Reasoning About Programs

Panagiotis Manolios
Northeastern University

September 13, 2021
Version: 121

Contents

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` and `definec` macros, which allow 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-lists` 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

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. [1]

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

As the above equation shows, if `f` is an ACL2s function of n arguments, and we want to apply it to $x_1, x_2, \ldots, x_n$, we write $(\texttt{f}\ x_1\ x_2 \cdots x_n)$. [2] Almost no other language satisfies the very strict rule of Leibniz, *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 Racket, 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. To reason about other languages, one typically generates an intermediate representation that is functional, as we will see later.

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 !>
```

You can now type in an expression, say

```
ACL2S !>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 !>
```

We recommend that as you read these notes, you also have ACL2s installed and follow along in the "ACL2S" mode. The prompt indicates that ACL2s is in the "ACL2S" mode.

[1] The term *functional programming language* has many different definitions, *e.g.*, it is often used to refer to languages in which functions are treated as first-class objects. ACL2s is not functional in this sense.

[2] This notation is used by Lisp-based languages, such as ACL2s.

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.*
- Characters: For example, `#\a`, `#\B`, `#\Space`.
- Strings: For example, `"Hello world"`.
- Conses: For example, `(1)`, `(1 2 3)`, `(cons 1 2)`, `(1 (1 2) 3)`.

The Rationals, Symbols, Characters, Strings 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 are used 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

$$[\![\mathit{expr}]\!]$$

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:

$$\texttt{if} : \mathsf{All} \times \mathsf{All} \times \mathsf{All} \rightarrow \mathsf{All}$$

The signature of `if` tells us that `if` takes three arguments, where all three 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 2 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 is *not* an expression.

```
(if t nil)
```

The list above does not satisfy the signature of `if`, which tells us that the function has an arity of three. 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.

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

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. For example, as we have seen `(if t nil)` is not an expression. If you try to evaluate it you will get an error message indicating that `if` requires three arguments.

We will almost always follow the convention that the first argument to `if` is a Boolean. Later on, we will see examples of how the ability to use non-Booleans in the test of an `if`, often referred to as *generalized Booleans*, can be useful.

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 $[\![\textit{test}]\!] \neq \texttt{nil}$, then ACL2s returns $[\![\textit{then}]\!]$.
3. Otherwise, it returns $[\![\textit{else}]\!]$.

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:

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

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

$$[\![\texttt{(if 2 3 4)}]\!] = 3$$

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. These functions are already predefined in ACL2s, but if you want to follow along and experiment, define them using a different function name.

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

$$\mathsf{All} \to \mathsf{Boolean}$$

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.*, Racket and Scheme use "`?`" (pronounced "huh") instead of "`p`."

Here is one way to 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.

```
(definec booleanp (x ...) ...
  (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.

In ACL2s, functions have input and output contracts. Contracts allow us to place constraints on arguments to functions and to make claims about the outputs generated by functions. Contracts are checked by ACL2s. A simple kind of contract that we will always use, is to specify the types of inputs and outputs. ACL2s has a powerful type system provided by `defdata`, so just the use of types gives us significant expressive power, but contracts in ACL2s are much more powerful. Types and contacts will be explained in this chapter.

What is the input type for `booleanp`?

It is `all` because there are no constraints on the input to the function. All *recognizers* will have an input type of `all`. A recognizer is a function that given any element of the

ACL2s universe recognizes whether it belongs to a particular subset of the universe. In the case of `booleanp`, the subset being recognized is the set of Booleans, *i.e.*, {`t`, `nil`}.

What about the output contract? Since `booleanp` is a recognizer it returns a Boolean! So, all together we have:

```
(definec booleanp (x :all) :boolean
  (if (equal x t)
      t
    (equal x nil)))
```

In addition to `definec`, ACL2s provides `defunc`, a lower-level utility for defining functions with contracts. In fact, `definec` is defined in terms of `defunc`. Here is how we could have defined `booleanp` using `defunc`.

```
(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.

$$ic \text{ implies } oc$$

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 ⇒ (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`. Notice that an equivalent way to write the input contract for the `defunc` version of `booleanp` is the following

```
  :input-contract (allp x)
```

because `(allp x)` is always `t`, so these two contracts are equivalent. Notice that in `defunc` contracts, we use recognizers (*e.g.*, `allp` and `booleanp`), but in `definec`, we use the corresponding type names (*e.g.*, `all` and `boolean`).

Let us continue with more basic definitions. To simplify code, ACL2s allows one to write `:bool` instead of `:boolean` in `definec` forms and `boolp` instead of `booleanp`.

Here is one way to define the other Boolean functions. This is *not* how they are really defined, as we will shortly see, but it is worth assuming, for now, that the functions are defined as shown below.

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

```
(definec and (a :bool b :bool) :bool
  (if a b nil))
```

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

```
(definec implies (a :bool b :bool) :bool
  (if a b t))
```

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

```
(definec or (a :bool b :bool) :bool
  (if a t b))
```

How do we evaluate expressions involving the above functions? Simple:

$\llbracket\texttt{(or t nil)}\rrbracket$

$= \{$ Definition of `or` $\}$

$\llbracket\texttt{(if t t nil)}\rrbracket$

$= \{$ Semantics of `if` $\}$

If $\llbracket\texttt{t}\rrbracket = \texttt{nil}$ then $\llbracket\texttt{nil}\rrbracket$ else $\llbracket\texttt{t}\rrbracket$

$= \{$ Constants evaluate to themselves $\}$

If $\texttt{t} = \texttt{nil}$ then `nil` else `t`

$= \{$ `t` is not `nil` $\}$

`t`

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

$$\texttt{not} : \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(definec not (a :bool) :bool
  (if a nil t))
```

$$\texttt{iff} : \mathsf{Boolean} \times \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(definec iff (a :bool b :bool) :bool
  (if a b (not b)))
```

$$\texttt{xor} : \mathsf{Boolean} \times \mathsf{Boolean} \to \mathsf{Boolean}$$

```
(definec xor (a :bool b :bool) :bool
  (if a (not b) b))
```

2.3 Numbers

We have the following built-in recognizers:

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

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

Here is what they mean.
$[\![\texttt{(integerp x)}]\!]$ is `t` iff $[\![\text{x}]\!]$ is an integer.
$[\![\texttt{(rationalp x)}]\!]$ is `t` iff $[\![\text{x}]\!]$ 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. ACL2s also includes other types of numbers, but for our purposes, we will assume that all numbers are rationals.
We also have the following functions.

$$\texttt{binary-+} : \textsf{Rational} \times \textsf{Rational} \to \textsf{Rational}$$

$$\texttt{binary-*} : \textsf{Rational} \times \textsf{Rational} \to \textsf{Rational}$$

$$\texttt{<} : \textsf{Rational} \times \textsf{Rational} \to \textsf{Boolean}$$

$$\texttt{unary--} : \textsf{Rational} \to \textsf{Rational}$$

$$\texttt{unary-/} : \textsf{Rational} \to \textsf{Rational}$$

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

$$\texttt{unary-/} : \textsf{Rational} \setminus \{0\} \to \textsf{Rational}$$

How do we express this kind of thing? Well, we have a type, `non-0-rational` corresponding to non-zero rationals. Here is how the recognizer is defined.

```
(definec non-0-rational (x :all) :boolean
  (and (rationalp x)
       (not (equal x 0))))
```

Now we can express the contracts using `definec`.

```
(definec unary-/ (a :non-0-rational) :rational
  ...)
```

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

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

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

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

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

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

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.

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

```
(definec posp (a :all) :bool
  (if (integerp a)
      (< 0 a)
    nil))
```

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

```
(definec posp (a :all) :bool
  (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`.

It turns out to be really useful if Boolean functions such as `and` and `or` are just abbreviations for `if`. This is actually the case. These Boolean "functions" are really *macros* that get expanded into `if` expressions. There is a lot to say about macros, but for our purposes, all we need to know is that macros give us abbreviation power. They also allow `and` and `or` to 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)`

This makes writing contracts simpler. We also have macros that allow us to use `=>` and `!` as abberviations for `implies` and `not`, respectively.

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 (< -6 x) (integerp x))
  :output-contract (booleanp (>5 x))
  (< 5 x))
```

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) (< -6 x))
  :output-contract (booleanp (>5 x))
  (< 5 x))
```

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 > -6. This assumption allows contract checking to pass on `(> 5)`. Contract checking also passes on `booleanp`, as it has an input contract of `t`.

We now reveal some new `definec` capabilities. The above definition of `>5` using `defunc` can also be written using `definec` in any of the following equivalent ways.

```
(definec >5 (x :all) :all
  :input-contract (and (integerp x) (< -6 x))
  :output-contract (booleanp (>5 x))
  (< 5 x))

(definec >5 (x :all) :all
  :input-contract (and (integerp x) (< -6 x))
  :output-contract :bool
  (< 5 x))

(definec >5 (x :all) :bool
  :input-contract (and (integerp x) (< -6 x))
  (< 5 x))

(definec >5 (x :int) :bool
  :input-contract (< -6 x)
  (< 5 x))
```

The last example above highlights why a good style that we will use throughout this book is to always use the strongest type in the keyword types appearing in a `definec`, *e.g.*, `(x :int)` instead of `(x :all)`, and `:bool` instead of `:all`. When `definec` processes input contracts, it does so in the order in which they appear, and remember we want recognizers to come first to avoid contract checking failures in the input and output contracts. We will also avoid using `:input-contract` and `:output-contract` forms in `definec` because they can often be avoided by defining types using `defdata` and they have consequences that will only become apparent once we discuss how the theorem proving engine works. For readers familiar with term-rewriting, they generate rewrite rules so considerations include the orientation of the rules and how they interact with existing rewrite rules.

Just like `and` and `or` are macros, ACL2s also provides the macros `*` and `+` that take an arbitrary number of arguments. For example,

1. `(*)` abbreviates `1`
2. `(* x)` abbreviates `(binary-* 1 x)`
3. `(* x y)` abbreviates `(binary-* x y)`
4. `(* x y z)` abbreviates `(binary-* x (binary-* y z))`
5. `(+)` abbreviates `0`
6. `(+ x)` abbreviates `(binary-+ 0 x)`
7. `(+ x y)` abbreviates `(binary-+ x y)`
8. `(+ x y z)` abbreviates `(binary-+ x (binary-+ y z))`

and so on. In addition, ACL2s provides the macros `<=`, `>`, `>=`, which are defined in terms of `<`.

Other ACL2s macros include the following.

1. `(== x y)` abbreviates `(equal x y)`
2. `(!= x y)` abbreviates `(not (equal x y))`

ACL2s also includes the functions `=` and `/=` which are equivalent to `==` and `!=`, but with input contracts that only allow the arguments to be numbers (rationals). Be careful between our use of $=$ (mathematical equality) and `=` (the ACL2s function).

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

We also have built-in

$$\texttt{numerator} : \mathsf{Rational} \to \mathsf{Integer}$$

$$\texttt{denominator} : \mathsf{Rational} \to \mathsf{Pos}$$

$[\![$`(numerator a)`$]\!]$ is the numerator of the number we get after simplifying $[\![\mathsf{a}]\!]$.
$[\![$`(denominator a)`$]\!]$ is the denominator of the number we get after simplifying $[\![\mathsf{a}]\!]$.
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 $gcd(|\mathsf{x}|, \mathsf{y})$ to obtain `a/b`. If $\mathsf{b} = 1$, 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.*,

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

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

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

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.

```
(definec even-natp (x :nat) :bool
  (or (= x 0)
      (! (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.*:

```
(definec even-natp (x :nat) :bool
  (natp (/ x 2)))
```

This is not a data-driven definition. It required more insight. In fact, in ACL2s mode, the functions `evenp` and `oddp` are built-in and operate on integers.

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.

$$\mathit{Int} : 0 \mid \mathit{Int} + 1 \mid \mathit{Int} - 1$$

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.

```
(definec even-integerp (x :int) :bool
  (cond ((= x 0) t)
        ((< x 0) (! (even-integerp (+ x 1))))
        (t (! (even-integerp (- x 1))))))
```

Note that ACL2s allows one to write `:int` instead of `:integer` in `definec` forms. The above function uses the function `zip`, whose signature is

$$\mathsf{Int} \to \mathsf{Bool}$$

and which returns `t` iff its input is equal to 0. Two related functions that have the same behavior but different signatures are `zp` and `zerop`.

$$\texttt{zp} : \textsf{Nat} \rightarrow \textsf{Bool}$$

$$\texttt{zerop} : \textsf{Rational} \rightarrow \textsf{Bool}$$

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.

2.4 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. ACL2s also includes other types of atoms, *e.g.*, complex numbers, but we do not use them in these notes.

2.5 Conses and Lists

Conses and lists allow us to create non-atomic data.

Our first built-in function is a recognizer for *conses*.

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

To create a Cons, we use the built-in function `cons`.

$$\texttt{cons} : \textsf{All} \times \textsf{All} \rightarrow \textsf{Cons}$$

A Cons is just a pair and ACL2s displays conses using *dot notation*, *e.g.*, `(cons 1 2)` is displayed as `(1 . 2)`.

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

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

```
(definec listp (l :all) :bool
  (or (consp l)
      (== l () )))
```

Here are the built-in functions for accessing the components of a Cons.[3]

$$\texttt{car} : \mathsf{List} \rightarrow \mathsf{All}$$

$$\texttt{cdr} : \mathsf{List} \rightarrow \mathsf{All}$$

A special type of list is a "true list." We will use TP to denote the objects recognized by the following function.

```
(definec true-listp (l :all) :bool
  (if (consp l)
      (true-listp (cdr l))
    (== l () )))
```

So, true lists are lists whose last `cdr` is `nil`. Since true lists are so widely used, ACL2s allows one to use `:tl` in `definec` forms to denote true lists (in addition to `:true-list`). Also one can write `tlp` instead of `true-listp`. Finally, ACL2s allows the use of `first` and `rest` instead of `car` and `cdr`. The general guideline is that when operating on true lists, we will use `first` instead of `car` and `rest` instead of `cdr`.

ACL2s has two rules that allow us to write cons pairs in a variety of ways. The first rule allows us to write the `(1 . nil)` as `(1)`: if a cons pair has the symbol `nil` as its cdr, you can drop the "dot" and the `nil`.

The second rule allows us to write `(0 . (1))` as `(0 1)`: if the cdr of a cons is also a cons, you can drop the dot and the balanced pair of parentheses following it. So, `(`x` . (`$\ldots$`))` can be written as `(`x $\ldots$`)`. When ACL2s displays conses, it first simplifies the dot notation using the above rules as many times as possible. For, the cons `(1 . ((2 . (3 . nil)) . (4 . nil)))` is displayed as `(1 (2 3) 4)`.

The semantics of the built-in functions is given by the following rules.

$$[\![\texttt{(cons x y)}]\!] = \texttt{(}[\![\mathsf{x}]\!]\ \texttt{.}[\![\mathsf{y}]\!]\texttt{)}$$

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

Notice that since `consp` is a recognizer it returns a Boolean. So, if $[\![\mathsf{x}]\!]$ 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 `car` and `cdr` is given with the following rules.

[3]The names `car` and `cdr` are related to the original implementation of Lisp on the IBM 704 and stand for "Contents of the Address Register" and the "Contents of the Decrement Register."

$$[\![\texttt{(car x)}]\!] = a,\ \text{when } [\![\texttt{x}]\!] = \texttt{(}a\ \texttt{.}\ b\texttt{)} \text{ and } \texttt{nil} \text{ otherwise}$$
$$[\![\texttt{(cdr x)}]\!] = b,\ \text{when } [\![\texttt{x}]\!] = \texttt{(}a\ \texttt{.}\ b\texttt{)} \text{ and } \texttt{nil} \text{ otherwise}$$

Here are some examples.

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

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

```
(== (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. In general

$\texttt{(list x}_1\ \texttt{x}_2\ \cdots\ \texttt{x}_n\texttt{)}$

abbreviates (or is shorthand for)

$\texttt{(cons x}_1\ \texttt{(cons x}_2\ \cdots\ \texttt{(cons x}_n\ \texttt{nil)}\ \cdots\texttt{))}$

Remember `tlp`, `first`, `rest` and `boolp`? They are also macros that just expand into `true-listp`, `car`, `cdr` and `booleanp`, respectively.

Sometimes we want versions of `car` and `cdr` that have stronger input contracts because we only ever expect their arguments to be conses and if their inputs are `nil`, that should be an error. Hence the following definitions.

```
(definec left (x :cons) :all
  (car x))

(definec right (x :cons) :all
  (cdr x))
```

The names `left` and `right` indicate that we selecting the left and right part of a cons pair, respectively.

Here is yet another variant in which we require that the arguments are non-empty true lists.

```
(definec head (x :ne-tl) :all
  (car x))

(definec tail (x :ne-tl) :tl
  (cdr x))
```

The type `non-empty-true-list` corresponds to non-empty true lists. We can also use the more abbreviated, but equivalent, `ne-tl`.

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.

```
(definec foo (a :int) :bool
  (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. Section 2.20 explores static vs dynamic checking in more depth.

2.7 Termination

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

For example, consider the following "definition."

```
(definec my-listp (a :all) :bool
  (if (consp a)
      (my-listp a)
    (== 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
(definec sum-n (n :int) :int
  (if (zerop n)
      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 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.

The function `atom` checks if its argument is an Atom, a non-Cons.

```
(definec atom (l :all) :bool
  (! (consp l)))
```

Now consider `endp`, a function that checks if a list is empty.

```
(definec endp (l :list) :bool
  (atom l))
```

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

Here is yet another variant.

```
(definec lendp (x :tl) :bool
  (atom x))
```

The functions `atom`, `endp` and `lendp` have the equivalent output contracts and bodies. `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 multiple 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; similarly we should use `lendp` when checking true lists. If we make a mistake, then by using `endp` (or `lendp`), 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.

Support for stricter checking when constructing conses is provided by the following function.

```
(definec lcons (x :all y :tl) :ne-tl
  (cons x y))
```

The `llen` function returns the length of a TL. This is the simplest example of a data-driven definition, so let's try defining it, with the caveat that the actual ACL2s definition, provided later, is syntactically different. The idea is to define `llen` using a template derived from the contract of its input variable, which is a TL. A TL is either empty or consists of the `first` element of the TL and the `rest` of the TL. Therefore our template also has two cases and in the second case, we can assume that `llen` returns the correct answer on the `rest` of the list, so all that is left is to add one to the answer. If we want to use the strictest functions, which is usually a good idea, we can use `lendp` and `tail` instead of `endp` and `rest`, respectively.

```
(definec llen (l :tl) :nat
  ; Returns the length of tl l.
  (if (lendp l)
      0
    (1+ (llen (tail 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.

ACL2s also provides the function `len` which takes anything as input and behaves the same as as `llen` on true lists, but also returns the length of conses that are not necessarily true lists.

```
(definec len (l :all) :nat
  ; Returns the length of l.
  (if (consp l)
      (1+ (len (cdr l)))
    0))
```

The `app` function appends two lists together. Let's try defining it using a data-driven definition, with the caveat that the actual ACL2s definition, which is provided later, is syntactically different. 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 `llen`, 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.

$$\texttt{app () Y} = \texttt{Y}$$

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

$$\texttt{app (cons a B) Y} = \texttt{aBY} = \texttt{cons a (app B Y)}$$

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`.

```
(definec app (x :tl y :tl) :tl
  ; App appends two lists together
  (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.

$$\texttt{app X ()} = \texttt{X}$$

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

$$\texttt{app X (cons a B)} = \texttt{XaB} = \texttt{(app (?? X a) B)}$$

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.

ACL2s has a function `binary-append` which allows the second argument to be anything and is defined as follows.

```
(definec binary-append (x :tl y :all) :all
  (if (endp x)
      y
    (cons (first x) (binary-append (rest x) y))))
```

It also has the macro `append` which take an arbitrary number of arguments. For example

```
(check= (append (list 1) (list 2) (list) (list 3)) (list 1 2 3))
```

Remember the caveats? In ACL2s, `app` is a macro which expands into `bin-app`, where `bin-app` is defined as follows.

```
(definec bin-app (x :tl y :tl) :tl
  (append x y))
```

The definitions of `binary-append` and `bin-app` are examples of a common pattern: when we have functions that differ only in their contracts we define the version with the most general contracts first and to use this version to define the rest of the functions. This allows us to prove reason about all the functions by proving theorems only for the general version; to reason about the other variants, we only use contract reasoning to expand the definitions into the most general version. This pattern is also used to reason about semantically equivalent functions, which comes up when reasoning about algorithms with different complexity. In ACL2s, `len` is defined as above and `llen` is defined using the pattern just described.

```
(definec llen (x :tl) :nat
  (len x))
```

We will see more examples of this pattern later.

The `rev` function reverses its argument.

```
(definec rev (x :all) :tl
  (if (consp x)
      (append (rev (cdr x)) (list (car x)))
    ()))
```

The `lrev` function reverses a true list and we define it using the previously mentioned pattern.

```
(definec lrev (x :tl) :tl
  (rev x))
```

2.9 Contracts, Part 1

Even though contracts can be quite complicated, we will mostly restrict the use of contracts by using `definec` as opposed to `defunc`.

Notice that `defunc` definitions of the following form can be written using only types with `definec`.

```
(defunc f ($x_1$ ... $x_n$)
  :input-contract t | ($R_1$ $x_i$) | (and ($R_1$ $y_1$) ... ($R_m$ $y_m$))
  :output-contract ($R$ (f $x_1$ ... $x_n$))
  ...)
```

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. This same argument applies to `definec` definitions.

2.10 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 `lrev` (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`.

2.11 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.

In fact, `let*` is really equivalent to nested `let` forms, *e.g.*,

```
(let* ((v1 x1)
       (v2 x2)
       ...
       (vn xn))
  body)
```

is equivalent to the following.

```
(let ((v1 x1))
  (let ((v2 x2))
    ...
    (let ((vn xn))
      body) ...))
```

2.12 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 ((expt 2 64) < _)))
```

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))
```

An equivalent way of writting the above is:

```
(defdata intstr (or integer string))
```

Recursive type expressions involve the `oneof` (or the equivalent `or`) combinator and product combinators, 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.

2.13 Properties

Instead of

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

we can write

```
(test? (== (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? (=> (natp n)
           (== (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? (=> 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? (=> (and (integerp n)
                 (< n 0))
           (== (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`:

```
(definec even-integerp (x :int) :bool
  (if (natp x)
      (even-natp x)
    (even-natp (* x -1))))
```

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

```
(test? (=> (and (natp n) (< 20/3 n))
           (== (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? (=> (< 20/3 n)
           (== (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 Contracts, Part 2

We can write more complicated contracts using `definec`. For example, here is a version of `app` with a stronger output contract.

```
(definec sapp (x :tl y :tl) :tl
  :output-contract (= (llen (sapp x y)) (+ (llen x) (llen y)))
  (if (lendp x)
      y
    (lcons (head x) (sapp (tail x) y))))
```

In lieu of `:input-contract`, you can use `:ic`, `:pre-condition`, `:pre`, `:require` or `:assume`. Similarly, you can use `:oc`, `:post-condition`, `:post`, `:ensure` or `:guarantee` in lieu of `:output-contract`. What to use is a matter of personal preference. Multiple input and output contract forms are also allowed and they are combined using conjunction, *i.e.*, `and`. For example, here is an ugly, but equivalent, way of defining `sapp`.

```
(definec sapp (x :all y :all) :all
  :ic (tlp x)
  :pre (tlp y)
  :post (tlp (sapp x y))
  :output-contract (= (llen (sapp x y)) (+ (llen x) (llen y)))
  (if (lendp x)
      y
    (lcons (head x) (sapp (tail x) y))))
```

Even though `definec` and `defunc` allow us to write complex output contracts, as shown above, until you read Chapter 7, the *output contracts* of all functions you define should consist of only a single recognizer. This is important because more complex output contracts lead to rewrite rules that program the theorem prover and if you do not understand how the theorem prover uses rewrite rules, you will wind up cause infinite rewrite loops. So, how should we define `sapp` if we want to make it clear that the length of the output is equal to the sum of the lengths of its inputs? By using properties, as shown below.

```
(definec sapp (x :tl y :tl) :tl
  (if (lendp x)
      y
    (lcons (head x) (sapp (tail x) y))))
```

The property can be specified using `test?` or `thm`.

```
(thm (=> (and (tlp x) (tlp y))
         (= (llen (sapp x y))
            (+ (llen x) (llen y)))))
```

By the way, the restriction above only applies to output contracts, not input contracts, *e.g.*, if we wanted to define a version of `app` that requires both inputs to have the same length, here is how we can do that.

```
(definec sapp (x :tl y :tl) :tl
  :ic (= (len x) (len y))
  (app x y))
```

We can say more out the output and we use a property to do so.

```
(thm (=> (and (tlp x) (tlp y) (= (len x) (len y)))
         (= (llen (sapp x y))
            (* 2 (llen x)))))
```

2.15 Concrete Testing

Consider the following function definition.

```
(definec foo (x :nat y :nat z :tl) ...
  ...)
```

How should we test this definition? We consider this question in more detail in this section. We start by considering *concrete tests* for `foo`, by which we mean tests of the form `(check (foo ...) ...)`. Later, we discuss more powerful kinds of tests, *symbolic tests*, which are based on `test?`. In this section, unless we say otherwise, test will mean concrete test.

The first method we discuss is *contract-based testing.* The idea is to use the input contract, which specifies the types of the inputs, to generate tests. For our example, contract-based testing indicates that we 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. Here is a list tests that provides complete contract-based testing for `foo`.

```
(check= (foo 0 0 nil) ...)
(check= (foo 0 0 '(a b)) ...)
(check= (foo 0 1 nil) ...)
(check= (foo 0 1 '(a b) ...)
(check= (foo 1 0 nil) ...)
(check= (foo 1 0 '(a b)) ...)
(check= (foo 1 1 nil) ...)
(check= (foo 1 1 '(a b)) ...)
```

Is this the best we can do? Notice that contract-based testing does not depend on the function definition. Suppose `foo` has the following definition.

```
(definec foo (x :nat y :nat z :tl) :int
  (cond ((zp x) (* y 2))
        ((zp y) (* x 2))
        ((< x y) (foo y x z))
        ((oddp (len z)) (* y 4))
        ((< y x) (* x 4))
        ((oddp (- x y)) -15)
        ((evenp (- x y)) 24)
        (t 43)))
```

Why are the above contract-based tests not enough? One reason is that they do not *cover* all of the code, *e.g.*, none of the above tests exercise the third `cond` case. This leads us to *coverage-based* testing. The goal is to have enough tests so that all of the code in the function definition is executed at least once. Try defining an appropriate set of tests.

The tests we already have cover the first two `cond` cases and the penultimate case, so out of the eight cases, contract-based testing covered less than half of the cases. This means that most of the contract-based tests above are redundant in terms of `cond` case coverage. We define the *expression coverage* of a set of concrete tests for a given function to be the number of unique expression occurrences in the function definition that are executed when running the set of tests over the number of unique expression occurrences in the function definition. Consider the third case in the above definition; it has 7 expression occurrences: `(< x y)`, `x`, `y`, `(foo y x z)`, `y`, `x` and `z`. Notice that `x` and `y` appear twice because there are two distinct occurrences of these expressions in this case and, in general, a test that leads to the execution of one occurrence of an expression does not necessarily lead to the execution of a different occurrence of the same expression.

Here are tests for cases $3, 4$ and 5.

```
(check= (foo 1 2 nil) ...)
(check= (foo 5 5 '(a)) ...)
(check= (foo 5 4 nil) ...)
```

Case 6 is interesting because there is no test that can satisfy the case. The expression "`-15`" is an example of *unreachable code*, code that will never be executed, no matter what input is provided to `foo`. Such code typically (but not always!) indicates an error. It will almost certainly be an error for any code you write for this class. The existence of unreachable code (sometimes also called *dead code*) means that 100% expression coverage is not always possible. Here is how we can test the claim using `test?`.

```
(test? (=> (and (natp x) (natp y) (tlp z)
                (! (zp x))
                (! (zp y))
                (! (< x y))
                (! (oddp (len z)))
                (! (< y x)))
           (! (oddp (- x y)))))
```

If case 6 is reachable, then there should be a counterexample to the above claim and we can use that counterexample to create a concrete test. However, not only does `test?` tell us that this is not the case, it also reports that it proved the conjecture, meaning that no test exists that will make case 6 true.

What about the rest of the cases?

Exercise 2.7 *Is case* 7 *reachable? Use* `test?` *to either show that it is not or to find a concrete test that satisfies the case.*

Exercise 2.8 *Is case* 8 *reachable? Use* `test?` *to either show that it is not or to find a concrete test that satisfies the case.*

Exercise 2.9 *Show that for every real number $r > 0$, there is an admissible ACL2s function* `f` *and a set of contact-based tests whose expression coverage is $< r$.*

We define the *maximal expression coverage* of a function to be the maximum possible expression coverage over all possible sets of concrete tests.

Exercise 2.10 *What is the maximal expression coverage for* `foo`*?*

Exercise 2.11 *Exhibit a set of tests with minimal cardinality that obtains maximal expression coverage for* `foo`*.*

Now consider the following function definition.

```
(definec bar (x :nat y :nat z :tl) :int
  (foo x y z))
```

We can obtain 100% expression coverage of `bar` with a single test, even though it is equivalent to `foo`. This example suggests that we may want a stronger notion of coverage. Given a function f, the *reachable* functions from f, denoted $R(f)$, are obtained by executing f on all legal inputs and collecting the set of functions that are called during any of these executions. For example, $R($`bar`$)$ includes `foo`, `zp`, `*`, `oddp`, `len` and `consp` (used to define `len`), among others. We define the *reachable expression coverage* of a set of tests for f to be the number of unique expression occurrences in definitions of the functions in $R(f)$ that are executed when running the set of tests over the number of unique expression occurrences in the definitions of the functions in $R(f)$. Notice that built-in functions do no have definitions, hence, they do not contribute expression occurrences in the definition of reachable expression coverage. We define the *maximal reachable expression coverage* of a function to be the maximum possible reachable expression coverage over all possible sets of concrete tests.

Exercise 2.12 *Exhibit a set of tests with minimal cardinality that obtains maximal expression coverage for* `foo`*.*

There are many ways to extend the above notions, *e.g.*, instead of considering all reachable functions, we may consider functions within a certain distance in the call graph of the function. We may also allow built-in functions to be annotated with a set of conditions that are used in lieu of definitions, *e.g.*, we may annotate `consp` with { `(consp x)`, `(atom x)` } indicating that for full expression covererage we want tests that make `consp` both true and false.

There are also many types of *coverage metrics* that have been defined, with many interesting subtleties. Hopefully, you will cover this topic in more detail in a good software engineering course.

2.16 Property-Based Testing

We now consider property-based testing, a significantly more powerful kind of testing. *Property-based testing* involves specifying a collection of properties that your code should satisfy. Such tests are sometimes called *symbolic tests* and are written, in ACL2s, using `test?`. When defining functions, you should use `test?` to write down properties that should be true of the functions you define.

