Logic and Computation **Lectures 1–6**
CSU 290 Spring 2009 (Pucella)

Introduction

This is a course about logic and computation, a course about the interaction between logic and computer science, a course to illustrate the usefulness of logic to computer science.

Let's start from the beginning. What is logic? Very roughly speaking, logic is the art and science of reasoning—the study of the laws of good deduction.

For the layman, logic is just what one uses when playing games such as Soduku, or those logic puzzles about 5 people working 5 different jobs living in 5 different houses driving 5 different kind of cars, and one has to determine who works which job living in which house and driving which car based on a collection of clues. For others, logic is what Sherlock Holmes uses to solve crimes—well, that, and keen observation skills.

Logic is the study of the laws that govern correct argumentation. To understand the role that logic plays for computer science, let me take a historical perspective. I find it helps put things in context. This is going to be understandably simplistic, bordering on the insulting. Feel free to investigate any of the topics and names I mention and dwelve deeper. In fact, I encourage it.

The ancient Greeks were the first to carefully think about how we reason, At least, so it is believed. They certainly were the first to write about it and influence civilization via teaching. The most common form of logic is deductive, the study of rules that govern drawing correct conclusions from some initial knowledge.

This study of logic culminated with Aristotle's study of syllogisms, capturing valid forms of reasoning.

Here are some classical syllogisms—convince yourselves that they are indeed correct arguments.

> All men are mortal
> Socrates is a man
> Therefore, Socrates is mortal.

and

> All men are mortal
> All Greeks are men
> Therefore, all Greeks are mortal.

Two things to note: first, the ancient Greek only considered men, and two, the validity of an argument does not depend on the truth of the initial knowledge. You can reason correctly from faulty data. The problem then is not the reasoning, it is the initial data.

Compare those valid arguments abot to the following faulty arguments, that have more or less the same form—can you figure out why they are not valid arguments?

> All men are intelligent
> All men are bipeds
> Therefore, all bipeds are intelligent.

or

> No dictator is benevolent
> Some kings are not dictators
> Therefore, some kinds are not benevolent

For the first one, think of birds—they are bipeds, and let us suppose that they are not intelligent in the sense we mean humans to be intelligent. What about the second one?

Aristotle explored in depth all possible forms of syllogisms, and identified those that were valid forms of reasoning, with arguments supporting their validity. It is very important to note that the validity of an argument does not depend on the meaning of the words involved. The following is a valid, if nonsensical argument:

> All borogoves are mome raths
> Mimsy the Tentacled Hamster is a borogove
> Therefore, Mimsy is a mouse rath.

Aristotle did such a good job with nailing down syllogistic reasoning that things did not evolve much beyond that until at least the Renaissance. (Part of the reason for this is that Aristotle became essentially a sacred figure throughout the middle ages, and there was no questioning him or critiquing him.) The Renaissance saw a renewed interest in trying to revisit the ancient knowledge, but it turned out to be difficult to do much better than what Aristotle did.

At least, that was the situation until 19th century, when Georges Boole, in the 1850s, showed that by basically "mathematizing" logic, you can get some leverage, and in fact study logic with the tools of mathematics, in particular algebra. He figured that reasoning can be captured in the form of an algebra (called a Boolean algebra), where there are two values, true and false, and there are operations on those values, operations such as "and", "or", "not", "implies", "is equivalent to", etc. For instance, "true and false" is equal to false, "true or false" is equal to true, and so on. Most importantly, he developed a notation for

logical formulas that divorced them from their natural language shakles, and led to being able to write syllogisms like the first one above as follows:

$$\forall x.(Man(x) \Rightarrow Mortal(x)) \wedge Man(Socrates) \Rightarrow Mortal(Socrates)$$

This doesn't seem like such a big deal—we've just replaced some English text with some mathematical symbolism that is hard to read unless you've been "initiated". But it was truly revolutionary. The move made it easier to manipulate formulas and to transform them into formulas that mean the same thing but look different, which at some level is what reasoning is all about.

