Recursion and Induction

J Strother Moore
Department of Computer Sciences
University of Texas at Austin
Austin, Tx 78712
moore@cs.utexas.edu (512) 471-9590

May, 2008 (updated, Jan, 2015)

1 Abstract

This paper introduces a formal mathematical theory that emphasizes recursive definition and inductive proof. Then it leads you through a series of exercises designed to help you learn how to prove theorems in such a setting. The exercises also stress how to define functions recursively in a way that makes their properties analyzable, how to state inductively provable theorems, how to decompose theorems into simpler lemmas, and how to simplify terms using previously proved lemmas. Answers to all the exercises are available in the companion document "Recursion and Induction: Answer Key." We urge you not to refer to the answer key except as a last resort.

2 Introduction

The language we will use is a subset of Lisp called ACL2. ACL2, which stands for "A Computational Logic for Applicative Common Lisp," is both a functional programming language based on Common Lisp and a first-order mathematical theory with induction. There is a mechanical theorem prover for ACL2, but we do not discuss it here. The best way to learn to use that theorem prover is first to master the art of recursive definition and inductive proof.

This paper uses a tiny fragment of full ACL2. We describe just enough formal machinery to study recursion and induction in an interesting context. See [3] for a precise description of the logic. See [2] for a description of how to use the mechanical theorem prover. See [1] for some interesting applications. There are also a number of resources on the web. See [4] for the ACL2 home page, including access to the source code, installation instructions, and hypertext documentation. Under the link "Books and Papers" on the home page, see "About ACL2" for a quick summary of some applications and an introduction to using the system. However, the best way to learn to use the system is to master the material here first!

3 Data Types

ACL2 provides five data types: numbers, characters, strings, symbols, and ordered pairs.

3.1 Numbers

The only numbers we will use are the integers, written in the usual way, e.g., `-3`, `0`, and `123`. ACL2 allows integers to be written in other ways, e.g., `00123`, `+123`, `246/2`, `#b1111011`, `#o173` and `#x7B` are all ways to write `123`. However, we will always write them in conventional decimal notation.

ACL2 also supports rationals, e.g., `1/3` and `22/7`, and complex rationals, `#c(5 2)` which is more commonly written $5 + 2i$. Lisp, but not ACL2, supports floating point numbers, e.g., `3.1415` and `31415E-4`.

3.2 Characters

We will not use character objects in this document. In case you come across such an object in your exploration of ACL2, some characters are `#\a`, `#\A`, `#\Newline` and `#\Space`. It is actually possible in ACL2 to "construct" and "deconstruct" characters in terms of naturals. For example, one can construct `#\A` from `65`, the ASCII code for uppercase 'A'.

3.3 Strings

Strings in our subset of ACL2 are written as sequences of ASCII characters between successive "string quotes." For example, here is a string `"Hello, World!"`. For the purposes of this document, we will treat strings as atomic objects, though it is actually possible to construct and deconstruct them in terms of lists of character objects.

3.4 Symbols

Unlike many languages, Lisp provides symbols as primitive data objects. Some example symbols are `t`, `nil`, `LOAD`, `STORE`, `ICONST_0`, `prime-factors`, `++`, and `file77`. For the purposes of this document, we will treat symbols as atomic objects, though it is actually possible to construct and deconstruct them in terms of strings.

For the purposes of this document, a symbol is a sequence of alphabetic characters, digits, and/or certain signs (specifically, `+`, `-`, `*`, `/`, `=`, `<`, `>`, `?`, `!`, `$`, `&`, and `_` (underscore)) that cannot be read as a number. Case is unimportant. Symbols are parsed to have the greatest length possible under the rules above. Thus, `xy` is one symbol, not two symbols (`x` and `y`) written without intervening whitespace!

Note that `t` and `T` are different ways to write the same symbol, as are `nil`, `Nil`, and `NIL`. `T` and `nil` are called the *Boolean symbols*. `T` is frequently used to denote "true" and `nil` is used to denote "false." For reasons that will become apparent, `nil` is also used as the "empty list."

3.5 Pairs

Pairs are written in Lisp's "dot notation." Instead of conventional Cartesian notation, e.g., $\langle 1, 2\rangle$, Lisp replaces the angle brackets with parentheses and the comma with a dot, `(1 . 2)`. Thus, `((1 . 2) . (3 . 4))` is the pair containing the pair `(1 . 2)` in its left component and the pair `(3 . 4)` in its right. In high school you might have written this object as $\langle\langle 1, 2\rangle, \langle 3, 4\rangle\rangle$.

In Lisp, pairs are called *conses*. Non-conses are called *atoms*. The left component is called the *car* and the right component is called the *cdr* (pronounced "cudder").[1] Lisp provides three conventions for writing parenthesized constants.

- `Nil` can be written `()`.
- A pair of the form `(`*x* `. nil)` may be written `(`*x*`)`.
- A pair of the form `(`*x* `. (`*y*...`))` may be written `(`*x y*...`)`.

Thus, the cons `(1 . (2 . (3 . nil)))` may be written `(1 2 3)`. This suggests the most common use of conses: to represent linked lists or sequences. The special role of `nil` in these conventions is the only sense in which `nil` is "the empty list."

Any object can be treated as a list! If a cons is being treated as a list, then its car is the first element and its cdr is treated as a list of the remaining elements. If an atom is treated as a list, it is treated as the empty list. `Nil` is just the most common atom used in this way.

Thus, the elements of `(1 . (2 . (3 . nil)))` are `1`, `2`, and `3`, respectively. The length of this list is three. If `((1 . 2) . (3 . 4))` is treated as a list, its elements are `(1 . 2)` and `3`; its length is two. The `4` is just the terminal atom. `((1 . 2) . (3 . nil))` and `((1 . 2) . (3 . 4))` are different objects, but when treated as lists they have the same two elements.

Here is a list of the symbols naming the summer months: `(June July August)`. It could also be written `(JUNE . (JULY . (AUGUST . NIL)))` and many other ways.

[1] The names come from the original implementation of Lisp on the IBM 704. That machine had a 36-bit word that was logically divided into two parts, the "address" and the "decrement." Lisp used such words to represent a `cons` cell. The address part pointed to the left component and the decrement part pointed to the right component. Thus, the operations were `car` ("contents of address of register") and `cdr` ("contents of decrement of register").

3.6 Identity

Since a pair can typically be written down in several different ways, you might ask how can you tell whether one display is equal to another? For example, how can you determine that `(1 . (2 . (3 . nil)))` is the same pair as `(1 2 3)`, which is also the same pair as `(1 . (2 3))`?

One way is to write each constant in a canonical form. If their canonical forms are different, the two constants are different. For integers, the standard canonical form is to use base 10 and to drop leading `0`s and "+" signs. For symbols, it is to write everything in upper case. For conses, the canonical form is to eschew the use of the three conventions noted above and to use dot notation exclusively.

3.7 Exercises

Problem 1.
Each of the utterances below is supposed to be a single object. Say whether it is a number, string, symbol, pair, or ill-formed (i.e., does not represent a single object in our language).

1. `Monday`
2. π
3. `HelloWorld!`
4. `--1`
5. `-1`
6. `*PI*`
7. `31415x10**-4`
8. `(A . B . C)`
9. `Hello World!`
10. `if`
11. `invokevirtual`
12. `((1) . (2))`
13. `<=`
14. `((A . 1) (B . 2) (C . 3))`
15. `Hello_World!`
16. `+`
17. `lo-part`
18. `31415926535897932384626433832795028841971693993751058209749445923`
19. `(1 . (2 . 3))`
20. `(1 . 2 3)`
21. `"Hello World!"`
22. `((1) (2) . 3)`
23. `()`

Problem 2.
Group the constants below into equivalence classes.

1. `(1 . (2 3))`
2. `(nil . (nil nil))`
3. `((nil nil) . nil)`
4. `(1 (2 . 3) 4)`
5. `(nil nil)`
6. `(1 (2 . 3) . (4 . ()))`
7. `(HelloWorld !)`
8. `(1 (2 3 . ()) 4)`
9. `((A . t) (B . nil)(C . nil))`
10. `(()())`
11. `(1 2 3)`
12. `(() () . nil)`
13. `(A B C)`
14. `(a . (b . (c)))`
15. `(HELLO WORLD !)`
16. `((a . t) (b) . ((c)))`

4 Terms

For the purposes of this document, a *term* is a variable symbol, a quoted constant, or a function application written as a sequence, enclosed in parenthesis, consisting of a function symbol of arity n followed by n terms.

If `car` is a function symbol of arity one and `cons` is a function symbol of arity two, then `(cons (car x) y)` is a term. In more conventional notation this term would be written $cons(car(x), y)$. We call `(car x)` and `y` the *actual expressions* or *actuals* of the *function call* `(cons (car x) y)`.

Semantically, terms are interpreted with respect to an assignment binding variable symbols to constants and an interpretation of function symbols as mathematical functions. For example, suppose the variables `x` and `y` are bound, respectively, to the constants `1` and `(2 3 4)`, and suppose the function symbol `cons` is interpreted as the function that constructs ordered pairs (as it always is). Then the meaning or *value* of the term `(cons x y)` is `(1 . (2 3 4))` or, equivalently, `(1 2 3 4)`. That is, the value of a variable is determined by the variable assignment; the value of a quoted constant is that constant; and the value of a function application, `(`$f\ \ a_1 \ldots a_n$`)`, is the result of applying the mathematical function assigned to f to the values of the actuals, a_i.

ACL2 provides an infinite number of *variable symbols*, whose syntax is that of symbols. Some example variable symbols are `x`, `a1`, and `temp`. The symbols `t` and `nil` are not legal variable symbols.

A *quoted constant* is written by prefixing an integer, string, or symbol by a single quote mark. For example, `'t`, `'nil`, `'-3`, `'"Hello World!"` and `'LOAD` are quoted constants. Note that we do not consider `'(1 2 3)` a quoted constant. This is a mere technicality. We will shortly introduce some abbreviations that allow us to write `'(1 2 3)` as an abbreviation for `(cons '1 (cons '2 (cons '3 'nil)))`.

ACL2 also has an infinite number of *function symbols* each of which has an associated *arity* or number of arguments. For the moment we will concern ourselves with six primitive function symbols, `cons`, `car`, `cdr`, `consp`, `if`, and `equal` described below. Note that we implicitly specify the arity of each primitive function symbol.

(`cons` x y) - construct and return the ordered pair (x . y).

(`car` x) - return the left component of x, if x is a pair; otherwise, return `nil`.

