       (implies (in a x)
                                (in a (app x y)))))
     (implies (and (consp x)
                   (not (equal a (first x)))
                   (implies (listp (rest x))
                            (implies (in a (rest x))
                                     (in a (app (rest x) y)))))
              (implies (listp x)
                       (implies (in a x)
                                (in a (app x y))))))
```

Is this true? If so, give a proof. Is it false? Is so, exhibit a counterexample.

Try this before reading further.

This conjecture fails contract checking. After contract completion, it is true and you should be able to prove it by breaking Conjecture 13 into three parts and proving each in turn.

In the cases we have seen so far, it was easy to decide if a conjecture was true or false, and with a good amount of testing, we would have identified the false conjectures.

Is this always the case?

No.

Anyone heard of Fermat's last theorem?

Conjecture 14 *For all positive integers x, y, z, and n, where $n > 2$, $x^n + y^n \neq z^n$.*

In 1637, Fermat wrote about the above:

> "I have a truly marvelous proof of this proposition which this margin is too narrow to contain."

This is called Fermat's Last Theorem. It took 357 years for a correct proof to be found (by Andrew Wiles in 1995).

We can use Fermat's last theorem to construct a conjecture that is hard to prove in ACL2s. After defining `expt`, a function that computes exponents, we define:

```
(defunc f (x y z n)
  :input-contract (and (posp x)
                       (posp y)
                       (posp z)
                       (natp n)
                       (> n 2))
  :output-contract (booleanp (f x y z n))
  (not (equal (+ (expt x n) (expt y n))
              (expt z n))))
```

```
(thm (f x y z n))
```

So, proving theorems may be hard.

Notice also that if the output contract was

```
  :output-contract (equal (f x y z n) t)
```

then ACL2s would have to prove a theorem that eluded mankind for centuries in order to even admit `f`!

But, it is easy to find a counterexample to a conjecture that is not a theorem, right?

That is not true either. There are many examples of conjectures that took a long time to resolve, and which turned out to be false.

4.5 Arithmetic

We can also reason about arithmetic functions. For example, consider the following conjecture

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

That is, summing up $0, 1, \ldots, n$ gives $\frac{n(n+1)}{2}$.

We can prove this using mathematical induction.

Here is how we do it in ACL2s. First, we have to define Σ.

```
(defunc sum (n)
  :input-contract (natp n)
  :output-contract (natp (sum n))
  (if (equal n 0)
      0
    (+ n (sum (- n 1)))))
```

We can prove that `(sum n)` $= \frac{\texttt{n(n+1)}}{2}$, which is formalized as:

```
(implies (natp n)
         (equal (sum n)
                (/ (* n (+ n 1)) 2)))
```

by mathematical induction. How?

First, we have the base case.

$$\texttt{(equal n 0)} \wedge \texttt{(natp n)} \;\Rightarrow\; \texttt{(sum n)} = \texttt{(/ (* n n+1) 2)} \tag{4.9}$$

Second, we have the induction step.

$$\texttt{n} > 0 \wedge \texttt{(natp n)} \wedge \texttt{(sum n-1)} = \texttt{(/ (* n-1 n) 2)} \;\Rightarrow\; \texttt{(sum n)} = \texttt{(/ (* n n+1) 2)} \tag{4.10}$$

Here is the proof, starting with (4.9).

Context.

C1. `(natp n)`

C2. `(equal n 0)`

Proof.

`(sum n)`

= { Def of `sum`, C2 }

`0`

= { Arithmetic, C2 }

`(/ (* n n+1) 2)`

Here is the proof of (4.10).

Context.

C1. `(natp n)`

C2. `n` $\neq 0$

C3. `(natp n-1)` $\Rightarrow$ `(sum n-1) = (/ (* n-1 n) 2)`

C4. `(natp n-1)` { C1, C2 }

C5. `(sum n-1) = (/ (* n-1 n) 2)` { C3, C4, MP }

Proof.

```
  (sum n)
= {  Def of sum, C2  }
   n + (sum (- n 1))
= {  C5  }
   n + (/ (* n-1 n) 2)
= {  Arithmetic  }
   (2n + n(n-1))/2
= {  Arithmetic  }
   (2n + n^2 -n)/2
= {  Arithmetic  }
   (n^2 +n)/2
= {  Arithmetic  }
   n(n+1)/2
```

Part IV

Definitions and Termination

5

Definitions and Termination

5.1 The Definitional Principle

We've already seen that when you define a function, say

```
(defunc f (x)
  :input-contract ic
  :output-contract oc
  body)
```

then ACL2s adds the definitional axiom

$$\texttt{ic} \;\Rightarrow\; \texttt{(f x)} = \texttt{body}$$

and the contract theorem

$$\texttt{ic} \;\Rightarrow\; \texttt{oc}$$

We now more carefully examine what happens when you define functions.

First, let's see why we have to examine anything at all.

In most languages, one is allowed to write functions such as the following:

```
(defunc f (x)
  :input-contract (natp x)
  :output-contract (natp (f x))
  (+ 1 (f x)))
```

This is a nonterminating recursive function.

Suppose we add the axiom

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{(f x)} = \texttt{(+ 1 (f x))} \tag{5.1}$$

Then, using the axiom, ACL2s can prove the contract theorem

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{(natp (f x))} \tag{5.2}$$

This is unfortunate because we now get a contradiction, *i.e.*, we can prove `nil` in ACL2s, all because we added the definitional axiom for `f` (5.1).

Here is how to derive a contradiction. First, notice that the following is an obvious arithmetic fact.

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{x} \neq \texttt{x} + 1 \tag{5.3}$$

ACL2s can prove this directly.

```
(thm (implies (natp x) (not (equal x (+ 1 x)))))
```

If we instantiate (5.3), we get

$$\texttt{(natp (f x))} \Rightarrow \texttt{(f x)} \neq \texttt{(+ 1 (f x))} \tag{5.4}$$

Together with (5.2), we have

$$\texttt{(natp x)} \Rightarrow \texttt{(f x)} \neq \texttt{(+ 1 (f x))} \tag{5.5}$$

Putting (5.1) and (5.5) gives us:

$$\texttt{(natp x)} \Rightarrow \texttt{nil} \tag{5.6}$$

But, now instantiating (5.6) with `((x 1))` gives us:

```
  t
= { (5.6) }
   (natp 1) ⇒ nil
= { Evaluation }
   nil
```

As we have seen, once we have `nil`, we can prove anything. Therefore, this nonterminating recursive equation introduced unsoundness. The point of the definitional principle in ACL2s is to make sure that new function definitions do not render the logic unsound. For this reason, ACL2s does not allow you to define nonterminating functions.

Almost all the programs you will write are expected to terminate: given some inputs, they compute and return an answer. Therefore, you might expect any reasonable language to detect non-terminating functions. However, no widely used language provides this capability, because checking termination is *undecidable*: no algorithm can always correctly determine whether a function definition will terminate on all inputs that satisfy the input contract.

We note that there are cases in which non-termination is desirable. In particular, *reactive systems*, which include operating systems and communication protocols, are intentionally non-terminating. For example, TCP (the Transmission Control Protocol) is used by applications to communicate on the Internet. TCP provides a communication service that is expected to always be available, so the protocol should *not* terminate. Does that mean that termination is not important for reactive systems? No, because reactive systems tend to have an outer, non-terminating, loop consisting of terminating actions. Can we reason about reactive systems in ACL2s? Yes, but how that is done will not be addressed in this chapter.

Question: does every non-terminating recursive equation introduce unsoundness?

Consider:

```
(defunc f (x)
  :input-contract t
  :output-contract t
  (f x))
```

This leads to the definitional axiom:

$$\texttt{(f x)} = \texttt{(f x)}$$

This cannot possibly lead to unsoundness since it follows from the reflexivity of equality.

Question: can terminating recursive equations introduce unsoundness?

Consider:

```
(defunc f (x)
  :input-contract t
  :output-contract t
  y)
```

This leads to the definitional axiom:

$$\texttt{(f x)} = \texttt{y} \tag{5.7}$$

Which causes problems, *e.g.*,

```
  t
= { Instantiation of (5.7) with ((y t) (x 0)) }
   (f 0)
= { Instantiation of (5.7) with ((y nil) (x 0)) }
   nil
```

We got into trouble because we allowed a "global" variable. It will turn out that we can rule out bad terminating equations with some simple checks.

So, modulo some checks we are going to get to soon, terminating recursive equations do not introduce unsoundness, because we can prove that if a recursive equation can be shown to terminate then there exists a function satisfying the equation.

The above discussion should convince you that we need a mechanism for making sure that when users add axioms to ACL2s by defining functions, then the logic stays sound.

That's what the *definitional principle* does.

Definitional Principle for ACL2s

The definition

```
(defunc f (x1 ... xn)
  :input-contract ic
  :output-contract oc
  body)
```

is *admissible* provided:

1. `f` is a new function symbol, *i.e.*, there are no other axioms about it. Functions are admitted in the context of a *history*, a record of all the built-in and defined functions in a session of ACL2s.

   Why do we need this condition? Well, what if we already defined `app`? Then we would have two definitions. What about redefining functions? That is not a good idea because we may already have theorems proven about `app`. We would then have to throw them out and any other theorems that depended on the definition of `app`. ACL2s allows you to undo, but not redefine.

2. The x_i are distinct variable symbols.

   Why do we need this condition? If the variables are the same, say `(defunc f (x x) ...)`, then what is the value of `x` when we expand `(f 1 2)`?

3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no variables freely other than the $\texttt{x}_i$;

   Why? Well, we already saw that global variables can lead to unsoundness. When we say that `body` is a term, we mean that it is a legal expression in the current history.

4. The function is terminating. As we saw, nontermination can lead to unsoundness.

There are also two other conditions that I state separately.

5. `ic` $\Rightarrow$ `oc` is a theorem.

6. The body contracts hold under the assumption that `ic` holds.

If admissible, the logical effect of the definition is to:

1. Add the *Definitional Axiom* for `f`: `ic` $\Rightarrow$ `[(f` $\texttt{x}_1$ ... $\texttt{x}_n$`)` $=$ `body]`.

2. Add the *Contract Theorem* for `f`: `ic` $\Rightarrow$ `oc`.

But, how do we prove termination?

A very simple first idea is to use what are called measure functions. These are functions from the parameters of the function at hand into the natural numbers, so that we can prove that on every recursive call the function terminates. Let's try this with `app`. What is a measure function for `app`?

How about the length of `x`? So, the measure function is `(len x)`.

Measure Function Definition: `m` is a measure function for `f` if all of the following hold.

1. `m` is an admissible function defined over the parameters of `f`;

2. `m` has the same input contract as `f`;

3. `m` has an output contract stating that it always returns a natural number; and

4. on every recursive call, `m` applied to the arguments to that recursive call decreases, under the conditions that led to the recursive call.

Here then is a measure function for `app`:

```
(defunc m (x y)
  :input-contract (and (listp x) (listp y))
  :output-contract (natp (m x y))
  (len x))
```

This is a non-recursive function, so it is easy to admit. Notice that we do not use the second parameter. That is fine and it just means that the second parameter is not needed for the termination argument.

Next, we have to prove that `m` decreases on all recursive calls of `app`, under the conditions that led to the recursive call. Since there is one recursive call, we have to show:

```
(implies (and (listp x)
              (listp y)
              (not (endp x)))
```

```
         (< (m (rest x) y) (m x y)))
```

which is equivalent to:

```
(implies (and (listp x)
              (listp y)
              (not (endp x)))
         (< (len (rest x)) (len x)))
```

which is a true statement.

More examples:

```
(defunc rev (x)
  :input-contract (listp x)
  :output-contract (listp (rev x))
  (if (endp x)
      nil
    (app (rev (rest x)) (list (first x)))))
```

Is this admissible? It depends if we defined `app` already. Suppose `app` is defined as above. What is a measure function?

`len`.

What about:

```
(defunc drop-last (x)
  :input-contract (listp x)
  :output-contract (listp (drop-last x))
  (if (equal (len x) 1)
      nil
    (cons (first x) (drop-last (rest x)))))
```

No. We cannot prove that it is non-terminating, *e.g.*, when `x` is `nil`, what is `(rest x)`? The real issue here is that we are analyzing a function that has body contract violations, *e.g.*, when `x` is `nil`, our function tries to evaluate `(first x)`. We can fix that in several ways. Here is one.

Exercise 5.1 *Define* `drop-last` *using the design recipe.*

```
(defunc drop-last (x)
  :input-contract (listp x)
  :output-contract (listp (drop-last x))
  (cond ((endp x) nil)
        ((endp (rest x)) nil)
        (t (cons (first x) (drop-last (rest x))))))
```

Exercise 5.2 *What is a measure function for* `drop-last`*?*

What about the following function?