When designing such 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 the properties 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.

Notice that the following symbolic test for `foo` is useless.

```
(test? (=> (and (natp x) (natp y) (tlp z))
           (intp (foo x y z))))
```

Why is it useless?

Because that is just the function contract and ACL2s already either proves or tests this contract. The general principle here is that you want to avoid redundant properties. A property is *redundant* if it is implied by the conjunction of the rest of the properties.

Properties related to output contracts are often useful. For example, we may want to check that the type specified for `foo` is the most restrictive type we can use. How can we do that? Well, we may want to check that it is possible for `foo` to return positive integers, negative integers and 0. Here is how to do that. The following form succeeds only if the `test?` form fails.

```
(must-fail
 (test? (=> (and (natp x) (natp y) (tlp z))
            (!= (foo x y z) 0))))
```

This example shows that we sometimes want properties that should fail because their failure means that something we care about is possible. In the above example, we wanted to see that it is possible for `foo` to return 0 and if `test?` can find such an example, then it will fail and the `must-fail` form will succeed.

What if we tried to avoid the use of `must-fail` by using this property?

```
(test? (=> (and (natp x) (natp y) (tlp z))
           (= (foo x y z) 0)))
```

Notice that this `test?` fails because `foo` does not always return 0.

Exercise 2.13 *Write a property using* `must-fail` *and* `test?` *to check that it is possible for* `foo` *to return a positive integer. Is this claim true?*

Exercise 2.14 *Write a property using* `must-fail` *and* `test?` *to check that it is possible for* `foo` *to return a negative integer. Is this claim true?*

Exercise 2.15 *Rewrite* `foo` *by removing all of the dead code. Use ACL2s to check your definition.*

Exercise 2.16 *Rewrite the definition of* `foo` *from Exercise 2.15 by restricting the output contract based on what you discovered in exercises 2.13 and* `ex:foo-neg`*. Use ACL2s to check your definition.*

Exercise 2.17 *Write a property using* `must-fail` *and* `test?` *to check that it is possible for* `foo` *to return an odd integer. Is this claim true?*

We define the *range* of a function to be the set of values it can return, given inputs that satisfy the function's input contract. Notice that the range of a function always satisfies the the output contract (if ACL2s admitted the function definition). However not evey value that satisfies a function's output contract is in the range of the function.

Exercise 2.18 *What is the range of* `foo`*?*

Exercise 2.19 *Show that the range of* `foo` *is a subset of the set of even natural numbers. You can do that by writing a property making this claim and checking it with ACL2s.*

Exercise 2.20 *Show that the set of even natural numbers is a subset of the range of* `foo`*. You can do that by defining a function* `foo-witness` *that given as input an even natural number, say* `n`*, returns a list containing a natural number, a natural number and a true list, say* `x`*,* `y` *and* `z` *such that* `(foo x y z)` *is* `n`*. This function generates a witness to the claim that* `n` *is in the range of* `foo`*. Next write a property formalizing the claim that* `foo-witness` *does what we want and check the property using ACL2s.*

In the above exercises we discovered that the range of `foo` is a strict subset of the revised output contract. Does that mean that we should change the output contract? No necessarily. The general guideline we will follow is that output contracts will only specify "type-like" constraints and these constraints should be as restrictive as we can make them. For example, using `:nat` instead of `:int` for the output contract of `foo` is in line with the above guidelines. If the property that `foo` always returns an even natural number is important, then we will add it as a property. This is only a guiding principle. Writing code is an art form and the artist is free to express themselves as they see fit. Nevertheless, if you decide that in a particular situation it makes sense to violate these guidelines, make sure you have a compelling reason for doing so.

2.17 Designing Programs

Consider the following function definition.

```
(definec foo (x :nat y :nat z :tl) ...
  ...)
```

You should write as per the above guidelines. 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.

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 true lists and natural numbers should be obvious, but here is a brief review. Recall the data definition for true lists.

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

We say that cons is a *constructor*. When we define recursive functions, we use the *destructors* `car` (or `first`) and `cdr` (or `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.

$$Nat : 0 \mid Nat + 1$$

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 true lists of existing types. For example, we can define a true list of rationals as follows. Notice `listof` gives us a *true* list.

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

If we define a function that recurs on one of its arguments, which is a true list of rationals, we just use the true 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.

```
(definec foo (px :propex ...) ...
  (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 true lists.

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

This definition is recursive, *i.e.*, *TL* 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

$$TL_0 = \{\texttt{()}\}$$

and then create all conses of the form

```
(cons x l)
```

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

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

and now we define *TL* to be the union of all the TL_i.

$$\mathit{TL} = \bigcup_{i \in \mathbb{N}} TL_i$$

When we define recursive functions using templates based on this data definition, we use the *destructors* `car` (or `first`) and `cdr` (or `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 TL_{i+1} (let $i+1$ be the smallest index such that the element is in TL_{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 TL_0) and get, at worst, an element in TL_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. *TL*! That is:

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

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

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

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 $f^i(\emptyset) = TL_i$. There is a rich theory of fixpoints and they play an important role in computer science.

2.18 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.

2.19 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.

```
(definec !! (x :int) :int
  (if (zip x)
      1
    (* x (!! (1- x)))))
```

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 going to program mode to avoid stack overflows, timeouts and other unpleasant consequences of testing non-terminating code.

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

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

```
(definec !! (x :int) :int
  (if (zip x)
      1
    (* x (!! (1- x)))))
```

Now you can test your code. For example:

```
(!! 10)
```

works as expected, but

```
(!! -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. These command control the behavior of `defunc` and `definec`. 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.

```
(definec !! (x :nat) :int
  (if (zerop x)
      1
    (* x (!! (1- x)))))
```

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.

2.20 Debugging Code

A very useful type of checking that ACL2s performs is contract checking. This includes function contracts, body contracts and contract contracts, as described previously. If you define a function in logic mode, say `f`, then ACL2s proves, statically that the following hold.

1. evaluating `f`'s input contract on any (well-formed) inputs whatsover will not lead to any contract violations, and
2. evaluating the body of `f` on any inputs that satisfy `f`'s input contract will never lead to a contract violation for any function that may be called during this evaluation, including functions that are called directly or indirectly, and
3. the evaluation of `f`'s body on any inputs that satisfy `f`'s input contract will terminate, and
4. the evaluation of `f`'s body on any inputs that satisfy `f`'s input contract will yield a value that satisfies `f`'s output contract.

Therefore, for logic mode definitions, ACL2s only needs to check input contract for "top-level" forms. For example, consider the following definition.

```
(definec tapp (x :tl y :tl) :tl
  (if (lendp x)
      y
    (lcons (head x) (tapp (tail x) y))))
```

Now, let us evaluate the following.

```
(tapp '(1 2 3) '(4))
```

ACL2s, as expected, returns the following result.

```
(1 2 3 4)
```

If we want to see how it does that, we can *trace* the appropriate set of functions, which in our case only includes `tapp`, as follows.

```
(trace$ tapp)
```

Now, when we evaluate the same expression:

```
(tapp '(1 2 3) '(4))
```

we see the following output.

```
1> (ACL2_*1*_ACL2S::TAPP (1 2 3) (4))
  2> (TAPP (1 2 3) (4))
    3> (TAPP (2 3) (4))
      4> (TAPP (3) (4))
        5> (TAPP NIL (4))
        <5 (TAPP (4))
      <4 (TAPP (3 4))
    <3 (TAPP (2 3 4))
  <2 (TAPP (1 2 3 4))
<1 (ACL2_*1*_ACL2S::TAPP (1 2 3 4))
(1 2 3 4)
```

Here is what is going on here. When we defined `tapp`, lots of things happened. One of those things is that we defined the function `ACL2_*1*_ACL2S::TAPP`. This is a function that includes code to check the input contract for `tapp`. Why do we need this? Because a user could type anything they want at the REPL, including something like this:

```
(tapp '(1 . 2) '(3))
```

Notice we have a contract violation and we get the following error.

```
1> (ACL2_*1*_ACL2S::TAPP (1 . 2) (3))
ACL2 Error in TOP-LEVEL:  The guard for the function call (TAPP X Y),
which is (AND (TRUE-LISTP X) (TRUE-LISTP Y)), is violated by the arguments
in the call (TAPP '(1 . 2) '(3)).
```

You can think of "guard" in the output generated by ACL2s as being synonymous with "contract." So, since we cannot stop a user from calling `tapp` (or any logic mode function) on arguments that violate the function's input contract, we check that these top-level forms satisfy the appropriate input contracts with the "star one star" functions. After that, no further checking is required since ACL2s already performed static contract checking, as described above.

What if we defined `tapp` as a `:program` mode function as follows?

```
(definec tapp (x :tl y :tl) :tl
  (declare (xargs :mode :program))
  (if (lendp x)
      y
    (lcons (head x) (tapp (tail x) y))))
```

Well, now ACL2s does not perform static contract checking. What if we trace `tapp` and run it on the same example above?

```
(tapp '(1 2 3) '(4))
```

We see the following output.

```
1> (ACL2_*1*_ACL2S::TAPP (1 2 3) (4))
  2> (ACL2_*1*_ACL2S::TAPP (2 3) (4))
    3> (ACL2_*1*_ACL2S::TAPP (3) (4))
      4> (ACL2_*1*_ACL2S::TAPP NIL (4))
      <4 (ACL2_*1*_ACL2S::TAPP (4))
    <3 (ACL2_*1*_ACL2S::TAPP (3 4))
  <2 (ACL2_*1*_ACL2S::TAPP (2 3 4))
<1 (ACL2_*1*_ACL2S::TAPP (1 2 3 4))
(1 2 3 4)
```

Notice that ACL2s is now checking contracts for all the function calls. This can incur a significant computational cost.

You can turn off all tracing as follows.

```
(untrace$)
```

You can also turn off tracing for specific functions as follows.

```
(untrace$ tapp)
```

Sometimes users attempt to define a function in ACL2s and get counterexamples but have difficulty determining why the counterexamples are problematic.

In such cases, if you evaluate the code in your head using the counterexamples ACL2s generates, you'll see what the issue is, but maybe that's hard and you want help.

Suppose you are trying to define a function, `f`, that has one argument, `l`, and you get an error like the following.

```
Query: Testing body contracts ...
**Summary of Cgen/testing**
We tested 3 examples across 1 subgoals, of which 3 (3 unique) satisfied
the hypotheses, and found 3 counterexamples and 0 witnesses.
We falsified the conjecture. Here are counterexamples:
[found in : "top"]
-- ((L '(A B C)))
-- ((L 'X))
-- ((L 'Y))
```

You could try evaluating the function on any of the counterexamples above, but maybe you still can't figure out what is going on. Here is what you can do instead.

First, turn off testing (described above) with the following command.

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

Then, switch to `:program` mode.

Then force ACL2s to accept the definition be resumbitting the `defunc` or `definec` form. Since testing is turned off and we are in `:program` mode, the function will be accepted by ACL2s.

Next, we trace the function, using `trace$` and then we can evaluate our function at the REPL using the counterexamples ACL2s previously generated for us. During this process, we may potentially trace even more functions, if we want to better visualize what is going on. One useful way to evaluate our function using the ACL2s counterexamples is to use `let`:

```
(let ((l '(a b c)))
  (f l))
```

Once we figure out what the problem is, we can undo the definitions, turn off tracing, fix the issue and resume.

2.21 Exercises

For all programming exercises, make sure to add tests and properties as per the testing guidelines.

Exercise 2.21 *Define the function* `rr`*.*

$$\texttt{rr} : \mathit{Nat} \times \mathit{TL} \rightarrow \mathit{TL}$$

`(rr n l)` *rotates the true list* `l` *to the right* `n` *times.*

```
(definec rr (n :nat l :tl) :tl
  ...)
```

Here are some initial tests.

```
(check= (rr 1 '(1 2 3)) '(3 1 2))
(check= (rr 2 '(1 2 3)) '(2 3 1))
(check= (rr 3 '(1 2 3)) '(1 2 3))
(check= (rr 5 '(1 2 3)) '(2 3 1))
```

Exercise 2.22 *Define the function* `err`*, an efficient version of* `rr`*.*

$$\texttt{err} : \mathit{Nat} \times \mathit{TL} \rightarrow \mathit{TL}$$

`(err n l)` *returns the list obtained by rotating* `l` *to the right* `n` *times but it does this efficiently because it actually never rotates more than* `(len l)` *times.*

```
(definec err (n :nat l :tl) :tl
  ...)
```

Make sure that `err` *is efficient by timing it with a large* `n` *and comparing the time with* `rr`*. Here is how you can do that.*

```
(time$ (rr  10000000 '(a b c d e f g)))
(time$ (err 10000000 '(a b c d e f g)))
```

Exercise 2.23 *Here is a data definition for a bitvector.*

```
(defdata bit (oneof 0 1))
(defdata bitvector (listof bit))
```

We can use bitvectors to represent natural numbers as follows. The list

`(0 0 1)`

corresponds to the number 4 *because the first* `0` *is the "low-order" bit of the number which means it corresponds to* $2^0 = 1$ *if the bit is* 1 *and* 0 *otherwise. The next bit corresponds to* $2^1 = 2$ *if the bit is* 1 *and to* 0 *otherwise and so on. So the above number is*

$$0 + 0 + 2^2 = 4.$$

As another example, 31 is

$$(11111)$$

or

$$(1111100)$$

or

$$\ldots$$

Define the function n-to-b that given a natural number, returns a bitvector list of minimal length, corresponding to that number.

```
(definec n-to-b (...)
  ...)
```

Here are some initial tests.

```
(check= (n-to-b 0)  '())
(check= (n-to-b 7)  '(1 1 1))
(check= (n-to-b 10) '(0 1 0 1))
```

Exercise 2.24 *Define the function* `b-to-n` *that given a bitvector, as defined in Exercise 2.23 returns the corresponding natural number.*

```
(definec b-to-n (...)
  ...)
```

Here are some initial tests.

```
(check= (b-to-n '(0 1 0 1)) 10)
(check= (b-to-n '(1 1 1)) 7)
(check= (b-to-n '(0 0 1)) 4)
(check= (b-to-n '(1 1 1 1 1)) 31)
(check= (b-to-n ()) 0)
```

Exercise 2.25 *Define the following properties relating the functions* `n-to-b` *and* `b-to-n` *from Exercises 2.23 and 2.24 and determine is they are true. If not, fix them in the minimally invasive way to make them true.*

Property 1: The function `b-to-n` *is the inverse of function* `n-to-b`.

Property 2: The function `n-to-b` *is the inverse of function* `b-to-n`.

Exercise 2.26 *The permutations of a (true) list are all the lists you can obtain by swapping any two of its elements (repeatedly). For example, starting with the list*

`(1 2 3)`

I can obtain

```
(3 2 1)
```

by swapping 1 and 3.

So the set of permutations of `(1 2 3)` *is:*

{`(1 2 3)`, `(1 3 2)`, `(2 1 3)`, `(2 3 1)`, `(3 1 2)`, `(3 2 1)`}.

Notice that if the list is of length n *and all of its elements are distinct, it has* $n!$ *(n factorial) permutations.*

Given a list, say `(a b c d e)`, *we can define any of its permutations using a list of* distinct *natural numbers from* 0 *to the length of the list* -1, *which tell us how to reorder the elements of the list. Let us call this list of distinct natural numbers an* arrangement. *For example applying the arrangement* `(4 0 2 1 3)` *to the list* `(a b c d e)` *yields* `(e a c b d)`.

Define the function `find-arrangement` *that given two non-empty true lists either returns* `nil` *if they are not permutations of one another or returns an arrangement such that applying the arrangement to the first list yields the second list. Note that if the lists have repeated elements, then more than one arrangement will work. In such cases, it does not matter which of these arrangements you return.*

```
(definec find-arrangement (...)
  ...)
```

Here are some initial tests.

```
(check= (find-arrangement '(a b c) '(a b b)) nil)
(check= (find-arrangement '(a b c) '(a c b)) '(0 2 1))
(check= (find-arrangement '(a a) '(a a)) '(0 1))
```

Note that you will have to define some helper functions. Make sure to write interesting properties.

Exercise 2.27 *This exercise depends on Exercise 2.26. The goal is to define a function to apply an arrangement to a list, thereby obtaining a permutation of the list.*

First, an arrangement has to be a list of positive integers (but not every list of positive integers is an arrangement!)

```
(defdata lop (listof pos))
```

Define `apply-arrangement`, *a function that takes a true-list,* `l`, *and an arrangement,* `a`, *(of type* `lop`*) as arguments and, assuming that* `a` *is really an arrangement, then it returns the result of applying the arrangement* `a` *to the list* `l`. *You might find the function* `nth` *useful. If finds the nth number in a list, using 0-indexing, so* `(nth 0 '(0 1 2 3))` *is* 0, `(nth 3 '(0 1 2 3))` *is* 3 *and if the number is too big,* `nil` *is returned,* e.g., `(nth 10 '(0 1 2 3))` *is* `nil`.

```
(definec apply-arrangement (l ... a ...) ...
  ...)

(check= (apply-arrangement '(a b c) '(1 3 2))
        '(a c b))
```

Exercise 2.28 *This exercise depends on Exercises 2.26 and 2.27. Use* `test?` *to formalize the claim that if* `x` *is a permutation of* `y` *then applying the arrangement (with* `apply-arrangement`*) returned by* `find-arrangement` *to* `x` *results in* `y`*. However, insted of doing this for arbitrary lists, assume that* `x` *and* `y` *are of type* `losyms`*, as defined below.*

```
(defdata syms (enum '(a b c)))
(defdata losyms (listof syms))
```

You can ask ACL2s to increase the testing it does with the following.

```
(acl2s-defaults :set num-trials 50000)
```

Use the above property to test your solutions to Exercises 2.26 and 2.27.

Exercise 2.29 *This exercise depends on Exercises 2.26, 2.27 and 2.28.*
Consider the following buggy solution for `find-arrangements` *from Exercise 2.26.*

```
;; When called by fa, if x is in y starting at some index which is >= i
;; and not in acc, then the index function returns the first such index.
;; Otherwise, it returns 0.
(definec index (x :all y :tl i :posp acc :tl) :nat
  (cond ((endp y) 0)
        ((and (== x (car y))
              (! (in i acc)))
         i)
        (t (index x (rest y) (+ i 1) acc))))

;; fa is a helper function for find-arrangement. It is called only if
;; x and y are lists of the same length. It finds an arrrangement if
;; one exists, otherwise it returns nil.
(definec fa (x :tl y :tl acc :tl) :tl
  (if (endp x)
      (rev acc)
    (let ((i (index (first x) y 1 acc)))
      (if (zp i)
          nil
        (fa (rest x) y (cons i acc))))))

(definec find-arrangement (x :ne-tl y :ne-tl) :tl
  (if (= (len x) (len y))
      (fa x y nil)
    nil))
(check= (find-arrangement '(a b c) '(a b b)) nil)
(check= (find-arrangement '(a b c) '(a c b)) '(1 3 2))
(check= (find-arrangement '(a a) '(a a)) '(1 2))
```

Notice that all the `check=` *forms pass. However, the definition of* `find-arrangement` *is buggy.*

We will use property-based testing to more thoroughly test this implementation of `find-arrangement`*.*

Use the property from Exercise 2.28 to find counterexamples. Use these counterexamples to fix the definition of `fa` *(do not modify any other definitions). Make sure that your proposed fix passes testing.*

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. [1]

[1] Many presentations of propositional logic use the term *connective* instead of operator. The use of "connective" is wide-spread and we also occassionally it in lieu of operator. Using it for binary operators is reasonable, but its use for unary operators ($\neg$) is misleading, since it is not connecting expressions.

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 of 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.*, an 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*. We will refer to rules that simplify formulas that have a constant in them as *constant propagation*.

1. $p \vee \mathit{true} \equiv \mathit{true}$
2. $p \wedge \mathit{true} \equiv p$
3. $p \Rightarrow \mathit{true} \equiv \mathit{true}$
4. $\mathit{true} \Rightarrow p \equiv p$
5. $(p \equiv \mathit{true}) \equiv p$

6. $(p \oplus \mathit{true}) \equiv \neg p$

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

7. $p \vee \mathit{false} \equiv p$

8. $p \wedge \mathit{false} \equiv \mathit{false}$

9. $p \Rightarrow \mathit{false} \equiv \neg p$

10. $\mathit{false} \Rightarrow p \equiv \mathit{true}$

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

12. $(p \oplus \mathit{false}) \equiv p$

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

13. $p \vee q \equiv q \vee p$

14. $p \wedge q \equiv q \wedge p$

15. $(p \equiv q) \equiv (q \equiv p)$

16. $(p \oplus q) \equiv (q \oplus p)$

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*:

17. $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$

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:

18. $p \Rightarrow q \equiv \neg p \vee q$

Also, it is sometimes useful to replace $\equiv$ by $\Rightarrow$, which is possible due to the following equality:

19. $(p \equiv q) \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$

The above equality allows us to prove a propositional equality by proving two implications. Dijkstra called this a *ping-pong* proof; where the forward direction $(p \Rightarrow q)$ is *ping* and the other direction $(q \Rightarrow p)$ is *pong.*

We will use the following implication all the time when reasoning about programs. This validity is called *Modus Ponens.*

20. $(p \Rightarrow q) \wedge p \Rightarrow q$

Equalities are often more useful than implications, *e.g.*, they are easier to use in *equational proofs*, proofs consisting of a sequence of equalities. Equational proofs will turn out to be really important! We can turn Modus Ponens into the following equality.

21. $(p \Rightarrow q) \wedge p \;\equiv\; p \wedge q$

Here are more equalities.

22. $\neg\neg p \equiv p$
23. $\neg \mathit{true} \equiv \mathit{false}$
24. $\neg \mathit{false} \equiv \mathit{true}$
25. $p \wedge p \equiv p$
26. $p \vee p \equiv p$
27. $p \Rightarrow p \equiv \mathit{true}$
28. $(p \equiv p) \equiv \mathit{true}$
29. $(p \oplus p) \equiv \mathit{false}$
30. $p \wedge \neg p \equiv \mathit{false}$
31. $p \vee \neg p \equiv \mathit{true}$
32. $p \Rightarrow \neg p \equiv \neg p$
33. $\neg p \Rightarrow p \equiv p$
34. $p \equiv \neg p \equiv \mathit{false}$
35. $p \oplus \neg p \equiv \mathit{true}$

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

36. $\neg(p \wedge q) \equiv \neg p \vee \neg q$
37. $\neg(p \vee q) \equiv \neg p \wedge \neg q$

What if we negate other connectives? The following equalities address that.

38. $\neg(p \Rightarrow q) \equiv p \wedge \neg q$
39. $\neg(p \equiv q) \equiv (p \oplus q)$
40. $\neg(p \oplus q) \equiv (p \equiv q)$

Another fundamental property is *associativity*:

41. $((p \vee q) \vee r) \equiv (p \vee (q \vee r))$
42. $((p \wedge q) \wedge r) \equiv (p \wedge (q \wedge r))$

43. $((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$

44. $((p \oplus q) \oplus r) \equiv (p \oplus (q \oplus r))$

Since $\equiv$ is associative and commutative, we do not need to use parentheses to disambiguate a sequence of $\equiv$'s. In fact, we can permute the arguments any way that we want to. Convince yourself that this is really the case. This result applies to any associative and commutative operator, including addition, *e.g.*, $a + b + c + d = c + a + d + b$ and similarly $p \vee q \vee r \vee s \equiv r \vee p \vee s \vee q$.

An important property is *distributivity*:

45. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$

46. $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

Another important property is *transitivity*:

47. $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$

48. $(p \equiv q) \wedge (q \equiv r) \Rightarrow (p \equiv r)$

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

49. $(p \oplus q) \wedge (q \oplus r) \Rightarrow (p \equiv r)$

50. $(p \equiv q) \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$

51. $(p \equiv q) \equiv (p \vee \neg q) \wedge (\neg p \vee q)$

52. $(p \oplus q) \equiv (p \wedge \neg q) \vee (\neg p \wedge q)$

53. $(p \oplus q) \equiv (p \vee q) \wedge (\neg p \vee \neg q)$

Make sure that not only you remember these equalities, but that you understand and can explain why they hold. Get into the habit of reading such formulas the way you would read text or programs. For example equation 49 states that if p differs from q $(p \oplus q)$ and q differs from r $(q \oplus r)$, then p must be equal to r $(p \equiv r)$.

We have the following *redundancy laws*:

54. $(p \vee q) \wedge (p \vee \neg q) \equiv p$

55. $(p \wedge q) \vee (p \wedge \neg q) \equiv p$

56. $p \wedge (\neg p \vee q) \equiv p \wedge q$

57. $p \vee (\neg p \wedge q) \equiv p \vee q$

Notice that equation 56 is equivalent to the equational version of Modus Ponens (equation 21).

We also have the related *absorption laws*:

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

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

Let's consider absorption more carefully. Here is a simple calculation, an equational proof:

Proof

$p \wedge (p \vee q)$

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

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

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

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

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

A very useful equality is based on the *Shannon expansion* of a formula: if f is a formula and p is an atom, then we have:

60. $f \equiv (p \wedge f|_{((p\ true))}) \vee (\neg p \wedge f|_{((p\ false))})$

By $f|_{((p\ x))}$ we mean, substitute x for atom p in f. Substitution has a higher binding power than all propositional connectives. For example, $g \vee f|_{((p\ x))}$ means $g \vee (f|_{((p\ x))})$, and *not* $(g \vee f)|_{((p\ x))}$. The right-hand side of the equality above is the Shannon expansion of f.

We now define substitution.

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 occurrence of a target atom by its image.

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

Let us apply Shannon decomposition to the formula above.

Proof

$p \vee (p \wedge q)$

$\equiv$ { Shannon expansion using p }

$(p \wedge (true \vee (true \wedge q))) \vee (\neg p \wedge (false \vee (false \wedge q)))$

$\equiv$ { Constant propagation }

$(p \wedge true) \vee (\neg p \wedge false)$

$\equiv$ { Constant propagation }

$p \vee false$

$\equiv$ { Constant propagation }

p $\square$

We showed the constant propagation steps in detail, but typically, we would just have one step that does all the constant propagation.

In general, Shannon expansion can double the size of formulas, but there are some special cases that always provide a win. These equalities are *very* useful for simplifying expressions. Make sure to remember them. If f is a formula, then we have:

61. $p \wedge f \equiv p \wedge f|_{((p\ true))}$

62. $\neg p \wedge f \equiv \neg p \wedge f|_{((p\ false))}$

63. $p \vee f \equiv p \vee f|_{((p\ false))}$

64. $\neg p \vee f \equiv \neg p \vee f|_{((p\ true))}$

We can combine the above equalities to derive new ones. For example, if f, g are formulas, then we have:

65. $p \Rightarrow f \equiv p \Rightarrow f|_{((p\ true))}$

66. $f \Rightarrow p \equiv f|_{((p\ false))} \Rightarrow p$

67. $p \wedge f \Rightarrow g \equiv p \wedge f|_{((p\ true))} \Rightarrow g|_{((p\ true))}$

We will have to manipulate formulas containing nested implications and the following two equalities will be very useful. The first is *exportation*:

68. $p \Rightarrow (q \Rightarrow r) \ \equiv\ p \wedge q \Rightarrow r$

The second is *negate and swap*:

69. $p \wedge q \Rightarrow r \ \equiv\ p \wedge \neg r \Rightarrow \neg q$

If we have a formula of the form $p_1 \wedge \cdots \wedge p_n \Rightarrow r$, we can negate and swap any p_i with r.

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*. Notice the similarity between case analysis and Shannon expansion.

Case Analysis (simple version): 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. The next validity can be used to justify case analysis.

70. $(p \Rightarrow q) \wedge (\neg p \Rightarrow q) \ \equiv\ q$

Notice that the above equality is just a restatement of one of the redundancy equalities.

Here is the promised proof.

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 }

$true$ $\square$

We can generalize the proof technique as follows.

Case Analysis (general version): If f is a formula and $g_1, \ldots, g_n$ are formulas such that $g_1 \vee \cdots \vee g_n$ is valid, then f is valid iff all of $(g_1 \Rightarrow f), \ldots, (g_n \Rightarrow f)$ are valid.

The intuition is that we are proving that f holds by considering the cases $g_1, \ldots, g_n$ and since the cases are exhaustive ($g_1 \vee \cdots \vee g_n \equiv \mathit{true}$), f always holds. Notice that when $g_1 = p,\ g_2 = \neg p$, then this is equivalent to the simple version of case analysis.

Another useful proof technique is *instantiation.*

Instantiation: If f is a valid formula, then so is $f|_\sigma$, where σ is a substitution.

Here is an application of instantiation. $a \vee \neg a$ is valid, so therefore, so is

$$(a \vee \neg a)|_{((a\ \ (p \vee (q \wedge r))))} = (p \vee (q \wedge r)) \vee \neg(p \vee (q \wedge r)).$$

A very important skill is learning to see a formula at different levels of abstraction and it is sometimes useful to take a complex formula, such as

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

and view it as a special case of the more abstract formula

$$a \vee \neg a$$

This allows us to apply instatiation.

All of the Boolean logic proof techniques are applicable to reasoning about more expressive logics. In the next few chapters, we will see how to use these proof techniques to reason about programs. As a simple example, we can prove that the following is valid.

```
(or (== x y) (! (== x y)))
```

The proof is just instantiation of the propositional equality

$$p \vee \neg p$$

using the substitution `((`p` (== x y)))`. Notice that `(== x y)` is not a propositional formula, but it is a Boolean-valued expression. Given any ACL2s expression, we can extract a *propositional skeleton* (or *Boolean skeleton*) and can reason about that using propositional logic. Here is another example. A propositional skeleton of

```
(=> (and (rationalp x) (rationalp y)) (= (+ x y) (+ y x)))
```

is

$$p \wedge q \Rightarrow r.$$

Given an ACL2s formula, say ϕ, we say that a Boolean formula f is a *propositional abstraction* (or *Boolean abstraction*) of ϕ if there is a substitution, σ such that $f|_\sigma = \phi$. If f is a Boolean abstraction of ϕ and there is no formula g with more propositional connectives that is also a Boolean abstraction of ϕ, we say that f is a propositional skeleton of ϕ. For example, the following formulas are Boolean abstractions of the above ACL2s formula, as the corresponding substitutions show.

p, `((`p` (=> (and (rationalp x) (rationalp y)) (= (+ x y) (+ y x)))))`

$p \Rightarrow q$, `((`p` (and (rationalp x) (rationalp y))) (`q` (= (+ x y) (+ y x))))`

$r \wedge p \Rightarrow q$, `((`r` (rationalp x)) (`p` (rationalp y)) (`q` (= (+ x y) (+ y x))))`

Only the last formula is a propositional skeleton because there is no further propositional structure to extract. We note that the definition of a substitution in the context of ACL2s formulas includes some subtleties that will be discussed in Chapter 4.

3.5 Decision Procedures

A problem that has a "yes" or "no" answer is a *decision problem.* Here are some examples of decision problems.