(`cdr` x) - return the right component of x, if x is a pair; otherwise, return `nil`.

(`consp` x) - return `t` if x is a pair; otherwise return `nil`.

(`if` x y z) - return z if x is `nil`; otherwise return y.

(`equal` x y) - return `t` if x and y are identical; otherwise return `nil`.

With these primitives we cannot do anything interesting with numbers, strings, and symbols. They are just tokens to put into or take out of pairs. But we can explore most of the interesting issues in recursion and induction in this setting!

Problem 3.
Which of the utterances below are terms?

1. `(car (cdr x))`
2. `(cons (car x y) z)`
3. `(cons 1 2)`
4. `(cons '1 '2)`
5. `(cons one two)`
6. `(cons 'one 'two)`
7. `(equal '1 (car (cons '2 '3)))`
8. `(if t 1 2)`
9. `(if 't '1 '2)`
10. `(car (cons (cdr hi-part) (car lo-part)))`
11. `car(cons x y)`
12. `car(cons(x,y))`
13. `(cons 1 (2 3 4))`

Problem 4.
For each constant below, write a term whose value is the constant.

1. `((1 . 2) . (3 . 4))`
2. `(1 2 3)`
3. `((1 . t) (2 . nil) (3 . t))`
4. `((A . 1) (B . 2))`

Problem 5.
For each term below, write the constant to which it evaluates.

1. `(cons (cons '1 '2) (cons (cons '3 '4) 'nil))`

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

3. `(cons 'nil (cons (cons 'nil 'nil) 'nil))`

4. `(if 'nil '1 '2)`

5. `(if '1 '2 '3)`

6. `(equal 'nil (cons 'nil 'nil))`

7. `(equal 'Hello 'HELLO)`

8. `(equal (cons '1 '2) (cons '1 'two))`

5 Substitutions

A *substitution* is a set $\{v_0 \leftarrow t_0\texttt{,}\ v_1 \leftarrow t_1\texttt{,}\ \ldots\}$ where each v_i is a distinct variable symbol and each t_i is a term. If, for a given substitution σ and variable v, there is an i such that v is v_i, we say v *is bound* by σ. If v is bound by σ, then the *binding* of v in σ is t_i, for the i such that v_i is v. (In set theory, a substitution is just a finite function that maps the v_i to their respective t_i.)

The result of *applying* a substitution σ to a term *term* is denoted $term/\sigma$ and is defined as follows. If `term` is a variable, then $term/\sigma$ is either the binding of *term* in σ or is *term* itself, depending on whether *term* is bound in σ. If *term* is a quoted constant, $term/\sigma$ is *term*. Otherwise, *term* is `(`$f\ \ a_1 \ldots a_n$`)` and $term/\sigma$ is `(`f $a_1/\sigma \ldots a_n/\sigma$`)`.

Problem 6.
Suppose σ is

$\{$`x` $\leftarrow$ `(car a),`
 `y` $\leftarrow$ `(cdr x)`$\}$

What term is `(car (cons x (cons y (cons '"Hello" z))))`$/\sigma$?

6 Abbreviations for Terms

If x is `t`, `nil`, an integer, or a string, and x is used where a term is expected, then x abbreviates the quoted constant `'`x.

If `'(`x `.` y`)` is used where a term is expected, it abbreviates `(cons '`x `'`y`)`.

If `''`x is used where a term is expected, it abbreviates `'(QUOTE` x`)`.

When `(list` $x_1 \ldots$`)` is used as a term, it abbreviates `(cons` x_1 `(list` $\ldots$`))`. When `(list)` is used as a term, it abbreviates `nil`. Thus `(list a b c)` abbreviates `(cons a (cons b (cons c nil)))`.

`And` and `or` will be defined as function symbols of two arguments. But if `and` is used as though it were a function symbol of more than two arguments, then it abbreviates the corresponding right-associated nest of `and`s. Thus, `(and p q r s)`, when used where a term is expected, abbreviates `(and p (and q (and r s)))`.

If `or` is used as though it were a function symbol of more than two arguments, then it abbreviates the corresponding right-associated nest of `or`s.

Problem 7.
Show the term abbreviated by each of the following:

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

2. `(equal "Hello" hello)`

3. (and (or a1 a2 a3) (or b1 b2 b3) (or c1 c2 c3))
4. (equal x '(or a1 a2 a3))
5. (cons cons '(cons cons 'cons))

The art of displaying a Lisp term in a way that it can be easily read by a person is called "pretty printing." We recommend the following heuristics. First, write clearly and count your parentheses. Second, try never to write a single line with more than about 30 non-blank characters on it. Third, if a function call will not fit on a line, break it into multiple lines, indenting each argument the same amount.

Below we show one term pretty printed with successively narrower margins. (Note that the display is a term only if `app` is a function symbol of arity two.) We find the second and third lines of the first display below excessively long. Each display shows exactly the same term. Note how we break terms to indent arguments the same amount and how we sometimes slide the arguments under the function symbol to save horizontal space. Personally, we find the second display below (the one labeled "`width 46`") the easiest to read. We show the others merely to illustrate how more space can be saved when one finds oneself close to the right margin.

```
|<---------------------------- width 70 ---------------------------->|
(IMPLIES (AND (CONSP A)
              (EQUAL (APP (APP (CDR A) B) C) (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C) (APP A (APP B C))))


|<---------------- width 46 ---------------->|
(IMPLIES (AND (CONSP A)
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C))))


|<------------ width 38 ------------>|
(IMPLIES
 (AND (CONSP A)
      (EQUAL (APP (APP (CDR A) B) C)
             (APP (CDR A) (APP B C))))
 (EQUAL (APP (APP A B) C)
        (APP A (APP B C))))


|<---------- width 34 ---------->|
(IMPLIES
 (AND
  (CONSP A)
  (EQUAL (APP (APP (CDR A) B) C)
         (APP (CDR A) (APP B C))))
 (EQUAL (APP (APP A B) C)
        (APP A (APP B C))))


|<------- width 28 ------->|
(IMPLIES
 (AND
  (CONSP A)
```

```
  (EQUAL
   (APP (APP (CDR A) B) C)
   (APP (CDR A) (APP B C))))
 (EQUAL
  (APP (APP A B) C)
  (APP A (APP B C))))
```

```
|<---- width 23 ----->|
(IMPLIES
 (AND
  (CONSP A)
  (EQUAL
   (APP (APP (CDR A) B)
        C)
   (APP (CDR A)
        (APP B C))))
 (EQUAL
  (APP (APP A B) C)
  (APP A (APP B C))))
```

7 Function Definitions

To define a function, we use the form `(defun` f $(v_1 \ldots v_n)$ β`)` where f is the function symbol being defined, the v_i are the formal variables or simply *formals*, and β is the body of the function.

Operationally, a definition means that to compute `(`f $a_1 \ldots a_n$`)` one can evaluate the actuals, a_i, bind the formals, v_i to those values, and compute β instead. Logically speaking, a definition adds the axiom that `(`f $v_1 \ldots v_n$`)` is equal to β.

Here are the Lisp definitions of the standard propositional logic connectives:

```
(defun not (p) (if p nil t))
(defun and (p q) (if p q nil))
(defun or (p q) (if p p q))
(defun implies (p q) (if p (if q t nil) t))
(defun iff (p q) (and (implies p q) (implies q p)))
```

Note that in Lisp, `and` and `or` are not Boolean valued. E.g., `(and t 3)` and `(or nil 3)` both return `3`. This is unimportant if they are only used propositionally, e.g., `(and t 3)` $\leftrightarrow$ `(and 3 t)` $\leftrightarrow$ `t`, if "$\leftrightarrow$" means `iff`. In Lisp, any non-nil value is propositionally equivalent to `t`.

Here is a recursive definition that copies a `cons`-structure.

```
(defun tree-copy (x)
  (if (consp x)
      (cons (tree-copy (car x))
            (tree-copy (cdr x)))
     x))
```

For example, the term `(tree-copy '((1 . 2) . 3))` has the value `((1 . 2) . 3)`.

In the exercises below you may wish to define auxiliary ("helper") functions as part of your solutions.

Problem 8.

Define `app` to concatenate two lists. For example `(app '(1 2 3) '(4 5 6))` is `(1 2 3 4 5 6)`.

Problem 9.
Define `rev` to reverse a list. For example, `(rev '(1 2 3))` is `(3 2 1)`.

Problem 10.
Define `mapnil` to "copy" a list, replacing each element by `nil`. Thus, `(mapnil '(1 2 3))` is `(nil nil nil)`.

Problem 11.
The result of "swapping" the pair `(`x `.` y`)` is the pair `(`y `.` x`)`. Define `swap-tree` to swap every cons in the binary tree `x`. Thus, `(swap-tree '((1 . 2) . (3 . 4)))` is `((4 . 3) . (2 . 1))`.

Problem 12.
Define `mem` to take two arguments and determine if the first one occurs as an element of the second. Thus, `(mem '2 '(1 2 3))` is `t` and `(mem '4 '(1 2 3))` is `nil`.

Problem 13.
Define the list analogue of subset, i.e., `(sub` x y`)` returns `t` or `nil` according to whether every element of x is an element of y.

Problem 14.
Define `int` to take two lists and to return the list of elements that appear in both. Thus `(int '(1 2 3 4) '(2 4 6))` is `(2 4)`.

Problem 15.
Define `(tip e x)` to determine whether `e` occurs as a tip of the binary tree `x`.

Problem 16.
Define `(flatten x)` to make list containing the tips of the binary tree `x`. Thus, `(flatten '((1 . 2) . (3 . 4)))` is `(1 2 3 4)`.

Problem 17.
Define `evenlen` to recognize lists of even length. Thus, `(evenlen '(1 2 3))` is `nil` and `(evenlen '(1 2 3 4))` is `t`.

8 Axioms

A formal mathematical theory is given by a formal syntax for "formulas," a set of formulas designated as "axioms," and some formula manipulation rules that allow one to derive "new" formulas from "old" ones. A "proof" of a formula p is a derivation of p from the given axioms using the given rules of inference. If a formula can be proved, it is said to be a "theorem." Formulas are given "semantics" similar to those described for terms. Given an "assignment" of values to variable symbols and an interpretation of the function symbols, every formula is given a truthvalue by the semantics. Given an interpretation, a formula is "valid" if it is given the value true under every possible assignment to the variable symbols. A "model" of a theory is an interpretation that makes all the axioms valid. Provided the rules of inference are validity preserving, every theorem is valid, i.e., "always true."

