Reasoning About Programs

Panagiotis Manolios
Northeastern University

February 24, 2016
Version: 94

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 and reasoning about computation, including propositional logic, recursion, induction, equational reasoning, termination analysis, 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, using 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 them produce incorrect code.

We introduce the ACL2s language. This is the language we use throughout the semester. Since ACL2s is very similar to Racket, this happens simultaneously with the programming review. During lectures, I will often point out the similarities and differences between these languages.

We introduce the semantics of the ACL2s language in a mathematical way. We show the syntax and semantics of the core language. 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. A key to helping students understand the material is to motivate and explain it by connecting it to their strong computational intuitions.

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, they are better prepared to ask for clarifications and they can better judge what notes they should take (if any).

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

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

2

A Simple Functional Programming Language

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

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

If $x_1 = y_1$ and $x_2 = y_2$ and $\cdots$ and $x_n = y_n$, then `(f` x_1 $x_2 \cdots x_n$`)` = `(f` y_1 $y_2 \cdots y_n$`)`

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

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

```
ACL2S BB !>
```

You can now type in an expression, say

```
ACL2S BB !>12
```

ACL2s reads and evaluates the expression and prints the result

```
12
```

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

```
ACL2S BB !>
```

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

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

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

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

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

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

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

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

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

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

2.1 Constants

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

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

The numeric constants include the natural numbers:

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

and the negative integers:

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

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

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

The numeric constants also include the rationals:

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

We will describe the evaluation of rationals in Section 2.3.

2.2 Booleans

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

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

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

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

Examples of `if` expressions include the following.

```
(if t nil t)
```

```
(if nil 3 4)
```

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

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

```
(if t nil)
```

```
(if 1 3 4)
```

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

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

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

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

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

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

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

ACL2s performs the following steps.

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

2. If $[\![test]\!] = \texttt{t}$, then ACL2s returns $[\![then]\!]$.

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

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

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

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

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

Here is a more complex `if` expression.

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

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

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

Therefore,

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

The next function we consider is `equal`.

$$\texttt{equal} : \mathsf{All} \times \mathsf{All} \to \mathsf{Boolean}$$

$[\![\texttt{(equal x y)}]\!]$ is `t` if $[\![x]\!] = [\![y]\!]$ and `nil` otherwise.
Notice that `equal` always evaluates to `t` or `nil`.
Here are some examples.

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

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

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

That's it for the built-in Booleans constants and functions.
Let us now define some utility functions.
We start with `booleanp`, whose signature is as follows.

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

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.*, Scheme uses "`?`" (pronounced "huh") instead of "`p`."

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

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

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

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

How do we evaluate `(booleanp 3)`?

$[\![$`(booleanp 3)`$]\!]$

= { Semantics of `booleanp` }

$[\![$`(if (equal 3 t) t (equal 3 nil))`$]\!]$

= { Semantics of `equal`, $[\![$`(equal 3 t)`$]\!]=$ `nil`, Semantics of `if` }

$[\![$`(equal 3 nil)`$]\!]$

= { Semantics of `equal`, $[\![$`(equal 3 nil)`$]\!]=$ `nil` }

`nil`

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

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

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

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

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

So, all together we have:

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

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

ic Implies oc

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

$$\texttt{t Implies (booleanp (booleanp } x\texttt{))}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

How do we evaluate the above? Simple:

$[\![\texttt{(or t nil)}]\!]$
$= \{$ Definition of `or` $\}$
$[\![\texttt{(if t t nil)}]\!]$
$= \{$ Semantics of `if` $\}$
If $[\![\texttt{t}]\!] = \texttt{nil}$ then $[\![\texttt{nil}]\!]$ else $[\![\texttt{t}]\!]$
$= \{$ 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.*

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

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

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

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

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

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

2.3 Numbers

We have the following built-in recognizers:

$$\texttt{integerp} : \mathsf{All} \to \mathsf{Boolean}$$
$$\texttt{rationalp} : \mathsf{All} \to \mathsf{Boolean}$$

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

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

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

We also have the following functions.

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

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

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

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

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

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

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

How do we express this kind of thing?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We also have built-in

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

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

$[\![\texttt{(numerator a)}]\!]$ is the numerator of the number we get after simplifying $[\![\texttt{a}]\!]$.
$[\![\texttt{(denominator a)}]\!]$ is the denominator of the number we get after simplifying $[\![\texttt{a}]\!]$.
To simplify an integer `x`, we return `x`.