1. Determining if an ACL2s object is an integer.
2. Determining if an ACL2s list of rational numbers is ordered.
3. Determining if a propositional formula is valid.
4. Determining if an ACL2s program terminates on all legal inputs.

A *decision procedure* is an algorithm that solves a decision problem. The algorithm has to terminate and it has to correctly solve the decision problem. If there is a decision procedure for a decision problem, then we say that the problem is *decidable.* Here are some examples.

1. `integerp` is a decision procedure for the decision problem of determining if an ACL2s object is an integer.
2. The decision problem of determining if an ACL2s list of rational numbers is ordered is decidable. To see this, note that an ACL2s function that checks this is a problem decision procedure for this decision problem.
3. There is a decision procedure for propositional validity.
4. There is no decision procedure for the decision problem of determining if an ACL2s program terminates on all legal inputs. How one goes about proving such *undecidability* results is an interesting, fundamental question that we will discuss in Chapter 5.

Let us consider decision problems and decision procedures in the context of propositional logic. The characterizations of formulas previously introduced are all decidable, as the exercises below ask you to show.

Exercise 3.4 *Write a decision procedure for propositional validity in ACL2s.*

Exercise 3.5 *Write a decision procedure for propositional satisfiability in ACL2s.*

Exercise 3.6 *Write a decision procedure for propositional falsifiability in ACL2s.*

Exercise 3.7 *Write a decision procedure for propositional unsatisfiability in ACL2s.*

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 $\neg f$) $\Box$

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 until you have internalized all the equalities.

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.

Often we want more than a "yes" or "no" answer. For example, 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 a decision procedure for propositional validity, as the exercises above ask you to do. In addition, we can also use it to write an algorithm that provides counterexamples when formulas are not valid.

Exercise 3.8 *Write an ACL2s program that determines the validity of propositional formulas and provides counterexamples when formulas are not valid.*

Exercise 3.9 *Write an ACL2s program that determines the satisfiability of propositional formulas and provide satisfying assignments when formulas are satisfiable.*

Exercise 3.10 *Write an ACL2s program that determines the unsatisfiability of propositional formulas and provides satisfying assignments when formulas are not unsatisfiable.*

Exercise 3.11 *Write an ACL2s program that determines the falsifiability of propositional formulas and provides falsifying assignments when formulas are falsifiable.*

Counterexamples can also be thought of as *certificates*. For example, a counterexample to validity is a certificate of falsifiability that can be checked by evaluation. The careful reader will have noticed that the above exercises only ask for certificates of satisfiability and falsifiability. It is clear that certificates that can be checked efficiently exist for showing satisfiability and falsifiability, because the size of the certificate is equal to the number of variables in the formula (so they are polynomial in the size of the formula) and checking the certificate can be done using evaluation (which can be done in polynomial time).

What about certificates for validity and unsatisfiability? This is an interesting question that is related to the $P = NP$ question, *e.g.*, if $P = NP$ then this implies that there are certificates that can be checked efficiently (in polynomial time). We currently do not know of any efficiently-checkable certificates for validity and unsatisfiability. Do any kinds of certificates exists? Yes; a proof of validity is a certificate, but such proofs may be very long, *i.e.*, no one has yet proved a theorem stating that proofs are always polynomial in size and no has yet proved that efficiently-checkable certificates for validity do not exist. If you want to know more, research the *co-NP* complexity class.

3.6 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.*

Exercise 3.12 *Come up with a way of representing an arbitrary Boolean function using full CNF. (Hint: Consider how we did this with full 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 3.13 *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 3.14 *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.

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

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

Exercise 3.15 *Represent the rest of the operators using ite.*

Exercise 3.16 *There are 16 binary Boolean operators. Are any of them complete? If so, exhibit such an operator and prove that it is complete. If not, prove that none of the operators is complete.*

We now consider two rules of inference that have a long, storied history.

The first rule of inference is *consensus.* Let C, D be conjunctive clauses and let $f = (p \wedge C) \vee (\neg p \wedge D) \vee g$. Then we have:

71. $f = f \vee (C \wedge D)$, *i.e.*, $C \wedge D \Rightarrow f$

Notice that $C \wedge D$ is a conjunctive clause and we say it is the *consensus* of conjunctive clauses $(p \wedge C)$ and $(\neg p \wedge D)$.

The second rule of inference is *resolution.* Let C, D be clauses and let $f = (p \vee C) \wedge (\neg p \vee D) \wedge g$. Then we have:

72. $f = f \wedge (C \vee D)$, *i.e.*, $f \Rightarrow C \vee D$

Notice that $C \vee D$ is a clause and we say it is the *resolvent* of clauses $(p \vee C)$ and $(\neg p \vee D)$. Modern SAT solvers are based on resolution.

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 (=> (and (boolp p) (boolp q))
         (iff (=> p q) (or (! p) q))))
```

Try it in ACL2s.

In fact, if a propositional formula is valid (that is, it is a theorem) then ACL2s will definitely prove it. We say that ACL2s includes a decision procedure for propositional validity. We saw that ACL2s indicates that it has determined that a formula is valid with "Q.E.D." [2]

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 (=> (and (boolp p) (boolp q))
         (iff (xor p q) (or p q))))
```

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

3.8 Valuations and Duality

We are often interested in the meaning of a propositional formula for a given assignment of values to the atoms of the formula. For example, consider the formula $p \Rightarrow q$. It is both satisfiable and falsifiable. However, if p is assigned *false*, then it is equivalent, under this assignment, to *true*. We define a *valuation* to be a substitution where all images are constants. A *Boolean valuation* is a valuation where all the images are Boolean constants. We say that a v is a *total valuation* for f when the domain of v includes all of the variables of f; in the case of Boolean valuations, the variables are the atoms. We may refer to valuations as assignments, but chose the term valuation, since it is rarer than assignment, thereby reducing the chance of ambiguity. [3]

If f is a Boolean formula, v is a valuation and $f|_v$ is equal to a constant, we say that v is *sufficient* for f. Notice that if f is any Boolean formula and v is a total valuation for f,

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

[3]Valuations are often presented as functions from atoms to Booleans, which has some advantages, *e.g.*, in situations where one considers infinite sets of formulas. For our purposes, there is no reason to not use the existing notion of a substitution.

then v is sufficient for f, *i.e.*, $f|_v$ is equal to either *true* or *false*. In fact, even if v is not a total valuation, we may be able to efficiently reduce $f|_v$ to a constant using just constant propagation, *e.g.*, consider $f = p \Rightarrow q$ and $v = ((p\ \mathit{false}))$. A valuation v is *minimal* for f if it is sufficient for f, but no sublist of v is sufficient for f. If f is either a tautology or unsatisfiable, then the empty valuation is sufficient and minimal for f, as we can reduce f to a constant by paying the computational cost of using a decision procedure. Given a valuation v, we define $\overline{v}$, the *negation* of v, to be the valuation obtained by negating images, *e.g.*, if $v = ((p\ \mathit{false})\ (q\ \mathit{true}))$, then $\overline{v} = ((p\ \mathit{true})\ (q\ \mathit{false}))$. When we say that a valuation v is total, sufficient or minimal for a set of formulas, we mean that it includes all the variables in the set, that under the valuation all of the formulas can be reduced to a constant or that the valuation is sufficient, but for any sublist there is at least one formula that cannot be reduced to a constant.

We now introduce duality, a very useful concept that is applicable in many parts of Mathematics. We consider only the propositional logic version. Given a formula f, we have seen that we can convert it to an equivalent formula consisting only of the operators $\{\neg, \wedge, \vee\}$; more generally we can use any complete Boolean base. Hence, for the remainder of this section we assume that formulas only contain these operators. If f is function, then its dual, $\hat{f}$, is obtained by replacing every occurrence of *false*, *true*, $\vee, \wedge$ with *true*, *false*, $\wedge, \vee$, respectively. For example, the dual of $(p \wedge \mathit{true}) \vee (q \vee \mathit{false})$ is $(p \vee \mathit{false}) \wedge (q \wedge \mathit{true})$.

Results involving duality and valuations are given as exercises. Reviewing instantiation will be helpful.

Exercise 3.17 *Prove that* $\hat{\hat{f}}$ *(the dual of the dual of* f*) is syntactically equal to* f*.*

Exercise 3.18 *Prove that* $f = g$ *iff for all total valuations of* f *and* g*,* v*, we have* $f|_v = g|_v$*.*

Exercise 3.19 *Prove that* $f = g$ *iff for all minimal valuations of* f *and* g*,* v*, we have* $f|_v = g|_v$*.*

Exercise 3.20 *Let* v *be a total valuation for* f*. Prove that* $\hat{f}|_v = \neg f|_{\overline{v}}$*.*

Exercise 3.21 *Let* v *be a sufficient valuation for* f*. Prove that* $\overline{v}$ *is a sufficient valuation for* $\hat{f}$*.*

Exercise 3.22 *Let* v *be a minimal valuation for* f*. Prove that* $\overline{v}$ *is a minimal valuation for* $\hat{f}$*.*

Exercise 3.23 *Let* v *be a sufficient valuation for* f*. Prove that* $\hat{f}|_v = \neg f|_{\overline{v}}$*.*

3.9 Connections to Set Theory

Set theory provides the foundations of mathematics. In this section, we will establish connections between Boolean (or propositional) logic and set theory. We will define a *Boolean algebra* of a non-empty set X to be a non-empty subset of the powerset of X closed under union, intersection and complementation (with respect to X).

We consider some examples. Let U be the ACL2s universe. Then $B = \{\emptyset, U\}$ is the smallest Boolean algebra of U. The largest Boolean algebra of U is the powerset of U, denoted 2^U.

In a Boolean algebra of X, *false* and *true* evaluate to $\emptyset$ and X and atoms correspond to elements of the algebra, *i.e.*, to subsets of X. Boolean algebras have the same operators as Boolean logic, with the following meaning. The Boolean operators $\vee$, $\wedge$ and $\neg$ correspond, respectively, to the set theory operators $\cup$ (union), $\cap$ (intersection) and $\neg$ (complementation relative to X). This is why Boolean algebras are required to be closed under union, intersection and complementation. Notice that the two-element Boolean algebra B is *isomorphic* to propositional logic, where $\emptyset$ corresponds to F and U corresponds to T. More generally, the smallest Boolean algebra of any non-empty set X will be isomorphic to propositional logic in the sense above.

Let us consider the Boolean algebra 2^U, whose elements are all the subsets of U, the ACL2s universe. Suppose that p and q are defined as follows.

$$p = \{x \in U : \texttt{(integerp } x\texttt{)}\}$$

$$q = \{x \in U : \texttt{(neg-rationalp } x\texttt{)}\}$$

$$r = \{x \in U : \texttt{(rationalp } x\texttt{)}\}$$

Then we have the following.

$$p \wedge q = \{x \in U : \texttt{(negp } x\texttt{)}\}$$

$$p \wedge \neg q = \emptyset$$

$$\neg p \vee r = U$$

A (Boolean algebra) formula is valid iff it is equal to U. Thus, $p \vee \neg p$ is valid in the Boolean algebra 2^U. A very interesting result is that the set of valid equalities of Boolean logic, some of which we presented in this chapter, is exactly equal to the set of valid equalities of Boolean algebra. This result (which we will not prove) can be useful when you are thinking about Boolean formulas. For example, consider the following absorption law (equation 58).

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

The above is valid iff the following set-theoretic formula is valid in 2^U (or any Boolean algebra).

$$p \cap (p \cup q) = p$$

Many find the set-theoretic version to be simpler to understand, which is probably due to the ubiquity of set theory.

The careful reader may wonder why we were able to replace $\equiv$ in equation 58 with $=$ in the corresponding set-theoretic formula. Since $\{\vee, \wedge, \neg\}$ is a complete Boolean base, we can define set-theoretic versions of all the other operators. For example, we can define $\Rightarrow$ and $\equiv$ using equations 18 and 19, respectively.

$$p \Rightarrow q \;\text{ is defined to be }\; \neg p \vee q$$

$$p \equiv q \;\text{ is defined to be }\; (p \Rightarrow q) \wedge (q \Rightarrow p)$$

Given the above, notice that the Boolean algebra formula $p \equiv q$ is valid iff the corresponding set-theoretic formula, $p = q$, holds.

Is the following formula valid?

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

This is equivalent to whether the following set theory formula is valid.

$$\neg(p \cup q) \cup r = (\neg p \cup r) \cap (\neg q \cup r)$$

The validity of the above set theory formula can be decided using Venn diagrams.

Is the following formula valid?

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

This is equivalent to whether the following set theory formula is valid.

$$(p \cap q) \subseteq (p \cup q)$$

We are using the observation that the Boolean algebra formula $p \Rightarrow q$ is valid iff the set-theoretic formula $p \subseteq q$ holds. We can once again use Venn diagrams to determine the validity of the above set theory formula.

Exercise 3.24 *Define a set-theoretic version of* $\oplus$.

Exercise 3.25 *Check that all of the Boolean logic equalities in this chapter also hold for Boolean algebras, using the definitions above and from Exercise 3.24.*

Exercise 3.26 *Prove that the Boolean algebra formula* $p \equiv q$ *is valid iff the set-theoretic formula,* $p = q$*, holds.*

Exercise 3.27 *Prove that the Boolean algebra formula* $p \Rightarrow q$ *is valid iff the set-theoretic formula* $p \subseteq q$ *holds.*

We have only scratched the surface. Boolean algebra can be futher generalized and that leads, among other things, to *lattice theory.* The interested reader is invited to explore this topic further.

3.10 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.

3.11 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.

3.11.1 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`.

3.11.2 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:

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

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.

$$\begin{gathered}\neg AC_1^1 \vee \neg AC_2^1 \\ \neg AC_1^1 \vee \neg AC_3^1\end{gathered}$$

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 2^{nd} 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.

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

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

$$\ldots$$

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

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

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, cryptography 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 `len`, a function we have already seen.

```
(definec len (l :all) :nat
  (if (consp l)
      (+ 1 (len (cdr l)))
    0))
```

Consider the following conjecture:

Conjecture 3 `(= (len (list x)) (len 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.*,

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

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

Here we can't find a counterexample.

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

```
  (len (cons x (list z)))
= {  Def len, instantiation  }
   (if (consp (cons x (list z))) (1+ (len (cdr (cons x (list z))))) 0)
= { car-cdr axioms }
   (if (consp (cons x (list z))) (1+ (len (list z))) 0)
= { consp axioms }
   (if t (1+ (len (list z))) 0)
= { if axioms }
   (1+ (len (list z)))
```

What we have shown so far is:

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

which we will feel free to write as

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

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

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

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

```
  (len (cons y (list z)))
= {  Def len, instantiation  }
   ...
= { if axioms }
   (+ 1 (len (list z)))
= {  Def 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:

$$\texttt{(len (cons y (list z)))} = 2 \tag{4.4}$$

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

$$\texttt{(len (cons x (list z)))} = \texttt{(len (cons y (list z)))}$$

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 `lrev`). 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 `lrev` 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

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

we usually write the more familiar

$$\texttt{(len (cons x z))} = \texttt{(len (cons y z))}$$

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 ≠ 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 `(len (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.

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

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

$$1 \neq 2$$

$$2/4 = 4/8$$

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

We will only use evaluation when there are no contract violations, *e.g.*, we will not be able to use evaluation to simplify `(endp 3)` because `(listp 3)` does not hold.

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

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

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

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

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

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

$$\texttt{(consp x)} \;\Rightarrow\; \texttt{x} = \texttt{(cons (car x) (cdr x))}$$

$$\texttt{(cons x1 y1)} = \texttt{(cons x2 y2)} \;\equiv\; \texttt{x1} = \texttt{x2} \wedge \texttt{y1} = \texttt{y2}$$

$$\neg\texttt{(consp x)} \;\Rightarrow\; \texttt{(car x)} = \texttt{nil}$$

$$\neg\texttt{(consp x)} \;\Rightarrow\; \texttt{(cdr x)} = \texttt{nil}$$

We have similar axioms for other built-in types such as characters, strings and numbers. For example, to reason about numbers, we have access to the following axioms (this is a very partial list).

$$\texttt{(rationalp x)} \;\Rightarrow\; \texttt{(posp (denominator x))}$$

$$\texttt{(rationalp x)} \;\Rightarrow\; \texttt{(integerp (numerator x))}$$

$$\texttt{(rationalp x)} \;\Rightarrow\; \texttt{(* (/ (denominator x)) (numerator x))} = \texttt{x}$$

$$\texttt{(integerp x)} \;\Rightarrow\; \texttt{(rationalp x)}$$

$$\texttt{(integerp x)} \;\Rightarrow\; \texttt{(integerp (+ x 1))}$$

`(integerp x)` $\Rightarrow$ `(integerp (+ x -1))`

Since our focus is on reasoning about computation and not on formally reasoning about number theory, we will allow ourselves to freely use any high-school level arithmetic theorems with the hint "Arith" or "Arithmetic." We will focus on reasoning about lists and numbers, so we will not review the axioms for strings, characters, etc. Once you can reason about lists and numbers, reasoning about strings, characters, etc. is a straight-forward extension.

One of our most powerful and widely-used rules of inference is instantiation.

Instantiation: If ϕ is a theorem and σ is a substitution, then $\phi|_\sigma$ is a theorem.

For example, since this is a theorem:

```
(== (car (cons x y)) x)
```

we can conclude that the following is a theorem by instantiation.

```
(== (car (cons (foo x) (bar z))) (foo x))
```

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

`((`var_1 $term_1$`)` ... `(`var_n $term_n$`))`

where the var_i are variables (symbols such as `x`, `y`, etc.) that we refer to as *target variables* and the $term_i$ are expressions that we refer to as *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.

Compare this version of instantiation with the propositional logic version.

The definition of a free occurrence of a variable will be provided with the help of the following examples. The first point is that there is a distinction between function symbols and variable symbols. Consider:

$\texttt{(f (g f g) 'f)}|_{\texttt{((f x) (x g) (g f))}} = \texttt{(f (g x f) 'f)}$

Notice that we only substitute an occurrence of a symbol that corresponds to a variable, which is why the occurrence of `f` in function position `(f ...)` was not changed. On the other hand, the occurrence of `f` in `(g f g)` is a variable and it was replaced by its image in the substitution. Also, `'f` is not a variable. It is a constant and it is not affected by any substitution. The general rule is that quoted objects are never modified by a substitution. One more thing to notice with this example is that substitutions are similar to `let`s: you do not keep recursively applying the substitution. For example the free occurrence of `g` became `f`, but we do not recursively keep applying the substitution, which would wind up leading to an infinite loop. If you want an algorithmic way of thinking about substitution, then just start copying the expression until you get to a free occurrence of a target variable and replace it by its image and then continue on with the rest of the expression. Once you get to the end of the expression, you are done. This part is exactly the same as substitutions in propositional logic, something we have already covered and seen many times.

Another point involves bound occurrences of variables. Substitution in the presence of binding forms such as `let` requires care. Consider:

$\texttt{(list a b c (let ((a 0) (c a)) (list a b c)))}|_{\texttt{((a b) (b a) (c a))}}$

In the body of the above `let`, the variables `a` and `c` are *bound*, but the variable `b` is *free*. Free variables are problematic, as we will see shortly, so ACL2s treats `let` expressions with free variables as abbreviations for expressions without free variables by adding identity bindings. For example, the above expression becomes:

$$\texttt{(list a b c (let ((a 0) (c a) (b b)) (list a b c)))}|_{\texttt{((a b) (b a) (c a))}}$$

Now, all variables in the body of the `let` are bound and substitution of `let` expressions only affects the second elements of the binding forms. Hence the above expression becomes:

```
(list b a a (let ((a 0) (c b) (b a)) (list a b c)))
```

Recall that `let` is a parallel binding so the second `a` in `(let ((a 0) (c b) (b a)) ...)` is free.

A `let*` form is equivalent to a nested `let` form, so we do not have to consider it as a special case.

Why is the definition of ACL2s substitution more complicated than it was in the case of propositional logic? Because the ACL2s logic is more powerful and because we want instantiation to be a valid rule of inference, as we use it all the time.

If we did not make distinctions between free and bound occurrences of variables, then instantiation would not be a valid rule of inference, *e.g.*, consider:

$$\varphi = \texttt{(=> (natp y) (= (let ((x 0)) (+ x y)) y))}, \sigma = \texttt{((x 1))}$$

Clearly φ is a theorem, as is:

$$\varphi|_\sigma = \texttt{(=> (natp y) (= (let ((x 0) (y y)) (+ x y)) y))}$$

But, if we were to replace every occurence of `x` with `1`, we would get the following, which is not even an expression, let alone a theorem.

```
(=> (natp y) (= (let ((1 0)) (+ 1 y)) y))
```

If we just replace the second occurrence of `x`, we get the following, which is an expression, but not a theorem.

```
(=> (natp y) (= (let ((x 0)) (+ 1 y)) y))
```

Consider the tricky substitution $\sigma' = \texttt{((y x))}$. Notice that the following is a theorem:

$$\varphi|_{\sigma'} = \texttt{(=> (natp x) (= (let ((x 0) (y x)) (+ x y)) x))}$$

But, if we did not add the identify binding for `y`, we would get the following, which is not a theorem.

```
(=> (natp x) (= (let ((x 0)) (+ x x)) x))
```

The above is an example of a problem often referred to as *variable capture* and there are many ways of dealing with this problem. For example, readers who have studied logic, will have probably seen definitions of substitution that use variable renaming to avoid variable capture. In ACL2s, we avoid the use of variable renaming.

To see why we distinguish between function symbols and variable symbols, consider the theorem

$$\varphi = \texttt{(consp (cons x y))}, \sigma = \texttt{((consp atom))}$$

Clearly, φ is a theorem as is

$$\varphi|_\sigma = \texttt{(consp (cons x y))}$$

But, if we were to replace every occurence of `consp` with `atom`, we would get the following, which is not a theorem.

`(atom (cons x y))`

Finally, to see why quoted objects are not variables, consider:

$$\varphi = \texttt{(symbolp 'x)}, \sigma = \texttt{((x 1))}$$

Clearly φ is a theorem, as is:

$$\varphi|_\sigma = \texttt{(symbolp 'x)}$$

But, if we were to replace every occurence of `x` with `1`, we would get the following, which is not a theorem.

`(symbolp '1)`

Notice that substitutions only replace variables. You cannot replace expressions or constants. To see why we have this restrictions, consider:

$$\varphi = \texttt{(= (+ 1 1) 2)}, \sigma = \texttt{((1 0))}$$

Clearly φ is a theorem, but, if we were to replace every occurence of `1` with `0`, we would get the following, which is not a theorem.

$$\varphi = \texttt{(= (+ 0 0) 2)}$$

Next consider:

$$\varphi = \texttt{(=> (natp x) (= (+ x x) (* 2 x)))}, \sigma = \texttt{(((+ x x) x))}$$

Clearly φ is a theorem, but, if we were to replace every occurence of `(+ x x)` with `x`, we would get the following, which is not a theorem.

$$\varphi = \texttt{(=> (natp x) (= x (* 2 x)))}$$

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

`(len (cons x (list z)))` $=$ `(len (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)
= { Def app, instantiation }
    (if (endp (cons x y)) z (cons (first (cons x y)) (app (rest (cons x y)) z)))
= { Def endp, consp axioms }
    (if nil z (cons (first (cons x y)) (app (rest (cons x y)) z)))
= { if axioms }
    (cons (first (cons x y)) (app (rest (cons x y)) z))
= { car-cdr axioms }
    (cons x (app y z))
```

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

Recall the definitions:

```
(definec true-listp (l :all) :bool
  (if (consp l)
      (true-listp (cdr l))
    (== l () )))

(definec endp (l :list) :bool
  (atom l))

(definec app (x :tl y :tl) :tl
  ; App appends two lists together
  (if (endp x)
      y
    (cons (first x) (app (rest x) y))))
```

Recall that `tlp` is abbreviation for `true-listp`, so we will be using the shorter `tlp` from now on. The definition of functions such as `app` give rise to *definitional axioms*. Here is the definitional axiom that `app` gives rise to:

```
(tlp x) ∧ (tlp y)
⇒
(app x y)
=
(if (endp x)
    y
  (cons (first x) (app (rest x) y)))
```

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

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

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

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 with `definec`. When using `defunc`, we get similar theorems. 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:

`(tlp (cons x y))` $\wedge$ `(tlp z)`

which is equivalent to:

`(tlp y)` $\wedge$ `(tlp z)`

So, what we really proved was:

Theorem 4.1 `(tlp y)` $\wedge$ `(tlp z)` $\Rightarrow$ `(app (cons x y) z)` $=$ `(cons x (app y z))`

When we write out proofs, we are not required to explicitly mention input contracts in hints 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. Just because we are not required to mention such hints does not mean that we must not mention them. If it seems clearer to mention hints, we will feel free to do so.

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.

Consider the following definition and (valid) conjecture.

```
(definec len2 (l :tl) :nat
  (if (endp l)
      0
    (+ 1 (len2 (tail l)))))
```

Conjecture 6 `(== x nil)` $\Rightarrow$ `(len2 x)` $\neq$ `0`

Contract checking passes, so there is no need for contract completion.

Our context is:

C1. `(== x nil)`

Consider the buggy following "proof" of the conjecture.

```
    (len2 x)
= {  Def len2, C1  }
    (if (endp x) 0 (+ 1 (len2 (tail nil))))
= {  Evaluation of (+ 1 (len2 (tail nil)))  }
    Contract Violation
```

The above "proof" is incorrect because we cannot evaluate expressions unless function contracts are satisfied. Just because we performed contract checking of the whole conjecture does not mean that we cannot wind up with subexpressions in our proof that fail contract checking. That is why you should always check contracts, even if you are not required to explicitly state in the hints that you did so. This applies to the use of any theorems, the expansion of function definitions, the use of evaluation, etc. This example is somewhat pathological and highlights why, as a general rule, we do not expand out definitions unless we can determine which case of the `if` or `cond` we wind up with. Notice that we *can* evaluate the whole `if` expression, thereby obtaining `0`. However, we do not restict how one can use evaluation, theorems, definitions, etc., because we want the flexibility to simplify subexpression when it makes sense to do so. The example is only meant to highlight that you do have check contracts even if you do not have to mention that you are doing so in a hint.

Let's look at another example:

Conjecture 7 `(endp x)` $\Rightarrow$ `(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 8 `(tlp x)` $\wedge$ `(tlp y)` $\wedge$ `(tlp 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:

```
(=> (and (tlp x)
         (tlp y)
         (tlp z)
         (endp x))
    (== (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 8 is:

C1. `(tlp x)`

C2. `(tlp y)`

C3. `(tlp 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 extend our context with a *derived context*:

D1. `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 and that all elements of our derived context will be given a label of the form Di, 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 Derived Context 1, we write D1.

```
 (app (app x y) z)
= {  Def app, D1, Def endp, if axioms  }
  (app y z)
= {  Def app, D1, Def 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 app  }
  (app (if (endp x) y (cons (first x) (app (rest x) y))) z)
= {  D1  }
  (app (if (endp nil) y (cons (first nil) (app (rest nil) y))) z)
= {  Def 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 8.

One other comment about the first proof of Conjecture 8. 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 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 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 8 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))
  (=> (and (tlp x)
           (tlp y)
           (tlp z)
           (endp x))
      (== (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?
 (=> (and (tlp x)
          (tlp y)
          (tlp z)
          (endp x))
     (== (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`.

```
(definec app2 (x :tl y :tl) :tl
  (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?
 (=> (and (tlp x) (tlp y))
     (== (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 many semantically 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?
 (=> (and (tlp x)
          (tlp y)
          (in a (app2 x y)))
     (in a (app x y))))

; check that if a is in app, it is in app2
(test?
 (=> (and (tlp x)
          (tlp 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 9

```
(consp x)
⇒
```

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

C is
```
[(tlp x) ∧ (tlp y) ∧ (tlp 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 9. Here's why. Remember *exportation*, a propositional validity we have already encountered.

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

We will use exportation almost all the time. We now use it to rewrite Conjecture 9 so that the context has as many conjunctions as possible. After applying exportation to Conjecture 9, we get:

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

Applying exportation again and rearranging conjuncts gives us:

Conjecture 10

```
[(consp x) ∧
 (tlp x) ∧
 (tlp y) ∧
 (tlp z) ∧
 [(tlp (rest x)) ∧ (tlp y) ∧ (tlp 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. `(tlp x)`

C3. `(tlp y)`

C4. `(tlp z)`

C5. `[(tlp (rest x))` $\wedge$ `(tlp y)` $\wedge$ `(tlp z)]` $\Rightarrow$
`[(app (app (rest x) y) z) = (app (rest x) (app y z))]`

Notice that we *cannot* use exportation 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 in the derived context. Here's how:

D1. `(tlp (rest x))` { Def `tlp`, C2, C1 }

D2. `(app (app (rest x) y) z)` = `(app (rest x) (app y z))` { C5, D1, C3, C4, MP }

So, notice what we did. First we added D1 to our derived context. How did we get D1? Well, we know `(tlp x)` (C2) and `(consp x)` (C1) so if we use the definitional axiom of `tlp`, we get D1: `(tlp (rest x))`.

Now, we have extended our context to include the antecedent of D1, so by propositional logic (*Modus Ponens*, abbreviated MP), we get that the conclusion also holds, *i.e.*, D2.

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 10.

```
    (app (app x y) z)
= {  Def 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))
= {  D2  }
     (cons (first x) (app (rest x) (app y z)))
= {  Def 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

$$\texttt{x} = 1 \;\Rightarrow\; 0 = 1 \tag{4.5}$$

Context:

C1. $\texttt{x} = 1$

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

So, now we have a "proof" of (4.5), but using (4.5) we can get:

$$\texttt{nil} \tag{4.6}$$

How?
Instantiate (4.5) with `((x 1))`, use Propositional logic, and Arithmetic.
Now we have a proof for any conjecture we want, *e.g.*,

$$\phi \tag{4.7}$$

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

$$\texttt{nil} \;\Rightarrow\; \phi$$

Now, ϕ follows using (4.6) and Modus Ponens.

The point is that a context, including the derived context, is *completely* different from a theorem. The context of (4.5) does not tell us that for all `x`, $\texttt{x} = 1$. It just tells us that $\texttt{x} = 1$ in the context of conjecture (4.5). 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 Undecidability of Equational Reasoning

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. In fact, ACL2s does that automatically.

Is this always the case?

No.
Consider Fermat's last theorem.

Conjecture 11 *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. We start with the following definition.

```
(definec f (x :pos y :pos z :pos n :nat) :bool
  :input-contract (> n 2)
  (!= (+ (expt x n) (expt y n))
      (expt z n)))
```

Now consider the following conjecture.

```
(thm (=> (and (posp x)
              (posp y)
              (posp z)
              (natp n)
              (> n 2))
         (f x y z n)))
```

So, proving theorems may be hard.