Think of it this way: imagine that you had to do all your arithmetic using English along the lines of "multiplying by 5 the sum of 3 and 8 is the same as adding the product of 5 and 3 to the product of 5 8", instead of the arguably cleaner $5(3 + 8) = 5 \times 3 + 5 \times 8$. Imagine doing long simplifications on English text representing arithmetic expressions, or even worse, imagine doing calculus without symbolisms. The mind boggles.

Thus was born symbolic logic, logic as a formal language amenable to mathematical analysis. Things progresses quickly from there. Partly, this was because around that time, towards the end of the 19th century and the beginning of the 20th, mathematics itself was in kind of a crisis. People were coming up with very strange beasts, such as everywhere continuous but nowhere differentiable functions (if you remember your calculus, you should think a second at how *strange* such a function ought to look), or one dimensional curves with no width that nonetheless can fill an entire plane leaving no spot uncovered. People were worried that math was inconsistent. They tried to reconstruct it carefully, coming up with set theory as a *lingua franca* for expressing all of mathematics, and then, Russell torn down the whole house of cards by showing that the naive set theory mathematicians wanted to use as a foundation allowed for nonsensical stuff. He showed you could construct a set R defined to be $\{x \mid x \notin x\}$. The problem with this set R is the following: is $R \in R$? If so, then $R \notin R$. But if not, then $R \in R$. So no whatever what you assume, you get a contradictory statement as a conclusion. So R simply cannot be. Yet clearly I can write down a definition for it.

(Think of the following example: suppose you create a special book, with the property that it contains a list of all the books that do not contain a reference to themselves—by reference, I mean a mention of the book itself, by title, in the book. The question about such a book, call it "The Book of Impossibility", is whether "The Book of Impossibility" itself is listed in it. If so, then the "Book of Impossibility" *does* contain a reference to itself, meaning that it shouldn't be listed in the first place, but if "The Book of Impossibility" does not list itself, then it does not refer to itself, meaning that "The Book of Impossibility" should in fact contain a listing of itself. Ouch. This demonstrates that a book such as "The Book of Impossibility" in fact can never exist—it is an impossible thing.)

To make a long story short, mathematicians turned to logic (under the guise of axiomatic set theory) to carefully formalize the basis of mathematics and help avoid paradoxes such as the one above. The fact that mathematics could be formalized in logic, with its rules of reasoning, led to the question of whether one could do mathematics purely mechanically,

by translating all mathematical statements into logic and reason using the rules of logic in a purely mechanical way. Gödel, in 1931, in a result that may be one of the most important mathematical results of the last century, showed that this was not to be the case. Roughly speaking, he proved that in any logic, there are true facts about the natural numbers that cannot be proved within the logic. Trying to understand exactly what are those things that can be proved, trying to understand what it means exactly to "reason purely mechanically", which in modern terminology we would say, "algorithmically", led to the invention of computer science. So, in a sense, computer science was born out of a desire to understand mathematics from a purely logical point of view. Funny how those things go.

What does that have to do with your job as programmers, you may well ask? Good question. The answer is, not that much. It's all historical background, which is interesting, but not always relevant. No, more practically, logic is interesting to you as programmers because logic is a precise way to reason about *stuff*, and one of that *stuff* you can reason about is *programs*, which are the bread and butter of programmers.

Let's consider a concrete example, simplistic, but that illustrates the main point Suppose your boss asks you to write a sorting function (let's not worry about why she doesn't ask you to pick one from a library instead...) You go off, you write one, you come back to her. One of the first things she will ask you is: does it work, that is, does it actually sort? How do you convince her that it does, how do you argue that it does?

A nice way to do so is to first precisely give what requirements you want sorting functions to satisfy, and then check that your function meets those requirements.

There are two points. Figuring out what the requirement is often the hard part. What does it mean to sort? Suppose you have a function `sort` that you claim sorts. Intuitively, you want that no matter what input list `l` you give `sort`, the result `(sort l)` be ordered, in increasing order. But that's not enough. Indeed, if I write `sort` in such a way that it returns the list `(1 2 3)` no matter what the input is, clearly the result is sorted, but obviously that function is not very useful. Intuitively, you also want that the result of `(sort l)` be a permutation of its input `l`—exactly what was in `l` appears in `(sort l)`, just possibly in a different order. This will be expressed by the requirement:

$$\mathit{IsOrdered}(\texttt{(sort l)}) \wedge \mathit{IsPermutation}(\texttt{l}, \texttt{(sort l)})$$

Given this requirement, we want to show that a particular implementation of `sort` actually obeys this requirement. This is the logic part—we will argue formally that it does (or doesn't, as the case may be).