We assume you know all that, and won't go into it further. The whole point of a practical formal theory is to use proof to determine truth: one way to determine if a formula is true is to prove it.

If α and β are terms, then $\alpha = \beta$ is a *formula*. If p and q are formulas, then each of the following is a formula:

- $p \rightarrow q$
- $p \wedge q$
- $p \vee q$
- $\neg p$
- $p \leftrightarrow q$.

If α and β are terms, then $\alpha \neq \beta$ is just an abbreviation for the formula $\neg(\alpha = \beta)$.

We extend the notation $term/\sigma$ in the obvious way so that we can apply substitution σ to formulas, replacing all the variables bound by σ in all the terms of the formula.

The axioms we will use for the initial part of our study are given below. Note that "Axioms" 1 and 8 are actually axiom schemas, i.e., they describe an infinite number of axioms.

Axiom	**1.**	`'`$\alpha \neq$ `'`β, where α and β are distinct integers, strings, or symbols
Axiom	**2.**	`x` $\neq$ `nil` $\rightarrow$ `(if x y z)` $=$ `y`
Axiom	**3.**	`x` $=$ `nil` $\rightarrow$ `(if x y z)` $=$ `z`
Axiom	**4.**	`(equal x y)` $=$ `nil` $\vee$ `(equal x y)` $=$ `t`
Axiom	**5.**	`x` $=$ `y` $\leftrightarrow$ `(equal x y)` $=$ `t`
Axiom	**6.**	`(consp x)` $=$ `nil` $\vee$ `(consp x)` $=$ `t`
Axiom	**7.**	`(consp (cons x y))` $=$ `t`
Axiom	**8.**	`(consp '`α`)` $=$ `nil`, where α is an integer, string, or symbol
Axiom	**9.**	`(car (cons x y))` $=$ `x`
Axiom	**10.**	`(cdr (cons x y))` $=$ `y`
Axiom	**11.**	`(consp x)` $=$ `t` $\rightarrow$ `(cons (car x) (cdr x))` $=$ `x`
Axiom	**12.**	`(consp x)` $=$ `nil` $\rightarrow$ `(car x)` $=$ `nil`
Axiom	**13.**	`(consp x)` $=$ `nil` $\rightarrow$ `(cdr x)` $=$ `nil`

One axiom described by Axiom (schema) 1 is `'t` $\neq$ `'nil`. Others are `'nil` $\neq$ `'3` and `'"Hello"` $\neq$ `'Hello`. We refer to all of these as Axiom 1.

One axiom described by Axiom (schema) 8 is `(consp 'nil)` $=$ `nil`. Others are `(consp '3)` $=$ `nil` and `(consp 'Hello)` $=$ `nil`. We refer to all of these as Axiom 8.

Note that if ϕ is an axiom or theorem and σ is a substitution, then ϕ/σ is a theorem, by the Rule of Instantiation.

We assume you are familiar with the rules of inference for propositional calculus and equality. For example, we take for granted that you can recognize propositititional tautologies, reason by cases, substitute equals for equals.

For example, we show a theorem below that you should be able to prove, using nothing but your knowledge of propositional calculus and equality (and Axiom 1).

The proof shown below uses the Deduction Law of propositional calculus: we can prove $p \rightarrow q$ by assuming p as a "Given" and deriving q.

Theorem.
`(consp x)` $=$ `t` $\wedge$ `x` $=$ `(car z)` $\rightarrow$ `(consp (car z))` $\neq$ `nil`

Proof.

1.	`(consp x)` $=$ `t`	{*Given*}
2.	`x` $=$ `(car z)`	{*Given*}
3.	`t` $\neq$ `nil`	{*Axiom 1*}
4.	`(consp x)` $\neq$ `nil`	{*Equality Substitution, line 1 into line 3*}
5.	`(consp (car z))` $\neq$ `nil`	{*Equality Substitution, line 2 into line 4*}

Q.E.D.

We will not write proofs in this style. We will simply say that the formula is a theorem "by propositional calculus, equality, and Axiom 1."

Recall that each function definition adds an axiom. The definition

```
(defun tree-copy (x)
  (if (consp x)
      (cons (tree-copy (car x))
            (tree-copy (cdr x)))
```

```
     x))
```

adds the axiom

Axiom `tree-copy`

```
(tree-copy x)
=
(if (consp x)
    (cons (tree-copy (car x))
          (tree-copy (cdr x)))
   x)
```

Thus, by the Rule of Instantiation

Theorem.

```
(tree-copy (cons a b))
=
(if (consp (cons a b))
    (cons (tree-copy (car (cons a b)))
          (tree-copy (cdr (cons a b))))
   (cons a b))
```

9 Terms as Formulas

Logicians make a careful distinction between terms (whose values range over objects in the domain, like the integers, etc.) and formulas (whose values range over the truthvalues). We have set up two systems of propositional calculus. At the level of formulas we have the traditional equality relation, =, and the logical operators $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$. At the level of terms, we have the primitive function `equal` and the defined propositional functions `and`, `or`, `not`, `implies`, and `iff`. In our term-level propositional calculus, `t` and `nil` play the role of truthvalues. Because terms can be written entirely with ASCII symbols (and easily entered on a keyboard!) we tend to write terms and use them as formulas.

For example, we might say that

```
(implies (and (consp x)
              (not (consp y)))
         (not (equal x y)))
```

is a theorem. But of course it cannot be a theorem because it is a term and only formulas are theorems!

If we use an ACL2 term `p` as though it were a formula then the term should be understood as an abbreviation for `p` $\neq$ `nil`. Thus, if we say term `p` is a theorem we mean it is a theorem that `p` is not `nil`. This abuse of terminology is justified by the following theorems.

Theorem. `NOT` is Logical Negation:
`(not p)` $\neq$ `nil` $\leftrightarrow \neg$ (`p` $\neq$ `nil`).

Proof. We handle the two directions of the $\leftrightarrow$.
Case 1.
`(not p)` $\neq$ `nil` $\rightarrow \neg$ (`p` $\neq$ `nil`).
This is equivalent to its contrapositive:
`p` $\neq$ `nil` $\rightarrow$ `(not p)` = `nil`.
By the definition of `not` and Axiom 2 and the hypothesis `p`$\neq$ `nil`, `(not p)` = `(if p nil t)` = `nil`.
Case 2.
$\neg$ (`p` $\neq$ `nil`) $\rightarrow$ `(not p)` $\neq$ `nil`.
The hypothesis is propositionally equivalent to `p` = `nil`. By substitution of equals for equals, the conclusion is `(not nil)` $\neq$ `nil`. By the definition of `not` and Axioms 3 and 1, `(not nil)` = `(if nil nil t)` = `t` $\neq$ `nil`.
Q.E.D.

Problem 18.
Prove
`(and p q)` $\neq$ `nil` $\leftrightarrow$ (`p` $\neq$ `nil`) $\wedge$ (`q` $\neq$ `nil`).

Problem 19.
Prove
`(or p q)` $\neq$ `nil` $\leftrightarrow$ (`p` $\neq$ `nil`) $\vee$ (`q` $\neq$ `nil`).

Problem 20.
Prove
`(implies p q)` $\neq$ `nil` $\leftrightarrow$ (`p` $\neq$ `nil`) $\rightarrow$ (`q` $\neq$ `nil`).

Problem 21.
Prove
`(iff p q)` $\neq$ `nil` $\leftrightarrow$ (`p` $\neq$ `nil`) $\leftrightarrow$ (`q` $\neq$ `nil`).

Problem 22.
Prove
`(equal x y)` $\neq$ `nil` $\leftrightarrow$ (`x` $=$ `y`)

Note that these theorems allow us to change the propositional functions to their logical counterparts as we move the "$\neq$ `nil`" into the term. Furthermore, we can always drop a "$\neq$ `nil`" anywhere it occurs in a formula since the term with which it appears would then be used as a formula and would mean the same thing.

Problem 23.
Using the theorems above, prove that

```
(implies (and p (implies q r))
         s)
```

is equivalent to

```
(p ∧ (q → r)) → s
```

which is equivalent to

```
 ((p ∧ ¬ q) → s)
∧
 ((p ∧ q ∧ r) → s)
```

When writing proofs on paper or the board, we tend to use formulas and the short symbols $=$, $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$ instead of the longer term notation.

Problem 24.
Prove

```
(equal (car (if a b c)) (if a (car b) (car c)))
```

that is, prove

```
(car (if a b c)) = (if a (car b) (car c))
```

Problem 25.
Prove

```
(equal (if (if a b c) x y)
```

```
(if a (if b x y) (if c x y)))
```

Problem 26.
Prove

```
(equal (tree-copy (cons a b))
       (cons (tree-copy a) (tree-copy b)))
```

10 Definitions, Revisited

Problem 27.
Suppose we define

```
(defun f (x) 1)
```

and then prove some theorems and then "redefine" `f` with

```
(defun f (x) 2)
```

Prove `(equal 'June 'July)`.

Problem 28.
Suppose we define

```
(defun f (x) (cons x y))
```

Prove `(equal 1 2)`.[2]

Problem 29.
Suppose we define

```
(defun f (x) (not (f x)))
```

Prove `(equal t nil)`.

These problems should disturb you! We want to use proof as a way to determine truth. But we know that `'June` and `'July` are different objects, as are `1` and `2` and `t` and `nil` – and yet we can prove them equal! Something has gone terribly wrong.

To prevent this kind of logical inconsistency, we impose some restrictions on our ability to introduce definitions. The restrictions that ACL2 enforces are somewhat different, but the ones below are sufficient to insure consistency. We do not explain in this document why these restrictions suffice.

A *car/cdr nest around* v is `(car` v`)`, `(cdr` v`)`, or a `car/cdr` nest around `(car` v`)` or `(cdr` v`)`. Thus, `(car (cdr (car x)))` is a `car/cdr` nest around `x`.

A term p *governs* an occurrence r of a term in term β if and only if

1. β is `(if` q x y`)`, r is in x, and p is q; or
2. β is `(if` q x y`)`, r is in y, and either p is `(not` q`)` or q is `(not` p`)`, or
3. β is (if q x y) and p governs r in x, or p governs r in y.

Thus, in the term `(if a (if b (h c) (h d)) (g c))`, both `a` and `b` govern the first occurrence of `c` and the occurrence of `(h c)`. In addition, `a` and `(not b)` govern `d` and `(h d)`, and `(not a)` governs the second occurrence of `c` and `(g c)`. Note that a consequence of this definition is that `p` does not govern the occurrence of `a` in `(car (if p a b))` but does govern the occurrence of `a` in the equivalent term `(if p (car a) (car b))`.