Notice also that if we added the following output contract:

```
  :output-contract (== (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.

In fact, the satisfiability problem for arithmetic expressions over the integers, using only $=, +, \cdot$ is *undecidable.* That means that no algorithm exists that given such an arithmetic expression returns "yes" if there is an assignment that makes the expression true and "no" otherwise. Notice that this also means that the validity problem is undecidable because ϕ is satisfiable iff $\neg\phi$ is not valid, *i.e.*, if we had a decision procedure for validity, we could use it to obtain a decision procedure for satisfiability. We will see a proof of a classic undecidability result (the undecidability of the halting problem) in a subsequent chapter.

We end by showing that even admitting `definec` functions in ACL2s is no easier than proving the validity of formulas. Consider any conjecture ϕ over variables $x_1, \ldots, x_n$. Now consider the following ACL2s code.

```
(defdata true t)
(definec f (x₁ :all ··· xₙ :all) :true
  φ)
```

ACL2s has to prove ϕ to admit `f` because the function contract is:

(=> (and (allp x_1) ... (allp x_n)) (truep ϕ))

but since `(allp x)` $=$ `t`, this is equivalent to:

(truep ϕ)

Once you have a logic that includes basic number theory there is no shortage of simple interesting questions whose answer is unknown.

Exercise 4.2 *Formalize the following false claim in ACL2s using* `test?` *and find a counterexample.*
Claim: The equation $x^3 + y^3 + z^3 = 29$ *does not have a solution in* $\mathbb{Z}$ *(*$\mathbb{Z}$ *is the integers).*

Exercise 4.3 *Formalize the following claim in ACL2s using* `test?`*. If false, find a counterexample. If true, prove it (any proof is fine).*
Claim: The equation $x^3 + y^3 + z^3 = 30$ *does not have a solution in* $\mathbb{Z}$ *(*$\mathbb{Z}$ *is the integers).*
Hint: This is hard.

Exercise 4.4 *Formalize the following claim in ACL2s using* `test?`*. If false, find a counterexample. If true, prove it (any proof is fine).*
Claim: The equation $x^3 + y^3 + z^3 = 33$ *does not have a solution in* $\mathbb{Z}$ *(*$\mathbb{Z}$ *is the integers).*
Hint: This is very hard and was an unsolved problem.

Exercise 4.5 *Formalize the following claim in ACL2s using* `test?`*. If false, find a counterexample. If true, prove it (any proof is fine).*
Claim: The equation $x^3 + y^3 + z^3 = 114$ *does not have a solution in* $\mathbb{Z}$ *(*$\mathbb{Z}$ *is the integers).*
Hint: This is very hard and is currently an unsolved problem.

Exercise 4.6 *Formalize the following claim in ACL2s using* `test?`*. If false, find a counterexample. If true, prove it (any proof is fine).*
Claim: The equation $x^3 + y^3 + z^3 = 95$ *does not have a solution in* $\mathbb{Z}$ *(*$\mathbb{Z}$ *is the integers).*

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.

```
(definec sum (n :nat) :nat
  (if (zp n)
      0
    (+ n (sum (1- n)))))
```

We can prove that `(sum n)` $= \frac{\texttt{n(n+1)}}{2}$, which is formalized as:

```
(=> (natp n)
    (= (sum n)
       (/ (* n (+ n 1)) 2)))
```

by mathematical induction. How?
First, we have the base case.

$$\texttt{(natp n)} \wedge \texttt{(= n 0)} \Rightarrow \texttt{(sum n)} = \texttt{(/ (* n n+1) 2)} \tag{4.8}$$

Second, we have the induction step.

$$\begin{aligned} &\texttt{(natp n)} \wedge \texttt{n} \neq 0 \wedge (\texttt{(natp n-1)} \Rightarrow \texttt{(sum n-1)} = \texttt{(/ (* n-1 n) 2)}) \\ \Rightarrow\ &\texttt{(sum n)} = \texttt{(/ (* n n+1) 2)} \end{aligned} \tag{4.9}$$

Here is the proof, starting with (4.8).
Context:

C1. `(natp n)`

C2. `(= n 0)`

Proof:

```
    (sum n)
= {  Def sum, C2  }
    0
= {  Arithmetic, C2  }
    (/ (* n n+1) 2)
```

Here is the proof of (4.9).
Context:

C1. `(natp n)`

C2. $\texttt{n} \neq 0$

C3. `(natp n-1)` $\Rightarrow$ `(sum n-1) = (/ (* n-1 n) 2)`

Derived Context:

D1. `(natp n-1)` { C1, C2 }

D2. `(sum n-1) = (/ (* n-1 n) 2)` { C3, D1, MP }

Proof:

```
    (sum n)
```

= { Def `sum`, C2 }
    n + `(sum n-1)`
= { D2 }
    n + `(/ (* n-1 n) 2)`
= { Arithmetic }
    $(2n + n(n-1))/2$
= { Arithmetic }
    $(2n + n^2 - n)/2$
= { Arithmetic }
    $(n^2 + n)/2$
= { Arithmetic }
    $n(n+1)/2$

4.6 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 (`car-cdr` axioms, `if` axioms, ...), all the definitional axioms (Def `app`, `len`, ...), all the contract theorems (Contract 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. `(tlp x)`

then, we create the derived context and populate it as follows.

D1. `x` $=$ `nil` { C1, C2 }

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

C1. `(consp x)`

C2. `(tlp x)`

C3. `(tlp (rest x))` $\Rightarrow$ ϕ

then the the derived context includes:
Derived Context:

D1. `(tlp (rest x))` { C1, C2, Def `tlp` }

D2. ϕ { C3, D1, MP }

where MP is Modus Ponens.

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

Conjecture 12 $x \leq xy$ *if* $y \geq 1$

Question: 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
  (=> (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))
  (=> (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?
  (=> (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
  (=> (and (rationalp x)
           (rationalp y)
           (>= y 1))
      (<= x (* x y))))
```

Of course, this isn't a theorem.

Let's consider another example:

Conjecture 13 *x(y+z) = xy + xz*

How do we write this in ACL2s?

```
(thm
  (=> (and (rationalp x)
           (rationalp y)
           (rationalp z))
      (= (* 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))
  (= (* x (+ y z))
     (+ (* x y) (* x z))))
```

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

```
(test?
 (=> (and (rationalp x)
          (rationalp y)
          (rationalp z))
     (= (* 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 (=> (and (rationalp x)
              (rationalp y)
              (rationalp z))
         (= (* 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

```
(definec app (a :tl b :tl) :tl
  (if (endp a)
      b
    (cons (first a) (app (rest a) b))))

(definec lrev (x :tl) :tl
  (if (endp x)
      nil
    (app (lrev (rest x)) (list (first x)))))

(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (== a (first X))
           (in a (rest X)))))

(definec del (a :all X :tl) :tl
  (cond ((endp X) nil)
        ((== a (car X)) (cdr X))
        (t (cons (car X) (del a (cdr X))))))
```

Conjecture 14

```
(tlp x)
⇒
(in a x)   ⇒   (! (in a (del a x)))
```

Using induction (something we will describe later), the above conjecture leads to the following proof obligation:

```
(and (=> (tlp x)
         (=> (endp x)
             (=> (in a x)
                 (! (in a (del a x))))))
     (=> (tlp x)
         (=> (and (consp x)
                  (== a (first x)))
             (=> (in a x)
                 (! (in a (del a x))))))
     (=> (tlp x)
         (=> (and (consp x)
                  (!= a (first x))
                  (=> (tlp (rest x))
                      (=> (in a (rest x))
                          (! (in a (del a (rest x)))))))
             (=> (in a x)
                 (! (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 14 is false, *e.g.*, consider

```
(let ((x '(1 1))
      (a 1))
  ...)
```

where ... in the above let is Conjecture 14.

What about the following conjecture?

Conjecture 15

```
(tlp x)
⇒
(in a x)   ⇒   (in a (app x y))
```

Which by induction leads to the following proof obligation:

```
(and (=> (tlp x)
         (=> (endp x)
             (=> (in a x)
                 (in a (app x y)))))
     (=> (tlp x)
         (=> (and (consp x)
                  (== a (first x)))
             (=> (in a x)
                 (in a (app x y)))))
     (=>
      (tlp x)
      (=>
       (and (consp x)
            (!= a (first x))
            (=> (tlp (rest x))
                (=> (in a (rest x))
                    (in a (app (rest x) y)))))
       (=> (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 15 into three parts and proving each in turn.

When reasoning about programs, it is often useful to decompose a proof into parts, so we introduce the decomposition proof technique.

Decomposition: If $\phi_1, \ldots, \phi_n$ are formulas then $\phi_1 \wedge \cdots \wedge \phi_n$ is valid iff all of $\phi_1, \ldots, \phi_n$ are valid.

Decomposition allows us to break up a proof of conjunction into a proof of its parts. To see why this is sound, note that from propositional logic $(\phi_1 \wedge \cdots \wedge \phi_n) \Rightarrow \phi_i$ for all i s.t. $1 \leq i \leq n$. Similarly, if each of $\phi_1, \ldots, \phi_n$ are valid, so is their conjunction.

Proofs of the three parts appear in Figures 4.1, 4.2 and 4.3. These proofs highlight the proof format we use when writing proofs on a computer. If we are writing proofs with paper and pencil, we will follow a similar proof format, but we will typically not explicitly write the exportation, contract completion and goal steps.

The proof of the first part, in Figure 4.1, shows that it is possible for us to prove a theorem in the context, if we can derive `nil`, because `nil` implies anything. The "QED" indicates the end of a proof and comes from the Latin *Quod Erat Demonstrandum*, which means "that which was to be demonstrated."

Conjecture 15-part-1:

```
(=> (tlp x)
    (=> (endp x)
        (=> (in a x)
            (in a (app x y)))))
```

Exportation:

```
(=> (and (tlp x)
         (endp x)
         (in a x))
    (in a (app x y)))
```

Contract Completion:

```
(=> (and (tlp x)
         (tlp y)
         (endp x)
         (in a x))
    (in a (app x y)))
```

Context:

```
C1. (tlp x)
C2. (tlp y)
C3. (endp x)
C4. (in a x)
```

Derived Context:

```
D1. (== x nil) { C1, C3, Def tlp }
D2. nil { Def in, C4, D1 }
```

QED

Figure 4.1: Proof format example using only context

The proof of the second part, in Figure 4.2, shows that the Derived Context section of the proof is optional. In fact, if no exportation is possible, the Exportation section is optional. If no contract completion is needed, then the Contract Completion section is optional. If we derive `nil`, even the Goal and Proof sections are optional.

The proof of the third part, in Figure 4.3 includes all possible sections that we currently have available (more sections will be introduced later). This proof also provides an example of nested implications. Notice that the fourth top-level hypothesis in the Exportation section has to remain an implication because exportation does not allow us to take its hypotheses and make them top-level hypotheses. Make sure you understand this! However, we did use exportation to simplify the fourth top-level hypothesis, so always apply exportation as much as possible everywhere you can, not just at the top level. During the exportation step, we allow any propositional simplification, as long as the resulting formula does not allow any further exportation simplifications.

When we have nested implications, as we do in C5, one of the goals when constructing the derived context is to get our hands of the consequent of such implications. The proof in Figure 4.3 shows an example of that. The idea is to establish all the hypotheses of C5 (D1 and D2 in our example) and to then use Modus Ponens (MP) to obtain the conclusion (D3). The conclusion is typically what we need in the proof.

Conjecture 15-part-2:

```
(=> (tlp x)
    (=> (and (consp x)
             (== a (first x)))
        (=> (in a x)
            (in a (app x y)))))
```

Exportation:

```
(=> (and (tlp x)
         (consp x)
         (== a (first x))
         (in a x))
    (in a (app x y)))
```

Contract Completion:

```
(=> (and (tlp x)
         (tlp y)
         (consp x)
         (== a (first x))
         (in a x))
    (in a (app x y)))
```

Context:

```
C1. (tlp x)
C2. (tlp y)
C3. (consp x)
C4. (== a (first x))
C5. (in a x)
```

Goal: `(in a (app x y))`

Proof:

```
    (in a (app x y))
= { Def app, C1, C3 }
    (in a (cons (first x) (app (rest x) y)))
= { Def in, car-cdr axioms, C3 }
    (or (== a (first x)) (in a (app (rest x) y)))
= { C4, PL }
    t
QED
```

Figure 4.2: Proof format with no derived context

Conjecture 15-part-3:

```
(=> (tlp x)
    (=> (and (consp x)
             (!= a (first x))
             (=> (tlp (rest x))
                 (=> (in a (rest x))
                     (in a (app (rest x) y)))))
        (=> (in a x)
            (in a (app x y)))))
```

Exportation:

```
(=> (and (tlp x)
         (consp x)
         (!= a (first x))
         (=> (and (tlp (rest x))
                  (in a (rest x)))
             (in a (app (rest x) y)))
         (in a x))
    (in a (app x y)))
```

Contract Completion:

```
(=> (and (tlp x)
         (tlp y)
         (consp x)
         (!= a (first x))
         (=> (and (tlp (rest x))
                  (in a (rest x)))
             (in a (app (rest x) y)))
         (in a x))
    (in a (app x y)))
```

Context:

```
C1. (tlp x)
C2. (tlp y)
C3. (consp x)
C4. (!= a (first x))
C5. (=> (and (tlp (rest x)) (in a (rest x)))
             (in a (app (rest x) y)))
C6. (in a x)
```

Derived Context:

```
D1. (tlp (rest x)) { Def tlp, C1, C3 }
D2. (in a (rest x)) { Def in, C6, C3, C4 }
D3. (in a (app (rest x) y)) { C5, D1, D2, MP }
```

Goal: `(in a (app x y))`

Proof:

```
  (in a (app x y))
= { Def app, C1, C3 }
  (in a (cons (first x) (app (rest x) y)))
= { Def in, car-cdr axioms, C3 }
  (or (== a (first x)) (in a (app (rest x) y)))
= { D3, PL }
  t
```

QED

Figure 4.3: Proof format with nested implications

4.7 Exercises

Exercise 4.7 *Use* `definec` *to define a function* `del-all` *that given* `a` *of type* `:all` *and* `X` *of type* `:tl` *deletes all occurrences of* `a` *from* `X`.

Prove the following conjectures (which will require contract completion).

```
Conjecture del-all-1:
(=> (endp x) (! (in a (del-all a x))))

Conjecture del-all-2:
(=> (consp x)
    (=> (== a (first x))
        (=> (=> (tlp (rest x))
                (! (in a (del-all a (rest x)))))
            (! (in a (del-all a x))))))

Conjecture del-all-3:
(=>
 (consp x)
 (=>
  (!= a (first x))
  (=> (=> (tlp (rest x))
          (! (in a (del-all a (rest x)))))
      (! (in a (del-all a x))))))
```

Consider the definition of `nodups`, a function that checks is a true-list has no duplicate elements.

```
(definec nodups (l :tl) :bool
  (or (endp l)
      (and (! (in (first l) (rest l)))
           (nodups (rest l)))))
```

The function `num-unique` determines how many unique elements a true-list contains.

```
(definec num-unique (l :tl) :nat
  (cond ((endp l) 0)
        ((in (first l) (rest l))
         (num-unique (rest l)))
        (t (+ 1 (num-unique (rest l))))))
```

Exercise 4.8 *Prove the following claim. Use propositional logic to break the claim into cases.*

```
(=> (tlp l)
    (=> (or (endp l)
            (and (! (endp l))
                 (in (first l) (rest l))
                 (=> (tlp (rest l))
                     (<= (num-unique (rest l))
                         (llen (rest l))))))
```

```
          (and (! (endp l))
               (! (in (first l) (rest l)))
               (=> (tlp (rest l))
                   (<= (num-unique (rest l))
                       (llen (rest l))))))
      (<= (num-unique l)
          (llen l))))
```

Exercise 4.9 *Prove the following claim. Use propositional logic to break the claim into cases.*

```
(=> (and (tlp l)
         (nodups l))
    (and (=> (endp l)
             (= (num-unique l) (llen l)))
         (=> (and (! (endp l))
                  (=> (and (tlp (rest l))
                           (nodups (rest l)))
                      (= (num-unique (rest l))
                         (llen (rest l)))))
             (= (num-unique l) (llen l)))))
```

Exercise 4.10 *You are given the following lemma.*

```
(=> (tlp l)
    (<= (num-unique l) (llen l)))
```

Prove the following claim. Use propositional logic to break the claim into cases.

```
(and
 (=> (and (tlp l) (endp l))
     (== (= (num-unique l) (llen l))
         (nodups l)))
 (=> (and (tlp l)
          (! (endp l))
          (in (car l) (cdr l))
          (=> (tlp (cdr l))
              (== (= (num-unique (cdr l))
                     (llen (cdr l)))
                  (nodups (cdr l)))))
     (== (= (num-unique l) (llen l))
         (nodups l)))
 (=> (and (tlp l)
          (! (endp l))
          (! (in (car l) (cdr l)))
          (=> (tlp (cdr l))
              (== (= (num-unique (cdr l))
                     (llen (cdr l)))
                  (nodups (cdr l)))))
     (== (= (num-unique l) (llen l))
         (nodups l))))
```

Exercise 4.11 *You are given the following lemma.*

```
(=> (and (tlp x) (tlp y))
    (== (in a (app x y))
        (or (in a x) (in a y))))
```

Prove the following claim. Use propositional logic to break the claim into cases.

```
(and
 (=> (and (tlp x) (tlp y))
     (=> (endp x)
         (<= (num-unique (app x y))
             (+ (num-unique x) (num-unique y)))))
 (=> (and (tlp x) (tlp y))
     (=> (consp x)
         (=>
          (=> (tlp (cdr x))
              (<= (num-unique (app (cdr x) y))
                  (+ (num-unique (cdr x)) (num-unique y))))
          (<= (num-unique (app x y))
              (+ (num-unique x) (num-unique y)))))))
```

Exercise 4.12 *You are given the following lemma.*

```
(=> (and (tlp x) (tlp y))
    (== (in a (app x y))
        (or (in a x) (in a y))))
```

Prove the following claim. Use propositional logic to break the claim into cases.

```
(and
 (=> (and (tlp x) (endp x))
     (== (in a (lrev x)) (in a x)))
 (=> (and (tlp x)
          (! (endp x))
          (=> (tlp (cdr x))
              (== (in a (lrev (cdr x)))
                  (in a (cdr x)))))
     (== (in a (lrev x)) (in a x))))
```

Exercise 4.13 *You are given the following lemmas.*

```
(=> (and (tlp x)
         (tlp y))
    (= (num-unique (app x y))
       (num-unique (app y x))))
(=> (tlp x)
    (== (in a (lrev x))
        (in a x)))
```

Prove the following claim. Use propositional logic to break the claim into cases.

```
(and
 (=> (and (tlp x) (endp x))
     (= (num-unique (lrev x))
        (num-unique x)))
 (=> (tlp x)
     (=> (and (consp x)
              (in (car x) (cdr x))
              (=> (tlp (cdr x))
                  (= (num-unique (lrev (cdr x)))
                     (num-unique (cdr x)))))
         (= (num-unique (lrev x))
            (num-unique x))))
 (=> (and (tlp x)
          (consp x)
          (=> (and (! (in (car x) (cdr x)))
                   (=> (tlp (cdr x))
                       (= (num-unique (lrev (cdr x)))
                          (num-unique (cdr x)))))
              (= (num-unique (lrev x))
                 (num-unique x))))))
```

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)
```

that ACL2s adds the definitional axiom

$$ic \;\Rightarrow\; \texttt{(f x)} = body$$

and the function contract theorem

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

We now more carefully examine what happens when you define functions.

The fundamental definitions and concepts of this chapter are somewhat easier to explain using `defunc`, as opposed to `definec`. Furthermore, since `definec` forms can be turned into `defunc` forms, it is enough to only consider `defunc` forms. However, we will use `definec` when considering specific examples.

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:

```
(definec f (x :nat) :nat
  (1+ (f x)))
```

This is a nonterminating recursive function.

Suppose we add the definitional axiom:

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{(f x)} = \texttt{(1+ (f x))} \tag{5.1}$$

and the function contract theorem:

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{(natp (f x))} \tag{5.2}$$

This is unfortunate because we can now prove `nil` in ACL2s. If `nil` is a theorem, that means that the ACL2s logic is *unsound*. Here is the proof of unsoundness. This is an interesting proof that just uses the derived context.

D1. `(f 1) = 1 + (f 1)` { Def `f` }

D2. `0 = 1` { D1, Contract `f`, Arith }

D3. `nil` { D2, evaluation }

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 in `:logic` mode using `defunc` and `definec`, unless you explicitly allow such definitions using methods we will not consider in this chapter.

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 nonterminating 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 will provide a careful proof of this, but, assuming the undecidability results in Chapter 4, you should be able to prove that checking termination is undecidable.

Exercise 5.1 *Show that checking termination of a proposed ACL2s function definition is undecidable. Assume the undecidability results in Chapter 4.*

Hint: Prove that if termination was decidable, then we could use it to prove that the satisfiability problem for arithmetic expressions over the integers is decidable, thereby contradicting the results in Chapter 4.

We note that there are cases in which nontermination is desirable. In particular, *reactive systems*, which include operating systems and communication protocols, are intentionally nonterminating. 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, nonterminating 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 nonterminating recursive equation introduce unsoundness?

Consider:

```
(definec f (x :all) :all
  (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:

```
(definec f (x :all) :all
  y)
```

This leads to the definitional axiom:

$$\texttt{(f x)} = \texttt{y} \tag{5.3}$$

Which causes problems, *e.g.*,

`t`

= { Instantiation of (5.3) with `((y t) (x 0))` }

`(f 0)`

= { Instantiation of (5.3) with `((y nil) (x 0))` }

`nil`

We got into trouble because we allowed a "global" variable. It will turn out that, modulo contracts, 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 regular users 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 (see the discussion of what a free variable is in Chapter 4) other than the 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. This means that if you evaluate the function on any inputs that satisfy the input contract, the function will terminate. As we saw, nontermination can lead to unsoundness.

There are also two other conditions that I state separately.

5. *ic* ⇒ *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* ⇒ `[(f` $\mathtt{x}_1$ `...` $\mathtt{x}_n$`) =` *body*`]`.

2. Add the *Contract Theorem* for `f`: *ic* ⇒ *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 under consideration into the natural numbers, so that we can prove that on every recursive call the function terminates. Let's try this with `app2`.

```
(definec app2 (x :tl y :tl) :tl
  (if (endp x)
      y
    (cons (car x) (app2 (cdr x) y))))
```

What is a measure function for `app2`?

How about the length of `x`? That works, *i.e.*, `(len x)` is a measure function for `app2`.

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 of `f`, `m` applied to the arguments to that recursive call decreases with respect to `<`, under the conditions that led to the recursive call.

Here is a measure function for `app2`:

```
(definec m (x :tl y :tl) :nat
  (len x))
```

If you try admitting `m` in ACL2s, you get an error because `y` is not used in the body of `m`. Here is one way to tell ACL2s to allow such definitions.

```
(definec m (x :tl y :tl) :nat
  (declare (ignorable y))
  (len x))
```

The above measure function is non-recursive, 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.

The astute reader may be wondering if is possible to ease the restriction that `m` is defined over the parameters of `f` and only require that `m` is defined over a subset of the parameters of `f`. That is possible; stay tuned.

Next, we have to prove that `m` decreases on all recursive calls of `app2`, under the conditions that led to the recursive call. Since there is one recursive call, we have to show:

```
(implies (and (tlp x)
              (tlp y)
              (not (endp x)))
         (< (m (rest x) y) (m x y)))
```

which is equivalent to:

```
(implies (and (tlp x)
              (tlp y)
              (not (endp x)))
         (< (len (rest x)) (len x)))
```

which is a true statement. How do we prove such statement? Using equational reasoning, of course.

Wait, what about `len`? How do we know that `len` is terminating? We have an axiom that the following function is admissible, hence terminating.

```
(definec cons-size (x :all) :nat
  (if (consp x)
      (+ 1 (cons-size (car x)) (cons-size (cdr x)))
    0))
```

This axiom tells us that cons trees, and therefore lists, are finite! In Lisp you can actually have circular trees and lists, in which case `cons-size` would be nonterminating, but that is not possible in ACL2s! We can use `cons-size` as the measure function for `len`.

More examples:

```
(definec rev2 (x :tl) :tl
  (if (endp x)
      nil
    (app2 (rev2 (rest x)) (list (first x)))))
```

Is this admissible? It depends if we defined `app2` already. Suppose `app2` is defined as above. What is a measure function?

`len`.

What about (`head` and `tail` were defined in Chapter 2):

```
(definec drop-last (x :tl) :tl
  (if (= (len x) 1)
      nil
    (cons (head x) (drop-last (tail x)))))
```

No. We cannot prove that it is nonterminating, *e.g.*, when `x` is `nil`, what is `(tail 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 `(head x)`. What about this version? Is it admissible?

```
(definec drop-last (x :tl) :tl
  (if (= (len x) 1)
      nil
    (cons (first x) (drop-last (rest x)))))
```

No. In fact it is nonterminating. Why?

We can fix that in several ways.

Exercise 5.2 *Define* `drop-last` *using a data-driven definition.*

Here is the solution to the above exercise.

```
(definec drop-last (x :tl) :tl
  (cond ((endp x) nil)
        ((endp (rest x)) nil)
        (t (cons (first x) (drop-last (rest x))))))
```

An equivalent definition is the following.

```
(definec drop-last (x :tl) :tl
  (if (endp (rest x))
      nil
    (cons (first x) (drop-last (rest x)))))
```

Why does the above pass contract checking?

Exercise 5.3 *Find a measure function for* `drop-last` *and prove that it works.*

What about the following function?

```
(definec prefixes (l :tl) :tl
  (if (endp l)
      '( () )
      (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 5.4 *What is a measure function for* `prefixes`*? Try to prove that it is a measure function. What happened?*

Does the following satisfy the definitional principle?

```
(definec f (x :int) :int
  (if (zip x)
      0
    (1+ (f (1- x)))))
```

No. It does not terminate.

What went wrong?

Maybe we got the input contract wrong. Maybe we really wanted natural numbers.

```
(definec f (x :nat) :int
  (if (zp x)
      0
    (1+ (f (1- x)))))
```

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 data definition:

```
(definec f (x :int) :int
  (cond ((zip x) 0)
        ((> x 0) (1+ (f (1- x))))
        (t (1+ (f (1+ x))))))
```

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 (x_1 ... x_n)
  :input-contract (and ... (tlp x_i) ...)
  :output-contract ...
  (if (endp x_i)
      ...
    (... (f ... (rest x_i) ...) ...)))
```

The above function has n parameters, where the i^{th} parameter, $\mathtt{x}_i$, is a list. The function recurs down the list $\mathtt{x}_i$. The ...'s in the body indicate non-recursive, well-formed code, and (rest $\mathtt{x}_i$) appears in the i^{th} position.

We can use (len $\mathtt{x}_i$) as the measure for any function conforming to the above scheme:

```
(defunc m (x_1 ... x_n)
  :input-contract (and ... (tlp x_i) ...)
  :output-contract (natp (m x_1 ... x_n))
  (len x_i))
```

That `m` is a measure function is obvious. The non-trivial part is showing that

```
(tlp x_i) ∧ (not (endp x_i))
⇒ (len (rest x_i)) < (len x_i)
```

which is easy to see.

So, this scheme is terminating. This is why all of the code you wrote in your introductory programming class that was based on the list data definition terminates.

We can generalize the above scheme, *e.g.*, consider:

```
(defunc f (x_1 x_2)
  :input-contract (and (tlp x_1) (tlp x_2))
  :output-contract (tlp (f x_1 x_2))
  (cond ((endp x_1) x_2)
        ((endp x_2) x_1)
        (t (list (f (rest x_1) (rest x_2))
                 (f (rest x_1) (app x_1 x_2 x_2))))))
```

We now have two recursive calls and two base cases. Nevertheless, the function terminates for the same reason: `len` decreases.

```
(defunc m (x_1 x_2)
  :input-contract (and (tlp $x_1$) (tlp $x_2$))
  :output-contract (natp (m $x_1$ $x_2$))
  (len $x_1$))
```

All recursive calls lead to the same proof obligation:

```
(tlp $x_1$) ∧ (not (endp $x_1$)) ∧ (not (endp $x_2$))
⇒ (len (rest $x_1$)) < (len $x_1$)
```

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.

Another interesting recursion scheme is the following.

```
(defunc f ($x_1$ ... $x_n$)
  :input-contract (and ... (natp $x_i$) ...)
  :output-contract ...
  (if (zp $x_i$)
      ...
      (... (f ... (1- $x_i$) ...) ...)))
```

The above is a function of n parameters, where the i^{th} 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 (1- x_i) appears in the i^{th} position.

We can use x_i as the measure for any function conforming to the above scheme:

```
(defunc m ($x_1$ ... $x_n$)
  :input-contract (and ... (natp $x_i$) ...)
  :output-contract (natp (m $x_1$ ... $x_n$))
  $x_i$)
```

That m is a measure function is obvious. The non-trivial part is showing that

```
(natp $x_i$) ∧ (not (zp $x_i$)) ⇒ (1- $x_i$) < $x_i$
```

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 5.5 *We can similarly construct a recursion scheme for integers. Do it.*

A recursion scheme for traversing cons trees is the following.

```
(defunc f ($x_1$ ... $x_n$)
  :input-contract (and ... (allp  $x_i$) ...)
  :output-contract ...
  (if (atom $x_i$)
      ...
    (... (f ... (car $x_i$) ...) ...
     ... (f ... (cdr $x_i$) ...) ...)))
```

The above function has n parameters, where the i^{th} parameter, x_i, can be anything. The function recurs down the car and cdr of x_i, if it is a cons. The ...'s in the body indicate non-recursive, well-formed code, and both (car x_i) and (cdr x_i) appear in the i^{th} position.

We can use (`cons-size` x_i) as the measure for any function conforming to the above scheme:

```
(defunc m (x₁ ... xₙ)
  :input-contract (and ... (allp xᵢ) ...)
  :output-contract (natp (m x₁ ... xₙ))
  (cons-size xᵢ))
```

That `m` is a measure function is obvious. The non-trivial part is proving the following two simple claims.

`(not (atom` x_i`))` $\Rightarrow$ `(cons-size (car` x_i`))` $<$ `(cons-size` x_i`)`
`(not (atom` x_i`))` $\Rightarrow$ `(cons-size (cdr` x_i`))` $<$ `(cons-size` x_i`)`

We can generalize the above scheme, in the way we generalized the recursion scheme for lists.

5.3 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 (roughly):

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 definition of "big-Oh" is a bit more complicated).

Thus, big-Oh analysis is just a refinement of termination, where in addition to being interested in whether a function terminates, we also want an upper bound on how long it will take for the function to terminate.

Exercise 5.6 *Claim: Let* `f` *be a function that has one argument,* `n`*, that is a* `nat`*. If a measure for* `f` *is* `n`*, then* `f` *is a linear time function. Prove or disprove before reading further.*

Consider:

```
(definec-no-test fib (n :nat) :nat
  (if (< n 2)
      n
    (+ (fib (1- n))
       (fib (- n 2)))))
```

The `definec-no-test` form is similar to a `definec` form, except that it does not perform any testing when it tries to admit `fib`.

Exercise 5.7 *A measure function for* `fib` *is one that just returns* `n`*. Prove that this is a measure function.*

The measure function in the above exercise tells us that there is no sequence of `fib` calls of length greater than `n`, *but* we can have a tree of calls, which we do in the case of `fib`, so even with such a simple measure function, the running time can be exponential. Thus the claim in Exercise 5.6 does not hold.

It should now be clear that a measure function does not count how many times a function is called recursively! A measure function tells us close to nothing about the running time of functions. To make this even clearer, consider the following definition.