```
(defunc prefixes (l)
  :input-contract (listp l)
  :output-contract (listp (prefixes l))
  (cond ((endp l) '( () ))
```

```
        (t (cons l (prefixes (drop-last l))))))
```

Is `prefixes` admissible?

Yes. It satisfies the conditions of the definitional principle; in particular, it terminates because we are removing the last element from `l`.

Exercise 5.3 *What is a measure function for* `prefixes`*?*

Does the following satisfy the definitional principle?

```
(defunc f (x)
  :input-contract (integerp x)
  :output-contract (integerp (f x))
  (if (equal x 0)
      0
    (+ 1 (f (- x 1)))))
```

No. It does not terminate.

What went wrong?

Maybe we got the input contract wrong. Maybe we really wanted natural numbers.

```
(defunc f (x)
  :input-contract (natp x)
  :output-contract (integerp (f x))
  (if (equal x 0)
      0
    (+ 1 (f (- x 1)))))
```

Another way of thinking about this is: What is the largest type that is a subtype of `integer` for which `f` terminates? Or, we could ask: What is the largest type for which `f` terminates?

But, maybe we got the input contract right. Then we used the wrong design recipe:

```
(defunc f (x)
  :input-contract (integerp x)
  :output-contract (integerp (f x))
  (cond ((equal x 0) 0)
        ((> x 0) (+ 1 (f (- x 1))))
        (t (+ 1 (f (+ x 1))))))
```

Now `f` computes the absolute value of `x` (in a very slow way).

The other thing that should jump out at you is that the output contract could be `(natp (f x))` for all versions of `f` above.

5.2 Admissibility of common recursion schemes

We examine several common recursion schemes and show that they lead to admissible function definitions.

The first recursion scheme involves recurring down a list.

```
(defunc f (x1 ... xn)
  :input-contract (and ... (listp xi) ...)
```

```
  :output-contract ...
  (if (endp xi)
      ...
      (... (f ... (rest xi) ...) ...)))
```

The above function has n parameters, where the ith parameter, x_i is a list. The function recurs down the list x_i. The . . .'s in the body indicate non-recursive, well-formed code, and (rest x_i) appears in the ith position.

We can use (len x_i) as the measure for any function conforming to the above scheme:

```
(defunc m (x1 ... xn)
  :input-contract (and ... (listp xi) ...)
  :output-contract (natp (m x1 ... xn))
  (len xi))
```

That m is a measure function is obvious. The non-trivial part is showing that

```
(listp xi) ∧ (not (endp xi)) ⇒ (len (rest xi)) < (len xi)
```

which is easy to see.

So, this scheme is terminating. This is why all of the code you wrote in your beginning programming class that was based on lists terminates.

We can generalize the above scheme, *e.g.*, consider:

```
(defunc f (x1 x2)
  :input-contract (and (listp x1) (listp x2))
  :output-contract (listp (f x1 x2))
  (cond ((endp x1) x2)
        ((endp x2) x1)
        (t (list (f (rest x1) (rest x2))
                 (f (rest x1) (f (rest x1) (cons x2 x2)))))))
```

We now have three recursive calls and two base cases. Nevertheless, the function terminates for the same reason: len decreases.

```
(defunc m (x1 x2)
  :input-contract (and (listp x1) (listp x2))
  :output-contract (natp (m x1 x2))
  (len x1))
```

All three recursive calls lead to the same proof obligation:

```
(listp x1) ∧ (not (endp x1)) ∧ (not (endp x2)) ⇒ (len (rest x1)) < (len x1)
```

Thinking in terms of recursion schemes and templates is good for beginners, but what really matters is termination. That is why recursive definitions make sense.

Let's look at one more interesting recursion scheme.

```
(defunc f (x1 ... xn)
  :input-contract (and ... (natp xi) ...)
  :output-contract ...
  (if (equal xi 0)
      ...
      (... (f ... (- xi 1) ...) ...)))
```

The above is a function of n parameters, where the ith parameter, x_i is a natural number. The function recurs on the number x_i. The ...'s in the body indicate non-recursive, well-formed code, and (- x_i 1) appears in the ith position.

We can use x_i as the measure for any function conforming to the above scheme:

```
(defunc m (x1 ... xn)
  :input-contract (and ... (natp xi) ...)
  :output-contract (natp (m x1 ... xn))
  xi)
```

That m is a measure function is obvious. The non-trivial part is showing that

(natp x_i) $\wedge$ (not (equal x_i 0)) $\Rightarrow$ (- x_i 1) $<$ x_i

which is easy to see.

So, this scheme is terminating. This is why all of the code you wrote in your beginning programming class that was based on natural numbers terminates.

Exercise 5.4 *We can similarly construct a recursion scheme for integers. Do it.*

5.3 Exercises

For each function below, you have to check if its definition is admissible, *i.e.*, it satisfies the definitional principle.

If the function does satisfy the definitional principle then:

1. Provide a measure that can be used to show termination.
2. Use the measure to prove termination.
3. Explain in English why the contract theorem holds.
4. Explain in English why the body contracts hold.

If the function does not satisfy the definitional principle then identify each of the 6 conditions above that are violated.

Exercise 5.5

```
(defunc f (x y)
  :input-contract (and (listp x) (natp y))
  :output-contract (listp (f x y))
  (cond ((equal y 0) nil)
        ((endp x) (list y))
        (t (f (cons y x) (- y 1)))))
```

Exercise 5.6 *Dead code example*

```
(defunc f (x y)
  :input-contract (and (natp x) (natp y))
  :output-contract (integerp (f x y))
  (cond ((equal x 0) 1)
        ((< x 0) (f -1 -1))
        (t (+ 1 (f (- x 1) y)))))
```

Notice that the second case of the `cond` above will never happen. Below are some generative recursion examples.

Exercise 5.7

```
(defunc f (x y)
  :input-contract (and (integerp x) (natp y))
  :output-contract (integerp (f x y))
  (cond ((equal x 0) 1)
        ((< x 0) (f (+ 1 y) (* x x)))
        (t (+ 1 (f (- x 1) y)))))
```

Exercise 5.8

```
(defunc f (x y)
  :input-contract (and (listp x) (integerp y))
  :output-contract (natp (f x y))
  (cond ((endp x) y)
        (t (f (rest x) (+ 1 y)))))
```

Exercise 5.9

```
(defunc f (x y)
  :input-contract (and (listp x) (integerp y))
  :output-contract (natp (f x y))
  (cond ((and (endp x) (equal y 0))
         0)
        ((and (endp x) (< y 0))
         (+ 1 (f x (+ 1 y))))
        ((endp x)
         (+ 1 (f x (- y 1))))
        (t
         (+ 1 (f (rest x) y)))))
```

Exercise 5.10

```
(defunc f (x)
  :input-contract (rationalp x)
  :output-contract (rationalp (f x))
  (if (< x 0)
      (f (+ x 1/2))
    x))
```

Exercise 5.11

```
(defunc f (x)
  :input-contract (rationalp x)
  :output-contract (natp (f x))
  (cond ((> x 0)   (f (- x 3/2)))
        ((< x 0)   (f (* x -1)))
        (t x)))
```

Exercise 5.12

```
(defunc f (x)
  :input-contract (rationalp x)
  :output-contract (natp (f x))
  (cond ((> x 0)   (f (- x 3/2)))
        ((< x 0)   (f 180))
        (t x)))
```

Exercise 5.13

```
(defunc f (x y)
  :input-contract (and (listp x) (rationalp y))
  :output-contract (natp (f x y))
  (cond ((> y 0)   (f x (- y 1)))
        ((consp x) (f (rest x) (+ y 1)))
        ((< y 0)   (f (list y) (* y -1)))
        (t y)))
```

Exercise 5.14

```
(defunc fib (n)
  :input-contract (natp n)
  :output-contract (natp (fib n))
  (if (< n 2)
      n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

Exercise 5.15

```
(defunc f (flg w r z s x y a b zs)
  :input-contract
  (and (integerp flg) (integerp w)
       (integerp r) (integerp z)
       (integerp s) (integerp x)
       (integerp y) (integerp a)
       (integerp b) (integerp zs))
  :output-contract (booleanp (f flg w r z s x y a b zs))
  (cond ((equal flg 1)
         (if (> z 0)
             (f 2 w r z 0 r w 0 0 zs)
           (equal w (expt r zs)))) ;expt is exponentiation
        ((equal flg 2)
         (if (> x 0)
             (f 3 w r z s x y y s zs)
           (f 1 s r (- z 1) 0 0 0 0 0 zs)))
        (t (if (> a 0)
               (f 3 w r z s x y (- a 1) (+ b 1) zs)
             (f 2 w r z b (- x 1) y 0 0 zs)))))
```

Exercise 5.16

```
(defunc f (x y)
  :input-contract (and (integerp x) (integerp y))
  :output-contract (natp (f x y))
  (cond ((< x y) (+ 1 (f (+ x 1) y)))
        ((> x y) (+ 1 (f x (+ y 1))))
        (t 0)))
```

f(x, y) =

Exercise 5.17

```
(defunc f (n)
  :input-contract (natp n)
  :output-contract (natp (f n))
  (cond ((<= n 1) n)
        ((integerp (/ n 2)) (f (/ n 2)))
        (t (f (+ n 1)))))
```

Exercise 5.18

```
(defunc f (n)
  :input-contract (natp n)
  :output-contract (natp (f n))
  (cond ((<= n 1) n)
        ((integerp (/ n 2)) (f (/ n 2)))
        (t (f (+ (* 2 n) 1)))))
```

Exercise 5.19

```
(defunc f (n)
  :input-contract t
  :output-contract t
  (cond ((or (not (integerp n))
             (<= n 1)) 0)
        ((integerp (+ 1/2 (/ n 2))) (f (+ n 1)))
        (t (+ 1 (f (/ n 2))))))
```

Exercise 5.20

```
(defunc ack (n m)
  :input-contract (and (natp n) (natp m))
  :output-contract (posp (ack x y))
  (cond ((equal n 0) (+ m 1))
        ((equal m 0) (ack (- n 1) 1))
        (t (ack (- n 1) (ack n (- m 1))))))
```

Exercise 5.21 *This is hard.*

```
(defunc m (x)
  :input-contract (integerp x)
  :output-contract (natp (m x))
  (if (< 100 x)
      (- x 10)
    (m (m (+ x 11)))))
```

5.4 Complexity Analysis

Remember "big-Oh" notation? It is connected to termination. How?