[2]The definition `f` in this problem has nothing to do with the definition of `f` in the previous problem! We tend to "re-use" function names like `f`, `g` and `h` from time to time simply to avoid inventing new names.

Principle of Structural Recursion: A definition, `(defun` f $(v_1 \ldots v_n)$ β`)` will be allowed (for now) only if it satisfies these four restrictions:

1. The symbol being defined, f, must be "new," i.e., not already in use as a function symbol in any axiom.
2. The formal variables, $v_1, \ldots, v_n$, must be distinct variable symbols.
3. The body, β, must be a term, it must use no new function symbol other than (possibly) f, and the only variable symbols in it are among the formals.
4. There is an i such that `(consp` v_i`)` governs every recursive call of f in β and for every recursive call (f $a_1 \ldots a_n$) in β, a_i is a `car`/`cdr` nest around v_i. We call v_i a *measured formal.*

An acceptable definition adds the axiom `(`$f v_1 \ldots v_n$`)` $= \beta$.

Problem 30.
Explain why these restrictions rule out the spurious definitions of `f` in the problems above.

Problem 31.
Is the following definition allowed under the above restrictions?

```
(defun f (x)
  (if (consp x)
      (if (consp (cdr x))
          (f (cdr (cdr x)))
          nil)
      t))
```

Problem 32.
Is the following definition allowed?

```
(defun f (x y)
  (if (consp x)
      (f (cons nil x) (cdr y))
      y))
```

Problem 33.
Is the following definition allowed?

```
(defun f (x y)
  (if (consp x)
      (f (cons nil y) (cdr x))
      y))
```

Problem 34.
Is the following definition allowed?

```
(defun f (x)
  (if (not (consp x))
      x
     (f (cdr (cdr x)))))
```

Problem 35.
Is the following sequence of definitions allowed?

```
(defun endp (x) (not (consp x)))
(defun f (x)
  (if (endp x)
      nil
      (cons nil (f (cdr x)))))
```

Problem 36.
Is the following definition allowed?

```
(defun f (x y)
  (if (consp x)
      (f (cdr x) (cons nil y))
      y))
```

Problem 37.
Is the following definition allowed?

```
(defun f (x y)
  (if (consp x)
      (f (cdr x)
         (f (cdr x) y))
      y))
```

Problem 38.
Is the following sequence of definitions allowed?

```
(defun f (x)
  (if (consp x)
      (g (cdr x))
      x))
(defun g (x)
  (if (consp x)
      (f (cdr x))
      x))
```

11 Structural Induction

Problem 39.
Given the definition

```
(defun f (x)
  (if (consp x)
      (f (cdr x))
      t))
```

can you prove the theorem `(equal (f x) t)` using the logical machinery we have described above?

ACL2 supports inductive proofs. Its Induction Principle is quite general and involves the notion of the ordinals and well-foundedness. We use a much simpler principle for now.

Principle of Structural Induction: The formula `(ψ x y)`, where `x` and `y` are variable symbols, can be proved by proving:

```
Base Case:
(implies (not (consp x)) (ψ x y))
```

and

```
Induction Step:
(implies (and (consp x)            ; test
              (ψ x1 α1)            ; induction hypothesis 1
              (ψ x2 α2)            ; induction hypothesis 2
              ...)
         (ψ x y))                  ; induction conclusion
```

where the x_i are `car`/`cdr` nests on `x`, and the α_i are arbitrary terms replacing the non-induction variable `y`.

Here is an example Induction Step.

```
(implies (and (consp x)
              (ψ (car x) (app x y))
              (ψ (cdr (cdr x)) (cons x y))
              (ψ (cdr (cdr x)) y))
         (ψ x y))
```

Let us use structural induction to prove a theorem about `tree-copy`. Recall the definition.

```
(defun tree-copy (x)
  (if (consp x)
      (cons (tree-copy (car x))
            (tree-copy (cdr x)))
    x))
```

Theorem `(equal (tree-copy x) x)`.

Proof.

Name the formula above *1.

We prove *1 by induction. One induction scheme is suggested by this conjecture – namely the one that unwinds the recursion in `tree-copy`.

If we let `(ψ x)` denote *1 above then the induction scheme we'll use is

```
(and (implies (not (consp x)) (ψ x))
     (implies (and (consp x)
                   (ψ (car x))
                   (ψ (cdr x)))
              (ψ x))).
```

When applied to the goal at hand the above induction scheme produces the following two nontautological subgoals.

Subgoal *1/2

```
(implies (not (consp x))
         (equal (tree-copy x) x)).
```

But simplification reduces this to `t`, using the definition of `tree-copy` and the primitive axioms.

Subgoal *1/1

```
(implies (and (consp x)                                ; hyp 1
              (equal (tree-copy (car x)) (car x))      ; hyp 2
```

```
          (equal (tree-copy (cdr x)) (cdr x)))  ; hyp 3
     (equal (tree-copy x) x)).
```

But simplification reduces this to `t`, using the definition of `tree-copy` and the primitive axioms.

That completes the proof of *1.

Q.E.D.

Let us look more closely at the reduction of Subgoal *1/1. Consider the left-hand side of the concluding equality. Here is how it reduces to the right-hand side under the hypotheses.

```
(tree-copy x)
=                       {def tree-copy}
(if (consp x)
    (cons (tree-copy (car x))
          (tree-copy (cdr x)))
    x)
=                       {hyp 1 and Axiom 6}
(if t
    (cons (tree-copy (car x))
          (tree-copy (cdr x)))
    x)
=                       {Axioms 2 and 1}
(cons (tree-copy (car x))
      (tree-copy (cdr x)))
=                       {hyp 2}
(cons (car x)
      (tree-copy (cdr x)))
=                       {hyp 3}
(cons (car x)
      (cdr x))
=                       {Axiom 11 and hyp 1}
x
```

This proof is of a very routine nature: induct so as to unwind some particular function appearing in the conjecture and then use the axioms and definitions to simplify each case to `t`.

The problems below refer to function symbols defined in previous exercises. Try to prove them for the definitions you wrote. But if you cannot, then use the definitions we use in our solutions. If the conjectures below are not theorems, show a counterexample! And then try to write the theorem "suggested" by the conjecture. For example, add a hypothesis that restricts some variable so that the conjecture holds; you may even need to introduce new concepts.

Problem 40.
Prove

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

Problem 41.
Prove

```
(equal (app a nil) a)
```

Problem 42.
Prove

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

Problem 43.
Prove

```
(equal (rev (mapnil x)) (mapnil (rev x)))
```

Problem 44.
Prove

```
(equal (rev (rev x)) x)
```

Problem 45.
Prove

```
(equal (swap-tree (swap-tree x)) x)
```

Problem 46.
Prove

```
(equal (mem e (app a b)) (or (mem e a) (mem e b)))
```

Problem 47.
Prove

```
(equal (mem e (int a b)) (and (mem e a) (mem e b)))
```

Problem 48.
Prove

```
(sub a a)
```

Problem 49.
Prove

```
(implies (and (sub a b)
              (sub b c))
         (sub a c))
```

Problem 50.
Prove

```
(sub (app a a) a)
```

Problem 51.
Define

```
(defun mapnil1 (x a)
   (if (consp x)
       (mapnil1 (cdr x) (cons nil a))
       a))
```

Formalize and then prove the remark "On lists of `nil`s, `mapnil1` is commutative.

Problem 52.
Define `(perm x y)` so that it returns `t` if lists `x` and `y` are permutations of each other; otherwise it returns `nil`.

Problem 53.
Prove

```
(perm x x)
```

Problem 54.
Prove

```
(implies (perm x y) (perm y x)).
```

Problem 55.
Prove

```
(implies (and (perm x y)
              (perm y z))
         (perm x z))
```

For several of the problems below it is necessary to have a total ordering relation. Let `<<=` be a non-strict total order, i.e., a Boolean function that enjoys the following properties:

```
(and (<<= x x)                  ; Reflexive
     (implies (and (<<= x y)    ; Anti-symmetric
                   (<<= y x))
              (equal x y))
     (implies (and (<<= x y)    ; Transitive
                   (<<= y z))
              (<<= x z))
     (or (<<= x y)              ; Total
         (<<= y x)))
```

Actually, there is such a function in ACL2 and it is called `lexorder`. But we use the more suggestive name "`<<=`" here. On the integers, `<<=` is just `<=`, but it orders all ACL2 objects.

Problem 56.
Define `(ordered x)` so that it returns `t` or `nil` according to whether each pair of adjacent elements of `x` are in the relation `<<=`. For example, `(ordered '(1 3 3 7 12)` would be `t` and `(ordered '(1 3 7 3 12))` would be `nil`.

Problem 57.
Define `(isort x)` to take an arbitrary list and return an `ordered` permutation of it.

Problem 58.
Prove

```
(ordered (isort x)).
```

Problem 59.
Prove

```
(perm (isort x) x).
```

Problem 60.
Prove

```
(equal (isort (rev (isort x)))
       (isort x))
```

I thank Pete Manolios for suggesting this problem.

Problem 61.
Define

```
(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
      a))
```

Prove

```
(equal (rev1 x nil) (rev x))
```

Problem 62.
Prove

```
(equal (mapnil1 x nil) (mapnil x))
```

Problem 63.
Prove

```
(not (equal x (cons x y)))
```

Problem 64.
Define

```
(defun mcflatten (x a)
  (if (consp x)
      (mcflatten (car x)
                 (mcflatten (cdr x) a))
    (cons x a)))
```

Prove

```
(equal (mcflatten x nil) (flatten x))
```

12 Arithmetic

12.1 Peano Arithmetic

Recall that the integers are being treated as atomic objects in this document. But we can explore elementary arithmetic by thinking of a list of n `nil`s as a representation for the natural number n. We will call such a list a "nat." Thus, `(nil nil nil)` is a nat, but `3` is a natural number.

Problem 65.
Define (nat x) to recognize nats.

Problem 66.
Define (plus x y) to take two arbitrary lists (even ones that are not nats) and to return the nat representing the sum of their lengths. By defining plus this way we insure that it always returns a nat and that it is commutative.

Problem 67.
Define (times x y) to take two arbitrary lists and to return the nat representing the product of their lengths.

Problem 68.
Define (power x y) to take two arbitrary lists and to return the nat representing the exponentiation of their lengths, i.e., if x and y are of lengths i and j, then (power x y) should return the nat representing i^j.