```
(definec-no-test f (n :nat) :nat
  (fib n))
```

The function that always returns 0 is a measure function for `f`, yet `f` takes exponential time.

Given the above discussion, there is no reason to make measure functions as small as possible. The goal is use measure functions that are easy to define and easy to reason about.

Let us test `fib` to make it clear that it really is not a linear-time function. After admitting the function here are some timing results.

```
(time$ (fib 40)) ; ~1 seconds = (fib 2) seconds
(time$ (fib 41)) ; ~2 seconds = (fib 3) seconds
(time$ (fib 42)) ; ~3 seconds = (fib 4) seconds
(time$ (fib 43)) ; ~5 seconds = (fib 5) seconds
(time$ (fib 44)) ; ~8 seconds = (fib 6) seconds
```

What if I tried this?

```
(time$ (fib 200))
```

This would take about `(fib 162)` seconds, which is 3210056809456107725247980776292056 seconds, which is more than 10^{26} years, which is more than 10^{15} times the age of our universe (from the big bang until now).

You may wonder "How does he even know that, since computing `(fib 162)` requires about `(fib 124)` seconds to compute, which is 36726740705505779255899443 seconds, which is more than 10^{18} years, which is more than 10^{7} times the age of our universe." Now, I'm wondering "How does the reader even know that, since computing `(fib 124)` requires ...." Enough of that; let's get back to reasoning about programs.

Well, ACL2s has a very nice feature, which allows you to memoize functions. This gives you language support for dynamic programming, a key idea in algorithms. Memoization works by recording the values of `fib` that you compute in a table, so you never have to compute `fib` on the same value more than once.

After defining `fib`, you can tell ACL2s to *memoize* the function with the following command.

```
(memoize 'fib)
```

Now, you can run `fib` on large numbers quickly. For example, the following form completes in 0 seconds.

```
(time$ (fib 200)) ; Much faster than universe-scale computations!
```

5.4 Undecidability of the Halting Problem

Turing's result that termination is undecidable is an amazing, fundamental result that highlights the limits of computation. In this section, we present a proof of the undecidability of the halting problem.

Recall that a problem that has a "yes" or "no" answer is a *decision problem* and a *decision procedure* is an algorithm that solves a decision problem. The algorithm has to terminate and it has to correctly solve the decision problem. If there is a decision procedure for a decision problem, then we say that the problem is *decidable.* We have seen that validity of propositional formulas is a decidable problem. We have mentioned that the admissibility of ACL2s functions is an undecidable problem. Now, we will see our first proof of undecidability.

We have seen that it would very useful if languages other than ACL2s could help us determine whether programs are terminating. Ideally, we want a decision procedure for termination: an algorithm that given a program returns "yes" iff that program terminates on all legal inputs and returns "no" otherwise. This is the halting problem. Turing proved that it is *undecidable*, *i.e.*, no algorithm exists that decides termination. You may wonder how to reconcile this with the termination analysis ACL2s performs. ACL2s will admit functions if it can prove that they terminate. By Turing's result, there must exist programs that are terminating, but which ACL2s will not be able to admit. That indeed is the case. When ACL2s is unable to prove termination, user input is required in the form of a measure function, a proof sketch, etc. Since almost no language includes a logic and a theorem prover, they also do not provide a means for checking termination.

5.4.1 Proof by Contradiction

We will use a proof technique referred to as *proof by contradiction* or *reductio ad absurdum*, which is Latin for "reduction to the absurd." Proof by contradiction is just a "cheap" (*i.e.*, simple) propositional trick.

Proof by contradiction: Let's say that we are trying to prove the validity of the formula:

$$\phi_1 \ \wedge \cdots \ \wedge \phi_n \ \Rightarrow \ \phi$$

We can do this by assuming the negation of the consequent and deriving a contradiction, *i.e.*, we instead prove:

$$\phi_1 \ \wedge \cdots \ \wedge \ \phi_n \ \wedge \ \neg\phi \ \Rightarrow \ \mathit{false}$$

Note that these two statements are equivalent by propositional logic. To see this, recall *negate and swap*:

$$A \wedge B \ \Rightarrow \ C \ \ \equiv \ \ A \wedge \neg C \ \Rightarrow \ \neg B$$

Then apply the above to

$$\phi_1 \ \wedge \cdots \ \wedge \ \phi_n \ \wedge \mathit{true} \ \Rightarrow \ \phi$$

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.

5.4.2 Diagonalization

Our proof will be based on *diagonalization*, a powerful proof technique. Diagonalization was introduced by Cantor in the late 1800's to show that there is a infinite tower of infinities.

One of the key theorems Cantor proved states that the *cardinality* (size) of the power set of the natural numbers is greater than the cardinality of the set of natural numbers. We use ω to denote the set of natural numbers, as is standard in set-theory. The cardinality of set X is denoted by $|X|$. It also turns out that ω is a *cardinal number*, a number denoting cardinality, and it is the first infinite cardinal number (natural numbers are also cardinal numbers), so $|\omega| = \omega$. If the size of a set is $\leq \omega$ then it is finite or its size is ω; we call such sets *countable*. We use 2^X to denote the powerset of X.

If two sets have the same cardinality, there is a *bijection* between them. A bijection between sets X and Y is a (total) function from X to Y such that every element in Y is related to exactly one element in X:

$$\text{for all } y \in Y : |\{x \in X \ : \ f(x) = y\}| = 1$$

With this definition (which works not only for finite sets, but also for infinite sets), notice that the set of integers is countable, *i.e.*, it has the same size as ω. To show this, we exhibit an appropriate bijection: $f(i) = 2i$ if $i \geq 0$ and $f(i) = -(2i + 1)$ otherwise.

Exercise 5.8 *Show that the set of even natural numbers is countable.*

Exercise 5.9 *Show that the set* $X = \{(a, b) \ : \ a, b \in \omega\}$ *is countable.*

Exercise 5.10 *Show that the set of rational numbers (including negative rational numbers) is countable.*

Exercise 5.11 *Show that the union of a countable set of countable sets is countable.*

Cantor's theorem can be stated succinctly as $|2^X| > |X|$. Instead of proving the general version of Cantor's theorem, we will instead use Cantor's diagonalization proof technique to show that the size of the set of real numbers between 0 and 1 is strictly greater than the size of the natural numbers.

First, we start by assuming the negation of the conjecture. Since our set of reals is infinite, its size has to be greater than or equal to ω. So we can assume that the size of the set of reals is exactly ω.

Now that means that there is a bijection, say f, from ω to the reals between 0 and 1. Any such bijection can be turned into an infinite table where the contents of row r correspond to the decimal expansion of the $f(r)$, *i.e.*, cell r, c contains the c^{th} decimal digit of $f(r)$, which we denote as $f(r)_c$. You can imagine that there is a "." before every row (since the numbers are between 0 and 1).

	0	1	2	...
0	$f(0)_0$	$f(0)_1$	$f(0)_2$	...
1	$f(1)_0$	$f(1)_1$	$f(1)_2$	...
2	$f(2)_0$	$f(2)_1$	$f(2)_2$	...
...	...			...

We now derive a contradiction by showing that a number that should be in the table is not. How do we do that? We define a real number, d, that differs from every number in the table. In order for d to differ from $f(0)$, it only needs to differ in one column, *i.e.*, for one of the digits in its decimal expansion. The idea is to define d so that it differs on the diagonal, *i.e.*, $d_i \neq f(i)_i$ and we can do that by defining $d_i = 5$ if $f(i)_i \neq 5$ and $d_i = 4$ otherwise. Note that we used 4 and 5 to avoid some tricky cases, *e.g.*, recall that .0999... = .1000.... But now, d is not in the table but it is a real number between 0 and 1 and we have a contradiction and that means that our assumption that there is a bijection must be wrong. Hence the cardinality of the real numbers between 0 and 1 is strictly greater than ω.

Exercise 5.12 *Use diagonalization to show that* $\omega < |2^w|$*,* i.e.*, that the cardinality of the set of natural numbers is strictly less than the cardinality of the powerset of natural numbers.*

Exercise 5.13 *Use diagonalization to show that* $|S| < |2^S|$*,* i.e.*, that the cardinality of any set is strictly less than the cardinality of its powerset.*

This is a simple example of diagonalization. We will see a more complex use of diagonalization shortly.

5.4.3 Basic Properties of Programs

We start with 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 (a list) indexed by the natural numbers in such a way that every program appears exactly once in the sequence. In fact, there is a computable function f that given a natural number, i, returns the i^{th} program. Let us say that we want to enumerate C programs. We can define f as follows: on input i, consider the binary representation of C programs and iterate over all bit vectors of length $1, 2, 3, \ldots$ in order; for each such bit vector, see if it compiles using GCC (or whatever compiler you want) and keep going until you find the i^{th} bitvector that compiles; that is the i^{th} program. So the cardinality of the set of programs is ω, the cardinal number corresponding to the size of the set of natural numbers.

The second observation is that we can treat all inputs and outputs as natural numbers (say by thinking of them as bit-vectors). So, a program is just a function from natural numbers to natural numbers. Since programs may *diverge* (fail to terminate) on some inputs, a program is really a *partial* function. More generally, if we want to allow non-deterministic behavior then a program is a relation over the natural numbers; while we get similar results in this case, for simplicity's sake, we will only consider deterministic programs.

With only these observations and some basic counting results, we can already prove that there are undecidable problems. Let us restrict ourselves to decision problems: a decision problem is given as input a natural number and returns 1 or 0. An example of a decision problem is the halting problem, where the input encodes a program and 1 means the program is terminating and 0 means there is at least one input to the program that leads to nontermination. How many such functions are there? There are 2^ω such functions where ω is the size of the natural numbers. Cantor showed that $2^\omega > \omega$. In fact $2^\omega \gg \omega$ *e.g.*, $2^\omega - \omega = 2^\omega$. Therefore, most decision problems are undecidable because the number of programs is ω, as is the number of decision procedures, so there are way less decision procedures than there are decision problems.

5.4.4 Proof of the Undecidability of the Halting Problem

Our proof will be based on diagonalization. First, we start by assuming the negation of 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 imagine an infinite table where rows and columns are indexed by natural numbers and cell r, c contains $F_r(c)$ where F_r is $F(r)$ and F is some function that returns a program (not necessarily f).

	0	1	2	...
0	$F_0(0)$	$F_0(1)$	$F_0(2)$	...
1	$F_1(0)$	$F_1(1)$	$F_1(2)$	...
2	$F_2(0)$	$F_2(1)$	$F_2(2)$	...
...	...			...

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, $F_r = f(g(r))$.

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 (so it must 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.

$$d(n) = F_n(n) + 1$$

That is, to determine $d(n)$, compute $f(g(n))$, which gives us the n^{th} 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 $F_k(k)$ (since $F_k = d$), but according to the definition of d, it is $F_k(k) + 1$, a contradiction.

	0	1	2	...			
0	$F_0(0)$	$F_0(1)$	$F_0(2)$	...	$d(0)$	$\neq$	$F_0(0)$
1	$F_1(0)$	$F_1(1)$	$F_1(2)$	...	$d(1)$	$\neq$	$F_1(1)$
2	$F_2(0)$	$F_2(1)$	$F_2(2)$	...	$d(2)$	$\neq$	$F_2(2)$
...	...			...			
...	...			...	$d(n)$	$\neq$	$F_n(n)$
...	...			...			

Wait, we derived *false*. And, we used $F_k(k) = F_k(k)+1$, which is exactly the function we showed leads to inconsistency in ACL2s when motivating the need for termination analysis! So, is *false* 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.

Exercise 5.14 *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 undecidability proof 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. That is not to say that it is hard to come up with simple functions whose termination status is unknown. Consider the following well-known "Collatz" function. Even after extensive effort, no one has been able to determine if it terminates or not.

```
(definec collatz (n :pos) :pos
  (cond ((= n 1) 1)
        ((evenp n) (collatz (/ n 2)))
        (t (collatz (1+ (* 3 n))))))
```

Exercise 5.15 *Define a function* `collatz-len` *that given as intput* `n`*, a* `pos`*, returns a* `clist`*, a list whose first element is the number of calls to* `collatz` *that input* `n` *gives rise to and whose second element is a list containing the arguments to* `collatz` *that input* `n` *gives rise to. Use the following definitions.*

```
(defdata lpos (listof pos))
(defdata clist (list pos lpos))
```

Here are some tests.

```
(check= (collatz-len 1) '(1 (1)))
(check= (collatz-len 8) '(4 (8 4 2 1)))
(check= (collatz-len 13) '(10 (13 40 20 10 5 16 8 4 2 1)))
```

Exercise 5.16 *Define a function that given a positive integer* `n` *returns a positive integer, say* `k`*, such that* `(car (collatz-len k))` $=$ `n`*. See Exercise 5.15.*

Exercise 5.17 *Define a function that given a positive integer* `n` *returns the smallest positive integer, say* `k`*, such that* `(car (collatz-len k))` $=$ `n`*. See Exercise 5.15.*

Exercise 5.18 *Define a function to determine what value of* `n` *less than or equal to* 2^{32} *that returns the largest number in the first element of* `(collatz-len n)`*. See Exercise 5.15.*

5.4.5 Consequences of Undecidability in Philosophy and Science

The undecidability result is a major result that has consequences beyond computer science. Humans have wondered for millennia what the limits of human cognition are. Are there problems that are too hard for humans to solve? How do human cognition and problem-solving capabilities compare with other animals, sentient beings or machines? These are some of the most important philosophical questions that humanity has grappled with. What Turing's undecidability result shows us is that there are decision problems that cannot be solved using a certain class of machines (computers running programs).

Are there more powerful entities in the universe (or even outside of the universe) that can solve such decision problems? What about humans? Aren't humans really just machines (whose code is their DNA) that are subject to the same rules of the universe that any other objects, beings and machines are subject to? If so, then not only do humans not have any super-Turing capabilities, but we can build machines that can simulate humans and can therefore do anything humans can, at least in principle.

If humans or other entities can solve undecidable decision problems, then they have supernatural powers and are more powerful than anything bound by the rules of our universe, assuming the universe is not radically different than our current understanding of it.

Even if there are powerful entities in or outside of the universe that can solve undecidable decision problems, they cannot produce decision procedures that solve these undecidable problems, because we just showed an example of a decision problem that has no decision procedure.

If you want to say something intelligent about the limits and power of human cognition, animal cognition, AI, any kind of machine operating in this universe, etc, in the context of philosophy, psychology, medicine, biology, physics, etc., you have to understand undecidability results.

5.5 Generalizing Measure Functions

The notion of measure functions presented in the previous sections is sufficiently powerful to prove termination of almost all the functions a typical undergraduate computer science student might be asked to write.

Are there functions that are known to be terminating that we cannot prove terminating using measure functions? Try to come up with examples before reading further.

Here is an interesting example.

```
(definec f (x :tl y :tl acc :tl) :tl
  (cond ((and (endp x) (endp y)) acc)
        ((endp x) (f x (rest y) (cons (first y) acc)))
        ((endp y) (f (rest x) y (cons (first x) acc)))
        (t (f x nil (f nil y acc)))))
```

This function is terminating because `(+ (len x) (len y))` is decreasing. Let us try to define a measure function and prove that it works.

```
(definec m (x :tl y :tl acc :tl) :nat
  (+ (len x) (len y)))
```

Consider the proof obligation for the last recursive call of `f`, after some simplification.

```
(implies (and (tlp x)
              (tlp y)
              (tlp acc)
              (consp x)
              (consp y))
         (< (m x nil (f nil y acc))
            (m x y acc)))
```

It would be nice to expand `m` since we then wind up with the following proof obligation, which is easy to discharge.

```
(implies (and (tlp x)
              (tlp y)
              (tlp acc)
              (consp x)
              (consp y))
         (< (len x)
            (+ (len x) (len y))))
```

Unfortunately, in order to expand `m`, we need to know that the arguments to `m` satisfy `m`'s input contract. But, how do we know that `(f nil y acc)` is a true-list? We have not admitted `f` yet, so we do not have its contract theorem available to us. We're stuck! How can we get around this problem? Well, we can relax the requirement that measure functions have to be defined over the same parameters as the function they are used to prove terminating. If we did that, we can define `m` as follows.

```
(definec m (x :tl y :tl) :nat
  (+ (len x) (len y)))
```

This leads to the following proof obligation for the last recursive call of `f`, after some simplification.

```
(implies (and (tlp x)
              (tlp y)
              (tlp acc)
              (consp x)
              (consp y))
         (< (m x nil)
            (m x y)))
```

Now we can expand `m` to get the proof obligation we wanted.

```
(implies (and (tlp x)
              (tlp y)
              (tlp acc)
              (consp x)
              (consp y))
         (< (len x)
            (+ (len x) (len y))))
```

Next, consider the following definition.

```
(definec f (x :tl y :tl acc :tl) :tl
  (cond ((and (endp x) (endp y)) acc)
        ((endp x) (f x (rest y) (cons (first y) acc)))
        ((endp y) (f y x acc))
        (t (f x nil (f acc nil y)))))
```

How do we prove that this is terminating? Try to come up with a measure function.

Here are some observations. If `(endp x)` and `(consp y)`, the length of `y` decreases by one on every subsequent recursive call until the function terminates; let's call this the first case. If `(endp y)` and `(consp x)` (the second case), the arguments are swapped and after one step we are back in the first case. If `(consp x)` and `(consp y)` then there are two recursive calls. After one step, the inner call gets us to the second case but with the arguments shuffled. After one step the outer call gets us to the second case, but with `y` and `acc` pushed into the third argument. With these considerations in mind, try to come up with a measure function.

Here is a potential measure function.

```
(definec m (x :tl y :tl acc :tl) :nat
  (cond ((and (endp x) (endp y))
         ;; no recursive calls, so 0
         0)
        ((endp x)
         ;; we will keep coming back to this case on recursive calls
         (len y))
        ((endp y)
         ;; after one call, we get to the 1st case, with swapped args, so
         (1+ (len x)))
        (t
         ;; for the inner call an upper bound is:
         ;; 1+m(recursive-call) =  (+ 2 (len acc))
         ;; for the outer call an upper bound is: (+ 2 (len x))
         ;; so an upper bound for both cases is:
         (+ 2 (len acc) (len x)))))
```

Notice that we needed all three parameters. Consider the proof obligation for the outer recursive call of last `cond` case. After some simplification we have the following.

```
(implies (and (tlp x)
              (tlp y)
              (tlp acc)
              (consp x)
              (consp y))
         (< (m x nil (f acc nil y))
            (m x y acc)))
```

The problem is that we cannot expand `(m x nil (f acc nil y))` because we do not know that `(tlp (f acc nil y))` holds since we have not admitted `f`. The solution is to change the type of `acc` in `m` to be `all`. This is reasonable because we only increased the number of inputs `m` accepts, so `m` can still be used to assign a measure to any legal inputs to `f`. Notice a subtle point here: during the termination proof of `f`, we allow terms that mention `f` as a function symbol, just like we allowed this in the body of `f` (see the definitional principle). With this change to `m`, we can now expand out the above formula and can prove termination.

Summarizing, our generalized measure function is defined as follows.

Generalized Measure Function Definition: `m` is a measure function for `f` if all of the following hold.

1. `m` is an admissible function defined over a subset of the parameters of `f`;
2. `f`'s input contract implies `m`'s input contract;
3. `m` has an output contract stating that it always returns a natural number; and
4. on every recursive call of `f`, `m` applied to the arguments to that recursive call decreases with respect to `<`, under the conditions that led to the recursive call; this proof obligation can mention `f` as a function symbol, but we have no axioms constraining `f`.

Can we generalize this further? What do you think?

The answer is yes. One very useful generalization is to allow measure functions to return more than natural numbers. Can we allow integers? No. Consider the following non-terminating function.

```
(definec f (x :nat) :nat
  (if (= x 100)
      x
    (f (1+ x))))
```

If we could use integers then here is a "measure" function.

```
(definec m (x nat) :int
  (- 100 x))
```

Notice that the following is a theorem.

```
(implies (and (natp x) (not (= x 100)))
         (< (- 100 (1+ x)) (- 100 x)))
```

What if measure functions could return non-negative rational numbers?

```
(defdata nn-rat (range rational (0 <= _ )))
```

Then consider the following terminating function.

```
(definec f (x :rational) :rational
  (if (>= x 100)
      x
    (f (+ x 1/100))))
```

If we allow measure functions to return non-negative rational numbers we have the following "measure" function.

```
(definec m (x :rational) :nn-rat
  (if (>= x 100)
      0
    (- 100 x)))
```

Notice that the following is a theorem.

```
(implies (and (rationalp x) (not (>= x 100)))
         (< (- 100 (+ x 1/100)) (- 100 x)))
```

Unfortunately, using rational numbers is unsound, for reasons that Zeno of Elea understood thousands of years ago!

Consider the following non-terminating function.

```
(definec f (x :rational) :rational
  (if (<= x 0)
      x
    (f (/ x 2))))
```

We have the following "measure" function.

```
(definec m (x :rational) :nn-rat
  x)
```

Notice that the following is a theorem.

```
(implies (and (rationalp x) (not (<= x 0)))
         (< (/ x 2) x))
```

The problem with integers and non-negative rationals is that there are infinite decreasing sequences over these domains, whereas this is not the case with the natural numbers.

Can we use measure functions to prove that functions over the rationals are terminating? Yes; see the following exercise.

Exercise 5.19 *Prove that the following function is terminating using measure functions.*

```
(definec f (x :rational) :rational
  (if (>= x 100)
      x
    (f (+ x 1/100))))
```

Can we generalize and use other domains with ordering relations which do not admit infinite length decreasing sequences? This is a valid way to generalize measure functions. We present a further generalization of measure functions, where the domain includes lists of natural numbers and the ordering relation is the *lexicographic ordering*. In order to support all of the measure functions we already defined, the domain includes natural numbers as well as non-empty lists of natural numbers. See the data definition for `lex` below. The lexicographic ordering is provided by the function `l<` below and is defined using `d<`, a function relating lists of natural numbers.

The datatype `lex` and the functions `d<` and `l<` are already defined in ACL2s. What is shown below are equivalent definitions.

```
(defdata lon (listof nat))
(defdata nlon (cons nat lon))
(defdata lex (oneof nat nlon))

(definec d< (x :lon y :lon) :bool
  (and (consp x)
       (consp y)
       (or (< (car x) (car y))
           (and (= (car x) (car y))
                (d< (cdr x)
                  (cdr y))))))

(definec l< (x :lex y :lex) :bool
  (or (< (len x) (len y))
      (and (= (len x) (len y))
           (if (atom x)
               (< x y)
             (d< x y)))))
```

Further Generalized Measure Function Definition: `m` is a measure function for `f` if all of the following hold.

1. `m` is an admissible function defined over a subset of the parameters of `f`;
2. `f`'s input contract implies `m`'s input contract;
3. `m` has an output contract stating that it always returns a `lex` ; and
4. on every recursive call of `f`, `m` applied to the arguments to that recursive call decreases with respect to `l<`, under the conditions that led to the recursive call; this proof obligation can mention `f` as a function symbol, but we have no axioms constraining `f`.

The next few exercises show that there are no infinite decreasing sequences using our new domain and relation (in contrast to the integer and non-negative rational case). They also show that our new ordering is significantly different from the previous ordering, which was based on the natural numbers.

Exercise 5.20 *Show that there are no infinite decreasing* `lex` *sequences, with respect to* `l<` *using standard mathematical reasoning.*

Exercise 5.21 *Show that there is a bijection from* `lex` *to* `nat` *using mathematical reasoning.*

Exercise 5.22 *Let* $P_{\texttt{nat}}$ *be a function with domain* `nat` *defined as follows:* $P_{\texttt{nat}}(x) = \{y : (\texttt{nat}\ y) \wedge (\texttt{<}\ y\ x)\}$*, i.e.,* $P_{\texttt{nat}}(x)$ *is the set of* predecessors *of* x*, under* `<`*. Show that there are no elements in* `nat` *that have an infinite number of predecessors, i.e., show that* $\{x : (\texttt{natp}\ x) \wedge |P_{\texttt{nat}}(x)| = \omega\} = \emptyset$*.*

Exercise 5.23 *Let* $P_{\texttt{lex}}$ *be a function with domain* `lex` *defined as follows:* $P_{\texttt{lex}}(x) = \{y : (\texttt{lexp}\ y) \wedge (\texttt{l<}\ y\ x)\}$*, i.e.,* $P_{\texttt{lex}}(x)$ *is the set of* predecessors *of* x*, under* `l<`*. Show that there are an infinite number of elements in* `lex` *that have an infinite number of predecessors, i.e., show that* $|\{x : (\texttt{lexp}\ x) \wedge |P_{\texttt{lex}}(x)| = \omega\}| = \omega$*.*

Can we generalize measure functions further? Yes. We can find other domains with ordering relations which do not admit infinite length decreasing sequences and which are more powerful than lexicographic orders on natural numbers. Pushing this idea to the current limits of human comprehension leads to measure functions that return infinite ordinal numbers. Ordinal numbers are at the heart of set theory, which, along with first-order logic, forms the foundations of modern mathematics.

Here is an informal introduction to ordinal numbers.[1] Ordinal numbers extend the natural numbers and, as is the case with the natural numbers, are totally ordered, *i.e.*, if α

[1] An observation that may help to avoid some confusion is that ordinal numbers differ from the cardinal numbers we have been using to measure the size of sets.

and β are two different ordinal numbers then either $\alpha < \beta$ or $\beta < \alpha$. So, an intuitive way to think about ordinal numbers is to list them in order. They start with the natural numbers:

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

But, why stop there? Let's add another element, the first infinite ordinal number, ω, to get:

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

Notice that the natural numbers have the nice property that every decreasing sequence of natural numbers is terminating. That is still true for the ordinal numbers we have seen so far. While it is true that the number of ordinal numbers less than ω is infinite, there are no infinitely decreasing sequences starting at ω. Why? Because if we start with ω, the next element of a decreasing sequence has to be a natural number. It turns out that a basic property of ordinal numbers is that they are terminating!

Why stop with ω? Let's keep extending the ordinals:

$$0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots$$

At the limit of the above, we get $\omega + \omega$ which happens to be $\omega \cdot 2$ and then we can keep going:

$$0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots, \omega + \omega = \omega \cdot 2, \omega \cdot 2 + 1, \ldots$$

Let's keep going, but at a faster rate.

$$0, 1, 2, \ldots, \omega, \omega + 1, \ldots, \omega \cdot 2, \omega \cdot 2 + 1, \ldots, \omega \cdot \omega = \omega^2, \omega^2 + 1, \ldots, \omega^3, \ldots, \omega^\omega$$

So, now we have infinite numbers raised to infinite powers. Can we keep going? Sure. Let's go at an even faster rate.

$$0, 1, 2, \ldots, \omega, \omega + 1, \ldots, \omega \cdot 2, \ldots, \omega^2, \ldots, \omega^\omega, \ldots, \omega^{\omega^\omega}, \ldots, \omega^{\omega^{\omega^{\cdots}}}$$

The last number in the above sequence is ε_0 and it is ω raised to ω an infinite number of times. Notice that it satisfies the seemingly contradictory equality $\varepsilon_0 = \omega^{\varepsilon_0}$. Ordinal arithmetic is strange! Wait, does it make sense to have an infinite stack of infinities? After all, raising the infinite number ω to any other exponent shown above (such as $2, \omega$ and ω^ω) gave us a number larger than the exponent, but how can it be that if we raise ω to ε_0, which is much larger than any of $2, \omega, \omega^\omega$, we just get the exponent, ε_0?

One of the goals of the foundations of mathematics and set theory is to figure out how far we extend the notion of an ordinal. This is relevant to us, because the notion of termination is fundamentally tied to ordinals, *i.e.*, every notion of termination ultimately can be reduced to a notion involving the ordinals (usually a very small subset of the ordinals). Another way of saying this is that the ordinals include all possible termination processes and every time we extend the notion of an ordinal, our ability to prove termination increases. Your "contradiction radar" should be going off right about now. Maybe ordinals that satisfy weird properties like the one that ε_0 satisfies lead to inconsistency. It is true that assuming the existence of ordinals that satisfy some "weird" properties leads to unsoundness, so if we extend the ordinals too far, we break mathematics (yes, that would be bad). However, we are nowhere near that point with ε_0, which is a baby ordinal.

ACL2s allows the use of ordinal numbers, but that's a topic for a more advanced class on logic and computer-aided reasoning. Some of the challenging exercises at the end of the chapter touch upon these issues. For example, consider exercises 5.50, 5.51 and 5.52. To learn more, take more advances courses in formal methods.

5.6 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 5.24

```
(definec f (x :tl y :nat) :tl
  (cond ((zp y) nil)
        ((endp x) (list y))
        (t (f (cons y x) (1- y)))))
```

Exercise 5.25 *Dead code example*

```
(definec f (x :nat y :nat) :int
  (cond ((zp x) 1)
        ((< x 0) (f -1 -1))
        (t (1+ (f (1- x) y)))))
```

Notice that the second case of the `cond` above is not reachable and will never be executed.

Exercise 5.26

```
(definec f (x :int y :nat) :int
  (cond ((zip x) 1)
        ((< x 0) (f (1+ y) (* x x)))
        (t (1+ (f (1- x) y)))))
```

Exercise 5.27

```
(definec f (x :tl y :int) :nat
  (cond ((endp x) y)
        (t (1+ (f (rest x) y)))))
```

Exercise 5.28

```
(definec f (x :tl y :int) :nat
  (cond ((endp x) y)
        (t (f (rest x) (1+ y)))))
```

Exercise 5.29

```
(definec f (x :tl y :int) :tl
  (cond ((zip y) x)
        (t (f (rest x) (1- y)))))
```

Exercise 5.30

```
(definec f (x :nat) :int
  (cond ((zp x) 1)
        ((< x 0) (f -1))
        (t (1+ (f (1- y))))))
```

Exercise 5.31

```
(definec f (x :tl y :int) :nat
  (cond ((and (endp x) (zip y))
         0)
        ((and (endp x) (< y 0))
         (1+ (f x (1+ y))))
        ((endp x)
         (1+ (f x (1- y))))
        (t
         (1+ (f (rest x) y)))))
```

Exercise 5.32

```
(definec f (x :rational) :rational
  (if (< x 0)
      (f (+ x 1/2))
    x))
```

Exercise 5.33

```
(definec f (x :rational) :nat
  (cond ((> x 0)   (f (- x 3/2)))
        ((< x 0)   (f (* x -1)))
        (t x)))
```

Exercise 5.34