In fact, you have already seen requirements—contracts that you used to write on functions in 211 are just a form of requirement, pretty simple ones at that. You never had to formally argue that your functions satisfied your contracts, but had to convince yourself (and the graders) that they did. Here, we will see tools for allowing you to make sure that this is the case.

In this class, to simplify our lives, we will write our programs in ACL2, which you can think

of as a variant of Scheme. Some of the design choices in ACL2 are there not for making it a nicer language to program in, but rather a nicer language for proving things about the programs you write. ACL2 comes with its own logic, a slight variant of first-order logic, and we will see how we can use it to reason about programs meeting their requirement. We will use the ACL2 logic by hand, and eventually we may get around to using some of the proof assistance that the programming environment provides.

Review: Design Recipe for Scheme

Steps:

(1) Analyze problem and define any requisite data types

(2) State contract and purpose for function that solves the problem

(3) Give examples of function use and result

(4) Select a template for the function body

(5) Write the function itself

(6) Test it, and confirm that tests succeeded

Here is an example: suppose you want to convert pounds to kilograms, where a pound is 2.2 kilograms.

```
;; Contract:
;;   kilograms: rational -> rational
;; Purpose:
;;   convert a measure in pounds to a measure in kilograms
;; Examples:
;;   (check-expect (kilograms 0) 0)
;;   (check-expect (kilograms 2.2) 1)
;;   (check-expect (kilograms 3.3) 1.5)
;; Definition
(define (kilograms x)
  (/ x 2.2))
```

More interesting templates arise with more interesting data definitions, of course. We shall review those later.

To give you a feel for the ACL2 language, here is the function `kilograms` in ACL2:

```
;; Contract:
;;   kilograms: rational -> rational
;; Purpose:
;;   convert a measure in pounds to a measure in kilograms
;; Examples:
;;   (check= (kilograms 0) 0)
;;   (check= (kilograms 22/10) 1)
;;   (check= (kilograms 33/10) 3/2)
;; Definition
(defun kilograms (x)
  (/ x 22/10))
```

The first thing to note is that instead of `(define (kilograms x) ...)`, we write `(defun kilograms (x) ...)`. This will be confusing for all of a couple of lectures, and then you'll get used to it.

The second thing to note is that ACL2 does not have floating point numbers such as `2.2`. You must use rationals, such as `22/10`, which is of course the same as `11/5`.

The ACL2 Language

Let's start our formal description of ACL2. Actually, I will rely a lot on your prior exposure to Scheme, so I won't have to go over the notion of binding for argument, the notion of a function, the notion of syntax and the prefix notation for application, etc, etc.

To a large extent, you can understand the particulars of a language by starting with the kind of values the language lets you manipulate.

There are six basic kinds of values in ACL2: Booleans, numbers, strings, characters, symbols, and cons cells. I'll focus on Booleans and numbers here, returning to cons cells in a couple of lectures, since it's a bit more complicated. Strings, written `"hello"` and symbols, written `'foo`, are as in Scheme. Characters we won't have much use for, feel free to explore those on your own. Every kind of value has a corresponding *recognizer* that checks whether a given value is of that kind.

Booleans

First, Booleans values, true and false. The value true is written `t`, while false is `nil`. (ACL2 is case-insensitive, so `T` and `NIL` and even `NiL` work just as well.) The recognizer for Booleans is `booleanp`: `(booleanp x)` returns `t` if `x` is a Boolean value, and `nil` otherwise. Most predicates (that is, functions that return true or false) will end with a `p` in ACL2—most but not all of them.

Functions that return Booleans include `=`, that checks whether or not two values are equals, as well as `and`, `or`, `not`, that perform the expected operations on Booleans. We'll have much more to say about these later in the course.