Problem 69.
Define (lesseqp x y) to return t or nil according to whether the length of x is less than or equal to that of y.

Problem 70.
Define (evennat x) to return t or nil according to whether the length of x is even.

Problem 71.
Prove

```
(implies (nat i)
         (equal (plus i nil) i))
```

Problem 72.
Prove

```
(equal (plus (plus i j) k)
       (plus i (plus j k)))
```

Problem 73.
Prove

```
(equal (plus i j) (plus j i))
```

Problem 74.
Prove

```
(equal (times (times i j) k)
       (times i (times j k)))
```

Problem 75.
Prove

```
(equal (times i j) (times j i))
```

Problem 76.
Prove

```
(equal (power b (plus i j))
```

```
       (times (power b i) (power b j)))
```

Problem 77.
Prove

```
(equal (power (power b i) j)
       (power b (times i j)))
```

Problem 78.
Prove

```
(lesseqp i i)
```

Problem 79.
Prove

```
(implies (and (lesseqp i j)
              (lesseqp j k))
         (lesseqp i k))
```

Problem 80.
Prove

```
(equal (lesseqp (plus i j) (plus i k))
       (lesseqp j k))
```

Problem 81.
Prove

```
(implies (and (evennat i)
              (evennat j))
         (evennat (plus i j)))
```

12.2 ACL2 Arithmetic

The techniques we have studied so far suffice to prove the most elementary facts of natural number arithmetic. In fact, we could conduct our entire study of recursion and induction in the domain of number theory. But it is more fun to deal with less familiar "data structures" where basic properties can be discovered. So we will skip past formal arithmetic with a few brief remarks.

ACL2 provides the numbers as a data type distinct from conses, symbols, strings, and characters. They are not lists of `nil`s! The naturals are among the integers, the integers are among the rationals, and the rationals are among the ACL2 numbers. The complex rationals are also among the ACL2 numbers; in fact they are complex numbers whose real and imaginary parts are rational and whose imaginary parts are non-`0`.

Here are a few commonly used functions in ACL2.

`(natp x)` - recognizes natural numbers

`(integerp x)` - recognizes integers

`(rationalp x)` - recognizes rationals

`(zp x)` - `t` if x is `0` or not a natural; `nil` otherwise

`(nfix x)` - `x` if `x` is a natural; `0` otherwise

`(+ x y)` - sum of the numbers `x` and `y`

`(- x y)` - difference of the numbers `x` and `y`

`(* x y)` - product of the numbers `x` and `y`

`(/ x y)` - rational quotient of the numbers `x` and `y`

`(< x y)` - predicate recognizing that the number `x` is less than the number `y`

`(<= x y)` - predicate recognizing that the number `x` is less than or equal to the number `y`

The functions `+`, `-`, `*`, `/`, `<`, and `<=` default their arguments to `0` in the sense that if some argument is not an ACL2 number then `0` is used instead.

The predicate `zp` is commonly used in recursive definitions that treat an argument as though it were a natural number and count it down to zero.

Here is a "definition" that accesses the $\texttt{n}^{th}$ element of a list, treating `n` as a natural. (This definition is unacceptable under our current Principle of Structural Recursion because `(consp x)` does not govern the recursive call. We will return to this point momentarily.)

```
(defun nth (n x)
  (if (zp n)
      (car x)
      (nth (- n 1) (cdr x))))
```

Thus, `(nth 2 '(A B C D))` is `C`. `(Nth 0 '(A B C D))` is `A`. Interestingly, `(nth -1 '(A B C D))` is also `A`, because `-1` satisfies `zp`. Thus, we can use `nth` with any first argument. (In ACL2, `nth` is defined differently, but equivalently.)

The numbers are axiomatized with the standard axioms for rational fields. See [3].

Henceforth, you may use arithmetic freely in your proofs and assume any theorem of ACL2 arithmetic. That is, you may assume any ACL2 theorem that can be written with the function symbols described above and use it in routine arithmetic simplification. But be careful about what you assume! For example, the following familiar arithmetic facts are not (quite) theorems:

```
(equal (+ x 0) x)                ; Additive Identity
(iff (equal (+ x y) (+ x z))     ; Additive Cancellation
     (equal y z))
```

In addition, the following strange fact is a theorem:

```
(not (equal (* x x) 2))
```

That is, we can prove that the square root of 2 is not rational and hence not in ACL2.

13 Inadequacies of Structural Recursion

Recall that to avoid logical contradictions introduced by "bad" definitions, we imposed four restrictions. The fourth restriction is very constraining: we can only recur on a `car`/`cdr` component of some argument and must ensure that that argument satisfies `consp` before the recursion.

The intent of this restriction was to guarantee that the newly defined function terminates. It is beyond the scope of this paper to explain why termination is linked to consistency, but the intuitive explanation is that if the recursion cannot go on forever then, for every combination of constants to which we apply the function, we could compute a value satisfying the definitional equation (given enough computational resources). From this observation we can conclude there exists a mathematical function satisfying the definitional equation – namely, the one that maps inputs to the computed outputs. Thus, given a model of the theory before we added the definition, we could extend it to a model of the theory with the new definition added. This establishes the consistency of the extended theory.[3]

[3] In fact, our definitions produce *conservative extensions*, which we will briefly discuss below.

The problem with the current version of our fourth restriction is that it is too syntactic – it insists, literally, on the use of `consp`, `car`, and `cdr`. In the ACL2 definitional principle, the fourth restriction is less syntactic: it requires that we be able to *prove* that the recursion terminates. That is, when we propose a new definition, a conjecture is generated and if this conjecture can be proved as a theorem, then we know the function terminates.

The basic idea of this conjecture is to establish that some measure of the function's arguments decreases in size as the function recurs, and this decreasing cannot go on forever. If the size were, say, a natural number, then we would know the decreasing could not go on forever, because the arithmetic less-than relation, `<`, is *well-founded* on the natural numbers. We discuss well-foundedness more in the next section.

Problem 82.
Define `(cc x)` to return the number of conses in `x`. The name stands for "cons count."

Problem 83.
Prove that `cc` always returns a non-negative `integer`.

Problem 84.
Suppose we define

```
(defun atom (x) (not (consp x)))
(defun first (x) (car x))
(defun rest (x) (cdr x))
```

then the following "definition" of `tree-copy` is logically equivalent to the acceptable version, but is considered unacceptable by our syntactic fourth restriction:

```
(defun tree-copy (x)
  (if (atom x)
      x
     (cons (tree-copy (first x))
           (tree-copy (rest x)))))
```

Write down a conjecture that captures the idea that the argument to `tree-copy` is getting smaller (as measured by `cc`) as the function recurs.

Problem 85.
Prove the conjecture above. Note that since `cc` is a natural number, this proof establishes that `tree-copy` terminates on all objects.

Problem 86.
Define `(rm e x)` to return the result of removing the first occurrence (if any) of `e` from `x`. Thus, `(rm 3 '(1 2 3 4 3 2 1))` is `(1 2 4 3 2 1)`.

Problem 87.
Show that the following function terminates.

```
(defun f23 (e x)
  (if (mem e x)
      (f23 e (rm e x))
     23))
```

Note that no `car`/`cdr` nest around `x` is equal to the result of `(rm 3 '(1 2 3))`. Thus, `f23` exhibits a kind of recursion we have not seen previously – but we know it terminates.

Problem 88.
It is obvious that `(f23 e x)` always return 23. Can you prove that with our current logical machinery?

The key to these termination proofs is that the less-than relation is well-founded on the natural numbers. But consider this famous function, known as Ackermann's function,

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

Observe that `ack` can generate some very large numbers. For example, `(ack 4 3)` is `65536`.

Problem 89.
`Ack` always terminates. Why? Don't feel compelled to give a formal proof, just an informal explanation.

In the next three sections of this document we will discuss a well-founded relation far more powerful than less-than on the natural numbers. We will then connect that well-foundedness machinery to a new version of the Definitional Principle, so that we can admit many interesting recursive functions, including `ack`. We will also connect the well-foundedness machinery to a new version of the Induction Principle, so that we can prove that `(f23 e x)` is `23` – and far more interesting theorems.

14 The Ordinals

The ordinals are an extension of the naturals that captures the essence of the idea of ordering. They were invented by George Cantor in the late nineteen century. While controversial during Cantor's lifetime, ordinals are among the richest and deepest mines of mathematics. We only scratch the surface here.

Think of each natural number as denoted by a series of strokes, i.e.,

0	0
1	|
2	||
3	|||
4	||||
...	...
ω	|||||...,

The limit of that progression is the ordinal ω, an infinite number of strokes.

Ordinal addition is just concatenation. Observe that adding one to the front of ω produces ω again, which gives rise to a standard definition of ω: the least ordinal such that adding another stroke at the beginning does not change the ordinal.

We denote by $\omega + \omega$ or $\omega \times 2$ the "doubly infinite" sequence that we might write as follows.

$$\omega \times 2 \quad ||||| \ldots ||||| \ldots$$

One way to think of $\omega \times 2$ is that it is obtained by replacing each stroke in 2 (||) by ω. Thus, one can imagine $\omega \times 3$, $\omega \times 4$, etc., which leads ultimately to the idea of $\omega \times \omega$, the ordinal obtained by replacing each stroke in ω by ω. This is also written as ω^2.

$$\omega^2 \quad ||||| \ldots ||||| \ldots ||||| \ldots ||||| \ldots ||||| \ldots \ldots$$

We can analogously construct ω^3 by replacing each stroke in ω by ω^2 (which, it turns out, is the same as replacing each stroke in ω^2 by ω). That is, we can construct ω^3 as ω copies of ω^2, and so on. This ultimately suggests ω^ω. We can then stack ωs, i.e., ω^{ω^ω}, etc. Consider the limit of all of those stacks,

$$\omega^{\omega^{\omega^{\omega^{\omega^{\omega^{\cdots}}}}}}.$$

That limit is ϵ_0. (As the subscript suggests, there are lots more ordinals! But ACL2 stops with ϵ_0.)