```
(definec f (x :rational) :nat
  (cond ((> x 0)   (f (- x 3/2)))
        ((< x 0)   (f 180))
        (t x)))
```

Exercise 5.35

```
(definec f (x :tl y :rational) :nat
  (cond ((> y 0)   (f x (1- y)))
        ((consp x) (f (rest x) (1+ y)))
        ((< y 0)   (f (list y) (* y -1)))
        (t y)))
```

Exercise 5.36

```
(definec f (x :tl y :nat) :nat
  (if (endp x)
      (if (zp y)
          0
        (1+ (f x (1- y))))
    (1+ (f (rest x) y))))
```

Exercise 5.37

```
(definec fib (n :nat) :nat
  (if (< n 2)
      n
    (+ (fib (1- n))
       (fib (- n 2)))))
```

Exercise 5.38

```
(definec f (x :int y :int) :nat
  (cond ((< x y) (1+ (f (1+ x) y)))
        ((> x y) (1+ (f x (1+ y))))
        (t 0)))
```

Exercise 5.39

```
(definec f (n :nat) :nat
  (cond ((<= n 1) n)
        ((even n) (f (/ n 2)))
        (t (f (1+ n)))))
```

Exercise 5.40

```
(definec f (n :nat) :nat
  (cond ((<= n 1) n)
        ((evenp n) (f (/ n 2)))
        (t (f (1+ (* 2 n))))))
```

Exercise 5.41

```
(definec e3 (x :nat y :pos) :nat
  (cond ((zp x) x)
        ((= y 1) y)
        ((> x y) (e3 y x))
        (t (e3 x (1- y)))))
```

Exercise 5.42

```
(definec foo (x :nat l :tl a :all) :all
  (cond ((endp l) a)
        ((zp x) 1)
        ((oddp x) (foo (1- x) l a))
        ((> x (len l)) (foo (/ x 2) l x))
        (t (foo x (rest l) (first l)))))
```

Exercise 5.43

```
(definec app-acc (x :tl y :tl acc :tl) :tl
  (cond ((and (endp x) (endp y)) acc)
        ((endp x) (app-acc x (rest y) (cons (first y) acc)))
        ((endp y) (app-acc (rest x) y (cons (first x) acc)))
        (t (app-acc x nil (app-acc nil y acc)))))
```

Exercise 5.44

```
(definec app-swap (x :tl y :tl acc :tl) :tl
  (cond ((and (endp x) (endp y)) acc)
        ((endp x) (app-swap x (rest y) (cons (first y) acc)))
        ((endp y) (app-swap y x acc))
        (t (app-swap x nil (app-swap acc nil y)))))
```

Exercise 5.45

```
(definec f (x :rational) :rational
  (cond ((<= x 0) x)
        ((>= x 2) (f (/ x 2)))
        ((>= x 1) (f (- x 1/100)))
        (t (f (/ 1 x)))))
```

Exercise 5.46

```
(definec f (x :rational) :rational
  (cond ((<= x 0) x)
        ((>= x 2) (f (/ x 2)))
        ((>= x 1) (f (- x 1/100)))
        (t (f (- x)))))
```

Exercise 5.47

```
(definec f (n :all) :all
  (cond ((or (not (integerp n))
             (<= n 1)) 0)
        ((integerp (+ 1/2 (/ n 2))) (f (1+ n)))
        (t (1+ (f (/ n 2))))))
```

Exercise 5.48 *This is hard.*

```
(defdata if-expr (oneof symbol (list 'if if-expr if-expr if-expr)))
(definec if-flat (x :if-expr) :if-expr
  (if (symbolp x)
      x
```

```
    (let ((test (second x))
          (true-branch (third x))
          (false-branch (fourth x)))
      (if (symbolp test)
          (list 'if test (if-flat true-branch) (if-flat false-branch))
        (if-flat (list 'if (second test)
                       (list 'if (third test) true-branch false-branch)
                       (list 'if (fourth test) true-branch false-branch)))))))
```

Exercise 5.49 *This is hard.*

```
(definec f
  (flg :int w :int r :non-0-integer z :int s :int x :int y :int
   a :int  b :int zs :int) :bool
  (cond ((= flg 1)
         (if (> z 0)
             (f 2 w r z 0 r w 0 0 zs)
           (= w (expt r zs))))
        ((= flg 2)
         (if (> x 0)
             (f 3 w r z s x y y s zs)
           (f 1 s r (1- z) 0 0 0 0 0 zs)))
        (t (if (> a 0)
               (f 3 w r z s x y (1- a) (1+ b) zs)
             (f 2 w r z b (1- x) y 0 0 zs)))))
```

Exercise 5.50 *This brings up interesting questions; consider using one of the generalized notions of measure functions.*

```
(definec ack (n :nat m :nat) :pos
  (cond ((zp n) (1+ m))
        ((zp m) (ack (1- n) 1))
        (t (ack (1- n) (ack n (1- m))))))
```

Exercise 5.51 *This brings up interesting questions and is hard.*

```
(definec mc (x :int) :nat
  (if (< 100 x)
      (- x 10)
    (mc (mc (+ x 11)))))
```

Exercise 5.52 *This brings up interesting questions and is hard.*

```
(definec tk (a :int b :int c :int) :int
  (if (<= a b)
      b
    (tk (tk (1- a) b c)
        (tk (1- b) c a)
        (tk (1- c) a b))))
```

Exercise 5.53 *This is related to Exercise 5.52.*

```
(definec tk4 (a :int b :int c :int d :int) :int
  (if (<= a b)
      b
    (tk4 (tk4 (1- a) b c d)
         (tk4 (1- b) c d a)
         (tk4 (1- c) d a b)
         (tk4 (1- d) a b c))))
```

Exercise 5.54 *This brings up interesting questions and is hard.*

```
(definec f (x :rational) :rational
  (if (< x 0)
      (- x)
    (/ (f (- x (f (1- x)))) 2)))
```

Part V

Induction

Induction

6.1 Introduction to Induction

Terminating functions give rise to induction schemes.

Consider the following function:

```
(definec nat-ind (n :nat) :nat
  (if (zp n)
      0
    (nat-ind (1- n))))
```

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 of `(nat-ind n)`. Our proof obligations are:

1. `(not (natp n))` $\Rightarrow \phi$
2. `(natp n)` $\wedge$ `(zp n)` $\Rightarrow \phi$
3. `(natp n)` $\wedge$ `(not (zp n))` $\wedge\ \phi|_{\texttt{((n n-1))}} \ \Rightarrow\ \phi$

A bit of terminology. The first proof obligation is the *contract case*. The first two proof obligations are *base cases* (so the contract case is a base case). The third proof obligation is an *induction step* because we get to assume that ϕ holds on smaller values. The last hypothesis of the third proof obligation, $\phi|_{\texttt{((n n-1))}}$, 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))` $\Rightarrow \phi$
2. `(natp n)` $\wedge$ `(zp n)` $\Rightarrow \phi$
3. `(natp n)` $\wedge$ `(not (zp n))` $\ \Rightarrow\ \phi$

The three cases above are exhaustive. When reasoning about programs, case analysis is a very useful proof technique, which we now define. Notice that it is the natural generalization of the synonymous proof technique we defined in the context of propositional logic.

Case Analysis: If ψ is a formula and $\phi_1, \ldots, \phi_n$ are formulas such that `(or` ϕ_1 `...` ϕ_n`)` is valid, then ψ is valid iff all of $(\phi_1 \Rightarrow \psi), \ldots, (\phi_n \Rightarrow \psi)$ are valid.

The intuition is the same as before. We are proving that ψ holds by considering the cases $\phi_1, \ldots, \phi_n$ and since the cases are exhaustive ($\phi_1 \vee \cdots \vee \phi_n \equiv t$), ψ always holds.

A commonly occurring example of where case analysis is useful is when we are proving a theorem of the following form.

$$\phi_1 \vee \cdots \vee \phi_n \;\Rightarrow\; \psi$$

It is often a good idea to use case analysis to instead prove the following set of (individually) simpler theorems.

$$\phi_1 \;\Rightarrow\; \psi,\; \ldots,\; \phi_n \;\Rightarrow\; \psi$$

Induction is more powerful than case analysis because it also allows us to assume induction hypotheses. This makes all the difference in the world when reasoning about programs.

Back to induction.

Why does induction work?

Suppose that we prove the above three cases, but ϕ is not valid.

Then, the set of objects in the ACL2 universe for which ϕ does not hold, say S, is non-empty. S 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:

$$\texttt{(natp } s\texttt{)} \wedge \texttt{(not (zp } s\texttt{))} \wedge \phi|_{\texttt{((n } s-1\texttt{))}} \;\Rightarrow\; \phi|_{\texttt{((n } s\texttt{))}}$$

Notice that $(\phi|_{\texttt{((n n-1))}})|_{\texttt{((n } s\texttt{))}} = \phi|_{\texttt{((n } s-1\texttt{))}}$. But, we have that `(natp` s`)` holds and $s \neq 0$. By the minimality of s, $\phi|_{\texttt{((n } s-1\texttt{))}}$ also holds. By MP and the above, so does $\phi|_{\texttt{((n } s\texttt{))}}$. 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. This is a proof by contradiction.

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 contradiction. Why do people use proofs by contradiction? 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 $\phi|_{\texttt{((n 0))}}$ all hold), so by MP $\phi|_{\texttt{((n 1))}}$ holds. Notice that we use the induction step to go from $\phi|_{\texttt{((n 0))}}$ to $\phi|_{\texttt{((n 1))}}$. Similarly, we can use $\phi|_{\texttt{((n 1))}}$ to get $\phi|_{\texttt{((n 2))}}$ and so on for all the natural numbers. We have just shown that for any natural number i, $\phi|_{\texttt{((n } i\texttt{))}}$ holds, so ϕ holds for all objects in the ACL2s universe. This is a direct proof that proof by induction is sound.

6.2 Induction in ACL2s

We will define induction in its full glory, but before we do, a few observations. First of all, induction in ACL2s will be more complicated that mathematical induction and the kinds of induction you have already seen (probably). Why is that? One reason is that the ACL2s universe includes more than numbers, *e.g.*, it includes lists and trees, and we want the ability to reason about arbitrary functions and properties. Another reason is that induction is intricately connected to termination because termination justifies induction. We have already invested a fair amount of effort on termination analysis in ACL2s and a wonderful result is that every terminating function gives rise to an induction scheme! We just need to define what that induction scheme is and then you can use it in your proofs for free. So, the upcoming definitions of induction schemes and proofs by induction may be a little hard to understand initially, but once you do understand them, you get a very powerful theorem proving weapon, a weapon you need if you are to reason about computation. So, make sure that you understand induction schemes and proof by induction well enough that you can easily generate and explain the definitions from memory.

6.2.1 Induction Schemes

Suppose we are given a function definition of the form:

```
(defunc f (x_1 ... x_n)
  :input-contract ic
  :output-contract oc
  body)
```

Start by expanding away all macros in *body*; this will make it easier to define induction schemes. An occurrence of an expression in *body* that does not contain any `if`s in it and which is not a subexpression of the test of any `if` in *body* is a *terminal.*

A terminal is *maximal* if it is not contained in any other terminal.

For every terminal, there is a corresponding *condition* that must hold for execution to reach the terminal.

For example, suppose that `f` has only one formal, `x`, and *body* is

```
(not (if (g x) (g x) (if (not (f (1- x))) (not (f (1- x))) (f (- x 2)))))
```

then the maximal terminals are `(g x)`, `(not (f (1- x)))` and `(f (- x 2))` (there are two occurrences of `(g x)` in *body* but only one is a terminal) and the corresponding conditions are `(g x)`, `(and (not (g x)) (not (f (1- x))))` and `(and (not (g x)) (f (1- x)))))`.

The set of *recursive calls* of a terminal contains all the calls to `f` that must be executed in order to execute the terminal. If the set of recursive calls of a terminal is empty, then the terminal is *basic*; otherwise it is *recursive.*

In our previous example, the recursive calls corresponding to the maximal terminals are $\{\}$, $\{$`(f (1- x))`$\}$ (notice that we do not distinguish between the two calls of `(f (1- x))`) and $\{$`(f (1- x))`, `(f (- x 2))`$\}$ (the first terminal is basic and the other two are recursive).

Let $\langle t_1, \ldots, t_m \rangle$ be a sequence containing `f`'s maximal terminals; let the corresponding sequence of conditions be $\langle c_1, \ldots, c_m \rangle$; and let the corresponding sequence of recursive calls be $\langle r_1, \ldots, r_m \rangle$, where r_i is $\{$`(f x`$_1$ `...` `x`$_n$`)`$|_{\sigma_i^j} : 1 \le j \le |r_i|\}$ (the σ_i^j's are substitutions and r_i is $\{\}$ if t_i is basic).

The function `f` gives rise to an *induction scheme* that is parameterized by a formula ϕ. The induction scheme of `(f` $\texttt{x}_1$ `...` $\texttt{x}_n$`)` for ϕ consists of the following formulas:

1. $\neg ic \;\Rightarrow\; \phi$
2. For all t_i that are basic terminals: $ic \wedge c_i \;\Rightarrow\; \phi$
3. For all t_i that are recursive terminals: $\big(ic \;\wedge\; c_i \;\wedge\; \bigwedge_{1\le j\le |r_i|} \phi|_{\sigma_i^j}\big) \;\Rightarrow\; \phi$

Proof by Induction: Let ϕ be any formula and let `f` be an n-ary function. If all of the formulas in the induction scheme of `(f` $\texttt{x}_1$ `...` $\texttt{x}_n$`)` for ϕ are valid, then so is ϕ.

The formulas in the induction scheme of a function are called *proof obligations* because if we prove that they are valid, it follows that ϕ is valid. The first formula is the *contract case*. The second class of formulas are *base cases*. We will also characterize the first formula as a base case. The last class of formulas are *induction steps*.

Consider the proof obligations generated by our running example.

1. $\neg ic \;\Rightarrow\; \phi$
2. $ic \wedge$ `(g x)` $\Rightarrow\; \phi$
3. $ic \wedge$ `(not (g x))` $\wedge$ `(not (f (1- x)))` $\wedge\, \phi|_{\texttt{((x (1- x)))}} \;\Rightarrow\; \phi$
4. $ic \wedge$ `(not (g x))` $\wedge$ `(f (1- x))` $\wedge\, \phi|_{\texttt{((x (1- x)))}} \wedge \phi|_{\texttt{((x (- x 2)))}} \;\Rightarrow\; \phi$

We allow some minor generalizations that we did not formalize to avoid notational clutter. Firstly, we allow you to use induction schemes for functions with any distinct variables as arguments. For example, the induction scheme of `(nat-ind k)` is generated as described above, but we make believe that `nat-ind` was defined using `k` instead of `n`, so we wind up with the induction scheme show previously, after applying the substitution `((n k))` to everything but ϕ. This is useful when we have formulas with many variables or with variables that differ from those used to define the function whose induction schemes we are using. We allow you to simplify the proof obligations using propositional logic, arithmetic, basic properties of equality and "type" theorems, *e.g.*, you can simplify away terms of the form `(allp` *term*`)`. Finally, we allow you to remove *subsumed* proof obligations, proof obligations that are implied by other proof obligations.

Exercise 6.1 *What induction schemes do the following functions give rise to? Simplify as much as possible.*

```
(definec f (x :all) :all
  x)

(definec g (x :tl) :all
  x)
```

Exercise 6.2 *Show that if there is a proof of the validity of formula ϕ that only uses induction schemes of non-recursive functions, then ϕ can be proven valid without using any induction schemes at all.*

Hint: Use case analysis instead and see Exercise 6.1.

Let us consider some examples. What is the induction scheme for the following function?

```
(definec in (a :all l :tl) :bool
  (and (consp l)
       (or (== a (car l))
           (in a (cdr l)))))
```

First, let us expand out macros, which gives us the following.

```
(definec in (a :all l :tl) :bool
  (if (consp l)
      (if (equal a (car l))
          (equal a (car l))
        (in a (cdr l)))
    nil))
```

After simplifying `allp` expressions, we have the following.

1. `(not (tlp l))` $\Rightarrow\ \phi$
2. `(tlp l)` $\wedge$ `(not (consp l))` $\Rightarrow\ \phi$
3. `(tlp l)` $\wedge$ `(consp l)` $\wedge$ `(equal a (car l))` $\Rightarrow\ \phi$
4. `(tlp l)` $\wedge$ `(consp l)` $\wedge$ `(not (equal a (car l)))` $\wedge\ \phi|_{\texttt{((l (cdr l)))}}\ \Rightarrow\ \phi$

What if we defined `in` like this?

```
(definec in (a :all l :tl) :bool
  (and (consp l)
       (or (in a (cdr l))
           (== a (car l)))))
```

First, let us expand out macros, which gives us the following.

```
(definec in (a :all l :tl) :bool
  (if (consp l)
      (if (in a (cdr l))
          (in a (cdr l))
        (equal a (car l)))
    nil))
```

After simplifying `allp` expressions, we have the following.

1. `(not (tlp l))` $\Rightarrow\ \phi$
2. `(tlp l)` $\wedge$ `(not (consp l))` $\Rightarrow\ \phi$
3. `(tlp l)` $\wedge$ `(consp l)` $\wedge$ `(in a (cdr l))` $\wedge\ \phi|_{\texttt{((l (cdr l)))}}\ \Rightarrow\ \phi$
4. `(tlp l)` $\wedge$ `(consp l)` $\wedge$ `(not (in a (cdr l)))` $\wedge\ \phi|_{\texttt{((l (cdr l)))}}\ \Rightarrow\ \phi$

We saw how to go from functions to induction schemes, but we can also play this game in reverse. Consider the following exercise.

Exercise 6.3 *Provide a function that gives rise to the following induction scheme.*

1. `(not (natp n))` $\Rightarrow$ ϕ
2. `(natp n)` $\wedge$ `(zp n)` $\Rightarrow$ ϕ
3. `(natp n)` $\wedge$ `(not (zp n))` $\wedge$ $\phi|_{((n\ n-1))}$ $\Rightarrow$ ϕ

Exercise 6.4 *One correct answer to the previous exercise is* `nat-ind`*, but there are infinitely many correct answers. Show this.*

Exercise 6.5 *Prove that no function gives rise to the following "induction scheme."*

1. `(not (natp n))` $\Rightarrow$ ϕ
2. `(natp n)` $\wedge$ `(zp n)` $\Rightarrow$ ϕ
3. `(natp n)` $\wedge$ `(not (zp n))` $\wedge$ $\phi|_{((n\ n+1))}$ $\Rightarrow$ ϕ

Why do we not allow "induction schemes" like the one in the above exercise? Because they are not sound! Remember that induction works because of termination. The only "functions" that give rise to the above "induction scheme" are nonterminating functions.

Exercise 6.6 *Prove that the "induction scheme" from Exercise 6.5 is not sound,* i.e.*, use the "induction scheme" to prove* `nil`*.*

We end by noting that ACL2s generates induction schemes from function definitions in a way that is very similar to what was explained here, but there are some differences. If you use the theorem prover and ask it to perform a particular induction and you wind up with proof obligations that seem strange, check what induction scheme was generated and if it is not what you expected, then read the documentation on `induction`, `rulers` and related topics.

6.3 Induction Examples

Recall the definition of `sumn`.

```
(definec sumn (n :nat) :nat
  (if (zp n)
      0
    (+ n (sumn (1- n)))))
```

Let's prove

$$\texttt{(sumn n)} = \mathtt{n}(\mathtt{n}+1)/2$$

Recall that the first thing we do is to contract check conjectures. After fixing the above conjecture, we get:

$$\texttt{(natp n)} \Rightarrow \texttt{(sumn n)} = \mathtt{n}(\mathtt{n}+1)/2 \tag{6.1}$$

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 `nat-ind`'s induction scheme (which is the same as `sumn`'s induction scheme). You have to identify both the function and the variables when using induction schemes! We are using the induction scheme of `(nat-ind n)`.

Our proof obligations are (we used some arithmetic reasoning, *e.g.*, we replaced `(zp n)` with $\texttt{n} = 0$):

1. `(not (natp n))` $\Rightarrow$ (6.1)
2. `(natp n)` $\wedge\ \texttt{n} = 0 \Rightarrow$ (6.1)
3. `(natp n)` $\wedge\ \texttt{n} \neq 0 \wedge (6.1)|_{\texttt{((n (1- n)))}} \Rightarrow$ (6.1)

Notice that the proof now goes through with just equational reasoning.

Since we know how to do equational reasoning, we will skip the equational proofs, but you can and should fill them in.

To see one reason why we need to identify variables, suppose we were asked to prove the following equivalent conjecture:

$$\texttt{(natp x)} \Rightarrow \texttt{(sumn x)} = \texttt{x}(\texttt{x}+1)/2$$

The induction scheme to use is now `(nat-ind x)`, which is the induction scheme we would get if `nat-ind` was defined using `x` instead of `n`.

Let's now reason about the following function definition.

```
(definec app2 (a :tl b :tl) :tl
  (if (endp a)
      b
    (cons (first a) (app2 (rest a) b))))
```

We want to prove that `app2` is associative.

$$\texttt{(app2 (app2 a b) c)} = \texttt{(app2 a (app2 b c))}$$

Contract checking gives:

$$\texttt{(tlp a)} \wedge \texttt{(tlp b)} \wedge \texttt{(tlp c)} \Rightarrow \texttt{(app2 (app2 a b) c)} = \texttt{(app2 a (app2 b c))} \quad (6.2)$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `app2`. 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 `tlp`'s induction scheme. Recall the definition of `tlp`.

```
(definec tlp (l :all) :bool
  (if (consp l)
      (tlp (cdr l))
    (equal l ())))
```

This brings up another reason why we need to identify variables in induction schemes. In (6.2), we have multiple variables that are true-lists, so which one are we using to generate an induction scheme? Let's say we use `a`, then the induction scheme of `(tlp a)` is:

1. $\neg$`(allp a)` $\Rightarrow$ (6.2)
2. `(allp a)` $\wedge$ `(consp a)` $\wedge$ (6.2)$|_{\texttt{((a (rest a)))}}$ $\Rightarrow$ (6.2)
3. `(allp a)` $\wedge$ $\neg$`(consp a)` $\Rightarrow$ (6.2)

This is equivalent to:

1. $\neg$`(consp a)` $\Rightarrow$ (6.2)
2. `(consp a)` $\wedge$ (6.2)$|_{\texttt{((a (rest a)))}}$ $\Rightarrow$ (6.2)

So, here we go. If you expand out the proof obligations, we have a problem we've seen before! Do the induction step.

Exercise 6.7 *We saw that the variables we use in proofs are irrelevant,* e.g., *even though* `tlp` *was defined over* `l`*, we can apply induction using* `a` *instead. Explain why this is the case.*

Exercise 6.8 *Assume that the output type for* `app2` *was defined to be* `all`*. This version of* `app2` *is admissible. Using this version of* `app2`*, use induction to prove* `(tlp (app2 x y))`*. (You have to perform contract completion first.)*

Exercise 6.9 *Prove the following conjecture.*

```
(implies (and (tlp x)
              (tlp y))
         (= (llen (app2 x y))
            (+ (llen x) (llen y))))
```

Exercise 6.10 *Prove the output contracts of all the functions we have defined. Some will require induction, but some can be proved using just equational reasoning.*

Exercise 6.11 *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 `rev2`. Here is the definition.

```
(definec rev2 (x :tl) :tl
  (if (endp x)
      nil
    (app2 (rev2 (rest x)) (list (first x)))))
```

Now we want to prove:

$$\texttt{(rev2 (rev2 x))} = \texttt{x}$$

Contract checking gives:

$$\texttt{(tlp x)} \Rightarrow \texttt{(rev2 (rev2 x))} = \texttt{x} \tag{6.3}$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `rev2`. 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 `tlp`'s induction scheme, which is:

1. $\neg$`(consp x)` $\Rightarrow$ (6.3)
2. `(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:

$$\texttt{(tlp x)} \wedge \texttt{(tlp y)} \Rightarrow \texttt{(rev2 (app2 x y))} = \texttt{(app2 (rev2 y) (rev2 x))} \qquad (6.4)$$

we could finish the proof.

Let's assume we have it and then finish the proof.

Notice that sometimes in the process of proving a theorem by induction, it is useful to prove lemmas, helper theorems that allow us to prove the theorem under consideration. This is similar to what happens when you try to define a function and realize that a helper function would be useful. Most of the techniques we will present for reasoning about programs have direct analogues to techniques for defining programs. Use these connections and your intuitions about programming to help help you internalize and more fully understand how to reason about programs.

Exercise 6.12 *Prove* (6.4)*. Use the induction scheme of* `(tlp x)`*.*

In the proof of (6.4), we needed to prove

$$\texttt{(tlp x)} \Rightarrow \texttt{(app2 x nil)} = \texttt{x}$$

So, even the proof of the lemma requires a lemma. How far can this go? Far! It's recursive. That should not be surprising, after all, suppose you were writing a complex program, say a compiler. Doing so will require helper functions, which themselves require helper functions. How far can this go? Far! It's recursive. It depends on the complexity of the compiler. Similarly, reasoning about programs requires lemmas, which require lemmas, and so on. Reasoning about programs is typically harder than writing them. For example, writing `collatz` was easy, but proving that it terminates is an open problem. Writing the following program is easy.

```
(definec f (x :int y :int z :int) :bool
  (/= (+ (expt x 3) (expt y 3) (expt z 3)) 33))
```

Determining if the following conjecture is a theorem is hard.

```
(implies (and (intp x) (intp y) (intp z))
         (f x y z))
```

Exercise 6.13 *What induction scheme does this give rise to?*

```
(definec fib (n :pos) :pos
  (if (<= n 2)
      n
    (+ (fib (1- n)) (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`, so why should the data definition give rise to the induction scheme we need?

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 schemes, then the induction schemes will also be based on the same recursion scheme.

Exercise 6.14 *Prove the previous conjecture using the induction scheme of* `fib`.

6.4 Induction Schemes for Defdata

What induction scheme do we get for data definitions? The basic idea is that we use the definition of the recognizer that `defdata` generates to determine the induction scheme.

Here is an example. Consider the following data definition.[1]

```
(defdata lor (listof rational) :do-not-alias t)
```

This form generates a recognizer, `lorp` and you can see its definition using the following command.

```
:pe lorp
```

Once you have the definition you can generate the induction scheme. The function is defined as follows (more or less).

```
(definec lorp (l :all) :bool
  (or (== l nil)
      (and (consp l)
           (rationalp (car l))
           (lorp (cdr l)))))
```

[1]The `:do-not-alias t` part forces ACL2s to generate a new definition; without it, ACL2s realizes that there is an existing data definition that is equivalent to this one.

Exercise 6.15 *What induction scheme does the above definition of* `lorp` *give rise to?*

6.5 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.*, `tlp` 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 `app2`, we get to assume that `app2` works correctly in the recursive case, even if its first input has 1,000,000 elements, *i.e.*, we get to assume that `app2` 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).

6.6 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 6.16 *Consider the following non-terminating function definition.*

```
(definec f (x :all) :all
  (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.*

6.7 Induction Like a Professional

In Exercise 6.12, we asked you to prove Conjecture 6.4.

$$\texttt{(tlp x)} \wedge \texttt{(tlp y)} \Rightarrow \texttt{(rev2 (app2 x y))} = \texttt{(app2 (rev2 y) (rev2 x))}$$

We told you what induction scheme to use (the one that `(tlp x)` gives rise to).

Typically, you will have to figure out what induction scheme to use. How do you go about doing that?

Look at the left-hand-side and of the equality in the conclusion. What variables control the recursive functions? On the LHS, `x`, due to `(app2 x y)` and on the right hand side it is both `x` and `y`, so `x` is a better choice.

Here is how I think about it. I can assume that the theorem I am trying to prove holds for "smaller" values of the arguments, so let me just not worry about exactly what smaller values to use and after I do the proof, find the induction scheme that I needed! Also, I will just sketch out the proof, without worrying about full justifications, which I will add later.

This is how professionals think about it and it is a more powerful way of thinking about induction.

So, here we go.

```
(rev2 (app2 x y))
= { Def app2 }
(rev2 (cons (first x) (app2 (rest x) y)))
= { Def rev2 }
(app2 (rev2 (app2 (rest x) y)) (list (first x)))
= { Using an instance of the theorem ((x (rest x))) }
(app2 (app2 (rev2 y) (rev2 (rest x))) (list (first x)))
= { Lemma assoc-append }
(app2 (rev2 y) (app2 (rev2 (rest x)) (list (first x))))
= { Def rev2 }
(app2 (rev2 y) (rev2 x))
```

Bingo! Induct on `(tlp x)` because we need an induction scheme that allows us to assume that the theorem holds when we replace `x` with `(rest x)`.

We still have to do the base case. That should be easy, right. If so, we often just say it is "obvious" using equational reasoning. But don't do that unless you are sure.

So, let us try us sketch out the base case.

```
C1. (tlp x)
C2. (tlp y)
C3. (not (consp x))
D1. x=nil

(rev2 (app2 x y))
= { Def app2, D1 }
(rev2 y)
= { ??? }
(app2 (rev2 y) nil)
= { Def rev2, D1 }
(app2 (rev2 y) (rev2 x))
```

But how do I justify the `???` step?

I can't just say `Def app2` because `app2` recurs down its first argument, so we have to prove

```
(app2 x nil) = x
```

Exercise 6.17 *Try proving the following conjecture using induction.*

```
    (implies (tlp x) (== (app2 x nil) x))
```

A. Try proving it with the induction scheme of `app2`*.*
B. Try proving it like a professional.
C. Try proving it with the induction scheme of `tlp`*.*

We now consider a more in-depth example. We will define insertion sort and prove that it returns an ordered permutation of its input.

Here is the definition of insertion sort.

```
(defdata lor (listof rational))

(definec insert (e :rational L :lor) :lor
  (cond ((endp L) (list e))
        ((<= e (car L)) (cons e L))
        (t (cons (car L) (insert e (cdr L))))))

(definec isort (L :lor) :lor
  (if (endp L)
      L
    (insert (car L) (isort (cdr L)))))
```

We will first prove that `isort` returns an ordered list, so let us define what it means for a list to be ordered.

```
(definec orderedp (L :lor) :bool
  (or (endp (cdr L))
      (and (<= (car L) (second L))
           (orderedp (cdr L)))))
```

Our goal is to prove the following theorem.

$$\varphi_0 : \texttt{(lorp L)} \Rightarrow \texttt{(orderedp (isort L))}$$

We will use the professional method. To remember where I make decisions relevant to the induction scheme I want to use, I will tag relevant hints with **.