To simplify a number of the form `x/y`, where `x` is an integer and `y` a natural number, we divide both `x` and `y` by the $gcd(|\texttt{x}|, \texttt{y})$ to obtain `a/b`. If $\texttt{b} = 1$, we return `a`; otherwise we return `a/b`. Note that `b` (the denominator) is always positive.

Since rational numbers can be represented in many ways, ACL2s returns the simplest representation, *e.g.*,

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

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

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

2.4 Other Atoms

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

2.5 Lists

Lists allow us to create non-atomic data.

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

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

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

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

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

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

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

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

The last built-in function is:

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

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

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

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

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

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

Here are some examples.

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

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

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

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

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

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

Here are some examples.

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

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

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

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

```
(1)
```

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

```
(list 1)
```

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

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

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

`(list` x_1 x_2 $\cdots$ x_n`)`

abbreviates (or is shorthand for)

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

2.6 Contract Violations

Consider

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

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

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

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

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

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

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

2.7 Termination

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

For example, consider the following "definition."

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

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

Let's look at another example.

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

Here are some possibilities:

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

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

2.8 Beginner Mode

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

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

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

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

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

and so on. This makes writing contracts simpler.

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

Here is how one might define the function.

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

ACL2s does not accept the definition. What's the problem? Well, if you use a function in the input contract, then ACL2s checks that it satisfies its contract! What's the contract for `>=`? That it is given rationals, but we don't know that `x` is a rational!

The point is that you should write your input contracts so that they accept anything as input.

Here's a second attempt.

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

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

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

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

Here is the definition.

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

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

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

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

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

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

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

The integer datatype can be characterized as follows.

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

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

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

This above `cond` is really a macro that expands into a nest of `if`s. In general,

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

expands into

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

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

2.9 Contracts

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

```
(defunc f (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.

2.10 Helpful Functions

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

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

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

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

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

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

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

The `len` function returns the length of a list. This is the simplest example of a data-driven definition. The idea is to define `len` using a template derived from the contract of its input variable, which is a list. A list is either empty or consists of the `first` element of the list and the `rest` of the list. Therefore our template also has two cases and in the second case, we can assume that `len` returns the correct answer on the `rest` of the list, so all that is left is to add one to the answer.

```
(defunc len (l)
  ; Returns the length of list l.
```

```
  :input-contract (listp l)
  :output-contract (natp (len l))
  (if (endp l)
      0
    (+ 1 (len (rest l)))))
```

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

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

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

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`.

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

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

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

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.

The `rev` function reverses a list.

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

2.11 Quote

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

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

2.12 Let

A `let` expression:

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

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

For example:

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

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

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

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

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

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

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

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

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

As does this further simplified expression.

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

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

2.13 Testing

Instead of

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

we can write

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

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

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

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

ACL2s also supports `thm` forms. If we use `thm` instead of `test?`, we are asking ACL2s to *prove* the property (not just test it). ACL2s will fail if it cannot find a proof (even if the

property is true). We will not use `thm` until later, when we get to theorem proving because `thm` can fail even if the property holds. Sometimes even `test?` will report that it proved the property, but all that is required for `test?` to succeed is that no counterexample is found.

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

```
:doc test?
```

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

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

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

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

`Test?` forms should be of the form

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

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

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

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

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

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

Now, consider a second `test?` form.

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

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

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

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

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

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

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

What about the following?

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

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

2.14 Data Definitions

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

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

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

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

```
(defdata one 1)
```

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

Enumerated types allow you to define finite types.

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

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

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

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

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

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

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

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

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

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

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

This defines the type consisting of lists of integers.

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

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

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

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

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

2.15 Design Recipe

Consider the following function definition.

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

The input contract, which specifies the types of the inputs, dictates that you should have at least 8 tests because for each variable, there should be as many tests as there are cases in the data definition of its type and all possible combinations of tests spanning multiple variables should be considered. That gives rise to $2 * 2 * 2 = 8$ tests. So you should have 8 tests of the form.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

If we have a more complex data definition, say:

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

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

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

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

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

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

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

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

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

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

and then create all conses of the form

$$\texttt{(cons x l)}$$

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

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

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

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

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

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

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

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

$$f(List) = List$$

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

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

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

2.16 Program Mode

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

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

```
:program
```

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

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

```
:logic
```

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

2.17 Dealing with Definition Failures

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

If this has ever happened to you, read on.

What do you do in such a situation?

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

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

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

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

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

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

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

Now you can test your code. For example:

```
(foo 10)
```

works as expected, but

```
(foo -1)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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