Well if the running time for a function is $O(n^2)$, say, then that means that:

1. the function terminates; and
2. there is a constant c s.t. the function terminates within $c \cdot n^2$ steps, where n is the "size" of the input

The big-Oh analysis is just a refinement of termination, where we are not interested in only whether a function terminates, but also we want an upper bound on how long it will take.

5.5 Undecidability of the Halting Problem

Turing's result that termination is undecidable is an amazing, fundamental result that highlights the limits of computation.

Here is a proof of the undecidability of the halting problem.

But first, a few basic observations about programs that will help us with the proof.

The first observation is that we can enumerate all programs. That means that we can create a sequence (list) indexed by the natural numbers in such a way that every program appears exactly once in the sequence. In fact, there is a function f that given a natural number, i, returns the ith program.

The second observation is that we can treat all inputs and outputs as natural numbers (say by thinking of them as bit-vectors).

With these observations, a program is just a function from natural numbers to natural numbers.

Our proof will be based on *diagonalization*, a powerful proof method. Here is how diagonalization works.

First, we start by negating the conjecture: the halting problem is decidable. So under this assumption, we have a program

$$h(i) = \text{ if Program } f(i) \text{ is terminating } \textit{ then } 1 \textit{ else } 0$$

Now we can construct infinite tables where rows and columns are indexed by natural numbers and cell n, m is $Fn(m)$ (where Fn is $F(n)$ and F is some function that returns a program (not necessarily f)).

```
        0       1       2       ...
0       F0(0)   F0(1)   F0(2)   ...
1       F1(0)   F1(1)   F1(2)   ...
2       F2(0)   F2(1)   F2(2)   ...
...     ...                     ...
```

Next, we derive a contradiction by defining a table like the one above and showing that a program that should be in the table is not.

Let $g(0), g(1), g(2), \ldots$ be the list of terminating program indices in order. Here is a more rigorous definition.

$$g(0) = \text{ smallest } i \text{ such that } f(i) \text{ is terminating.}$$

$$g(n+1) = \text{ smallest i } > g(n) \text{ such that } f(i) \text{ is terminating.}$$

Notice that this is well defined because there are an infinite number of terminating programs! Notice also that since h is decidable, g is a computable, terminating function, so for some i we have that $f(i) = g$.

In the table we will use in our proof, $Fn = f(g(n))$.

So, *every* terminating function appears somewhere in the table and the table tells us what *every* terminating function returns on *every* possible input.

Now, we are ready for our contradiction. We will define d so that it is a terminating program (and so should be in the table), but it also differs along the diagonal, *i.e.*, it differs with every program in our table on at least one input.

$$d(n) = Fn(n) + 1$$

That is, to determine $d(n)$, compute $g(n)$, which gives us the nth terminating program. Then run that program on n and add 1 to the result. Notice that d is a terminating program! (Because $g(n)$ is the index of a terminating program, $f(g(n))$ terminates on all inputs.)

Now, since d is a terminating program there is some k for which $Fk = d$. But what is $d(k)$? Well it should be $Fk(k)$ (since $Fk = d$), but according to the definition of d, it is $Fk(k) + 1$, a contradiction.

```
        0       1       2       ...
0       F0(0)   F0(1)   F0(2)   ...   d(0) != F0(0)
1       F1(0)   F1(1)   F1(2)   ...   d(1) != F1(1)
2       F2(0)   F2(1)   F2(2)   ...   d(2) != F2(2)
...     ...                     ...   d(n) != Fn(n)
```

Wait, we derived *false*. (We used $Fk(k) = Fk(k) + 1$, which is exactly the function we showed leads to inconsistency in ACL2s when motivating the need for termination analysis!) So, is *false* is a theorem? Of course not. What we showed is that if we assume that termination is decidable, then we can prove *false*. So, termination is not decidable. This is a proof by contradiction, a key proof technique.

This is also a proof by *diagonalization*, another key proof technique introduced by Cantor in the late 1800's, where he used it to show that there is a infinite tower of infinities.

Exercise 5.22 *How would you write a program that checks if other programs terminate? We just proved that there is no decision procedure for termination, so your program will*

return "yes", indicating that the input program always terminates, "no", indicating that the input program fails to terminate on at least one input, or "unknown", indicating that your program cannot determine whether the input program is terminating. A really simple solution is to always return "unknown." The goal is to write a program that returns as few "unknown"s as possible.

Given the result in this section, we know that there are terminating functions for which ACL2s (or any other termination analysis engine) will fail to prove termination. However, we expect that for almost all of the programs we ask you to write in logic mode, ACL2s will be able to prove termination automatically. If not, send email and we will help you.

Part V

Induction

6

Induction

Terminating functions give rise to induction schemes.

Consider the following function:

```
(defunc nind (n)
  :input-contract (natp n)
  :output-contract t
  (if (equal n 0)
      0
    (nind (- n 1))))
```

This function is admissible. Given a natural number `n` it counts down to 0 and returns, therefore it is terminating.

Induction is justified by termination: every terminating function gives rise to an induction scheme. For example, suppose we want to prove φ using the induction scheme from `(nind n)`. Our proof obligations are:

1. `(not (natp n))` $\Rightarrow \varphi$
2. `(natp n)` $\wedge$ `(equal n 0)` $\Rightarrow \varphi$
3. `(natp n)` $\wedge$ `(not (equal n 0))` $\wedge\ \varphi|_{\texttt{((n n-1))}} \ \Rightarrow \ \varphi$

A bit of terminology. Cases 1 and 2 are *base cases*. Case 3 is an *induction step* because we get to assume that φ holds on smaller values. The last hypothesis of case 3, $\varphi|_{\texttt{((n n-1))}}$, is called an *induction hypothesis*. Notice that the induction hypothesis is what distinguishes induction from case analysis, *i.e.*, we could try to prove φ using case analysis as follows:

1. `(not (natp n))` $\Rightarrow \varphi$
2. `(natp n)` $\wedge$ `(equal n 0)` $\Rightarrow \varphi$
3. `(natp n)` $\wedge$ `(not (equal n 0))` $\ \Rightarrow \ \varphi$

The three cases above are exhaustive. Induction is more powerful than case analysis because it also allows us to assume the induction hypothesis.

Back to induction.

Why does induction work?

First, let me describe a proof technique that I'm going to use (and that is widely used): Proof by contradiction.

What does a proof by contradiction look like? It's just a "cheap" propositional trick. Let's say that we are trying to prove:

$$\varphi_1 \ \wedge \cdots \ \wedge \varphi_n \ \Rightarrow \ \varphi$$

We do this by assuming the negation of the consequent and deriving a contradiction, *i.e.*, we instead prove:

$$\varphi_1 \;\wedge \cdots \;\wedge\; \varphi_n \;\wedge\; \neg\varphi \;\Rightarrow\; \mathit{false}$$

Note that these two statements are equivalent by propositional logic. To see this, note

$$A \wedge B \;\Rightarrow\; C \;\;\equiv\;\; A \wedge \neg C \;\Rightarrow\; \neg B$$

Then apply the above to

$$\varphi_1 \;\wedge \cdots \;\wedge\; \varphi_n \;\wedge \mathit{true} \;\Rightarrow\; \varphi$$

Proof by contradiction is often referred to *reductio ad absurdum*, Latin for "reduction to the absurd."

Here is a great quote about proof by contradiction from Godfrey Harold Hardy's *A Mathematician's Apology* (1940).

> Reductio ad absurdum, which Euclid loved so much, is one of a mathematician's finest weapons. It is a far finer gambit than any chess gambit: a chess player may offer the sacrifice of a pawn or even a piece, but a mathematician offers the game.

Back to our question.

Suppose that we prove the above three cases, but φ is not valid.

Then, the set, S, of objects in the ACL2 universe for which φ does not hold is non-empty. The set can only contain positive natural numbers, as case 1 rules out there being any non-natural numbers in S and case 2 rules out 0 being in S. Consider the smallest (natural) number $s \in S$. Now instantiate the induction step (3), replacing `n` by s:

$$\texttt{(natp } s\texttt{)} \wedge \texttt{(not (equal } s \texttt{ 0))} \wedge \varphi|_{\texttt{((n } s-1\texttt{))}} \;\Rightarrow\; \varphi|_s$$

Notice that $(\varphi|_{\texttt{((n n-1))}})|_{\texttt{((n } s\texttt{))}} = \varphi|_{\texttt{((n } s-1\texttt{))}}$. But, we have that `(natp` s`)` holds and $s \neq 0$. By the minimality of s, $\varphi|_{\texttt{((n } s-1\texttt{))}}$ also holds. By MP and the above, so does $\varphi|_s$. So, $s \notin S$! That is our contradiction, so our assumption that φ is false led to a contradiction, *i.e.*, φ is in fact valid.

Two observations.

1. We used a nice property of the natural numbers: they are *terminating*: every decreasing sequence is finite. This is equivalent to saying that every non-empty subset has a minimal element. Induction works as long as we have termination. We can prove termination for any kind of ACL2s function, using measure functions. For example, measure functions allow us to prove termination of functions that operate on lists. Notice that they do this by relating what happens on lists to numbers.

2. We used a proof by counterexample. Why do people use proofs by counterexample? It seems like an elaborate way of proving φ. In some sense it is, but it is a nice technique to have in your arsenal because it often helps you focus on the goal: prove false.

Here is yet another way to see why induction works. Suppose, as before, that we prove the three proof obligations above. Now, as before, the first two proof obligations directly show that φ holds for all non-natural numbers and for 0. If we instantiate the third proof obligation, the induction step, with the substitution `((n 1))`, then the hypotheses hold (as `(natp 1)`, $1 \neq 0$, and $\varphi|_{\texttt{((n 0))}}$ all hold), so by MP $\varphi|_{\texttt{((n 1))}}$ holds. Notice that we use the

induction step to go from $\varphi|_{((\texttt{n}\ 0))}$ to $\varphi|_{((\texttt{n}\ 1))}$. Similarly, we can use $\varphi|_{((\texttt{n}\ 1))}$ to get $\varphi|_{((\texttt{n}\ 2))}$ and so on for all the natural numbers. We have just shown that for any natural number i, $\varphi|_{((\texttt{n}\ i))}$ holds, so φ holds for all objects in the ACL2s universe. This is a direct proof that proof by induction is sound.