Numbers

There are four kinds of numbers in ACL2: naturals, integers, rationals, and complex rationals.

Naturals numbers: `0`, `1`, `2`, ... They can be written in several different ways, including different bases, such as `#b11101` for binary, or `#xffa3` for hexadecimal. The recognizer for natural number is `natp`: `(natp x)` returns `t` if `x` is a natural number, and `nil` otherwise.

Integers include all the natural numbers, and their negated counterparts. Thus, `3` is an integer, and `-3` is also an integer. The recognizer for integers is `integerp`: `(integerp x)` return `t` if `x` is an integer (including natural numbers, of course), and `nil` otherwise.

Rational numbers are good old fractions: `3/4`, `-1/2`. Because every integer can be represented as a fraction, e.g, `2` is just `2/1`, rational numbers include the integers. The recognizer for rational numbers is `rationalp`: `(rationalp x)` return `t` if `x` is a rational number (including integers), and `nil` otherwise.

Complex rationals are complex numbers whose components are rational numbers. They are written `#c(3 4)`, for the complex rational $3 + 4i$. We're not going to use complex rationals in this course, so we won't have much to say about them. The recognizer for complex rationals is `complex-rationalp`, and it behaves a bit differently from the above recognizers: `(complex-rationalp x)` returns `t` if `x` is a complex rational that is *not* also a rational, and `nil` otherwise. Thus, `(complex-rationalp #c(3 4))` returns `t`, but `(complex-rationalp 10)` returns `nil`.

Note that whenever you use a number, ACL2 will internally reduce it to its simplest expression. This affects how recognizers appear to work. For instance, it is not the case that if something looks like a fraction it is necessarily not an integer. The rational number `4/2` gets simplified internally to `2`, which is of course an integer. Therefore, `(integerp 4/2)` returns `t`. In fact, `(natp 4/2)` returns `t`. Similarly for complex numbers: `(natp #c(3 0))` returns `t`.

This is especially clear for the operations `numerator` and `denominator`. Clearly, `(numerator 3/2)` yields `3`, and `(denominator 3/2)` yields `2`. Slightly more interestingly, but still not unexpectedly, `(numerator 6/4)` also yields `3`, since the rational number `6/4` gets simplified to `3/2` before `numerator` is applied to it; in the same way, `(denominator 6/4)` yields `2`. More interestingly, `(numerator 2)` yields `2`, because `2` is equivalent to the fraction `2/1`, which has `2` as a numerator; in the same way, `(denominator 2)` yields `1`, again because `2` can be written as `2/1`.

Numbers have the usual operations defined on them: `+`, `-`, `*`, `/`, modulo, `<`, `<=`, `>`, `>=`, etc. I

will let you look them up in the reference manual.

Discussing those operations brings us to what is probably the most interesting design aspect of the ACL2 language, pretty much unlike any other language you will encounter. **Functions in ACL2 are required to be total**, that is, they are required to return a result no matter what input you give them.[1] In particular, it is clear what `(+ 1 2)` gives as a result. But what should `(+ "hello" t)` give as a result?

Every operation, intuitively, has a notion of an intended domain, that is, a class of values for which it is intended to return a result that is well defined. For addition, for instance, the intended domain is numbers. What happens when one of the inputs is not in the intended domain? It is *coerced* into a value in the intended domain. Which value depends a bit on the intended domain, but there is often a reasonable defaults. For the built-in operations, when the intended domain is a number, any value which is not a number is coerced to `0`. Thus, in our example, `(+ "hello" t)` yields `0`, since `"hello"` and `t` both get coerced to `0`.

What is the intended domain for any given operation is often obvious, but not always. When in doubt, try it out, or ask. (Although my answer will often be to try it out...)

Termination

The fact that functions must be total affects how you write your own functions, of course. In particular, you have to make sure that your functions return a value no matter what their input. This is more subtle than you might think at first. Consider the following simple example, that of a factorial function. Remember factorial, written $!n$, which computes $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$? If we assume that $!0 = 1$, then here is the obvious definition of `fact` translated directly from Scheme to ACL2:

```
;; Contract:
;;   fact: natural -> natural
;; Purpose:
;;   compute the factorial of a natural number
;; Examples:
;;   (check= (fact 0) 1)
;;   (check= (fact 2) 2)
;;   (check= (fact 6) 720)
;; Definition
(defun fact (n)
  (cond ((= n 0) 1)
        (t (* n (fact (- n 1))))))
```

[1] You have to give them the right number of inputs, though. If `foo` has been defined to take two arguments and you try to call `(foo 1)`, the system will complain loudly with a syntax error.