```
(orderedp (isort L))
= { Def isort, (consp L) } **
(orderedp (insert (car L) (isort (cdr L))))
```

Now, I can assume `(orderedp (isort (cdr L)))`, but how do I deal with the `insert`? I need a lemma.
What lemma? Think about it before reading ahead.

$$\varphi_1 : \texttt{(rationalp e)} \wedge \texttt{(lorp L)} \wedge \texttt{(orderedp L)} \Rightarrow \texttt{(orderedp (insert e L))}$$

Let us assume that this lemma holds and continue with the proof.

= { $\varphi_0|_{\texttt{((L (cdr L)))}}, \varphi_1$ } **
t

So, what induction scheme works? The simplest is `(tlp L)` because we need a recursive case where the condition is `consp` and the function calls itself recursively on `(cdr L)`.

This is interesting because if we used the type of `L`, we would induct on `(lorp L)` and then we would have to prove the following five proof obligations.

0. $\neg$`(allp L)` $\Rightarrow \varphi_1$ `(trivial)`
1. `L` $=$ `nil` $\Rightarrow \varphi_1$
2. `L` $\neq$ `nil` $\wedge$ $\neg$`(consp l)` $\Rightarrow \varphi_1$
3. `L` $\neq$ `nil` $\wedge$ `(consp l)` $\wedge\neg$ `(rationalp (car l))` $\Rightarrow \varphi_1$
4. `L` $\neq$ `nil` $\wedge$ `(consp l)` $\wedge$ `(rationalp (car l))` $\wedge\varphi_1|_{\texttt{((l (cdr l)))}} \Rightarrow \varphi_1$

Notice that proof obligation 1 above is the not that the base case for `tlp`. With proof obligations 2, 3, we derive `nil` in the context.

So, the professional method told us to use an induction scheme from a function that doesn't even appear in φ_0!

Once you use the professional method to determine the proof, go back and write it up assuming you knew what induction scheme to use. Always do this to make sure you did not skip any steps.

Exercise 6.18 *Finish the proof.*

What about the proof of φ_1? Let's again use the professional method.

```
(orderedp (insert e L))
= { Def insert, (consp L), e<=(car l) } **
(orderedp (cons e L))
= { Def orderedp, (consp L), car-cdr axioms }
(<= e (car L)) ∧ (orderedp L)
= { e<=(car L), context }
```

```
t
```

In the above I assumed `e <= (car l)`, so I still have to consider the following.

```
(orderedp (insert e L))
= { Def insert, (consp L), ¬(e<=(car l)) } **
(orderedp (cons (car L) (insert e (cdr L))))
= { Def orderedp, (consp L), Def insert, car-cdr axioms }
(<= (car L) (car (insert e (cdr L)))) ∧ (orderedp (insert e (cdr l)))
= { φ1|((L (cdr L))) } **
(<= (car L) (car (insert e (cdr L))))
= { φ2 }
t
```

What function gives rise to the induction scheme I used? `(insert e L)`.
We still have to prove the following lemma.

```
φ2 : (lorp L) ∧ (consp L) ∧  ¬(e<=(car L)) ∧ (orderedp L) ⇒
      (<= (car L) (car (insert e (cdr L))))
```

The proof is by case analysis using the following cases.

```
1. (endp (cdr L))
2. (consp (cdr L)) ∧ e<=(second l)
3. (consp (cdr L)) ∧  ¬(e<=(second l))
```

Exercise 6.19 *Prove* φ_2*.*

Now we will prove that `isort` returns a permutation of its input. To do that, we first define the following functions.

```
(definec in (a :rational L :lor) :bool
  (and (consp L)
       (or (== a (car l))
           (in a (cdr L)))))

(definec del (a :rational L :lor) :lor
  (cond ((endp L) L)
        ((== a (car L)) (cdr L))
        (t (cons (car L) (del a (cdr L))))))

(definec permp (x :lor y :lor) :bool
  (if (endp x)
      (endp y)
    (and (in (car x) y)
         (permp (cdr x) (del (car x) y)))))
```

The goal is to prove the following.

$$\varphi_4 : \texttt{(lorp L)} \Rightarrow \texttt{(permp (isort L) L)}$$

Let's use the professional method.

```
(permp (isort L) L)
```

```
= { Def isort, (consp L) } **
(permp (insert (car L) (isort (cdr L))) L)
= { (consp L), car-cdr axioms }  **
(permp (insert (car L) (isort (cdr L))) (cons (car L) (cdr L)))
```

At this point, we can assume

```
(permp (isort (cdr L)) (cdr L))
```

But, how do we use that to make progress? We need a lemma!

$$\varphi_5 : \texttt{(rationalp e)} \wedge \texttt{(lorp x)} \wedge \texttt{(lorp y)} \wedge \texttt{(permp x y)} \Rightarrow$$
$$\texttt{(permp (insert e x) (cons e y))}$$

So, assuming we have φ_5, we continue as follows.

= { $\varphi_5|_{\texttt{((L (cdr L)))}}$, `Def lorp, context` } `**To be filled in later`
`t`

So, what induction scheme does the professional method tell us to use?

Well, we need a function of the form

```
(definec f (L :xxx ):...
  (if (consp L)   **Assumption
      ... (f (cdr L)) ... **IH
     ...))
```

where ... doesn't matter, as long as it is not recursive, and `xxx` matters, but we have flexibility.

So, again `tlp` works! Thus, we induct on `(tlp L)`. As was the case previously, `tlp` does not show up in φ_4!

Once you figure out what induction scheme to use, recall that you have to go back and do the proof carefully using the induction scheme. If you are doing this on a computer, you can cut and paste!

Exercise 6.20 *Prove* φ_4*.*

What about the proof of the φ_5?

Let us again use the Professional Method.

```
(permp (insert e x) (cons e y))
= { Def insert, (consp x), e<=(car x) } **
(permp (cons e x) (cons e y))
= { Def permp, del }
(permp x y)
= { context }
t

(permp (insert e x) (cons e y))
= { Def insert, (consp x), ¬[e<=(car x)] } **
(permp (cons (car x) (insert e (cdr x))) (cons e y))
= { Def permp, del }
(permp (insert e (cdr x)) (del (car x) (cons e y)))
```

```
***Note: not yet ready to use an IH due to (del ...)
= { Def del, Arithmetic, car-cdr axioms }
(permp (insert e (cdr x)) (cons e (del (car x) y)))
= { IH|((x (cdr x)) (y (del (car x) y))), context } **
t
```

What induction scheme do we wind up with now? We need a function of the following form.

```
(definec f (e :rational x :lor y :lor) : ...
   (cond ((endp x) ...)
         ((<= e (car x)) ...)
         (t ... (f e (cdr x) (del (car x) y)))))
```

Do we have such a function available? No. Is such a function admissible? Sure. It fits one of the recursion schemes we have considered.

```
(definec f (e :rational x :lor y :lor) :lor
  (cond ((endp x) y)
        ((<= e (car x)) x)
        (t (f e (cdr x) (del (car x) y)))))
```

Exercise 6.21 *Finish the proof of* φ_5.

Exercise 6.22 *If I used the following definitions, then* φ_4 *still holds. Prove it.*

```
(definec in (a :all L :tl) :bool
  (and (consp L)
       (or (== a (car l))
           (in a (cdr L)))))

(definec del (a :all L :tl) :tl
  (cond ((endp L) L)
        ((== a (car L)) (cdr L))
        (t (cons (car L) (del a (cdr L))))))

(definec permp (x :tl y :tl) :bool
  (if (endp x)
      (endp y)
    (and (in (car x) y)
         (permp (cdr x) (del (car x) y)))))
```

Hint. The idea is to show that the same theorems we had with the more restricted definitions still hold.

So, we need lemmas showing that when we restrict these functions to `lor`*'s and* `rational`*'s good things happen,* e.g.,

$$\texttt{(rationalp a)} \wedge \texttt{(lorp L)} \Rightarrow \texttt{(lorp (del a L))}$$

Exercise 6.23 *Use* `sig` *to prove Exercise 6.22 in ACL2s.*

6.8 Generalization

Consider the following definitions.

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (== a (first X))
           (in a (rest X)))))

(definec subset? (x :tl y :tl) :bool
  ; checks if every element in x is in y
  (or (endp x)
      (and (in (first x) y)
           (subset? (rest x) y))))
```

Try to prove the following theorem:

$$\texttt{(tlp x)} \Rightarrow \texttt{(subset? x x)} \tag{6.5}$$

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:

$$\texttt{(tlp x)} \wedge \texttt{(tlp y)} \wedge \texttt{(subset? x y)} \Rightarrow \texttt{(subset? x (cons a y))} \tag{6.6}$$

Now, we can prove (6.6) using induction.

Exercise 6.24 *Prove* (6.6)

Exercise 6.25 *Prove* (6.5)

Exercise 6.26 *Prove* `(tlp x)` $\wedge$ `(tlp y)` $\wedge$ `(tlp z)` $\wedge$ `(subset? x y)` $\wedge$ `(subset? y z)` $\Rightarrow$ `(subset? x z)`

6.9 Reasoning About Accumulator-Based Functions

Let's start with a simple definition we have already seen.

```
(definec rev2 (x :tl) :tl
  (if (endp x)
      nil
    (app2 (rev2 (rest x)) (list (first x)))))
```

The problem with this definition is that it requires $O(n^2)$ conses. Why?

Because `(app2 x y)` requires `(len x)` conses (as we have seen previously).

In the recursive case of `rev2`, we have `app2` applied to `(rev2 (rest x))` which requires `(len (rev2 (rest x)))` conses plus one cons for the list, *i.e.*, `(len x)` conses. Since `rev2` is called on `x`, then `(rest x)`, then `(rest (rest x))`, …, it requires `(len x)` + `(len x)`-1 + `(len x)`$-2 + \cdots + 1$ 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.

```
(definec revt (x :tl acc :tl) :tl
  (if (endp x)
      acc
    (revt (rest x) (cons (first x) acc))))
```

But, we want a function with the same interface as `rev2` and `revt` takes 2 arguments. Hence, we define:

```
(definec rev* (x :tl) :tl
  (revt x nil))
```

Exercise 6.27 *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 `rev2` 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 `rev2`?

Let's prove that they are equal.

$$\texttt{(tlp x)} \Rightarrow \texttt{(rev* x)} = \texttt{(rev2 x)} \tag{6.7}$$

Is it true?

Can we solve this with equational reasoning? No. Why not?

Notice that proving (6.7) will require proving:

$$\texttt{(tlp x)} \Rightarrow \texttt{(revt x nil)} = \texttt{(rev2 x)} \tag{6.8}$$

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 of `(tlp x)`.

1. $\neg$`(consp x)` $\Rightarrow$ (6.8)
2. `(consp x)` $\wedge$ (6.8)$|_{\texttt{((x (rest x)))}}$ $\Rightarrow$ (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. `(tlp x)`

C3. `(tlp (rest x))` $\Rightarrow$ `(revt (rest x) nil)` $=$ `(rev2 (rest x))`

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

C5. `(revt (rest x) nil)` $=$ `(rev2 (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 all instantiations of the theorem we want to prove (which is how induction hypotheses are generated) 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.

`(tlp x)` $\wedge$ `(tlp acc)` $\Rightarrow$ `(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:

`(tlp x)` $\wedge$ `(tlp acc)` $\Rightarrow$ `(revt x acc)` $=$ `(app2 (rev2 x) acc)` (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. `(tlp x)`

C2. `(tlp acc)`

C3. `(consp x)`

C4. `(tlp (rest x))` $\wedge$ `(tlp acc)` $\Rightarrow$
    `(revt (rest x) acc)` = `(app2 (rev2 (rest x)) acc)`

D1. `(tlp (rest x))` {Def `tlp`, C1, C3}

D2. `(revt (rest x) acc)` =
    `(app2 (rev2 (rest x)) acc)` {C4, D1, C2, 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

`(revt (rest x) acc)` = `(app2 (rev2 (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 of `(revt x acc)`. That gives us the following context:

C1. `(consp x)`

C2. `(tlp x)`

C3. `(tlp acc)`

C4. `(tlp (rest x))` $\wedge$ `(tlp (cons (first x) acc))` $\Rightarrow$
    `(revt (rest x) (cons (first x) acc))` =
    `(app2 (rev2 (rest x)) (cons (first x) acc))`

D1. `(tlp (rest x))` {C2, Def `tlp`, C1}

D2. `(tlp (cons (first x) acc))` {C3, Def `tlp`, cons axioms}

D3. `(revt (rest x) (cons (first x) acc))` =
    `(app2 (rev2 (rest x)) (cons (first x) acc))` {C4, D1, D2, MP}

Proof:

```
 (revt x acc)
```

= { By C1 and the definition of `revt` }

```
    (revt (rest x) (cons (first x) acc))
```

= { By D3 }

```
    (app2 (rev2 (rest x)) (cons (first x) acc))
```

= { Def of `app2` (working on pulling `acc` out to match RHS) }

```
    (app2 (rev2 (rest x)) (app2 (list (first x)) acc))
```

= { Associativity of `app2` (now we pull `acc` out) }

```
    (app2 (app2 (rev2 (rest x)) (list (first x))) acc)
```

= { Def of `rev2`, C1 }

```
    (app2 (rev2 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 of `ft`.
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.

6.10 Exercises

6.10.1 List Theory

The following functions are used in the exercises below.

```
(definec rem-dups (a :tl) :tl
  (cond ((endp a) nil)
        ((in (car a) (cdr a))
         (rem-dups (cdr a)))
        (t (cons (car a) (rem-dups (cdr a))))))
```

```
(definec no-dups (a :tl) :bool
  (or (endp a)
      (and (not (in (car a) (cdr a)))
           (no-dups (cdr a)))))
```

Exercise 6.28 *Prove the following.*

```
;theorem llen-app2
(implies (and (tlp x)
              (tlp y))
         (= (llen (app2 x y))
            (+ (llen x) (llen y))))
```

Exercise 6.29 *Prove the following.*

```
;theorem llen-rev2
(implies (tlp x)
         (= (llen (rev2 x))
            (llen x)))
```

Exercise 6.30 *Prove the following.*

```
;theorem llen-rem-dups
(implies (tlp x)
         (<= (llen (rem-dups x))
             (llen x)))
```

Exercise 6.31 *Prove the following.*

```
;theorem llen-rem-dups-no-dups
(implies (and (tlp x)
              (no-dups x))
         (= (llen (rem-dups x))
            (llen x)))
```

Exercise 6.32 *Prove the following.*

```
;theorem in-app2
(implies (and (tlp x)
              (tlp y))
         (== (in a (app2 x y))
             (or (in a x) (in a y))))
```

Exercise 6.33 *Prove the following.*

```
;theorem in-rev2
(implies (tlp x)
         (== (in a (rev2 x))
             (in a x)))
```

Exercise 6.34 *Prove the following.*

```
;theorem in-rem-dups
(implies (tlp x)
         (== (in a (rem-dups x))
             (in a x)))
```

Exercise 6.35 *Prove the following.*

```
;theorem rem-dups-no-dups
(implies (and (tlp x)
              (no-dups x))
         (== (rem-dups x)
             x))
```

Exercise 6.36 *Prove the following.*

```
;theorem rem-dups-has-no-dups
(implies (tlp x)
         (no-dups (rem-dups x)))
```

Exercise 6.37 *Prove the following.*

```
;theorem rem-dups-idempotent
(implies (tlp x)
         (== (rem-dups (rem-dups x))
             (rem-dups x)))
```

6.10.2 Set Theory

The following functions are used in the exercises below.

```
(definec == (x :tl y :tl) :bool
  ; checks if x and y have exactly the same set of elements
  (and (subset? x y)
       (subset? y x)))

(definec union (x :tl y :tl) :tl
; the union of x and y
  (app2 x y))

(definec intersect (x :tl y :tl) :tl
  ; the intersection of x and y
  (cond ((endp x)  nil)
        ((in (car x) y)
         (cons (car x) (intersect (cdr x) y)))
        (t (intersect (cdr x) y))))

(definec cardinality (x :tl) :nat
; the number of distinct elements in x
  (llen (rem-dups x)))
```

For all the exercises in this section, either prove the conjecture or exhibit a counterexample. You may find it useful to do the exercises in an order that is different than the order in which they are presented.

Exercise 6.38 *Prove the following.*

```
;theorem intersect-same
(implies (tlp x)
         (== (intersect x x)
             x))
```

Exercise 6.39 *Prove the following.*

```
;theorem intersect-x-sub-y
(implies (and (tlp x)
              (tlp y)
              (subset? x y))
         (== (intersect x y)
             x))
```

Exercise 6.40 *Prove the following.*

```
;theorem intersect-llen
(implies (and (tlp x)
              (tlp y)
              (tlp z)
              (subset? x y)
              (subset? y z))
         (= (llen (intersect x (intersect y z)))
            (llen x)))
```

Exercise 6.41 *Prove the following.*

```
;theorem in-intersect
(implies (and (tlp x)
              (tlp y))
         (== (in a (intersect x y))
             (and (in a x) (in a y))))
```

Exercise 6.42 *Prove the following.*

```
;theorem intersect-associative
(implies (and (tlp x)
              (tlp y)
              (tlp z))
         (== (intersect (intersect x y) z)
             (intersect x (intersect y z))))
```

Exercise 6.43 *Prove the following.*

```
;theorem intersect-llen-lemma
(implies (and (tlp x)
              (tlp y)
              (tlp z)
              (tlp w))
         (= (llen (intersect (intersect x y) (intersect w z)))
            (llen (intersect x (intersect (intersect y w) z)))))
```

Exercise 6.44 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (subset? (rev2 x) y))
         (subset? x (rev2 y)))
```

Exercise 6.45 *Prove the following.*

```
(implies (tlp x)
         (== (union x x) x))
```

Exercise 6.46 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (== (union x y)
             (union y x)))
```

Exercise 6.47 *Prove the following.*

```
(implies (tlp x)
         (== (intersect x x) x))
```

Exercise 6.48 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (<= (cardinality (union x y))
             (+ (cardinality x) (cardinality y))))
```

Exercise 6.49 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (tlp z))
         (== (intersect (intersect x y) z)
             (intersect x (intersect y z))))
```

Exercise 6.50 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (tlp z))
         (== (union (union x y) z)
             (union x (union y z))))
```

Exercise 6.51 *Prove the following.*

```
(implies (tlp x)
         (<= (cardinality x)
             (llen x)))
```

Exercise 6.52 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (= (cardinality (union x x))
            (cardinality x)))
```

Exercise 6.53 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (= (cardinality (intersect x x))
            (cardinality x)))
```

Exercise 6.54 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (<= (cardinality (union x y))
             (+ (cardinality x) (cardinality y))))
```

Exercise 6.55 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (= (cardinality (union x y)) (cardinality y)))
         (= (cardinality (union y x))
            (cardinality y)))
```

Exercise 6.56 *Prove the following.*

```
(implies (and (tlp x)
              (= (cardinality x) (llen x)))
         (== (rem-dups x) x))
```

Exercise 6.57 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (subset? x y)
              (<= n (cardinality x)))
         (<= n (llen y)))
```

Exercise 6.58 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (< 0 (cardinality x))
              (== x y))
         (consp y))
```

Exercise 6.59 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (subset? (union x y) x)
              (< (cardinality x) (cardinality y)))
         (not (== x x)))
```

6.10.3 Sorting

In this section, the exercises require you define functions and formalize claims written in (rigorous) English.

Exercise 6.60 *Use* `defdata` *to define* `lor`*, a (true) list of rationals.*

Exercise 6.61 *Define insertion sort, where the input is a* `lor`*.*

Exercise 6.62 *Define quicksort, where the input is a* `lor`*.*

Exercise 6.63 *Define merge sort, where the input is a* `lor`*.*

Exercise 6.64 *Define bubble sort, where the input is a* `lor`*.*

Exercise 6.65 *Define* `orderedp`*, a function that given an* `lor` *checks that the elements of the list are in non-decreasing order.*

Exercise 6.66 *Define* `permp`*, a function that given a two true-lists (not necessarily* `lor`*s) checks that they are permutations of each other.*

Exercise 6.67 *Show that insertion sort returns an ordered permutation.*

Exercise 6.68 *Show that quicksort returns an ordered permutation.*

Exercise 6.69 *Show that merge sort returns an ordered permutation.*

Exercise 6.70 *Show that bubble sort returns an ordered permutation.*

Exercise 6.71 *Show that if* `x` *and* `y` *are permutations of each other and they are both ordered, then they are equal.*

Exercise 6.72 *Show that all the sorting algorithm are equal to each other.*

Exercise 6.73 *Without using the definition of* `permp` *(or any theorems that depend on* `permp`*) show that insertion sort is equal to quicksort.*

6.10.4 Tail Recursion

Use the recipe for reasoning about accumulator-based functions for all of the exercises in this section.

Exercise 6.74 *Define and prove the correctenss of a tail recursive version of* `sumn`*.*

Exercise 6.75 *Define and prove the correctenss of a tail recursive version of* `insert`*.*

Exercise 6.76 *Define and prove the correctenss of a tail recursive version of* `del`*.*

Exercise 6.77 *Define and prove the correctenss of a tail recursive version of* `isort`*.*

Exercise 6.78 *Define and prove the correctenss of a tail recursive version of* `fib`*.*

Exercise 6.79 *Define and prove the correctenss of a tail recursive version of* `app2`*.*

Exercise 6.80 *Define and prove the correctenss of a tail recursive version of* `rem-dups`*.*

Exercise 6.81 *Define and prove the correctenss of a tail recursive version of* `intersect`*.*

6.10.5 Miscellaneous Exercises

Exercise 6.82 *Prove that there is a bijection from* `lex` *to* `nat` *using the ACL2s logic (paper and pencil). See Exercise 5.21. Do this by defining an ACL2s function from* `lex` *to* `nat` *and formalizing what it means for it to be a bijection.*

Part VI

Steering the ACL2 Sedan

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 (`$class_1 \ldots class_n$`))`

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)
```

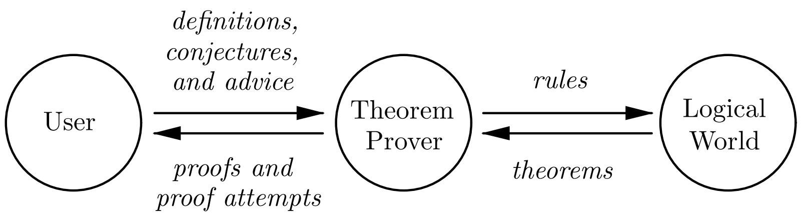
definitions, conjectures, and advice
User
Theorem Prover
Logical World
rules
proofs and proof attempts
theorems

Figure 7.1: Data Flow in the Theorem Prover

```
  (if (endp x)
      nil
    (append (rev (rest x)) (list (first x)))))

(defthm rev-rev
  (implies (tlp x)
           (equal (rev (rev x)) x)))
```

Think of `defun` as `definec` without contracts. 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.

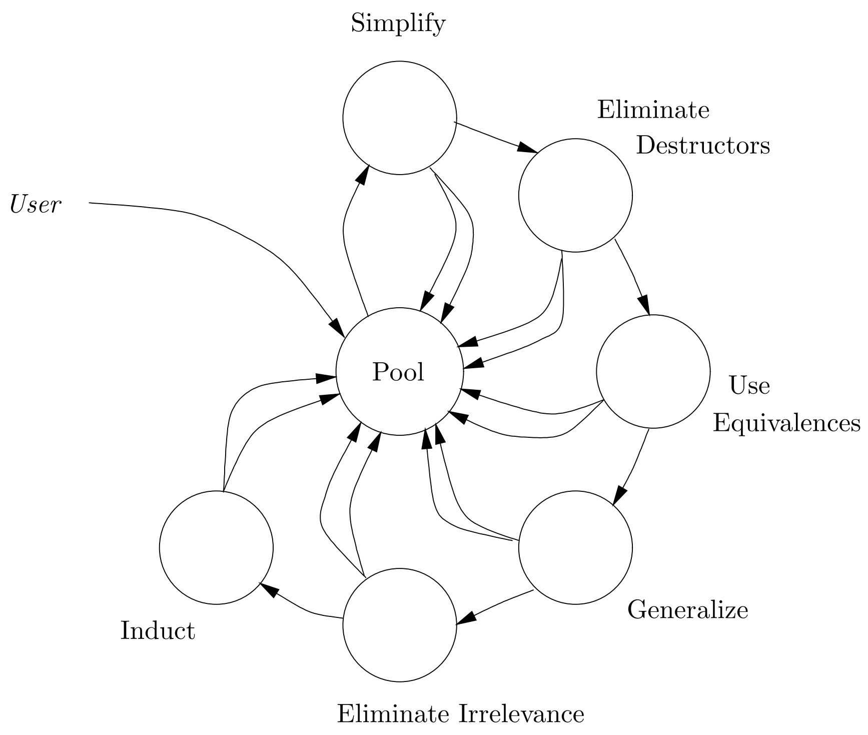

Figure 7.2: Organization of the Theorem Prover

Here is a user-level description of how the rewriter works. The following description is not altogether accurate but is relatively simple and predicts the behavior of the rewriter in nearly all cases you will encounter.

If given a variable or a constant to rewrite, the rewriter returns it. Otherwise, it is dealing with a function application, `(`f a_1 ... 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 (tlp x) (tlp y) (tlp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))

(defthm app-associative2
  (implies (and (tlp x) (tlp y) (tlp 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 (tlp x) (tlp y) (tlp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

What does the following get rewritten to?

```
(implies (and (tlp a) (tlp b) (tlp c) (tlp 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 (tlp x) (tlp 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 (tlp 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.

7.4 Exercises

Prove all of the exercises in Chapter 6 using the ACL2s theorem prover.

For example, here is the exercise corresponding to Exercise 6.82

Exercise 7.1 *Prove the theorems in Exercise 6.82 using the ACL2s theorem prover.*

Part VII

Advanced Topics

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.

```
(definec !! (n :nat) :pos
  (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))

; Data definition of an empty stack
(defdata empty-stack nil)

; Data definition of a non-empty stack
(defdata non-empty-stack (cons element stack))

; Empty, non-empty stacks are stacks
(defdata-subtype empty-stack stack)
(defdata-subtype non-empty-stack stack)

; The push operation inserts e on the top of the stack s
(definec stack-push (e :element s :stack) :non-empty-stack
  (cons e s))

; The pop operation removes the top element of a non-empty stack
(definec stack-pop (s :non-empty-stack) :stack
  (rest s))

; The head of a non-empty stack
(definec stack-head (s :non-empty-stack) :element
  (first s))

; Stack creation: returns an empty stack
(definec new-stack () :empty-stack
  nil)
```

While we now have an implementation, we do not have an implementation-independent characterization of stacks. In fact, we will see two such characterizations.

To simplify the reasoning we have to do later, we assume that `app2` and `rev2`, as well as the various theorems we have proved about them have been defined. In addition, we add some theorems about `app2` and `rev2` (that ACL2s knows about the default versions of these functions, `app` and `rev`). These theorems state that if you append or reverse a list of some type then you get back a list of the same type. We also have a macro and a theorem about `stackp`.

```
(sig app2 ((listof :a) (listof :a)) => (listof :a))
(sig rev2 ((listof :a)) => (listof :a))
(defthm stack-tlp (equal (stackp l) (tlp l)))
(defmacro stack (a) `(listof ,a))
```

8.2 Algebraic Data Types

The first idea is to characterize stacks using only the algebraic properties they satisfy.

What the user of our library will be able to see is the following.

1. The subtype theorems:

   ```
   (defdata-subtype empty-stack stack)
   (defdata-subtype non-empty-stack stack)
   ```

   These theorems state that `empty-stack` and `non-empty-stack` are subtypes of `stack`. That is any object that satisfies `empty-stackp` also satisfies `stackp` and similarly for `non-empty-stackp`.

2. The following signatures and theorems. The signatures correspond to the contracts of the functions defined, parameterized over the kinds of elements the stack contains. That is, think of `:a` as a type; if it is `nat`, then the signature for `stack-push` tells us that calling `stack-push` on a `nat` and a stack of `nat`s gives us back a stack of `nat`s. The `satisfies` clauses for `stack-pop` and `stack-head` tell us that their first arguments (denoted by `x1`) are non-empty stacks.

   ```
   (sig stack-push (:a (stack :a)) => (stack :a))
   (sig stack-pop ((stack :a)) => (stack :a)
        :satisfies (non-empty-stackp x1))
   (sig stack-head ((stack :a)) => :a
        :satisfies (non-empty-stackp x1))
   ```

   The theorems tell us that `new-stack` return an empty stack, that `elementp` is a recognizer and that `stack-push` returns a non-empty stack.

   ```
   (thm (empty-stackp (new-stack)))
   (thm (booleanp (elementp e)))
   (thm (implies (and (elementp e) (stackp s))
                 (non-empty-stackp (stack-push e s))))
   ```

   Notice that the user does not see the data definition of a stack, because how we represent stacks is implementation-dependent. They also do not see the data definition of `element`, since that that can be any recognizer.

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 head-push
  (implies (and (stackp s)
                (elementp e))
           (equal (stack-head (stack-push e s))
                  e)))

(defthm push-pop
  (implies (non-empty-stackp s)
           (equal (stack-push (stack-head s) (stack-pop s))
                  s)))

(defthm empty-stack-unique
  (implies (empty-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))

; Data definition of an empty stack
(defdata empty-stack nil)

; Data definition of a non-empty stack
(defdata non-empty-stack (cons element stack))

; Empty, non-empty stacks are stacks
(defdata-subtype empty-stack stack)
(defdata-subtype non-empty-stack stack)

; Stack creation: returns an empty stack
(definec new-stack () :empty-stack
  nil)

; The push operation inserts e on the top of the stack s
; The :otf-flg directive tells ACL2s to use induction on multiple
; initial subgoals (see the documentation for details)
(definec stack-push (e :element s :stack) :non-empty-stack
  :otf-flg t
  (app2 s (list e)))
```

To admit the next function, we need some lemmas about `stackp`, `app2` and `rev2`.

```
(defthm stack-app
  (implies (and (stackp x)
                (stackp y))
           (stackp (app2 x y))))

(defthm stack-rev
  (implies (stackp x)
           (stackp (rev2 x))))

; The pop operation removes the top element of a non-empty stack
(definec stack-pop (s :non-empty-stack) :stack
  (rev* (rest (rev* s))))

; The head of a non-empty stack
```

```
(definec stack-head (s :non-empty-stack) :element
  (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 are considering a basic question: how do we specify computational systems?

In the context of a stack library, one of the key ideas is that of an abstract data type. What this means is that we will restrict what a user of the library can do. Instead of having access to the underlying implementation, a user can only create, access and modify stacks using a set of functions (methods, procedures, etc.) that we have defined. That is why the data type is "abstract." The reasons for this are numerous and include the following.

1. Separation of concerns. We remove dependencies between our implementation of a stack and code that uses stacks. For example, a customer of the library can write lots of code that depends on our library and years later we change the library without having to worry about all the code that depends on it failing.

2. It makes debugging easier. If you have a clear interface, and a bug is found, it is now much easier to determine who is responsible. Otherwise, you run into the situation where the library owner can claim that the behavior is a feature, not a bug and the the library user can claim that it is a bug, not a feature.

3. It makes development easier. One can parallelize development once you have an interface. If we agree on an interface, then one team can start developing a library while another develops an application that depends on the library.

However, knowing what the functions are and what their signatures are is not enough to characterize the data type. We will consider two qualitatively different approaches for proceeding from here.

1. The first approach is based on properties. We create a list of properties that the data structure and its functions should satisfy. That is what we did with stacks in the previous sections. We defined a collection of "algebraic properties," hence the term "algebraic data types." We considered notions of redundancy, independece and completeness. We saw how to deal with redundancy and independece (but not completeness).

2. The other approach is based on *refinement*. We will define a simple implementation, which will be the specification. We can make that available to users of the library. Then, we 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. Customers of the library have only one guarantee: that the actual implementation is going to provide the same externally visible behavior.

We have already seen an example of these two approaches in the context of sorting lists of numbers.

Using the property-based approach, we agreed that a sorting algorithm is correct if it returns an ordered permutation of its input. Arriving at this specification was not trivial and alternative proposals for characterizing correctness are often wrong, *e.g.*, this specification is wrong: the algorithm must return an ordered list, every element in the output list must be in the input list, every element in the input list must be in the output list and the length of the lists must be equal.

Using the refinement-based approach, we agreed that a sorting algorithm is correct if it is equal to insertion sort.

An advantage of the refinement specification is that it is clearly complete, as it tells us exactly what a correct sorting algorithm should return for every legal input. On the other hand, it is much harder to determine if a set of properties is complete.

We now consider how to use the refinement-based approach to characterize 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. `empty-stackp`: 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` (if `pop` or `head` are called on an empty stack).