Now that we have motivated induction, here it is in its full glory.

Induction Scheme: Suppose we are given a function definition of the form:

```
(defunc foo (x1 ... xn)
  :input-contract ic
  :output-contract oc
  (cond (t1 c1)
        (t2 c2)
        ...
        (tm cm)
        (t cm+1)))
```

where none of the $\texttt{c}_i$'s have any `if`s in them.

Notice that any function definition can be written in this form.

If $\texttt{c}_i$ contains a call to `foo`, we say it is a *recursive* case; otherwise it is a *base* case. If $\texttt{c}_i$ is a recursive case, then it includes at least one call to `foo`. Say there are R_i calls to `foo` and they are $(\texttt{foo x}_1 \ \ldots \ \texttt{x}_n)|_{\sigma_i^j}$, for $1 \leq j \leq R_i$ (the σ_i^j's are substitutions).

Let $\texttt{t}_{m+1}$ be `t`.

Let Case$_i$ be $\texttt{t}_i \wedge \neg\texttt{t}_j$ for all $j < i$, *e.g.*,

- Case$_1$ is $\texttt{t}_1$
- Case$_2$ is $\texttt{t}_2 \wedge \neg\texttt{t}_1$
- Case$_3$ is $\texttt{t}_3 \wedge \neg\texttt{t}_1 \wedge \neg\texttt{t}_2$
- Case$_{m+1}$ is $\texttt{t} \wedge \neg\texttt{t}_1 \wedge \neg\texttt{t}_2 \wedge \cdots \wedge \neg\texttt{t}_m$

The function `foo` gives rise to the following induction scheme:

To prove φ, you can instead prove

1. $\neg\texttt{ic} \Rightarrow \varphi$
2. For all $\texttt{c}_i$ that are base cases: $[\texttt{ic} \wedge \text{Case}_i] \Rightarrow \varphi$
3. For all $\texttt{c}_i$ that are recursive cases: $[\texttt{ic} \wedge \text{Case}_i \wedge \bigwedge_{1\leq j\leq R_i} \varphi|_{\sigma_i^j}] \Rightarrow \varphi$

We can play this game in reverse. For example, if I were to ask you to write a function that gives rise to the following induction scheme:

1. $\texttt{(not (natp n))} \Rightarrow \varphi$
2. $\texttt{(natp n)} \wedge \texttt{(equal n 0)} \Rightarrow \varphi$
3. $\texttt{(natp n)} \wedge \texttt{(not (equal n 0))} \wedge \varphi|_{((n\ n-1))} \Rightarrow \varphi$

Exercise 6.1 *One correct answer is* `nind`*, but there are infinitely many correct answers. Show this.*

6.1 Induction Examples

Recall the definition of `sumn`.

```
(defunc sumn (n)
  :input-contract (natp n)
  :output-contract (natp (sumn n))
  (if (equal n 0)
      0
    (+ n (sumn (- n 1)))))
```

Let's prove

$$\texttt{(sumn n)} = \texttt{n}(\texttt{n}+1)/2$$

Recall that the first thing we do is to contract check conjectures. After fixing the above conjecture, we get:

$$\texttt{(natp n)} \Rightarrow \texttt{(sumn n)} = \texttt{n}(\texttt{n}+1)/2 \qquad (6.1)$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `sumn`. When you find yourself in such a situation, induct!

What induction scheme?

The one that the data definition gives rise to. In this case, we are reasoning about natural numbers, so `nind`'s induction scheme (which is the same as `sumn`'s induction scheme).

Our proof obligations are then:

1. $\texttt{(not (natp n))} \Rightarrow (6.1)$
2. $\texttt{(natp n)} \wedge \texttt{n} = 0 \Rightarrow (6.1)$
3. $\texttt{(natp n)} \wedge \texttt{n} \neq 0 \wedge (6.1)|_{\texttt{((n (- n 1)))}} \Rightarrow (6.1)$

Notice that the proof now goes through with just equational reasoning.

So, since we know how to do equational reasoning, we will skip the equational proofs, but you can and should fill them in.

Let's now reason about the following function definition.

```
(defunc app (a b)
  :input-contract (and (listp a) (listp b))
  :output-contract (and (listp (app a b))
                        (equal (len (app a b))
                               (+ (len a) (len b))))
  (if (endp a)
      b
    (cons (first a) (app (rest a) b))))
```

We want to prove that `app` is associative.

$$\texttt{(app (app a b) c)} = \texttt{(app a (app b c))}$$

Contract checking gives:

$$\texttt{(listp a)} \wedge \texttt{(listp b)} \wedge \texttt{(listp c)} \Rightarrow \texttt{(app (app a b) c)} = \texttt{(app a (app b c))} \quad (6.2)$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `app`. When you find yourself in such a situation, induct!

What induction scheme?

The one that the data definition gives rise to. In this case, we are reasoning about lists, so `listp`'s induction scheme. Recall

```
(defunc listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
  (if (consp l)
      (listp (rest l))
    (equal l ())))
```

So the induction scheme is:

1. $\neg\texttt{t} \Rightarrow (6.2)$
2. $\texttt{t} \wedge \texttt{(consp a)} \wedge (6.2)|_{\texttt{((a (rest a)))}} \Rightarrow (6.2)$
3. $\texttt{t} \wedge \neg\texttt{(consp a)} \Rightarrow (6.2)$

This is equivalent to:

1. $\neg\texttt{(consp a)} \Rightarrow (6.2)$
2. $\texttt{(consp a)} \wedge (6.2)|_{\texttt{((a (rest a)))}} \Rightarrow (6.2)$

Exercise 6.2 *Notice that the variables we use in proofs are irrelevant,* e.g.*, even though* `listp` *was defined over* `l`*, we can apply induction using* `a` *instead. Explain why this is the case.*

So, here we go. If you expand out the proof obligations, we have a problem we've seen before! Do the induction step.

Exercise 6.3 *Assume that the output contract for* `app` *is* `t`*. This version of* `app` *is admissible. Using this version of* `app`*, use induction to prove the first conjunct in the output contract of the definition given earlier (*`(listp (app x y))`*).*

Exercise 6.4 *Assume that the output contract for* `app` *is* `t`*. This version of* `app` *is admissible. Using this version of* `app`*, use induction to prove the second conjunct in the output contract of the definition given earlier.*

Exercise 6.5 *Play the same game of proving the output contracts of the functions we have defined. Some will require induction, but some can be proved using just equational reasoning.*

Exercise 6.6 *Formalize (using ACL2s) and prove the following theorem (n is a natural number):*

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Next, we will play around with `rev`. Here is the definition.

```
(defunc rev (x)
  :input-contract (listp x)
  :output-contract (and (listp (rev x))
                        (equal (len (rev x))
                               (len x)))
  (if (endp x)
      nil
    (app (rev (rest x)) (list (first x)))))
```

Now we want to prove:

$$\texttt{(rev (rev x))} = \texttt{x}$$

Contract checking gives:

$$\texttt{(listp x)} \Rightarrow \texttt{(rev (rev x))} = \texttt{x} \tag{6.3}$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `rev`. When you find yourself in such a situation, induct!

What induction scheme?

The one that the data definition gives rise to. In this case, we are reasoning about lists, so `listp`'s induction scheme, which is:

1. $\neg\texttt{(consp x)} \Rightarrow (6.3)$
2. $\texttt{(consp x)} \wedge (6.3)|_{\texttt{((x (rest x)))}} \Rightarrow (6.3)$

Try the proof. You will get stuck.

Now what? Well, we need a lemma. If we had the following:

$$\texttt{(listp x)} \wedge \texttt{(listp y)} \Rightarrow \texttt{(rev (app x y))} = \texttt{(app (rev y) (rev x))} \tag{6.4}$$

we could finish the proof.

Let's assume we have it and then finish the proof.

So, notice that sometimes in the process of proving a theorem by induction, we have to prove lemmas in order for the proof to go through.

Exercise 6.7 *Prove* (6.4). *Use the induction scheme* `(listp x)` *gives rise to.*

In the proof of (6.4), we needed to prove

$$\texttt{(listp x)} \Rightarrow \texttt{(app x nil)} = \texttt{x}$$

So, even the proof of the lemma requires a lemma. How far can this go? Far! It's recursive.

Exercise 6.8 *What induction scheme does this give rise to?*

```
(defunc fib (n)
  :input-contract (posp n)
  :output-contract (posp (fib n))
  (cond ((equal n 1)
         1)
        ((equal n 2)
         2)
        (t (+ (fib (- n 1)) (fib (- n 2))))))
```

Let's prove the following:

$$\texttt{(fib n)} \geq n$$

Contract checking gives us:

$$\texttt{(posp n)} \Rightarrow \texttt{(fib n)} \geq n$$

Now what?

Can we prove this using induction on the natural numbers? Try it. The base case is trivial, but the induction step winds up requiring case analysis.

A better idea is to use the induction scheme `fib` gives rise to.

Why should you not be surprised? Well, because we didn't define `fib` using the data definition for `Nat`. `fib` is an example of generative recursion.

The point is that the induction scheme one should use to prove a conjecture is often related to the recursion schemes of the functions appearing in the conjecture. If all of the functions in the conjecture are based on a common recursion scheme (or design recipe), then the induction schemes will also be based on the same recursion scheme.

Exercise 6.9 *Prove the previous conjecture using the induction scheme* `fib` *gives rise to.*

6.2 Data-Function-Induction Trinity

Every admissible recursive definition leads to a valid induction scheme. What underlies both recursion and induction is *termination*. So, terminating functions give us both recursion schemes and induction schemes.

Notice a wonderful connection.

The data-function-induction (DFI) trinity:

1. Data definitions give rise to predicates recognizing such definitions. These predicates must be shown to terminate. (Otherwise they are inadmissible by the Definitional Principle.) Their bodies give rise to a *recursion scheme*, *e.g.*, `listp` gives rise to the common recursion scheme for iterating over a list.

2. Functions over these data types are defined by using the *recursion scheme* as a template. Templates allow us to define correct functions by assuming that the function we are defining works correctly in the recursive case. For example, in the definition of `app`, we get to assume that `app` works correctly in the recursive case, even if its

first input has 1,000,000 elements, *i.e.*, we get to assume that `app` applied to 999,999 elements works and all we have to do is to figure out what to do with the first element. This is about as simple an extension to straight-line code as we can imagine. Recursion provides us with a *significant* increase in expressive power, allowing us to define many functions that are not expressible using only straight-line code.

3. The *Induction Principle*: Proofs by induction involving such functions and data definitions should use the same *recursion scheme* to generate proof obligations. Non-recursive cases are proven directly. For each recursive case, we assume the theorem under *any* substitutions that map the formals to arguments in that recursive call. Induction provides us with a *significant* increase in theorem proving power over equational reasoning, analogous to the increase in definitional power we get when we move from straight-line code to recursive code. Notice also that induction and recursion are tightly related, *e.g.*, when defining recursive functions, we get to assume that the function works on smaller inputs; when proving theorems with induction, we get to assume that the theorem holds on smaller inputs (the induction hypothesis).