(Note that the `cond` forms does not take an `else` as the last condition as it does in Scheme, but takes a `t`, which is of course the condition that is always true.) This looks all nice and well, but does it return a value for every value you give it? What happens at `(fact -1)`? Well, if you try it out on paper, you see that the function never terminates—it keeps on trying to compute the factorial of more and more negative numbers. Now, a function that never terminates on an input certainly is not returning a value for that input—therefore, the function is not total. ACL2 will reject those functions, will refuse to even consider them.

So how do you fix that. There are a few ways of doing so (testing for `<= 0` instead of `= 0`, testing to make sure the input is a `natp`, and so on), but we will advocate a particular way of fixing the problem in this course. The motivation is two-fold: first, we don't want to change the way you would write the functions in Scheme in the first place; two, we want to keep the fix as simple as possible, because later on when we try to prove things about our functions, we will benefit greatly from having the functions be particularly simple.

So our fix is to use the same approach that the built-in operations: coerce the result to the intended domain. Here, the intended domain is captured by the contract you wrote for the function—the intended domain is that of the natural numbers. The idea, then, is to coerce any input which is not a natural number to be a natural number, and something reasonable, like `0`, otherwise.

To help you, ACL2 defines a couple of coercion functions:

- `(nfix x)` returns `x` if `x` is a natural number and `0` otherwise;
- `(ifix x)` returns `x` if `x` is an integer and `0` otherwise;
- `(rfix x)` returns `x` if `x` is a rational number and `0` otherwise.

Thus, the condition in our `fact` function should really be `(= (nfix n) 0)`, so that the condition is true for any non-natural-number value, at which point the function terminates immediately with `1`.

Because `(= (nfix x) 0)` and `(= (ifix x) 0)` are so common, the system defines the functions `(zp x)` to be just `(= (nfix x) 0)` and `(zip x)` to be just `(= (ifix x) 0)`. Thus, the function `fact` we advocate should look like this:

```
;; Contract:
;;   fact: natural -> natural
;; Purpose:
;;   compute the factorial of a natural number
;; Examples:
;;   (check= (fact 0) 1)
;;   (check= (fact 2) 2)
;;   (check= (fact 6) 720)
;;   (check= (fact -1) ??)
```

```
;; Definition
(defun fact (n)
  (cond ((zp 0) 1)
        (t (* n (fact (- n 1))))))
```

Note how small a change it is from the direct version—we only slightly changed the condition on the base case of the recursion. Also, note that we will ask you to **put in a test case or example for a value outside the intended domain of the function**, so that you are reminded to check and make sure your function returns a value for those inputs too. (What value is returned is usually completely unimportant, hence the `??` in the above.)

This is in fact our first function template, modified for ACL2, for recursion over the natural numbers:

```
(defun nat-recursion-template (n)
  (cond ((zp n) ...)
        (t ... (nat-recursion-template (-n 1)) ...)))
```

You may wonder why we care about functions being total. The reason is purely and simply to help us prove facts about functions, as we will see later in this course. Every programming language designer must make choices about what the programming language allows. Often, those design decisions are based on what the programming language is for. Often, programming languages are intended for writing large-scale programs, in which case the programming language will provide good features for modular programming, perhaps at the cost of other features. Programming languages for writing web-based scripts will have primitive for helping writing web-based scripts, again at the cost of other features. ACL2 is intended as a programming language for writing programs that can be reasoned about, and therefore design decisions such as forcing every function to be total are made, at the cost of other features, such as the ability to catch wrong inputs and cause errors.

True Lists

The kinds of values we saw above, Booleans, numbers, strings, symbols, charaacters, are the so-called *atomic data types*. The recognizer `atom` returns `t` exactly when its input is a value of one of those types.