```
(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 `o`.

```
(definec external-observation (s :stack o :operation) :observation
  (cond ((equal o 'empty?)
          (list (empty-stackp s)))
        ((consp o) (list (second o)))
        ((equal o 'pop) (if (empty-stackp s) 'error nil))
        (t (if (empty-stackp 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 (non-empty-stackp s))
                (equal (external-observation s 'pop)
                       nil)))

(test? (implies (empty-stackp s)
                (equal (external-observation s 'pop)
                       'error)))

(test? (implies (empty-stackp s)
                (equal (external-observation s 'head)
                       'error)))

(test? (implies (and (stackp s) (elementp e))
                (equal (external-observation (stack-push e s) 'head)
                       (list e))))

(test? (implies (non-empty-stackp s)
                (equal (external-observation s 'empty?)
                       (list nil))))

(test? (implies (empty-stackp 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.

```
(definec update-stack (s :stack op :operation) :stack
  (cond ((in op '(empty? head))
         s)
        ((equal op 'pop)
         (if (empty-stackp s)
             (new-stack)
           (stack-pop s)))
        (t (stack-push (second op) s))))

(definec external-observations (s :stack l :lop) :lob
  (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)

; Data definition of a queue: a list of elements
(defdata queue (listof element))

; Data definition of an empty queue
(defdata empty-queue nil)

; Data definition of a non-empty queue
(defdata non-empty-queue (cons element queue))
```

```
; Empty, non-empty queues are queues
(defdata-subtype empty-queue queue)
(defdata-subtype non-empty-queue queue)

; Queue creation: a new queue is just the empty list
(definec new-queue () :empty-queue
  nil)

; The head of a queue. Let's decide that the head of the queue
; will be the first.
(definec queue-head (q :non-empty-queue) :element
  (first q))

; Dequeueing can be implemented with rest
(definec queue-dequeue (q :non-empty-queue) :queue
  (rest q))

; Enqueueing to a queue requires putting the element at the
; end of the list.
(definec queue-enqueue (e :element q :queue) :non-empty-queue
  :otf-flg t
  (app2 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 last element the list. We will define a new version of all the previous queue-functions.

```
(defdata element2 all)

(defdata queue2 (listof element2))

; Data definition of an empty queue2
(defdata empty-queue2 nil)

; Data definition of a non-empty queue2
(defdata non-empty-queue2 (cons element2 queue2))

; Empty, non-empty queue2s are queue2s
(defdata-subtype empty-queue2 queue2)
(defdata-subtype non-empty-queue2 queue2)

; A new queue2 is just the empty list
(definec new-queue2 () :empty-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*.
(definec queue2-head (q :non-empty-queue2) :element2
  (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. Since we have rev*, we will use that to make
; this more efficient that if we were to use rev2.
(definec queue2-dequeue (q :non-empty-queue2) :queue2
  :otf-flg t
  (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.
(definec queue2-enqueue (e :element2 q :queue2) :non-empty-queue2
  (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` (if `h` `d` are called on an empty queue).

```
(defdata observation (oneof (list boolean) (list element) nil 'error))
```

Next, 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 (app2 q (list x)))))

(definec queue-run (l :lop q :queue) :lob
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
             (if (empty-queuep q)
                 (list 'error)
               (cons nil (queue-run (rest l) (queue-dequeue q)))))
            ((equal i 'h)
             (if (empty-queuep q)
                 (list 'error)
               (cons (list (queue-head q)) (queue-run (rest l) q))))
            ((equal i 'e?)
             (cons (list (empty-queuep 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 (rev2 (rest (rev2 q))))))

(definec queue2-run (l :lop q :queue2) :lob
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
             (if (empty-queue2p q)
                 (list 'error)
               (cons nil (queue2-run (rest l) (queue2-dequeue q)))))
            ((equal i 'h)
             (if (empty-queue2p q)
                 (list 'error)
               (cons (list (queue2-head q)) (queue2-run (rest l) q))))
            ((equal i 'e?)
             (cons (list (empty-queue2p q)) (queue2-run (rest l) q)))
            (t (cons (list (second i))
                     (queue2-run (rest l) (queue2-enqueue (second i) q))))))))
```

Here is a 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`.

```
(definec queue2-to-queue (q :queue2) :queue
 (rev* q))
```

We need some lemmas.

```
(defthm queue2-queue-rev
  (implies (queue2p x)
           (queuep (rev2 x))))

(defthm app2-non-empty
  (implies (tlp x)
           (app2 x (list y))))
```

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)))))
```

Reasoning about Imperative Code

We have seen how to reason about programs written in the ACL2s language using the ACL2s logic and the ACL2s theorem prover. How do we reason about programs written in other languages? Can we use what we have learned so far or do we have to define a logic for the language and then a theorem prover for that language and logic? We will explore these questions in this chapter.

In Section 9.1 we introduce a simple imperative language (SIP) and define its semantics using ACL2s. In contrast to ACL2s, imperative languages allow us to change the value of variables, so procedures in imperative languages are not functions, *e.g.*, they do not satisfy the axioms of equality that ACL2s functions satisfy.

In Section 9.2, we show how to define the semantics of SIP programs by compiling SIP programs into ACL2s programs. This allows us to easily generate executable code from our SIP programs. It also allows us to use all the nice features of ACL2s to analyze SIP programs, including tracing and property-based testing.

In Section 9.3, we discuss how to reason about SIP programs. First we reason about the ACL2s versions of SIP programs and then we show how to build systems that allow us to directly reason about SIP programs using invariants, loop invariants, preconditions, postconditions, assumptions and guarantees. These systems work by generating verification conditions based on invariants provided by programmers and do not require any expertise with the reasoning engine used to discharge these proof obligations. Section 9.4 includes a set of exercises reinforcing this kind of reasoning and includes a link to an invariant discovery game that allows one to reason about SIP programs.

9.1 SIP: A Simple Imperative Language

We introduce SIP, a simple imperative language, via an example. Consider the following simple SIP program.

```
program multiply
main(n, m: nat)
{ var res, cnt: int;
  res := 0;
  cnt := 0;
  while  (cnt < m)
  { cnt  := cnt+1;
    res  := res+n;
  }
  return res;
}
```

SIP programs have a name; the above program's name is `multiply`. The SIP programs we consider have one procedure called `main`. In the program above, main has two inputs, `n` and `m`. Both are of type `nat`. The SIP language has two types. The type `int` corresponds to integers of arbitrary precision, *i.e.*, there is no minimal integer and there is no maximal integer. SIP also supports the type `nat`, which corresponds to integers greater than or equal to 0.

All non-input variables need declarations. The variables `res` and `cnt` are declared to be of type `int`. We use the assignment operator (`:=`) to assign initial values to the variables.

Next we have a while loop, consisting of the loop condition `(cnt < m)` and the body of the loop, which updates the variables `cnt` and `res`.

Finally we have a `return` statement that indicates what the procedure returns. In `multiply`, the value of `res` is returned.

Before we formalize the semantics of such programs, let us start by considering program traces. We will do that with the simple approach of adding `print` statements as follows.

```
program multiply
main(n, m: nat)
{ var res, cnt: int;
  res := 0;
  cnt := 0;
  while  (cnt < m)
  { print(m, cnt, n, res);
    cnt  := cnt+1;
    res  := res+n;
  }
  print(m, cnt, n, res);
  return res;
}
```

Now, imagine running the program with the inputs `n = 7` and `m = 4`. The trace our program generates is the following (without the header).

```
m  cnt   n  res
4    0   7    0
4    1   7    7
4    2   7   14
4    3   7   21
4    4   7   28
```

The trace makes it clear that the program computes `n * m` by repeatedly adding `m` to `res`.

9.2 Semantics of SIP

To define the semantics of SIP, we will write a compiler that takes a SIP program and generates an ACL2s representation. This compiler can be written in any language. We will not define this compiler, but we will show what it generates for the `multiply` program.

```
(sip-program
 multiply
 ((natp n) (natp m))
 ((intp cnt) (intp res))
 ((res 0) (cnt 0))
 (< cnt m)
 ((print m cnt n res)
  (cnt . (+ 1 cnt))
  (res . (+ res n)))
 ((print m cnt n res))
 (res))
```

The above is a call of the ACL2s macro `sip-program` that includes the name of the SIP program (`multiply`), the input variables and their types, the declared variables and their types, the initialization code, the while loop condition, the while loop body, where we either have a `print` statement or an assignment operation. Note that we use alists to denote how a variable gets updated. Then we have any post-loop statements and finally what, if anything, gets returned.

Notice that the ACL2s representation is very close to the original SIP program and that is by design because we want this step to be as simple and transparent as possible.

Now, the `sip-program` macro will wind up generating a list of definitions that provide the semantics of `multiply`. The macro generates a list of utilities. One of those utilities allows us to generate traces. More of these utilities will be introduced later. We will not define the macro, but we will reveal what it generates.

First, the macro generates `sip-multiply`, an ACL2s version of `multiply`. In order to define such a function, we need to deal with the while loop and this is done via the use of a recursive helper function, `sip-multiply-loop`, as follows.

```
:program

(set-ignore-ok t)

(definec sip-multiply-loop (n :nat m :nat cnt :int res :int) :int
  (if (< cnt m)
      (let ((cnt (+ 1 cnt))
            (res (+ res n)))
        (sip-multiply-loop n m cnt res))
    res))

(definec sip-multiply (n :nat m :nat) :int
  (let ((res 0)
        (cnt 0))
    (sip-multiply-loop n m cnt res)))
```

Now, we can run the program as the following examples show.

```
(sip-multiply 7 4)
(sip-multiply 127 671)
```

Not only do we now have executable versions of SIP programs such as `multiply`, but we also have access to all ACL2s capabilities. For example, we can use `check=` and `test?` forms as shown below.

```
(check= (sip-multiply 7 4) (* 7 4))

(test? (implies (and (natp n) (natp m))
                (= (sip-multiply n m) (* n m))))
```

The `sip-program` macro also generates utility functions for generating traces.

```
(definec sip-multiply-loop-trace (n :nat m :nat cnt :int res :int) :tl
  (if (< cnt m)
      (cons (list m cnt n res)
            (let ((cnt (+ 1 cnt))
                  (res (+ res n)))
              (sip-multiply-loop-trace n m cnt res)))
    `((,m ,cnt ,n ,res))))

(definec sip-multiply-trace (n :nat m :nat) :tl
  (let ((res 0)
        (cnt 0))
    (app '((multiply-trace)
           (m cnt n res))
         (sip-multiply-loop-trace n m cnt res))))
```

Here is an example of the trace utility functions in action.

```
(sip-multiply-trace 7 4)
(sip-multiply-trace 21 18)
```

In order to develop robust, usable tools, we have to deal with syntax checking, type checking, error reporting, termination analysis, etc. during the compilation step. We will ignore these issues as considering them in any depth will take us too far afield. We therefore assume that SIP programs are free of errors and that they are terminating.

Exercise 9.1 *Define a compiler that given a SIP program generates the corresponding* `sip-program` *form. You can use your favorite language to do that.*

Exercise 9.2 *Define a version of the* `sip-program` *macro that generates the forms shown above.*

Exercise 9.3 *You may wonder why the functions generated by* `sip-program` *are defined in* `:program` *mode. Come up with a SIP program that is terminating, but for which one of the ACL2s functions generated by* `sip-program` *is non-terminating.*

9.3 Reasoning About SIP Programs

Now that we can compile SIP programs to ACL2s programs, we can reason about SIP programs using ACL2s. Try the following exercise before continuing.

Exercise 9.4 *Prove the following conjecture using paper and pencil. Then prove it using ACL2s. Assume that* `sip-program` *generated* `:logic` *mode functions.*

```
(implies (and (natp n)
              (natp m))
         (= (sip-multiply n m) (* n m)))
```

Notice that since `sip-multiply` is a non-recursive program that calls `sip-multiply-loop`, we really need a lemma about `sip-multiply-loop`. Also, for reasons we have already discussed in the context of tail recursive functions, we will not be able to prove a lemma about `sip-multiply-loop` that includes constants: we will have to generalize. Once we have the appropriate lemma, then the main theorem follows via equational reasoning.

There are many lemmas that can be used to prove the main theorem. We know we have to generalize, so one idea is to prove a lemma that before contract completion is as close as possible to this, *i.e.*, we are trying to determine what `sip-multiply-loop` does given arbitrary inputs.

```
(= (sip-multiply-loop n m cnt res)
   ...)
```

We can use ACL2s to help us discover an appropriate lemma (of course we need to perform contract completion). Since we keep adding to `res` during the loop, we expect an expression of the form `(+ ... res)`. We are adding `n` each time through the loop, so we should get something like `(+ (* n ...) res)`. How many times do we add `n`? At the beginning of the loop it is `m`, but at an arbitrary point during the loop we have already gone through `cnt` iterations, so there should be `(- m cnt)` iterations left. So, we can try the following.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res))
          (= (sip-multiply-loop n m cnt res)
             (+ (* n (- m cnt)) res))))
```

We get counterexamples, but they involve assignments where `cnt` is greater than `m`, which never happens. We can add an extra hypothesis to account for that and we get.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res)
               (<= cnt m))
          (= (sip-multiply-loop n m cnt res)
             (+ (* n (- m cnt)) res))))
```

No counterexamples are reported, so let us see if ACL2s can prove it.

```
(defthm mult-lemma
  (implies (and (natp n) (natp m) (intp cnt) (intp res)
                (<= cnt m))
           (= (sip-multiply-loop n m cnt res)
              (+ (* n (- m cnt)) res))))
```

That worked, so we can try the main theorem, which now goes through using only equational reasoning.

```
(thm
  (implies (and (natp n) (natp m))
           (= (sip-multiply n m) (* n m))))
```

Here is another way to go about proving the main theorem. We will try to prove a lemma of the following form.

```
(implies ...
         (= (sip-multiply-loop n m cnt res)
            (* n m)))
```

The idea here is that, in addition to contract completion hypotheses, we want to identify all the invariants that hold before and after `sip-multiply-loop`, as these invariants should imply that the final result we get is `(* n m)`. Again, we can use ACL2s to help us find these invariants. We start with `(<= cnt m)` since we needed that in our first attempt.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res)
               (<= cnt m))
          (= (sip-multiply-loop n m cnt res)
             (* n m))))
```

ACL2s generates counterexamples that include negative numbers. We can rule such examples by observing that `cnt` and `res` are always non-negative.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res)
               (<= cnt m) (<= 0 cnt) (<= 0 res))
          (= (sip-multiply-loop n m cnt res)
             (* n m))))
```

ACL2s still generates counterexamples. There are counterexamples where `res` is not even a multiple of `n`. Actually, our hypotheses do not relate `res` with the input variables at all, so of course we are going to get counterexamples! What is the relationship between `res` and the other variables? The answer to the question leads us to the following conjecture.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res)
               (<= cnt m) (<= 0 cnt) (<= 0 res) (= res (* cnt n)))
          (= (sip-multiply-loop n m cnt res)
             (* n m))))
```

ACL2s did not find any counterexamples and it can prove the conjecture. In fact, we do not even need all of the hypotheses, as the following theorem shows.

```
(defthm mult-lemma
  (implies (and (natp n) (natp m) (intp cnt) (intp res)
                (<= cnt m) (= res (* cnt n)))
           (= (sip-multiply-loop n m cnt res)
              (* n m))))
```

Again, the main theorem goes through using only equational reasoning.

```
(thm
  (implies (and (natp n) (natp m))
           (= (sip-multiply n m) (* n m))))
```

So, we can reason about SIP programs. Are we done? No because the current state of affairs is not entirely satisfactory, for several reasons. One technical problem is that, as Exercise 9.3 shows, we cannot expect `sip-program` to generate `:logic` mode functions, unless we do more work to ensure we can prove termination. But, the main objection to our current approach is that a SIP programmer should not have to learn ACL2s in order to reason about SIP programs.

We need another approach, one where the reasoning is done directly on SIP programs. We use our running example to introduce the key ideas.

First, we need a mechanism by which we can express what it is that `main` is supposed to do. We will use *`Guarantee`* statements at the end of `main` to specify what `main` is supposed to do. For our example, `main` is supposed to set `res` to the product of `n` and `m`, so we have the following *`Guarantee`* statement.

`Guarantee:` ⟦ `res = n*m` ⟧

The brackets are there to make it clear that this is not program text; it is an *invariant*, a Boolean-valued statement that holds whenever program execution reaches the statement (in this case, when `main` ends).

There is a dual statement that expresses what we can assume about the inputs, beyond their types. That statement is the *`Assume`* statement. For example, if `n` and `m` were declared to be of type `int`, then we could have added the following assumption at the beginning of the program.

`Assume:` ⟦ `n >=0 & m >= 0`⟧

So, the specification of procedures can include both assumptions and guarantees. A procedure is correct if whenever it is given inputs that satisfy the types and the assumptions, then when the procedure finishes execution, the guarantees holds. Notice the similarity between this and contracts in ACL2s.

In many verification systems, the terms *`Precondition`* and *`Postcondition`* are used instead of *`Assume`* and *`Guarantee`*.

Now that we know how to specify what SIP programs are supposed to do, how do we prove that they are correct? We are going to design a system where programmers do not have to provide proofs. Instead, they only have to provide key insights. This is similar to what we did when reasoning about the ACL2s version of the code using ACL2s. Once we identified the main lemma, ACL2s took care of all the tedious equational reasoning for us. SIP programmers will not even have to write out lemmas. Instead, they will only provide *loop invariants*: invariants that hold right before the loop condition is evaluated. Suppose that the user has provided loop invariant I, then they are claiming that I holds before and after the loop, *i.e.*, that I holds whenever program execution reaches the program locations indicated below.

```
program multiply
main(n, m: nat)
{ var res, cnt: int;
  res := 0;
  cnt := 0;
  while ⟦I⟧ (cnt < m)
  { print(m, cnt, n, res);
    cnt  := cnt+1;
    res  := res+n;
    ⟦I⟧
  }
  print(m, cnt, n, res);
  return res;
  Guarantee: ⟦ res = n*m ⟧
}
```

We already saw that we can prove that the ACL2s version of the `multiply` program is correct using `mult-lemma`. The same idea works for when reasoning about the SIP program directly. Instead of requiring the user to specify types, we will infer the types from the variable declarations, so we will only require the user to set I to `cnt <= m & res = cnt * n` and the solver (ACL2s, but this is not something the programmer needs to know) can prove correctness. In order to prove correctness, we first need to show that I is a *loop invariant*, which means it satisfies the following two conditions.

1. I holds when program execution reaches the loop for the first time, and
2. Given any state where program execution has reached I and I holds, then I also holds after one iteration of the loop. There are two cases. If the loop condition is false then the loop body is not executed, so I trivially holds after the loop. Otherwise, the loop condition is true, and then I must hold after the loop body is executed. Notice that when the loop body is executed, variables get updated and program execution loops back the beginning of the loop, so I has to hold for the updated values of the variables.

Once we prove that I is a loop invariant, we use it to show the guarantee. Our proof obligation here is that whenever program execution reaches the end of the program, the guarantee holds. To get to the end of the program, we have to get past the loop, which means we satisfy I and the loop condition fails. If any such state satisfies the guarantee, then we have proven program correctness.

This process of generating proof obligations is often called *verification condition generation.* Many tools for reasoning about various programming languages operate in this way. While we will not define such a verification condition generator here, it is not hard to define such a tool using ACL2s; in fact, the `sip-program` macro is what generates all the code needed to generate verification conditions. We will show the proof obligations generated when I is `cnt <= m & res = cnt * n`.

```
; Show that the invariant holds initially.
(thm
  (let ((res 0) (cnt 0))
    (implies (and (intp cnt)
                  (intp res)
                  (natp n)
                  (natp m))
             (and (<= cnt m)
                  (= res (* cnt n))))))

; Show that the invariant is inductive
(thm
  (implies (and (and (intp cnt)
                     (intp res)
```

```
                    (natp n)
                    (natp m))
               (< cnt m)
               (and (<= cnt m) (= res (* cnt n))))
          (and (<= (+ 1 cnt) m)
               (= (+ res n) (* (+ 1 cnt) n)))))

; Show that the loop invariant implies the guarantee
(thm
  (implies (and (and (intp cnt)
                    (intp res)
                    (natp n)
                    (natp m))
               (not (< cnt m))
               (<= cnt m)
               (= res (* cnt n)))
          (= res (* n m))))
```

Exercise 9.5 *Provide a paper and pencil proof of the above proof obligations.*

9.4 SIP Exercises

Exercise 9.6 *Play the invariant discovery game* `http://invgame.atwalter.com/` *to gain experience with loop invariants.*

Exercise 9.7 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program multiply-by-1000
main(k: int)
{ var res, i: int;
  i := 0;
  res := 10;
  while ⟦I⟧  (i < 1000)
  { print(k, i, res);
    res := res + k;
    i := i + 1;
    ⟦I⟧
  }
  print(k, i, res);
  Guarantee: ⟦ res=10+k*1000 ⟧
}
```

Exercise 9.8 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program mult-of-6
main(n: nat)
```

```
{ var cnt, res: int;
  cnt := 0;
  res := 0;
  while ⟦I⟧ (cnt < n)
  { print(n, cnt, res);
    cnt := cnt + 1;
    res := res + (cnt * cnt);
   ⟦I⟧
  }
  print(n, cnt, res);
  Guarantee: ⟦ res = (n * (n + 1) * (1 + (2 * n))) / 6 ⟧
}
```

Exercise 9.9 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program summation
main(n: nat)
{ var i, sum: int;
  sum := 0;
  i   := 1;
  while ⟦I⟧ (i <= n)
  { print(n, i, sum);
    sum := sum + i;
    i   := i + 1;
    ⟦I⟧
  }
  print(n, i, sum);
  Guarantee: ⟦ sum = n*(n+1)/2 ⟧
}
```

Exercise 9.10 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program summation2
main(n: nat)
{ var sum, cnt, mys: int;
  sum := 0;
  mys := 0;
  cnt := 1;
  while ⟦I⟧ (cnt <= n)
  { print(n, mys, cnt, sum);
    sum := sum + cnt;
    mys := cnt;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print(n, mys, cnt, sum);
  Guarantee: ⟦ sum = n*(n+1)/2  ⟧
}
```

Exercise 9.11 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program mult-by-add
main (j, k: nat)
{ var i: int;
  i := 0
  while ⟦I⟧ (i < j*k)
  { print(j, k, i);
    i := i + 1;
    ⟦I⟧
  }
  print(j, k, i);
  Guarantee: ⟦ i = j*k  ⟧
}
```

Exercise 9.12 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program square-times-2
main(k: nat)
{ var res, cnt: int;
  cnt := 0;
  res := 0;
  while ⟦I⟧ (cnt < k)
  { print(k, cnt, res);
    res := res + 2*k;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print(k, cnt, res);
  Guarantee: ⟦ res=2*k ^2 ⟧
}
```

Exercise 9.13 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program square-times-const
main(k: nat, j: int)
{ var res, cnt: int;
  cnt := 0;
  res := 0;
  while ⟦I⟧ (cnt < k)
  { print(k, j, cnt, res);
    res := res + j*k;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print(k, j, cnt, res);
  Guarantee: ⟦ res=j*k ^2 ⟧
}
```

Exercise 9.14 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program cube
main(n: nat)
{ var cnt, res: int;
  cnt := 0;
  res := 0;
  while ⟦I⟧ (cnt < n)
  { print (n, cnt, res);
    res := res + (3 * ((cnt+1) ^2)) + ((cnt+1) * -3) + 1;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print (n, cnt, res);
  Guarantee: ⟦ res = n ^3 ⟧
}
```

Exercise 9.15 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program cube2
main(n: nat)
{ var cnt, res, a, b: int;
  cnt := 0;
  res := 0;
  a := 1;
  b := 6;
  while ⟦I⟧ (cnt < n)
  { print(n, cnt, a, b, res);
    cnt := cnt + 1;
    res := res + a;
    a := a + b;
    b := b + 6;
    ⟦I⟧
  }
  print(n, cnt, a, b, res);
  Guarantee: ⟦ res = n ^3 ⟧
}
```

Exercise 9.16 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program int-square-root
```

```
main(n: nat)
{ var cnt, sqr, odd: int;
  cnt := 0;
  sqr := 1;
  odd := 1;
  while ⟦I⟧ (sqr <= n)
  { print (n, sqr, odd, cnt);
    cnt := cnt+1;
    odd := odd+2;
    sqr := sqr+odd;
    ⟦I⟧
  }
  print (n, sqr, odd, cnt);
  Guarantee: ⟦ cnt ^2 <= n & n < (cnt+1) ^2 ⟧
}
```

Exercise 9.17 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee for the following SIP program.*

```
program binary-product
main(a, b: nat)
{ var dig, cnt, res: int;
  dig := a;
  cnt := b;
  res := 0;
  while ⟦I⟧ (cnt != 0)
  { print(a, b, dig, cnt, res);
    if (cnt % 2 == 1) {
      res := res + dig;
      cnt := cnt - 1;
    } else {
      dig := 2*dig;
      cnt := cnt / 2;
    }
    ⟦I⟧
  }
  print(a, b, dig, cnt, res);
  Guarantee: ⟦ res = a*b ⟧
}
```

9.5 Reasoning about C-like languages

The C programming language was developed in the early 1970's and is widely used for systems programming, *e.g.*, it is used to define operating systems, system utilities and device drivers. Security is a major consideration for all of these applications, so we will consider the semantics of C with a focus on security. After a brief introduction to C, we will explore the issues involved in formalizing and reasoning about a short C program.

Here is a simple C program, which highlights the kind of issues we run into when reasoning about C code involving arithmetic. The program assigns `x` and `y` the result of starting with the binary representation of `1` and left-shifting 10 times, which results in $2^{10} = 1024$. It then assigns `z` the sum of `x` and `y` and prints the result.

```
#include <stdlib.h>
#include <stdio.h>
int main ()
  unsigned int x = 1<<10;
  unsigned int y = 1<<10;
  unsigned int z = x + y;
  printf("%u + %u = %u\n", x, y, z);
```

C does not have a REPL, so you have to compile programs using a C compiler. Suppose that the file `add.c` contains the above program, you can invoke the C compiler from the command line with a command similar to the following.

```
cc add.c
```

This will generate an executable file `a.out` which you can run at the command line with the following command.

```
./a.out
```

You will see the following output.

```
1024 + 1024 = 2048
```

Now, let us try changing the values of `x` and `y` as follows.

```
#include <stdlib.h>
#include <stdio.h>
int main ()
  unsigned int x = 1<<31;
  unsigned int y = 1<<31;
  unsigned int z = x + y;
  printf("%u + %u = %u\n", x, y, z);
```

After compiling and running the program again, we may see the following.

```
2147483648 + 2147483648 = 0
```

This may be a surprise. Only a few languages, like ACL2s, provide native support for what is called *arbitrary precision arithmetic*, which means that arithmetic operators over integers and rationals in ACL2s corresponds to their mathematical definitions. In contrast, in most languages integers are represented using a fixed number of bits; this is called *fixed precision arithmetic*. This is true for C, Java, Fortran, C#, C++, etc. Notice there are libraries for languages that provide arbitrary precision arithmetic.

Why do language designers create languages with fixed precision arithmetic? Mostly because efficiency was prioritized over other considerations. That means that the onus is on you—the programmer—to not make a mess of it. As we will see, fixed precision arithmetic is a source of many bugs, including security bugs, and places a significant burden on programmers because reasoning about fixed precision arithmetic can be subtle and difficult.

We note that fixed precision arithmetic is but one of many reasons why reasoning about C is difficult. We focus on this issue in this chapter because our goal is to present a cautionary tale and what better way to do that than by using arithmetic, something we all have understood since elementary school. We note that a full accounting of the semantics of C is beyond the scope of this chapter.

The C language reference does not even specify how many bits should be used to represent integers, so your compiler gets to decide. If 32 bits are used, as we will assume for now, you will see the behavior above; otherwise, you will see something else.

Once we limit the number of bits that can be used to represent numbers, we wind up having to consider situations that we do not have to worry about when reasoning about languages that provide arbitrary precision arithmetic. For example, what if we add two integers whose sum is not a 32-bit number? This is called an *overflow.* The C language reference manual specifies that with unsigned numbers, when we have an overflow, we should return the 32 lower-order bits of the result, *i.e.*, we should perform modular arithmetic (mod 2^{32}). That is why the sum is reported to be `0` above. Overflow can occur with other arithmetic operators, including multiplication. In fact, notice that if we select integers randomly and uniformly, then with very high probability (over 0.9999) their product will cause an overflow.

Reasoning about arithmetic operators can be tricky. It also has to be done. For example Shamir (the "S" in "RSA") wrote a note that showed that RSA can be broken if multiplication reports the wrong answer on even one pair of numbers. RSA is what enables much of the secure e-commerce on the internet. Everything that determines how multiplication works needs to be considered here, especially the hardware, but also the software, which depends on the semantics of the programming languages used and on the correctness of the compilers used to generate machine code.

Here is an example of the issues that arise with fixed precision arithmetic. [1] The code is actually C++ code, but the same issues arise in C.

```
void* ArrayBuffer::tryAllocate(unsigned numElements, unsigned elementByteSize)
{
  void* result;
  // Do not allow 32-bit overflow of the total size
  if (numElements) {
      unsigned totalSize = numElements * elementByteSize;
      if (totalSize / numElements != elementByteSize)
          return 0;
  }

  if (WTF::tryFastCalloc(numElements, elementByteSize).getValue(result))
      return result;
  return 0;
}
```

[1] This example came from the following blog post: `http://seanhn.wordpress.com/2010/11/05/augment-your-auditing-`