6.3 The Importance of Termination

Notice how important termination turns out to be.

1. Termination is the non-trivial proof obligation for admitting function definitions.
2. Termination is what justifies common recursion schemes and the design recipe.
3. Termination is what justifies generative recursive function definitions.
4. Complexity analysis is just a refinement of termination.
5. Termination is what justifies mathematical induction.
6. Terminating functions give rise to induction schemes. In fact, the only induction schemes we will use are the ones we can extract from terminating functions.

Exercise 6.10 *Consider the following non-terminating function definition.*

```
(defunc f (x)
  :input-contract t
  :output-contract t
  (f x))
```

Were we to admit `f` *(which we really can't because it is non-terminating), we would get the definitional axiom* `(f x) = (f x)`*.*

We have seen that this axiom does not lead to unsoundness. However, the "induction scheme" this function gives rise to does lead to unsoundness. Prove `nil` *using the induction scheme* `f` *gives rise to. Notice that there is a stronger relationship between induction and termination than there is between admissibility and termination. This relationship is an important reason why ACL2s does not admit non-terminating function definitions.*

6.4 Generalization

Consider the following definitions.

```
(defunc in (a X)
  :input-contract (listp x)
  :output-contract (booleanp (in a X))
  (cond ((endp x) nil)
        ((equal a (first X)) t)
        (t (in a (rest X)))))

(defunc subsetp (x y)
  ; checks if every element in x is in y
  :input-contract (and (listp x) (listp y))
  :output-contract (booleanp (subsetp x y))
  (cond ((endp x) t)
        ((in (first x) y)
         (subsetp (rest x) y))
        (t nil)))
```

Try to prove the following theorem:

$$\texttt{(listp x)} \Rightarrow \texttt{(subsetp x x)} \tag{6.5}$$

Notice that we can't prove this by induction. Why? Because whatever we do, we have to substitute for `x` and we want to distinguish the two occurrences of `x`. Unfortunately, we can't do that.

The solution?

Generalize: prove a theorem that we can prove by induction and that can then be used to prove the theorem we really want.

Here is the generalization:

$$\texttt{(listp x)} \wedge \texttt{(listp y)} \wedge \texttt{(subsetp x y)} \Rightarrow \texttt{(subsetp x (cons a y))} \tag{6.6}$$

Now, we can prove (6.6) using induction.

Exercise 6.11 *Prove* (6.6)

Exercise 6.12 *Prove* (6.5)

Exercise 6.13 *Prove* `(listp x)` $\wedge$ `(listp y)` $\wedge$ `(listp z)` $\wedge$ `(subsetp x y)` $\wedge$ `(subsetp y z)` $\Rightarrow$ `(subsetp x z)`

6.5 Reasoning About Accumulator-Based Functions

Let's start with a simple definition we have already seen.

```
(defunc rev (x)
  :input-contract (listp x)
```

```
  :output-contract (listp (rev x))
  (if (endp x)
      nil
    (app (rev (rest x)) (list (first x)))))
```

The problem with this definition is that it requires $O(n^2)$ conses. Why?

Because `(app x y)` requires `(len x)` conses (as we have seen previously).

In the recursive case of `rev`, we have `app` applied to `(rev (rest x))` which requires `(len (rev (rest x)))` conses plus one cons for the list, *i.e.*, `(len x)` conses. Since `rev` is called on `x`, then `(rest x)`, then `(rest (rest x))`, ..., it requires `(len x)` + `(len x)`-1 + `(len x)`$-2 + \ldots + 1$ conses, which is $O(n^2)$, for $n =$ `(len x)`.

This is a horrible function from an efficiency point of view, so we want to do better. One way of doing better is to define a tail-recursive function that uses an accumulator.

Here is such a definition.

```
(defunc revt (x acc)
  :input-contract (and (listp x) (listp acc))
  :output-contract (listp (revt x acc))
  (if (endp x)
      acc
    (revt (rest x) (cons (first x) acc))))
```

But, we want a function with the same interface as `rev` and `revt` takes 2 arguments. Hence, we define:

```
(defunc rev* (x)
  :input-contract (listp x)
  :output-contract (listp (rev* x))
  (revt x nil))
```

Exercise 6.14 *Show that* `(rev* l)` *requires only* $O(n)$ *conses, where* $n =$ `(len l)`.

We are now in a situation that computer scientists often find themselves in. We have one function definition `rev` that is simple, but inefficient. We also have another function definition that is more complex, but efficient. We want to show that these two functions are related in some way.

What relationship do we want to establish between `rev*` and `rev`?

Let's prove that they are equal.

$$\texttt{(listp x)} \Rightarrow \texttt{(rev* x)} = \texttt{(rev x)} \tag{6.7}$$

Is it true?

Can we solve this with equational reasoning? No. Why not?

Notice that proving (6.7) will require proving:

$$\texttt{(listp x)} \Rightarrow \texttt{(revt x nil)} = \texttt{(rev x)} \tag{6.8}$$

It is the recursive definitions that we have to worry about!

We will try to prove correctness using what we already know. Our proof attempt will run into several hurdles and we will have to analyze what went wrong and how to proceed. By the end of this section, we will have constructed a little recipe for reasoning about accumulator-based functions in the future.

Let's try proving (6.8) using the induction scheme `listp` gives rise to.

1. ¬`(consp x)` ⇒ (6.8)
2. `(consp x)` ∧ (6.8)$|_{\texttt{((x (rest x)))}}$ ⇒ (6.8)

Let's try to prove this.
Proof?
The base case is simple. Here is an attempt at proving the induction step.
The context is:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp (rest x))` ⇒ `(revt (rest x) nil)` = `(rev (rest x))`

C4. `(listp (rest x))` {C1, C2, Def `listp`}

C5. `(revt (rest x) nil)` = `(rev (rest x))` {C3, C4, MP}

Proof:
`(revt x nil)`
= { By C1 and the definition of `revt` }
`(revt (rest x) (cons (first x) nil))`

But, now what? Our induction hypothesis tells us something about

`(revt (rest x) nil)`

but we really need to know something about

`(revt (rest x) (cons (first x) nil))`

so we're stuck. The point is that when we expand

`(revt x nil)`

we get an expression that is of the form

`(revt ... (cons ...))`

The second argument is not `nil`, but any way of instantiating the theorem we want to prove (as we do to get the induction hypothesis) will give us a `nil` in the second argument.

We need a handle on that second argument, but we're not going to get it if the theorem we want to prove has a `nil` there; we need a variable. We call this a generalization step because we are now going to prove a theorem about `(revt x acc)`, whereas before we were proving a theorem about a special case of the above, namely about `(revt x nil)`. But, now we have to figure out what the new theorem we want to prove is.

`(listp x)` ∧ `(listp acc)` ⇒ `(revt x acc)` = `???`

What is `???` ? Think about the role of `acc` in the definition of `revt`. The accumulator corresponds to a partial result: it should be the reverse of the elements of the original argument to `revt` that we have seen already, so we wind up with:

$$\texttt{(listp x)} \wedge \texttt{(listp acc)} \Rightarrow \texttt{(revt x acc)} = \texttt{(app (rev x) acc)} \qquad (6.9)$$

Suppose we prove (6.9). Can we use it to prove (6.7) and (6.8)?
Yes. Why?
The base case is simple. Here is a proof of the induction step. First, our context is:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp acc)`

C4. `(listp (rest x))` ∧ `(listp acc)` ⇒
`(revt (rest x) acc)` = `(app (rev (rest x)) acc)`

C5. `(listp (rest x))` {C1, C2, Def `listp`}

C6. `(revt (rest x) acc)` =
`(app (rev (rest x)) acc)` {C5, C3, C4, MP}

Here is the proof. It starts the same way as before.
`(revt x acc)`
= { By C1 and the definition of `revt` }
`(revt (rest x) (cons (first x) acc))`

But, now we have yet another problem. The induction hypothesis doesn't match the above, since, as stated it is

$$\texttt{(revt (rest x) acc)} = \texttt{(app (rev (rest x)) acc)}$$

and we don't want `acc` as the second argument of `revt`; we want `(cons (first x) acc)`.

So, we can either define a function that gives rise to this induction scheme, or we can see if we have such a function available to us. In fact, we do: `revt`! So, let's use the induction scheme `revt` gives rise to. That gives us the following context:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp acc)`

C4. `(listp (rest x))` ∧ `(listp (cons (first x) acc))` ⇒
`(revt (rest x) (cons (first x) acc))` =
`(app (rev (rest x)) (cons (first x) acc))`

C5. `(listp (rest x))` {C1, C2, Def `listp`}

C6. `(listp (cons (first x) acc))` {C3, Def `listp`}

C7. `(revt (rest x) (cons (first x) acc))` =
`(app (rev (rest x)) (cons (first x) acc))` {C5, C6, C4, MP}

Proof:

```
    (revt x acc)
= { By C1 and the definition of revt }
    (revt (rest x) (cons (first x) acc))
= { By C7 }
    (app (rev (rest x)) (cons (first x) acc))
= { Def of app (working on pulling acc out to match RHS) }
    (app (rev (rest x)) (app (list (first x)) acc))
= { Associativity of app (now we pull acc out) }
    (app (app (rev (rest x)) (list (first x))) acc)
= { Def of rev, C1 }
    (app (rev x) acc)
```

Based on our experience above, here is a little recipe for reasoning about accumulator-based functions.

Part 1: Defining functions

1. Start with a function `f`.
2. Define `ft`, a tail-recursive version of `f` with an accumulator.
3. Define `f*`, a non-recursive function that calls `ft` and is logically equivalent to `f`, *i.e.*, this is a theorem

   hyps ⇒ `(f* ...)` = `(f ...)`

Part 2: Proving theorems

4. Identify a lemma that relates `ft` to `f`. It should have the following form:

   hyps ⇒ `(ft ... acc)` = ... `(f ...)` ...

   Remember that you have to generalize, so all arguments to `ft` should be variables (no constants). The RHS should include `acc`.
5. Assuming that the lemma in 4 is true, and using only equational reasoning, prove the main theorem

   hyps ⇒ `(f* ...)` = `(f ...)`

   If you have to prove lemmas, prove them later.
6. Prove the lemma in 4. Use the induction scheme `ft` gives rise to.
7. Prove any remaining lemmas.

You might wonder why we bother to define `f*`? So that we can use `f*` as a replacement for `f`: `ft` won't do because it has a different signature than `f`.

You might want to swap steps 5 and 6. Don't because you want to first make sure that the lemma from 4 is the one you need.

If you want to swap steps 6 and 7, that's fine.

Part VI

Steering the ACL2 Sedan

7

Steering the ACL2 Sedan

7.1 Interacting with ACL2s

Most of the material in this chapter comes from the Computer-Aided Reasoning book. As depicted in Figure 7.1, the theorem prover takes input from both you and a database, called the *logical world* or simply *world*. The world embodies a theorem proving strategy, developed by you and codified into *rules* that direct the theorem prover's behavior. When trying to prove a theorem, the theorem prover applies your strategy 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 database is the `defthm` command.

```
(defthm name formula
  :rule-classes (class_1 ... class_n))
```

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 database in each of the ways specified by the $class_i$. To find out details of the various rule classes, see the documentation topic `rule-classes`.