As you'll recall from 211, there are types of data that are not atomic. You saw structures in 211, which we will not see here, and you also saw lists and other recursive kinds of data. Let us look at lists. In ACL2, what you know as lists are called *true lists*—we will see why in a couple of lecture when we look at things are list-like but not true lists, and they have a data definition which is very similar to the one you saw in 211.

Let us start with true lists of integers. A true list of integers, or `loi`, is one of:

- `nil`, or
- `(cons i l)` where `i` is an integer and `l` is a `loi`.

Note that we use `nil` for the empty list, as well as for the Boolean value false. This is somewhat unfortunate, but is due to historical reasons. (Most of the design choices for ACL2 as a programming language is due to such historical reasons.)

Thus, `nil` is a true list of integers, `(cons 1 nil)` is a true list of integers, `(cons 1 (cons 2 nil))` is a true list of integers, and so on. Like in Scheme, you can use the notation `'(1 2 3)` to write lists, as well as the `list` function, so that the list containing `1`, `2`, and `3` can be written `(list 1 2 3)`, which is just an abbreviation for `(cons 1 (cons 2 (cons 3 nil)))`.

Functions over lists are generally recursive functions, with a design template that follows from the data definition above. Here is the design template we will be using for true lists of any kind:

```
(defun true-list-template (l)
  (cond ((endp l) ...)
        (t ... (car l) ... (true-list-template (cdr l)) ...)))
```

Note that `car` and `cdr` are used to access the first element of list `l` and the rest of list. You can also use `first` and `rest`, like you did in Scheme.

The important thing to note is that `endp` is used in the base case, instead of `empty?`. Intuitively, `endp` is true if its input is not a cons. Meaning, in particular, if you have `nil`, indicating the end of the list, and also any non-list value, such as an integer. Very roughly, this use of `endp` implicitly does a coercion of anything which is not a list into `nil`, and otherwise checks if a list is `nil`. (I am somewhat lying here, but for the time being this is not a bad mental model to have. More on this later.)

The most general kind of true list is just that, true lists. A *true list* is one of:

- `nil`, or
- `(cons x l)` where `x` is anything and and `l` is a true list.

True lists can containing anything, including other true lists, as elements.

Let's look at some examples. First, computing the length of a list.

```
;; Contract:
;;   length: true-list -> natural
;; Purpose:
;;   count the number of items in a true list
```

```
;; Examples:
;;   (check= (length nil) 0)
;;   (check= (length '(1 2)) 2)
;;   (check= (length '(1 2 3 4)) 4)
;;   (check= (length 10) ??)
;; Definition
(defun length (l)
  (cond ((endp l) 0)
        (t (+ 1 (length (cdr l))))))
```

You should check that the examples actually work as advertised. Since `10` is not a true list, `(endp 10)` returns `t`, meaning that `(length 10)` returns `0`.

More interesting, appending two lists together, which requires a version of the design template with more than one argument. Here, you have to decide which argument drives the function—this depends on the function, and it is not always clear until you think about it some. For append, the first argument can be seen as driving the recursion. As a general rule, in the recursive call, the arguments that are not driving the recursion are generally unchanged. If you find yourself needing to change those arguments, pause and think carefully. (One of the only times when you will be changing those arguments during a recursive call is when you write accumulator-based functions.)

```
;; Contract:
;;   app : true-list true-list -> true-list
;; Purpose:
;;   append two true lists together
;; Examples:
;;   (check= (app nil nil) nil)
;;   (check= (app '(1 2) nil) '(1 2))
;;   (check= (app nil '(1 2)) '(1 2))
;;   (check= (app '(1 2) '(3 4)) '(1 2 3 4))
;;   (check= (app 3 4) ??)
;; Definition
(defun app (x y)
  (cond ((endp x) y)
        (t (cons (car x) (app (cdr x) y)))))
```