Despite the plethora of ordinals, we can represent all the ones below ϵ_0 in ACL2, using lists. Below we begin listing some ordinals up to ϵ_0; the reader can fill in the gaps at his or her leisure. We show in the left column the conventional notation and in the right column the ACL2 object representing the corresponding ordinal.

ordinal	ACL2 representation
0	0
1	1
2	2
3	3
$\ldots$	...
ω	((1 . 1) . 0)
$\omega+1$	((1 . 1) . 1)
$\omega+2$	((1 . 1) . 2)
$\ldots$	...
$\omega \times 2$	((1 . 2) . 0)
$(\omega \times 2)+1$	((1 . 2) . 1)
$\ldots$	...
$\omega \times 3$	((1 . 3) . 0)
$(\omega \times 3)+1$	((1 . 3) . 1)
$\ldots$	...
ω^2	((2 . 1) . 0)
$\ldots$	...
$\omega^2+\omega \times 4+3$	((2 . 1) (1 . 4) . 3)
$\ldots$	...
ω^3	((3 . 1) . 0)
$\ldots$	...
ω^ω	((((1 . 1) . 0) . 1) . 0)
$\ldots$	...
$\omega^\omega+\omega^{99}+\omega \times 4+3$	((((1 . 1) . 0) . 1) (99 . 1) (1 . 4) . 3)
$\ldots$	...
ω^{ω^2}	((((2 . 1) . 0) . 1) . 0)
$\ldots$	...
ω^{ω^ω}	((((((1 . 1) . 0) . 1) . 0) . 1) . 0)
$\ldots$	...

We say an ordinal is "finite" if it is not a cons and we define `(o-finp x)` to recognize finite ordinals. Of course, if `x` is an ordinal and finite, it is a natural number. But by defining `o-finp` this way we insure that if an ordinal is not finite we can recur into it with `cdr`.

To manipulate ordinals we define functions that access the first exponent, the first coefficient, and the rest of the ordinal:

```
(defun o-first-expt (x)
  (if (o-finp x) 0 (car (car x))))
(defun o-first-coeff (x)
  (if (o-finp x) x (cdr (car x))))
(defun o-rst (x) (cdr x))
```

For example, if `x` is the representation of $\omega^e \times c + r$ then `(o-first-expt x)` is e, `(o-first-coeff x)` is c and `(o-rst x)` is r.

Here is the definition of `o-p`, the function that recognizes ordinals.

```
(defun o-p (x)
  (if (o-finp x)
      (natp x)
      (and (consp (car x))
```

```
            (o-p (o-first-expt x))
            (not (equal 0 (o-first-expt x)))
            (natp (o-first-coeff x))
            (< 0 (o-first-coeff x))
            (o-p (o-rst x))
            (o< (o-first-expt (o-rst x))
                (o-first-expt x)))))
```

(The ACL2 definition is syntactically different but equivalent.)

The function `o<` is the "less than" relation on ordinals. We show its definition below. But study the definition of `o-p` first. It says that an ordinal is a list of pairs, terminated by a natural number. Each pair `(e . c)` consists of an exponent `e` and a coefficient `c` and represents $(\omega^e) \times c$. The exponents are themselves ordinals and the coefficients are non-0 naturals. Importantly, the exponents are listed in strictly descending order. The list represents the ordinal sum of its elements plus the final natural number. Thus, ordinals are a kind of generalized polynomial.

By insisting on the ordering of exponents we can readily compare two ordinals, using `o<` below, in much the same way we can compare polynomials.

```
(defun o< (x y)
  (if (o-finp x)
      (or (not (o-finp y))
          (< x y))
    (if (o-finp y)
        nil
      (if (equal (o-first-expt x)
                 (o-first-expt y))
          (if (equal (o-first-coeff x)
                     (o-first-coeff y))
              (o< (o-rst x)
                  (o-rst y))
            (< (o-first-coeff x)
               (o-first-coeff y)))
        (o< (o-first-expt x)
            (o-first-expt y))))))
```

(The ACL2 definition is syntactically different but equivalent.)

Problem 90.
Which is smaller, ordinal a or ordinal b?

1. a= `23`, b= `100`
2. a= `1000000`, b= `((1 . 1) . 0)`
3. a= `((2 . 1) . 0)`, b= `((1 . 2) . 0)`
4. a= `((3 . 5) (1 . 25) . 7)`, b= `((3 . 5) (2 . 1) . 3)`
5. a= `((((2 . 1) . 0) . 5) . 3)`, b= `((((1 . 1) . 0) . 5) (1 . 25) . 7)`

Problem 91.
The `o<` operation can be reduced to lexicographic comparison. Define `m2` so that it constructs "lexicographic ordinals" from two arbitrary natural numbers. Specifically, show that the following is a theorem:

```
(implies (and (natp i1)
              (natp j1)
              (natp i2)
              (natp j2))
         (and (o-p (m2 i1 j1))
```

```
(iff (o< (m2 i1 j1)
         (m2 i2 j2))
     (if (equal i1 i2)
         (< j1 j2)
       (< i1 i2)))))
```

The crucial property of `o<` is that it is *well-founded on the ordinals*. That is, there is no infinite sequence of ordinals, x_i such that $\ldots x_3$ `o<` x_2 `o<` x_1 `o<` x_0.

Problem 92.
What is the longest decreasing chain of ordinals starting from the ordinal `10`? What is the longest decreasing chain of ordinals starting from the ordinal `((1 . 1) . 0)`?

Problem 93.
Construct an infinitely descending `o<` chain of objects. Note that by the well-foundedness of `o<` on the ordinals, your chain will not consist entirely of ordinals!

Problem 94.
Prove that `o<` is well-founded on our ordinals, i.e., those recognized by `o-p`.

Caution: Using the logical machinery we have developed here, it is not possible to state that `o<` is well-founded on the ordinals: that requires an existential quantifier and infinite sequences. However, it can be done in a traditional set theoretic setting. That is, the theorem that `o<` is well-founded is a "meta-theorem", it can be proved about our system but it cannot be proved within our system.

Our definitional and induction principles are built on the assumption that `o<` is well-founded on the ordinals recognized by `o-p`. Thus, if some ordinal measure of the arguments of a recursive function decreases according to `o<` in every recursive call, the recursion cannot go on forever.

The representation of ordinals described here is a version of Cantor's Normal Form. See [5] or the online documentation topics `ordinals` and `o-p` in [4], from which some of the examples above have been chosen.

15 The Definitional Principle

Below we give a new definitional principle that subsumes the previously given Principle of Structure Recursion.

The definition

`(defun` f `(`$v_1 \ldots v_n$`)` β`)`

is *admissible* if and only if

1. f is a new function symbol,
2. the v_i are distinct variable symbols,
3. β is a term that mentions no variable other than the v_i and calls no new function symbol other than (possibly) f, and
4. there is a term m (called the *measure*) such that the following are theorems:
   - *Ordinal Conjecture*

     `(o-p` m`)`

   - *Measure Conjecture(s)* For each recursive call of `(`$fa_1 \ldots a_n$`)` in β and the conjunction q of tests governing it,

     `(implies` q `(o<` m/σ m`))`

     where σ is $\{v_1 \leftarrow a_1, \ldots, v_n \leftarrow a_n\}$.

Admissible definitions add the axiom:

`(`$fv_1 \ldots v_n$`) =` β.

In each of the problems below, admit the proposed definitions, i.e., identify the measure and prove the required theorems.

Problem 95.

```
(defun tree-copy (x)
  (if (atom x)
      x
     (cons (tree-copy (first x))
           (tree-copy (rest x)))))
```

Problem 96.

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

Problem 97.

Recursion like that in `ack` allows us to define functions that cannot be defined if we are limited to "primitive recursion" where a given argument is decremented in every recursive call. That is, the new definitional principle is strictly more powerful than the old one. This can be formalized and proved within our system (after we extend the principle of induction below). If you are inclined towards metamathematics, feel free to pursue the formalization and ACL2 proof of this. The existence of non-primitive recursive functions, dating from 1928, by Wilhelm Ackermann, a student of David Hilbert's, was one of the important milestones in our understanding of the power and limitations of formal mathematics culminating in Godel's results of the early 1930s.

Problem 98.

```
(defun f1 (i j)
  (if (and (natp i)
           (natp j)
           (< i j))
      (f1 (+ 1 i) j)
    1))
```

Problem 99.

```
(defun f2 (x)
  (if (equal x nil)
      2
    (and (f2 (car x))
         (f2 (cdr x)))))
```

Problem 100.

```
(defun f3 (x y)
  (if (and (endp x)
           (endp y))
      3
    (f3 (cdr x) (cdr y))))
```

Problem 101.
Suppose `p`, `m`, `up`, and `dn` ("down") are undefined functions. Suppose however that you know this about `p`, `m`, and `dn`:

Theorem `dn-spec`
```
(and (o-p (m x))
     (implies (p x)
              (o< (m (dn x)) (m x))))
```
Then admit

```
(defun f4 (x y q)
  (if (p x)
      (if q
          (f4 y (dn x) (not q))
        (f4 y (up x) (not q)))
    4))
```

Note that `f4` is swapping its arguments. Thus, if `q` starts at `t`, say, then in successive calls the first argument is `x`, `y`, `(dn x)`, `(up y)`, `(dn (dn x))`, `(up (up y))`, etc. I thank Anand Padmanaban for helping me think of and solve this problem.

16 The Induction Principle

The Induction Principle allows one to derive an arbitrary formula, ψ, from

- *Base Case*:

  `(implies (and (not` q_1`) ... (not` q_k`))` ψ`),` and

- *Induction Step(s)*: For each $1 \leq i \leq k$,

  `(implies (and` q_i $\psi/\sigma_{i,1}$ ... $\psi/\sigma_{i,h_i}$`)`
  ψ`) ,`

provided that for terms m, $q_1, ...q_k$, and substitutions $\sigma_{i,j}$ $(1 \leq i \leq k, 1 \leq j \leq h_i)$, the following are theorems:

- *Ordinal Conjecture*:

  `(o-p` m`)` , and

- *Measure Conjecture(s)*: For each $1 \leq i \leq k$, and $1 \leq j \leq h_i$,

  `(implies` q_i `(o<` $m/\sigma_{i,j}$ m`))` .

17 Relations Between Recursion and Induction

Informally speaking, a recursive definition is "ok" if there is an ordinal measure that decreases in every recursive call. Thus, the recursion cannot go on forever. In a simple recursion on naturals down to 0 by -1, the value of the function on 5 is determined recursively by its value on 4, which is determined recursively by its value on 3, which is determined recursively by its value on 2, which is determined recursively by its value on 1, which is determined recursively by its value on 0, which is specified explicitly in the definition.

An inductive proof is "ok" if there is an ordinal measure that decreases in every induction hypothesis. Thus, any concrete instance of the conjecture could be proved by "pumping" forward a finite number of times from the base cases. Given a simple inductive proof over the naturals, the conjecture is true on 0 because it was proved explicitly in the base case, so it is true on 1 by the induction step, so it is true on 2 by the induction step, so it is true on 3 by the induction step, so it is true on 4 by the induction step, so it is true on 5 by the induction step.