You have lots of control over what the theorem prover can do. For example, 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 the documentation topic `hints`), you can change the strategy embodied in the world.

7.2 The Waterfall

So, how does ACL2 work? Let's look at the classic example that shows the ACL2 waterfall. This is in `ACL2s` mode.

```
(defun rev (x)
```

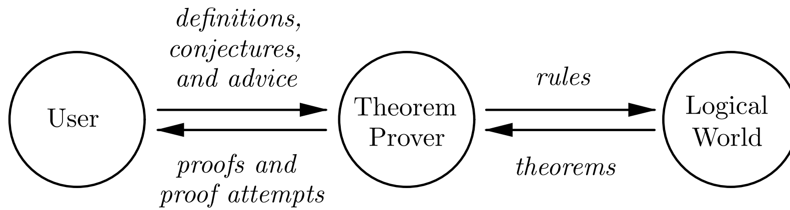

Figure 7.1: Data Flow in the Theorem Prover

```
  (if (endp x)
      nil
    (append (rev (rest x)) (list (first x)))))

(defthm rev-rev
  (implies (true-listp x)
           (equal (rev (rev x)) x)))
```

Think of `defun` as `defunc` without contracts and think of `true-listp` as `listp`. See section 8.2 of the Computer-Aided Reasoning book for an in-depth discussion.

The `rev-rev` example nicely highlights the organization of ACL2, which is shown in Figure 7.2. At the center is a *pool* of formulas to be proved. The pool is initialized with the conjecture you are trying to prove. Formulas are removed from the pool and processed using six proof techniques. If a technique can reduce the formula, say to a set of $n \geq 0$ other formulas, then it deposits these formulas into the pool. In the case where n is 0, the formula is proved by the technique. If a technique can't reduce the formula, it just passes it to the next 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 called "the waterfall."

Go over the proof output for the above theorem in Theorem Proving Beginner Mode in ACL2s.

7.3 Term Rewriting

It is easy to be impressed with what ACL2 can do automatically, and you might think that it 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 have to use your knowledge of the mechanization to figure out what the system is missing. And, an understanding of how simplification, and in particular rewriting, works is a requirement.

We are going to focus on rewriting, as the successful use of the theorem prover requires successful control of the rewriter.

You have to understand how the rewriter works and how the theorems you prove affect the rewriter in order to develop a successful proof strategy that can be used to prove the theorems you are interested in formally verifying.

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

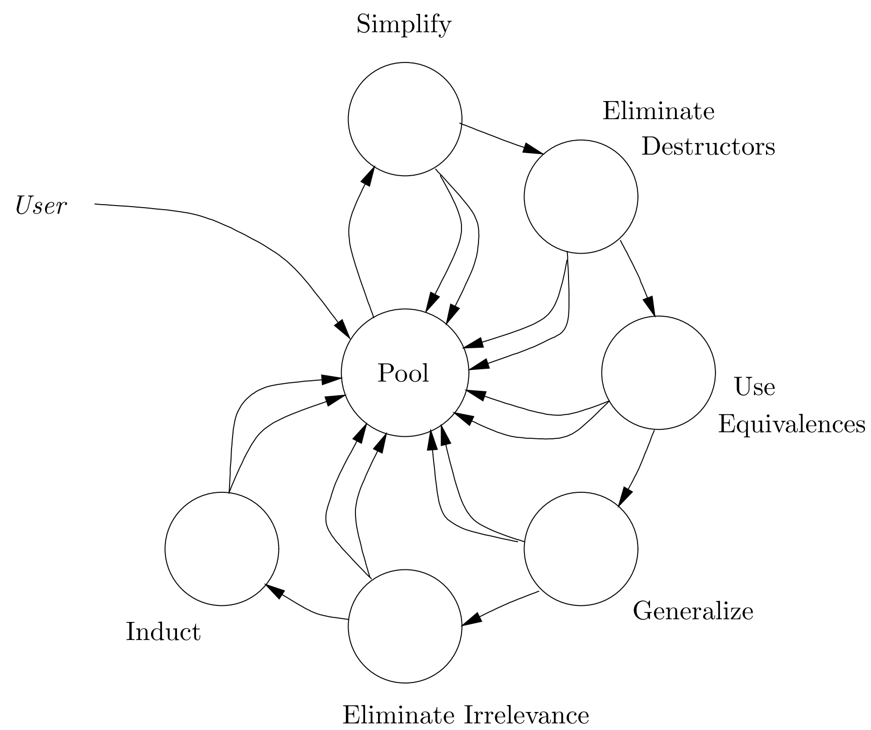

Figure 7.2: Organization of the Theorem Prover

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 ... 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 ... a'_n`)`, as described below.

But a few functions are handled specially. For example 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 whether we can establish if a'_1 is `nil`.

Now we explain how rewrite rules are applied to `(`f a'_1 ... 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)
         (equal (f b1 ... bn)
                rhs))
```

Note that the definition of f is of this form, where $k = 0$.

Aside: A theorem concluding with a term of the form `(not (`p ...`))` is considered, for these purposes, to conclude with `(iff (`p ...`) nil)`. A theorem concluding with `(`p ...`)`, where p is not a known equivalence relation and not `not`, is considered to conclude with `(iff (`p ...`) t)`.

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 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)
         (equal (f a'1 ... a'n)
                rhs'))
```

To apply the instantiated rule the rewriter must establish its hypotheses. To do so, rewriting 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.

7.3.1 Example

Suppose you just completed a session with ACL2s where you proved theorems leading to the following rewrite rules.

These rewrite rules were admitted in the give order, *i.e.*, 1 was admitted first, then 2, then 3, then 4.

1. `(f (h a) b) = (g a b)`

2. `(g a b) = (h b)`
3. `(g c d) = (h c)`
4. `(f (h a) (h b)) = (f (h b) a)`

- ♦ Some of the rewrite rules above can *never* be applied to *any* expression. Which rules are they?
- ♦ Show what ACL2s rewrites the following expression into. Show all steps, in the order that ACL2s will perform them.

```
(equal (f (f (h b) (h a)) b) (h b))
```

Answer:
1. Rule 2 cannot be applied because rule 3 will always match first.
2. Here are all the steps.

```
  (equal (f (f (h b) (h a)) b) (h b))
= { By Rule 4 }
   (equal (f (f (h a) b) b) (h b))
= { By Rule 1 }
   (equal (f (g a b) b) (h b))
= { By Rule 3 }
   (equal (f (h a) b) (h b))
= { By Rule 1 }
   (equal (g a b) (h b))
= { By Rule 3 }
   (equal (h a) (h b))
```

So, ACL2s will not prove the above conjecture. But, does there exist a proof of the above conjecture?

Yes! For example, if we use rule 2 instead of rule 3 at the last step, then ACL2s will be left with

```
(equal (h b) (h b))
```

which it will rewrite to `t` (using the reflexivity of `equal`). The point is that ACL2s is not a decision procedure for arbitrary properties of programs. In fact, this is an undecidable problem, so there can't be a decision procedure.

7.3.2 Three Rules of Rewriting

Let's consider how we can take what we learned to make effective use of the theorem prover.

Remember the three rules.

1. We saw that ACL2 uses lemmas (theorems) as rewrite rules. Rewrite rules are oriented, *i.e.*, they are applied only from left to right. We saw that theorems of the form

```
(implies ... (equal (foo ...) ...))
```

are used to rewrite occurrences of

```
(foo ...)
```

2. Rules are tried in reverse-chronological order (last first) until one that matches is found. If the rule has hypotheses, then we backchain, trying to discharge those hypotheses. If we discharge the hypotheses, we apply the rule, and recursively rewrite the result.

3. We saw that rewriting proceeds inside-out, *i.e.*, first we rewrite the arguments to a function before rewriting the function.

7.3.3 Pitfalls

Suppose we have both of the following rules.

```
(defthm app-associative
  (implies (and (listp x) (listp y) (listp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))

(defthm app-associative2
  (implies (and (listp x) (listp y) (listp z))
           (equal (app x (app y z))
                  (app (app x y) z))))
```

Ignoring hypotheses for the moment, when we try to rewrite

```
(app x (app y z))
```

We wind up getting into an infinite loop. So notice that you can have non-terminating rewrite rules. ACL2 does not check that rewrite rules are non-terminating, so if you admit the above two rules, you can easily cause ACL2 to chase its tail forever. Non-termination here does not cause unsoundness; all that happens is that ACL2 becomes unresponsive and you have to interrupt it, but non-terminating rewrite rules will *never* allow you to prove `nil`. Contrast this with non-terminating function definitions, which, as we have seen, can lead to unsoundness.

7.3.4 Examples

Suppose that we have the following rule.

```
(defthm app-associative
  (implies (and (listp x) (listp y) (listp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

What does the following get rewritten to?

```
(implies (and (listp a) (listp b) (listp c) (listp d))
```

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

Here is another separate example. Suppose that we have the following two rules.

```
(defthm +-commutative
  (implies (and (rationalp x) (rationalp y))
           (equal (+ x y)
                  (+ y x))))
```

Oops. This seems like a really bad rule. Why?

ACL2 is smart enough to identify rules like this that permute their arguments. It recognizes that they will lead to infinite loops, so it only applies when the term associated with `y` is "smaller" than the term associated with `x`. The details are not relevant. What is relevant is that this does not lead to infinite looping. Also a variable is smaller than another if it comes before it in alphabetical order, and a variable is smaller than a non-variable expression, *e.g.*, `x` is smaller than `(f y)`.

We also have

```
(defthm +-associative
  (implies (and (rationalp x) (rationalp y) (rationalp z))
           (equal (+ (+ x y) z)
                  (+ x (+ y z)))))
```

What does the following get rewritten to?

```
(implies (and (rationalp a) (rationalp b) (rationalp c))
         (equal (+ (+ b a) c)
                (+ a (+ c b))))
```

What does this get rewritten to?

```
(implies (and (rationalp a) (rationalp b) (rationalp c))
         (equal (+ a (+ b c))
                (+ (+ c b) a)))
```

Can you prove using equational reasoning that the above conjecture is true?

Yes, but rewriting does not discover this fact. Because rewriting is directed. What does one do in such situations?

Add another rule, *e.g.*, :

```
(defthm +-swap
  (implies (and (rationalp x) (rationalp y) (rationalp z))
           (equal (+ x (+ y z))
                  (+ y (+ x z)))))
```

Now what happens to the above?

7.3.5 Generalize!

The previous example shows that sometimes we have to add rewrite rules in order to prove conjectures (by rewriting).

As a general rule, we may have many options for adding rewrite rules and we want to do it in the most general way.

For example, suppose that during a proof, we are confronted with the following *stable* subgoal, where a subgoal is stable if none of our current rewrite rules can be applied to any subexpression of the subgoal.

```
(... (app (rev (app x y)) nil) ...)
```

We realize that `(app (rev (app x y)) nil)` is equal to `(rev (app x y))`, so we need a rewrite rule that allows us to simplify the above further. One possibility is:

```
(defthm lemma1
  (implies (and (listp x) (listp y))
           (equal (app (rev (app x y)) nil)
                  (rev (app x y)))))
```

But a better, and more general, lemma is the following.