Here is another function, to reverse a list, that we can write using `app`—although you and I know there is a much better way to write a reverse function, which we will see in a little bit.

```
;; Contract:
;;   rev : true-list ->  true-list
;; Purpose:
```

```
;;   reverse a true list
;; Examples:
;;   (check= (rev nil) nil)
;;   (check= (rev '(1)) '(1))
;;   (check= (rev '(1 2 3)) '(3 2 1))
;;   (check= (rev 3) ??)
;; Definition
(defun rev (l)
  (cond ((endp l) nil)
        (t (app (rev (cdr l)) (list (car l))))))
```

What about recognizers for true lists? It is easy to write one, following the data definition.

```
;; Contract:
;;   true-listp: anything -> Boolean
;; Purpose:
;;   check if a value is a true list
;; Examples:
;;   (check= (true-listp nil) t)
;;   (check= (true-listp '(1 2 3)) t)
;;   (check= (true-listp 20) nil)
;; Definition
(defun true-listp (l)
  (cond ((endp l) (= l nil))
        (t (true-listp (cdr l)))))
```

Note the base case of the recursion, and make you understand what it is saying.

All of the above are straight applications of the design template for true lists. Learn it, and use it.

Here is an interesting example: `list-ref`, which returns the nth element of a list. There are two arguments, a natural number and a true list, and you have to chose which drives the recursion. It turns out, in this case, that either choice works. Let's do the one where the true list argument drives the recursion.

```
;; Contract:
;;   list-ref : true-list natural ->  anything
;; Purpose:
;;   (list-ref l n) returns the (n+1)th element of the nonempty list l
;; Examples:
;;   (check= (list-ref '(1 2 3) 0) 1)
;;   (check= (list-ref '(1 2 3) 1) 2)
;;   (check= (list-ref '(1 2 3) 2) 3)
```

```
;;   (check= (list-ref '(1 2 3) 10) ??)
;;   (check= (list-ref nil 3) ??)
;; Definition
(defun list-ref (l n)
  (cond ((endp l) nil)
        (t (if (= n 0) (car l) (list-ref (cdr l) (- n 1))))))
```

This is one of those functions that requires you to change the non-driving argument in the recursive call. As I said, everytime you do that, make sure you understand what you are doing.

Exercise 1 *Write another version of* ***`list-ref`*** *where the natural number argument drives the recursion. Note that you should use the design template for recusion over the natural numbers that we saw earlier.*

To finish the topic of recursion over lists, we recall the notion of writing recursive functions over accumulators. The idea is to use an auxiliary function that has an extra argument into which we build a partial result. When we are done, the partial result will be the final result. As you know, accumulator-based functions can sometimes be easier to come up with than non-accumulator-based functions. Also, as in the case of factorial or reverse, they can be faster to execute.

In ACL2, there are no nested functions (no `local` definitions), therefore, our auxiliary function will be a function like any other. The auxliary function should use the design template for whatever kind of recursion is called for, though.

Without further ado, here is the accumulator-based version of factorial:

```
;; Contract:
;;   fact-acc: natural -> natural
;; Purpose:
;;   compute the factorial of a natural number using an accumulator
;; Examples:
;;   (check= (fact-acc 0) 1)
;;   (check= (fact-acc 2) 2)
;;   (check= (fact-acc 6) 720)
;;   (check= (fact-acc -1) ??)
;; Definition
(defun fact-aux (n acc)
  (cond ((zp n) acc)
        (t (fact-aux (- n 1) (* n acc)))))

(defun fact-acc (n)
  (fact-aux n 1))
```

Note the use of the design template for recursion over the natural numbers in auxiliary function `fact-aux`.

Here is the accumulator-based version of reverse:

```
;; Contract:
;;   rev-acc : true-list ->  true-list
;; Purpose:
;;   reverse a true list using an accumulator
;; Examples:
;;   (check= (rev-acc nil) nil)
;;   (check= (rev-acc '(1)) '(1))
;;   (check= (rev-acc '(1 2 3)) '(3 2 1))
;;   (check= (rev-acc 3) ??)
;; Definition
(defun rev-aux (l acc)
  (cond ((endp l) acc)
        (t (rev-aux (cdr l) (cons (car l) acc)))))
(defun rev (l)
  (rev-aux l nil))
```

I will refer you to your 211 notes for hints and tips on how to design accumulator-based functions in the first place. Everything that I would have to say about it would be redundant here anyways.