Clearly these two concepts are duals. The formal statements of the two principles look more different than they are. They both require us to prove that a measure returns an ordinal and that some substitutions make the measure decrease under some tests. But there seems to be a lot less indexing going on in the Definitional Principle than in the Induction Principle. That is due to language and the two different uses of the principles. The Definitional Principle is designed to tell us whether a definitional equation is ok. The Induction Principle is designed to tell us whether a set of formulas is an ok inductive argument. So the Definitional Principle talks about the tests in `IF`s and the substitution built from each recursive call, whereas the Induction Principle talks about the tests in the i^{th} formula and the substitution that created the j^{th} induction hypothesis of the i^{th} formula.

But the key insight is: *Every ok definition suggests an ok induction*! We call this the induction *suggested* by the definition. It is easist to see this by considering a particular, generic definition and thinking about what had to be proved to admit it, what induction it suggests, what has to be proved for that induction to be legal, and when the suggested induction might be useful.

Suppose the following definition has been admitted, justified by measure term `(m x a)`. Note that the body is an `IF`-tree, there are three tips in the `IF`-tree; the first tip contains two recursive calls, the second tip contains one recursive call, and the third tip contains no recursive calls.

```
(defun f (x a)
  (if (test1 x a)
      (if (test2 x a)
          (h                              ; tip 1
           (f (d1 x a) (a1 x a))          ;    rec call 1,1
           (f (d2 x a) (a2 x a)))         ;    rec call 1,2
        (g                                ; tip 2
         (f (d3 x a) (a3 x a))))          ;    rec call 2,1
    (b x a)))                             ; tip 3
```

To admit this definition we had to prove:

Ordinal Conjecture

```
(o-p (m x a))
```

Measure Conjecture 1,1

```
(implies (and (test1 x a) (test 2 x a))
         (o< (m (d1 x a) (a1 x a))
             (m x a)))
```

Measure Conjecture 1,2

```
(implies (and (test1 x a) (test 2 x a))
         (o< (m (d2 x a) (a2 x a))
             (m x a)))
```

Measure Conjecture 2,1

```
(implies (and (test1 x a) (not (test 2 x a)))
         (o< (m (d3 x a) (a3 x a))
             (m x a)))
```

Suppose we want to prove `(p x a)`, by induction according to the scheme "suggested" by `(f x a)`. Here is the scheme:

Base Case ; for tip 3

```
(implies (not (test1 x a))
         (p x a))
```

Induction Step 1 ; for tip 1

```
(implies (and (test1 x a)
              (test2 x a)
              (p (d1 x a) (a1 x a))   ;   for rec call 1,1
              (p (d2 x a) (a2 x a))) ;   for rec call 1,2
         (p x a))
```

Induction Step 2 ; for tip 2

```
(implies (and (test1 x a)
              (not (test2 x a))
              (p (d3 x a) (a3 x a))) ;   for rec call 2,1
         (p x a))
```

This induction is produced by the following parameter choices in the Induction Principle:

ψ	`(p x a)`
m	`(m x a)`
q_1	`(and (test1 x a) (test2 x a))`
q_2	`(and (test1 x a) (not (test2 x a)))`
$\sigma_{1,1}$	{`x` ← `(d1 x a)`, `a` ← `(a1 x a)`}
$\sigma_{1,2}$	{`x` ← `(d2 x a)`, `a` ← `(a2 x a)`}
$\sigma_{2,1}$	{`x` ← `(d3 x a)`, `a` ← `(a3 x a)`}

It should be obvious to you how these choices are determined from the definition of `f` with measure `(m x a)`. For example, q_1 is the conjunction of the tests leading to the first tip containing recursive calls of `f`, and the substitutions $\sigma_{1,j}$ are the substitutions derived from the recursive calls in that tip.

The measure conjectures required by the Induction Principle for this choice of parameters are the exactly[4] the same as the measure conjectures verified when the Definitional Principle was used to admit `f`! That is, to use an induction suggested by an already-admitted recursive function, no additional measure conjectures have to be proved.

But why might this induction be interesting or useful for proving `(p x a)`? The answer depends on `(p x a)`, of course. But the most common situation is that we choose the induction scheme suggested by some recursive function used in the conjecture to be proved. So suppose `(f x a)` occurs in `(p x a)`. Why is the suggested induction likely to be helpful? Consider the Base Case and the two Induction Steps.

In the Base Case, the `(f x a)` occurring in `(p x a)` can be replaced by tip 3 of the definition of `f`, `(b x a)` because the test in the Base Case of the induction is the test leading to the non-recursive exit from the definition. So `f` has been eliminated from the proof of the Base Case.

Now consider Induction Step 1. The `(f x a)` occurring in `(p x a)` can be replaced by tip 1 of the definition of `f`, namely

```
(h                           ; tip 1
 (f (d1 x a) (a1 x a))       ;    rec call 1,1
 (f (d2 x a) (a2 x a)))      ;    rec call 1,2
```

because the tests in Induction Step 1 are the tests leading to tip 1 of the definition. But notice that the induction hypothesis labeled "for rec call 1,1" in Induction Step 1 gives us a hypothesis about recursive call 1,1 — because the occurrence of `(f x a)` in `(p x a)` becomes an occurrence of `(f (d1 x a) (a1 x a))` when we apply the substitution $\sigma_{1,1}$ to it. Similarly, the induction hypothesis labeled "for rec call 1,2" gives us a hypothesis about call 1,2. Thus, the proof of Induction Step 1 boils down to proving, "if the two recursive calls in this tip have the property we're proving, then `h` of those two calls have the property." While not exactly eliminating `f` from the proof, it provides us with all the information we have a right to suppose about `f` in this case. Usually the proof of this step requires a lemma about `p` and `h`, e.g., "if `a` and `b` have property `p`, then so does `(h a b)`." Such a lemma would eliminate `f` and if we had that lemma the proof of Induction Step 1 would be done. The proof of Induction Step 2 is analogous.

Thus, we see that there may well be some heuristic value in using the induction suggested by `(f x a)` whenever you are trying to prove a property of `(f x a)`. Occasionally it is necessary to use an induction suggested by a function not appearing in the conjecture, but when that occurs it is usually some easily recognized "mash up" of other functions appearing in the conjecture.

Problem 102.
Recall the previously admitted

```
(defun f1 (i j)
  (if (and (natp i)
           (natp j)
```

[4] We have propositionally simplified the defining condition for the Base Case. The Induction Principle says it is `(and (not` q_1`) (not` q_2`))` and the literal instantiation of that here would be `(and (not (and (test1 x a) (test2 x a))) (not (and (test1 x a) (not (test2 x a)))))`, but that is propositionally equivalent to `(not (test1 x a))`.

```
          (< i j))
      (f1 (+ 1 i) j)
    1))
```

Prove (equal (f1 i j) 1).

Problem 103.
Recall the previously admitted

```
(defun f2 (x)
  (if (equal x nil)
      2
    (and (f2 (car x))
         (f2 (cdr x)))))
```

Prove (equal (f2 x) 2).

Problem 104.
Recall the previously admitted

```
(defun f3 (x y)
  (if (and (endp x)
           (endp y))
      3
    (f3 (cdr x) (cdr y))))
```

Prove (equal (f3 x y) 3).

Problem 105.
Recall the previously admitted

```
(defun f4 (x y q)
  (if (p x)
      (if q
          (f4 y (dn x) (not q))
        (f4 y (up x) (not q)))
    4))
```

Prove (equal (f4 x y q) 5).

These simple inductive exercises drive home the point that once a function has been admitted (proved to terminate) then we can do inductions to "unwind" it. Students so frequently see induction limited to "$p(n)$ implies $p(n+1)$" that it is easy to forget that every total recursive function give rises to an induction that is appropriate for it.

18 More Problems

Problem 106.
Here is a way to flatten a binary tree without using an auxiliary function. Admit this definition.

```
(defun flatten! (x)
  (if (atom x)
      (cons x nil)
    (if (atom (car x))
        (cons (car x) (flatten! (cdr x)))
      (flatten! (cons (caar x) (cons (cdar x) (cdr x)))))))
```

Problem 107.
Prove `(equal (flatten! x) (flatten x))`.

Problem 108.
Here is a clever way to determine if two binary trees have the same fringe. Admit this function (and its subroutine).

```
(defun samefringe (x y)
 (if (or (atom x)
         (atom y))
     (equal x y)
   (and (equal (car (gopher x))
               (car (gopher y)))
        (samefringe (cdr (gopher x))
                    (cdr (gopher y))))))
```

where

```
(defun gopher (x)
  (if (or (atom x)
          (atom (car x)))
      x
    (gopher (cons (caar x) (cons (cdar x) (cdr x))))))
```

Problem 109.
Prove

```
(equal (samefringe x y)
       (equal (flatten x)
              (flatten y)))
```

Problem 110.
The curious recursions in `gopher` and `samefringe` are due to John McCarthy, who viewed `gopher` as a model of a co-routine. Explain what he was thinking.

Problem 111.
Below is a model of the functional behavior of Quick Sort. Note that `rel` is defined as a "higher order" function that can apply any of four relations.

```
(defun rel (fn x y)
  (if (equal fn '<<=)
      (<<= x y)
    (if (equal fn '>>=)
        (<<= y x)
      (if (equal fn '<<)
          (and (<<= x y) (not (equal x y)))
        (and (<<= y x) (not (equal x y)))))))
(defun filter (fn x e)
  (if (endp x)
      nil
    (if (rel fn (car x) e)
        (cons (car x) (filter fn (cdr x) e))
        (filter fn (cdr x) e))))
(defun qsort (x)
```

```
  (if (endp x)
      nil
    (if (endp (cdr x))
        x
      (app (qsort (filter '<< (cdr x) (car x)))
           (cons (car x)
                 (qsort (filter '>>= (cdr x) (car x))))))))
```

Prove that `qsort` produces an ordered permutation of its input.

Problem 112.
Prove that the length of a list of distinct natural numbers is no greater than its maximum element plus one. This is sometimes called the Pigeon Hole Principle.