```
(defthm lemma2
  (implies (listp x)
           (equal (app x nil)
                  x)))
```

Why is the second lemma better? Because given the contracts for `app` and `rev`, any time we can apply lemma1, we can apply lemma2, but not the other way around. That means that lemma2 allows us to simplify more expressions than lemma1.

Part VII

Abstract Data Types and Observational Equivalence

Abstract Data Types and Observational Equivalence

8.1 Abstract Data Types

Let's just jump right in and consider a simple example: stacks.

Think about how you interact with trays in a cafeteria. You can take the top tray (a "pop" operation) and you can add a tray (a "push" operation).

Think about how you respond to interruptions. If you are working on your homework and someone calls, you suspend your work and pick up the phone (push). If someone then knocks on the door, you stop talking and open the door (push). When you finish talking, you continue with the phone (pop), and when you finish that (pop), you go back to your homework.

Think about tracing a recursive function, say the factorial function.

```
(defunc ! (n)
  :input-contract (natp n)
  :output-contract (posp (! n))
  (if (equal n 0)
      1
    (* n (! (- n 1)))))
```

Consider the call `(! 3)`. It involves a call to `(! 2)` (push) which involves a call to `(! 1)` (push) which involves a call to `(! 0)` 0 (push) which returns 1 (pop), which is multiplied by 1 to return 1 (pop) which is multiplied by 2 to return 2 (pop) which is multiplied by 3 to return 6 (pop). If you trace `!`, and evaluate `(! 3)`, ACL2s will show you the stack.

```
(trace* !)
(! 3)
```

So the idea of a stack is that it is a data type that allows several operations, including:

- `stack-push` :add an element to the stack; return the new stack
- `stack-head`: return the top element of a non-empty stack
- `stack-pop`: remove the head of a non-empty stack; return the new stack

We are going to think about stacks in an implementation-independent way. There are two good reasons for doing this. First, a user of our stacks does not have to worry about how stacks are implemented; everything they need to know is provided via a set of operations we provide. Second, we can change the implementation if there is a good reason to do so and, as long as we maintain the guarantees we promised, our changes cannot affect the behavior of the code others have written using our stack library.

If you think about what operations a user might need, you will see that also the following operations are needed.

- `new-stack`: a constructor that creates an empty stack; without this operation, how does a user get their hands on a stack?
- `stackp`: a recognizer for stacks

The above description is still vague, so let's formalize it in ACL2s with an implementation.

We start by defining what elements a stack can hold.

```
(defdata element all)

; Data definition of a stack: a list of elements
(defdata stack (listof element))

; A stack is empty iff it is equal to nil
(defunc stack-emptyp (s)
  :input-contract (stackp s)
  :output-contract (booleanp (stack-emptyp s))
  (equal s nil))

; Stack creation: returns an empty stack
(defunc new-stack ()
  :input-contract t
  :output-contract (and (stackp (new-stack))
                        (stack-emptyp (new-stack)))
  nil)

; The push operation inserts e on the top of the stack s
(defunc stack-push (e s)
  :input-contract (and (elementp e) (stackp s))
  :output-contract (and (stackp (stack-push e s))
                        (not (stack-emptyp (stack-push e s))))
  (cons e s))

; The pop operation removes the top element of a non-empty stack
(defunc stack-pop (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (stackp (stack-pop s))
  (rest s))

; The head of a non-empty stack
(defunc stack-head (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (elementp (stack-head s))
  (first s))
```

While we now have an implementation, we do not have an implementation-independent characterization of stacks. In fact, we will see two such characterizations.

8.2 Algebraic Data Types

The first idea is to characterize stacks using only the algebraic properties they satisfy.

So, what the user of our library will be able to see is:

1. The data definition for element. In fact, almost everything we do below does not depend on the definition of element. However, to use the implementation, a user must know what elements they can push onto the stack. They do not need to see the data definition of a stack, because how we represent stacks is implementation-dependent.
2. The contracts for all the operations given above.
3. The (algebraic) properties that stacks satisfy. These properties include the contract theorems for the stack operations and the following properties:

```
(defthm pop-push
  (implies (and (stackp s)
                (elementp e))
           (equal (stack-pop (stack-push e s))
                  s)))

(defthm stack-head-stack-push
  (implies (and (stackp s)
                (elementp e))
           (equal (stack-head (stack-push e s))
                  e)))

(defthm push-pop
  (implies (and (stackp s)
                (not (stack-emptyp s)))
           (equal (stack-push (stack-head s) (stack-pop s))
                  s)))

(defthm empty-stack-new-stack
  (implies (and (stack-emptyp s)
                (stackp s))
           (equal (new-stack)
                  s)))
```

There are numerous interesting questions we can now ask. For example,

1. How did we determine what these properties should be?
2. Are these properties independent? We can characterize properties as either being *redundant*, meaning that they can be derived from existing properties, or *independent*, meaning that they are not provable from existing properties. How to show redundancy is clear, but how does one show that a property is independent? The answer is to come up with two implementations, one which satisfies the property and one which does not. Since we already have an implementation that satisfies all the properties, to show that some property above is independent of the rest, come up with an implementation that satisfies the rest of the properties, but not the one in question.

3. Are there any other properties that are true of stacks, but that do not follow from the above properties, *i.e.*, are independent?

Exercise 8.1 *Show that the above four properties are independent.*

Exercise 8.2 *Find a property that stacks should enjoy and that is independent of all the properties we have considered so far. Prove that it is independent.*

Exercise 8.3 *Add a new operation* `stack-size`*. Define this in a way that is as simple as possible. Modify the contracts and properties in your new implementation so that we characterize the algebraic properties of* `stack-size`*.*

Exercise 8.4 *Change the representation of stacks so that the size is recorded in the stack. Note that you will have to modify the definition of all the other operations that modify the stack so that they correctly update the size. This will allow us to determine the size without traversing the stack. Prove that this new representation satisfies all of the properties you identified in Exercise 8.3.*

Let's say that this is our final design. Now, the user of our implementation can only depend on the above properties. That also means that we have very clear criteria for how we can go about changing our implementation. We can do so, as long as we still provide exactly the same operations and they satisfy the same algebraic properties identified above.

Let's try to do that with a new implementation. The new implementation is going to represent a stack as a list, but now the head will be the last element of the list, not the first. So, this is a silly implementation, but we want to focus on understanding algebraic data types without getting bogged down in implementation details, so a simple example is best. Once we understand that, then we can understand more complex implementations where the focus is on efficiency. Remember: correctness first, then efficiency.

Try defining the new implementation and show that it satisfies the above properties.

Here is an answer.

```
(defdata element all)

; Data definition of a stack: a list of elements
(defdata stack (listof element))

; A stack is empty iff it is equal to nil
(defunc stack-emptyp (s)
  :input-contract (stackp s)
  :output-contract (booleanp (stack-emptyp s))
  (equal s nil))

; Stack creation: returns an empty stack
(defunc new-stack ()
  :input-contract t
  :output-contract (and (stackp (new-stack))
                        (stack-emptyp (new-stack)))
  nil)

; The push operation inserts e on the top of the stack s
```

```
(defunc stack-push (e s)
  :input-contract (and (elementp e) (stackp s))
  :output-contract (and (stackp (stack-push e s))
                        (not (stack-emptyp (stack-push e s))))
  (app s (list e)))

; The pop operation removes the top element of a non-empty stack
(defunc stack-pop (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (stackp (stack-pop s))
  (rev (rest (rev s))))

; The head of a non-empty stack
(defunc stack-head (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (elementp (stack-head s))
  (first (rev s)))
```

Exercise 8.5 *Provide the lemmas ACL2s needs to admit all of these definitions.*

Exercise 8.6 *Prove that the above implementation of stacks satisfies all of the stack theorems.*

8.3 Observational Equivalence

We now consider yet another way of characterizing stacks.

We will define the notion of an external observation. The idea is that we will define what an external observer of our stack library can see. Such an observer cannot see the implementation of the library, just how the stack library responds to stack operations for a particular stack.

The observer can see what operations are being performed and for each operation what is returned to the user. More specifically below is a list of operations and a description of what the observer can see for each.

1. `stack-emptyp`: what is observable is the answer returned by the library, which is either `t` or `nil`.
2. `stack-push`: what is observable is only the element that was pushed onto the stack (which is the element the user specified).
3. `stack-pop`: If the operation is successful, then nothing is observable. If the operation is not successful, *i.e.*, if the stack is empty, then an error is observable.
4. `stack-head`: If the operation is successful, then the head of the stack is observable, otherwise an error is observable.

If a stack operation leads to a contract violation, then the observer observes the error, and then nothing else. That is, any subsequent operations on the stack reveal absolutely nothing.

Our job now is to define the observer. Use the first definition of stacks we presented above.

First, we start by defining the library operations. Note that they have different names than the functions we defined to implement them.

```
(defdata operation (oneof 'empty? (list 'push element) 'pop 'head))

; An observation is a list containing either a boolean (for
; empty?), an element (for push and head), or nothing (for
; pop). An observation can also be the symbol 'error (pop,
; head).
(defdata observation (oneof (list boolean) (list element) nil 'error))

; We are now ready to define what is externally observable given a
; stack s and an operation.
(defunc external-observation (s o)
  :input-contract (and (stackp s) (operationp o))
  :output-contract (observationp (external-observation s o))
  (cond ((equal o 'empty?)
         (list (stack-emptyp s)))
        ((consp o) (list (second o)))
        ((equal o 'pop) (if (stack-emptyp s) 'error nil))
        (t (if (stack-emptyp s) 'error (list (stack-head s))))))

; Here are some simple tests.
(check= (external-observation '(1 2) '(push 4))
        '(4))
(check= (external-observation '(1 2) 'pop)
        '())
(check= (external-observation '(1 2) 'head)
        '(1))
(check= (external-observation '(1 2) 'empty?)
        '(nil))
```

But we can do better. It should be the case that our code satisfies the following properties. Notice that each property corresponds to an infinite number of tests. `(test? ...)` allows us to test a property. ACL2s can return one of three results.

1. ACL2s proves that the property is true. Note that `test?` does not use induction. In this case, the `test?` event succeeds.

2. ACL2s falsifies the property. In this case, `test?` fails and ACL2s provides a concrete counterexample.

3. ACL2s cannot determine whether the property is true or false. In this case all we know is that ACL2s intelligently tested the property on a specified number of examples and did not find a counterexample. The number of examples ACL2s tries can be specified. A summary of the analysis is reported and the `test?` event succeeds.

```
(test? (implies (and (stackp s) (elementp e))
                (equal (external-observation s (list 'push e))
                       (list e))))

(test? (implies (and (stackp s)
                     (not (stack-emptyp s)))
                (equal (external-observation s 'pop)
                       nil)))

(test? (implies (and (stackp s)
                     (stack-emptyp s))
                (equal (external-observation s 'pop)
                       'error)))

(test? (implies (and (stackp s)
                     (stack-emptyp s))
                (equal (external-observation s 'head)
                       'error)))

(test? (implies (stackp s)
                (equal (external-observation (stack-push e s) 'head)
                       (list e))))

(test? (implies (and (stackp s)
                     (not (stack-emptyp s)))
                (equal (external-observation s 'empty?)
                       (list nil))))

(test? (implies (and (stackp s)
                     (stack-emptyp s))
                (equal (external-observation s 'empty?)
                       (list t))))

; Now we want to define what is externally observable for a
; sequence of operations. First, let's define a list of operations.
(defdata lop (listof operation))

; Next, let's define a list of observations.
(defdata lob (listof observation))

; Now, let's define what is externally visible given a stack s
; and a list of observations.

(defunc update-stack (s op)
  :input-contract (and (stackp s) (operationp op))
  :output-contract (stackp s)
  (cond ((or (equal op 'empty?) (equal op 'head))
         s)
        ((equal op 'pop) (if (stack-emptyp s) nil (stack-pop s)))
```

```
        (t (stack-push (second op) s))))

(defunc external-observations (s l)
  :input-contract (and (stackp s) (lopp l))
  :output-contract (lobp (external-observations s l))
  (if (endp l)
      nil
    (let* ((op (first l))
           (ob (external-observation s op)))
      (if (equal ob 'error)
          '(error)
        (cons ob (external-observations (update-stack s op) (rest l)))))))

; Here are some instructive tests.
(check= (external-observations
         (new-stack)
         '(head ))
        '(error))

(check= (external-observations
         (new-stack)
         '( (push 1) pop (push 2) (push 3)
            pop head empty? pop empty? ))
        '( (1) () (2) (3) () (2) (nil) () (t) ))

(check= (external-observations
         (new-stack)
         '( (push 1) pop pop pop empty? ))
        '( (1) () error))

(check= (external-observations
         (new-stack)
         '( (push nil) (push error) (push pop) empty? head pop
            empty? head pop empty? head pop empty? head pop))
        '( (nil) (error) (pop) (nil) (pop) () (nil) (error) ()
         (nil) (nil) () (t) error))
```

Exercise 8.7 *What happens when we use a different implementation of stacks?*

Suppose that we use the second implementation of stacks we considered. Then, we would like to prove that an external observer cannot distinguish it from our first implementation.

Prove this.

Exercise 8.8 *Prove that the implementation of stacks from Exercise 8.4 is observationally equivalent to the above implementation, as long as the observer cannot use* `stack-size`*. This shows that users who do not use* `stack-size` *operation cannot distinguish the stack implementation from Exercise 8.4 with our previous stack implementations.*

Exercise 8.9 *Prove that the implementation of stacks from Exercise 8.4 is observationally equivalent to the implementation of stacks from Exercise 8.3. Extend the observations that can be performed to account for* `stack-size`*.*

8.4 Queues

We will now explore queues, another abstract data type.

Queues are related to stacks. Recall that in a stack we can push and pop elements. Stacks work in a LIFO way (last in, first out): what is popped is what was most recently pushed. Queues are like stacks, but they work in a FIFO way (first in, first out). A queue then is like a line at the bank (or the grocery store, or an airline terminal, ...): when you enter the line, you enter at the end, and you get to the bank teller when everybody who came before you is done.

Let's start with an implementation of a queue, which is going to be similar to our implementation of a stack.

```
; A queue is a true-list (like before, with stacks)
(defdata element all)

(defdata queue (listof element))

; A queue is empty iff it is nil
(defunc queue-emptyp (q)
  :input-contract (queuep q)
  :output-contract (booleanp (queue-emptyp q))
  (equal q nil))

; A new queue is just the empty list
(defunc new-queue ()
  :input-contract t
  :output-contract (and (queuep (new-queue))
                        (queue-emptyp (new-queue)))
  nil)

; The head of a queue. Let's decide that the head of the queue
; will be the first.
(defunc queue-head (q)
  :input-contract (and (queuep q) (not (queue-emptyp q)))
  :output-contract (elementp (queue-head q))
  (first q))

; Dequeueing can be implemented with rest
(defunc queue-dequeue (q)
  :input-contract (and (queuep q) (not (queue-emptyp q)))
  :output-contract (queuep (queue-dequeue q))
  (rest q))

; Enqueueing to a queue requires putting the element at the
; end of the list.
(defunc queue-enqueue (e q)
  :input-contract (and (elementp e) (queuep q))
  :output-contract (and (queuep (queue-enqueue e q))
(not (queue-emptyp (queue-enqueue e q))))
  (app q (list e)))
```

We're done with this implementation of queues.

Instead of trying to prove a collection of theorems that hold about queues, we are going to define another implementation of queues and will show that the two implementations are observationally equivalent.

We'll see what that means in a minute, but first, let us define the second implementation of queues. The difference is that now the head of the queue will be the first element of a list. We will define a new version of all the previous queue-functions.

```
(defdata element2 all)

(defdata queue2 (listof element2))

; A queue2 is empty iff it satisfies endp
(defunc queue2-emptyp (q)
  :input-contract (queue2p q)
  :output-contract (booleanp (queue2-emptyp q))
  (equal q nil))

; A new queue2 is just the empty list
(defunc new-queue2 ()
  :input-contract t
  :output-contract (and (queue2p (new-queue2))
                        (queue2-emptyp (new-queue2)))
  nil)

; The head of a queue2 is now the last element of the list
; representing the queue2. What's a simple way of getting our
; hands on this? Use rev.
(defunc rev (x)
  :input-contract (listp x)
  :output-contract (listp (rev x))
  (if (endp x)
    nil
    (app (rev (rest x)) (list (first x)))))

; Here are the basic theorems about rev that we already
; established.

(defthm rev-app
  (implies (and (listp x) (listp y))
           (equal (rev (app x y))
                  (app (rev y) (rev x))))
  :hints (("goal" :induct (listp x))))

(defthm rev-rev
  (implies (listp x)
           (equal (rev (rev x))
                  x)))

; The head of a queue2 is the last element in q
```

```
(defunc queue2-head (q)
  :input-contract (and (queue2p q) (not (queue2-emptyp q)))
  :output-contract (element2p (queue2-head q))
  (first (rev q)))

; Dequeueing (removing) can be implemented as follows. Recall that
; in this implementation, the first element of a queue2 is the last
; element of the list. Also, we don't care about efficiency at
; this point. We can make it more efficient later. We care about
; specification.
(defunc queue2-dequeue (q)
  :input-contract (and (queue2p q) (not (queue2-emptyp q)))
  :output-contract (queue2p (queue2-dequeue q))
  (rev (rest (rev q))))

; Enqueueing (adding an element to a queue2) can be implemented
; with cons. Note that the last element of a queue2 is at the
; front of the list.
(defunc queue2-enqueue (e q)
  :input-contract (and (element2p e) (queue2p q))
  :output-contract (and (queue2p (queue2-enqueue e q))
(not (queue2-emptyp (queue2-enqueue e q))))
  (cons e q))
```

Let's see if we can prove that the two implementations are equivalent. To do that, we are going to define what is observable for each implementation.

```
; We start with the definition of an operation.
; 'e? is the empty check, 'e is enqueue, 'h is head
; and 'd is dequeue
(defdata operation (oneof 'e? (list 'e element) 'h 'd))

; Next, we define a list of operations.
(defdata lop (listof operation))

; An observation is a list containing either a boolean (for
; e?), an element (for 'e and 'h), or nothing (for
; 'd). An observation can also be the symbol 'error ('h, 'd).

(defdata observation (oneof (list boolean) (list element) nil 'error))

; Finally, we define a list of observations.
(defdata lob (listof observation))

; Now we want to define what is externally observable given a
; sequence of operations and a queue. It turns out we need a
; lemma for ACL2s to admit queue-run. How we came up with the
; lemma is not important. (But in case it is useful, there was a
; problem proving the contract of queue-run, so I admitted it
; with the output-contract of t and then tried to prove the
```

```
; contract theorem and noticed (using the method) what the
; problem was).

(defthm queue-lemma
  (implies (queuep q)
           (queuep (app q (list x)))))

(defunc queue-run (l q)
  :input-contract (and (lopp l) (queuep q))
  :output-contract  (lobp (queue-run l q))
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
             (if (queue-emptyp q)
                 (list 'error)
               (cons nil (queue-run (rest l) (queue-dequeue q)))))
            ((equal i 'h)
             (if (queue-emptyp q)
                 (list 'error)
               (cons (list (queue-head q)) (queue-run (rest l) q))))
            ((equal i 'e?)
             (cons (list (queue-emptyp q)) (queue-run (rest l) q)))
            (t (cons (list (second i))
                     (queue-run (rest l) (queue-enqueue (second i) q))))))))

; Now we want to define what is externally observable given a
; sequence of operations and a queue2. We need a lemma, as
; before. (It was discovered using the same method).

(defthm queue2-lemma
  (implies (queue2p q)
           (queue2p (rev (rest (rev q))))))

(defunc queue2-run (l q)
  :input-contract (and (lopp l) (queue2p q))
  :output-contract (lobp (queue2-run l q))
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
             (if (queue2-emptyp q)
                 (list 'error)
               (cons nil (queue2-run (rest l) (queue2-dequeue q)))))
            ((equal i 'h)
             (if (queue2-emptyp q)
                 (list 'error)
               (cons (list (queue2-head q)) (queue2-run (rest l) q))))
            ((equal i 'e?)
```

```
            (cons (list (queue2-emptyp q)) (queue2-run (rest l) q)))
           (t (cons (list (second i))
                    (queue2-run (rest l) (queue2-enqueue (second i) q))))))))
```

```
; Here is one test.

(check=
 (queue-run '( e? (e 0) (e 1) d h (e 2) h d h) (new-queue))
 (queue2-run '( e? (e 0) (e 1) d h (e 2) h d h) (new-queue2)))
```

But, how do we prove that these two implementations can never be distinguished? What theorem would you prove?

```
(defthm observational-equivalence
  (implies (lopp l)
           (equal (queue2-run l (new-queue2))
                  (queue-run l (new-queue)))))
```

But, we can't prove this directly. We have to generalize. We have to replace the constants with variables. How do we do that?

First, note that we cannot replace (new-queue2) and (new-queue) with the same variable because they are manipulated by different implementations. Another idea might be to use two separate variables, but this does not work either because they have to represent the same abstract queue. The way around this dilemma is to use two variables but to say that they represent the same abstract queue. The first step is to write a function that given a queue2 queue returns the corresponding queue.

```
(defunc queue2-to-queue (q)
  :input-contract (queue2p q)
  :output-contract (queuep (queue2-to-queue q))
 (rev q))

; We need a lemma

(defthm queue2-queue-rev
  (implies (queue2p x)
           (queuep (rev x))))

; Here is the generalization.

(defthm observational-equivalence-generalization
  (implies (and (lopp l)
                (queue2p q2)
                (equal q (queue2-to-queue q2)))
           (equal (queue2-run l q2)
                  (queue-run l q))))

; Now, the main theorem is now a trivial corollary.

(defthm observational-equivalence
  (implies (lopp l)
           (equal (queue2-run l (new-queue2))
                  (queue-run l (new-queue)))))
```