Problem 113.
Imagine a simple list being treated as a "memory." If a is a natural number less than the length of the memory, then a is an "address" and we can use `nth` to fetch the contents. This is "dereferencing" a. If a is not an address, we will say it is "data." Now consider the idea of taking an object and a memory and dereferencing until we get to data. The following function counts the steps; it returns the symbol `infinite` if a loop is detected.

```
(defun deref-cnt (ptr mem seen)
  (if (addressp ptr mem)
      (if (mem ptr seen)
          'infinite
        (inc (deref-cnt (nth ptr mem) mem (cons ptr seen))))
    0))
```

where

```
(defun len (x)
  (if (consp x)
      (+ 1 (len (cdr x)))
      0))
(defun inc (x)
  (if (integerp x) (+ 1 x) 'infinite))
(defun addressp (ptr m)
  (and (natp ptr)
       (< ptr (len m))))
```

Admit `deref-cnt`.

Problem 114.
This is a fact every undergraduate knows. The number of times you can dereference (without being in a loop) is bounded by the size of the memory (plus one for the initial probe). Prove it.

```
(<= (deref-cnt ptr mem nil)
    (+ 1 (len mem)))
```

Since `deref-cnt` returns the non-number `infinite` if it is a loop, and since ACL2 arithmetic treats non-numbers as `0`, this theorem is trivial when the process loops. You may therefore explicitly add the hypothesis `(integerp (deref-cnt ptr mem nil))` if you are uncomfortable dealing with the non-numeric defaults.

Problem 115.
Here is the familiar definition of the Fibonacci function and a more efficient one ("fast Fibonacci").

```
(defun fib (i)
  (if (zp i)
      0
    (if (equal i 1)
        1
      (+ (fib (- i 1))
```

```
         (fib (- i 2))))))
(defun ffib (i j k)
  (if (zp i)
      j
    (if (equal i 1)
        k
      (ffib (- i 1) k (+ j k)))))
```

Prove they are equal in the following sense

```
(equal (ffib n 0 1) (fib n))
```

Problem 116.
Prove

```
(equal (rotn (len x) x) x)
```

where

```
(defun rot (x)
  (if (endp x)
      nil
    (app (cdr x) (cons (car x) nil))))
(defun rotn (n x)
  (if (zp n)
      x
    (rotn (- n 1) (rot x))))
```

Problem 117.
In this problem we explore binary arithmetic. Let a *bit vector* be a list of Booleans; it is convenient to suppose that the least significant bit is the car. A bit vector represents a natural number in the usual binary encoding. Define `(vadd x y)` to return a bit vector representing the sum of the numbers represented by the bit vectors `x` and `y`. Formally specify and prove that `vadd` is correct. Obviously, this problem can be expanded to include other binary arithmetic operations and implementations.

19 More Inadequacies of the Definitional Principle

19.1 Mutual Recursion

Problem 118.
Mutual recursion is (still) not allowed by our statement of the Definitional Principle. However, suppose that somehow the axioms produced by the following pair of definitions were available.

```
(defun fx (x)
  (if (consp x)
      (cons (fx (car x))
            (gx (cdr x)))
    x))
(defun gx (x)
  (if (consp x)
      (cons (gx (car x))
            (fx (cdr x)))
    x))
```

Prove

```
(equal (fx x) x)
```

Problem 119.
Develop a methodology for dealing with mutual recursion. That is, explain how you can use our Definitional Principle to introduce two functions `fx` and `gx` that can be shown to satisfy the "defining equations" for `fx` and `gx` above. Your methodology should work for any clique of mutually recursive functions that can be shown to terminate under the `o<` relation.

Problem 120.
We say `x` is an *expression* if (a) `x` is a symbol or (b) `x` is a list of the form ($f\ e_1 \ldots e_n$), where f is a symbol and the e_i are expressions. Define `(expr x)` to recognize expressions.

Problem 121.
An *expression substitution* is a list of pairs of the form `(`v *expr*`)`, where v is a symbol and *expr* is an expression. Define `substitution` to recognize expression substitutions.

Problem 122.
Define `(slash x s)` so that it substitutes the expression-substitution `s` into the expression `x`, e.g., it returns `x/s`.

Problem 123.
Prove that if `s` is an expression substitution and `x` is an expression, then `(slash x s)` is an expression.

19.2 Problematic Nested Recursion

Problem 124.
The following function cannot be admitted. Explain why.

```
(defun f5 (x)
  (if (zp x)
      0
    (+ 1 (f5 (f5 (- x 1))))))
```

When the function being defined, f, is called recursively on the value of a nested recursive call, e.g., `(`f ... `(`f `...))`, we say the definition exhibits *nested recursion.* We have already seen some examples of nested recursion:

```
(defun mcflatten (x a)
  (if (consp x)
      (mcflatten (car x)
                 (mcflatten (cdr x) a))
    (cons x a)))


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

Problem 125.
What makes the recursion in `f5`, which cannot be admitted, different from that in `mcflatten` and `ack`, which can be admitted?

It is often possible to deal with problematic nested recursion, by admitting a different but related definition and then proving that it is equivalent to the one you wanted. The next few problems illustrate this.

Problem 126.
Admit the following

```
(defun f5 (x)
  (if (zp x)
      0
    (if (< (nfix (f5 (- x 1))) (nfix x))
        (+ 1 (f5 (f5 (- x 1))))
      'undef)))
```

Problem 127.
Prove

```
(implies (natp x) (<= (f5 x) x))
```

Problem 128.
Prove

```
(equal (f5 x)
       (if (zp x)
           0
         (+ 1 (f5 (f5 (- x 1))))))
```

You may think that problematic nested recursion never arises in actual formal models. Think again! Consider a graph reachability algorithm, as might be implemented in the mark phase of a garbage collector. Suppose a memory location can hold a pair of addresses, a single address, or data. Given an initial address, mark all the reachable addresses, avoiding any possible cycles in the data. When the mark algorithm arrives at an address, *ptr*, containing a pair, it marks that address, explores and marks one of the addresses, and then explores and marks the other. The algorithm terminates because marking increases the number of marked addresses in memory and the number of addresses is bounded. But note that when the algorithm explores and marks the second address it does so on the memory produced by exploring and marking the first address and the parent. This is problematic nested recursion: if the inner recursion *unmarked* some of the marked addresses, the outer sweep might get caught in a cycle. But we cannot prove that the algorithm does not unmark things until we have admitted the model!

Problem 129.
Let `m` be a list representing a RAM, with 0-based addressing modeled by `nth`. An address is legal if it is a natural less than the length of the memory. If an address contains a pair, we explore both the `car` and the `cdr`. If an

address contains an address, we explore that address. Cycles may be present. The function below collects the list of all addresses reachable from a given one. This definition exhibits problematic nested recursion.

```
(defun reachables (ptr m seen)
  (if (addressp ptr m)
      (if (mem ptr seen)
          seen
        (if (consp (nth ptr m))
            (reachables (car (nth ptr m))
                        m
                        (reachables (cdr (nth ptr m))
                                    m
                                    (cons ptr seen)))
          (reachables (nth ptr m) m (cons ptr seen))))
    seen))
```

Use the method suggested by the `f5` problems to define a function satisfying the equation above.

Problem 130.
Consider the problem of reversing a list. Our standard definition, `rev`, uses an auxiliary function, `app`. The tail-recursive version, `rev1`, uses an additional formal parameter. Below is a definition of reverse that has only one parameter and no auxiliary functions. This definition was proposed by Rod M. Burstall in the early 1970s as an entertaining puzzle. It exhibits problematic nested recursion.

```
(defun rmb (x)
  (if (consp x)
      (if (consp (cdr x))
          (cons (car (rmb (cdr x)))
                (rmb (cons (car x)
                           (rmb (cdr (rmb (cdr x)))))))
        (cons (car x) nil))
    nil))
```

Show how this definition can be derived from an admissible one.

Problem 131.
Suppose `rmb` is known to satisfy the "defining equation" that would be added by the `defun` above had it been admissible. That is, suppose that is the only equation known for `rmb`. Prove

```
(equal (rmb x) (rev x))
```

20 Still More Problems

The next several problems will lead you to define a tautology checker in ACL2 and to prove that it is sound and complete. This is an excellent exercise if you are interested in theorem proving.

Problem 132.
An *IF-expression* is a cons whose car is `IF`. A *quote* is a cons whose care is `QUOTE`. A *variable* is anything besides an `IF`-expression or a quote. An *expression* is a variable, a quote, or an `IF`-expression.

Note: By these definitions, any object is an expression. But we typically think of `IF`-expressions as being of the form (`IF` a_1 a_2 a_3) and quotes being of the form (`QUOTE` $a1$). But rather than check that they are of this form we will just use `car` and `cdr` to chew into `IF`s and quotes to access the desired substructure, which we will call "arguments" one, two, and three.

An *assignment* is a list of pairs, (*var* . *val*), where *var* is a variable and *val* is an arbitrary object (but is typically a Boolean).

We define the *value* of an expression under an assignment as follows. The *value* of a variable is the *val* associated with that variable, or `nil` is the variable is not bound. The *value* of a quote is the first argument of the quote. The value of an IF-expression depends on the value, v, of the first argument. If v is nil, the value of the IF-expression is the value of its third argument; otherwise, it is the value of its second argument.

Define these concepts.

Problem 133.
An expression is said to be in *IF-normal form* if no IF-expression in the expression has an IF-expression in its first argument. Thus, (IF A (IF B C D) D) is in IF-normal form, but (IF (IF A B 'NIL) C D) is not. It is possible to put an expression into IF-normal form while preserving its value, by repeatedly transforming it with the rule:

```
(IF (IF a b c) x y) = (IF a (IF b x y) (IF c x y))
```

Define the notion of IF-normal form and admit the function norm that normalizes an expression while preserving its value.

Problem 134.
Given an expression in IF-normal form, define the function tautp that explores all feasible branches through the expression and determines that every output is non-nil. We say a branch is *infeasible* if, in order to traverse it during evaluation under an assignment, the assignment would have to assign some variable both nil and non-nil.

Problem 135.
Define tautology-checker to recognize whether an expression has a non-nil value under all possible assignments.

Problem 136.
Prove that your tautology-checker is *sound*: if it says an expression is a tautology, then the value of the expression is non-nil, under any assignment.

Problem 137.
Prove that your tautology-checker is *complete*: if it fails to recognize an expression, then some assignment falsifies the expression.

References

[1] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.

[2] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[3] M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. In *http://www.cs.utexas.edu/users/moore/publications/km97a.ps.gz*. Dept. of Computer Sciences, University of Texas at Austin, 1997.

[4] M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2004.

[5] P. Manolios and D. Vroon. Ordinal arithmetic in acl2. In *ACL2 Workshop 2003*, Boulder, Colorado, July 2003. http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/.