Recursion and Induction

Basics

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

University of Texas, Austin

Fall, 2021

Adapted from J Moore's Recursion and Induction Notes.
Slides created and edited by Warren A. Hunt, Jr.
This content closely mirrors Moore's R&I Notes.

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.

The best way to learn to use that theorem prover to productive ends is first to master the art of recursive definition and inductive proof.

We will start by learning a subset of the ACL2 language and a subset of the rules of inference.

Data Types

ACL2 provides five data types:

- Numbers: integers, rationals and complex rationals
- Characters: examples in ACL2 (Common Lisp) include `#\a`, `#\A`, and `#\Newline`
- Strings: ASCII characters written between "string quotes."
- Symbols: are primitive data objects. Some symbols are "overloaded," such as **t** and **nil**, but most just "represent themselves," e.g., **LOAD**, **Store**, **fence**.
- Pairs: The only mechanism to aggreate objects is to pair one object with another object. Lisp provides many abbreviation mechanisms to help write aggreate terms.
  - `nil` can be written `()`.
  - `(`x `. nil)` may be written `(`x`)`.
  - `(`x `. (`y `...))` may be written `(`x y `...)`.
- Identity: We will use a canonical form to determine object equality.

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!`

Exercises

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)))`

Terms

For our purposes, a *term* is a variable symbol, a quoted constant, or a function application written as **(fn** *arg1* **...** *argn***)**

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.

Semantically, terms are interpreted with respect to an assignment binding variable symbols to constants and an interpretation of function symbols as mathematical functions.

- Semantically, terms are interpreted with respect to an assignment binding variable symbols to constants and an interpretation of function symbols as mathematical functions.

- Suppose the variables `x` is bound to `1` and `y` is bound to `(2 3 4)`
- Suppose `cons` is interpreted as the function that constructs ordered pairs
- Then, the meaning or *value* of the term `(cons x y)` is `(1 . (2 3 4))` or, equivalently, `(1 2 3 4)`.

ACL2 provides an infinite number of *variable symbols*, whose syntax is that of symbols.

Terms, continued

A *quoted constant* is written by prefixing an integer, string, or symbol by a single quote mark.

`'t`, `'nil`, `'-3`, `'"Hello World!"` and `'LOAD` are quoted constants.

We do not consider `'(1 2 3)` a quoted constant; this is a 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 has an infinite number of *function symbols* each with an associated *arity*. We now concern ourselves with six primitive function symbols:

- `(cons` *x y*`)` - construct and return the ordered pair `(`*x* `.` *y*`)`.
- `(car` *x*`)` - return left component of *x*, if *x* is a pair; otherwise, return `nil`.
- `(cdr` *x*`)` - return right component of *x*, if *x* is a pair; otherwise, `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))`

Substitutions

A *substitution* is a set $\{v_0 \leftarrow t_0,\ v_1 \leftarrow t_1,\ \ldots\}$ 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.

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))))`$/\sigma$?

Abbreviations for Terms

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

If '() is used as a term, it abbreviates `'nil`.

If `'(`x`)` is used as a term it abbreviates `(cons '`x `'nil)`.

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

If `'(`x_1 x_2 ... x_n`)` is used where a term is expected, it abbreviates `(cons '`x_1 `(cons '`x_2 ... `(cons '`x_n `'nil)...))`. Thus, for example, `'(MON TUE WED)` abbreviates `(cons 'MON (cons 'TUE (cons 'WED 'NIL)))`.

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

Symbols `and` and `or` are used as function symbols of two arguments.

- If `and` is used as though it were a function 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, it abbreviates `(and p (and q (and r s)))`.
- And, similarly, `(or a b c)` abbreviates `(or a (or b c))`.

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))`

Displaying Terms

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.

- Write clearly and count your parentheses.
- Don't write a line with more than about 30 non-blank characters.
- If a function call will not fit on a line, break it into multiple lines, indenting each argument the same amount.

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)))
```

Fuctions Definitions, continued

Note that in Lisp, `and` and `or` are not Boolean valued.

`(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))
```

The term `(tree-copy '((1 . 2) . 3))` has the value `((1 . 2) . 3)`.

Exercises

In the exercises that follow 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`.

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 (of inference) 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.

Axioms, continued

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."

A model, given a set of functions, is an association of each function symbol to an actual function on that set.

Given an assignment, a mapping of variables to elements of the domain (e.g., natural numbers), one can use the model to determine the value of the terms.

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.

Axioms, continued

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.

Axioms, continued

Axiom 1. $\texttt{'}\alpha \neq \texttt{'}\beta$,
where α and β are distinct integers, strings, or symbols

Axiom 2. $\texttt{x} \neq \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{y}$

Axiom 3. $\texttt{x} = \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{z}$

Axiom 4. $\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$

Axiom 5. $\texttt{x} = \texttt{y} \leftrightarrow \texttt{(equal x y)} = \texttt{t}$

Axiom 6. $\texttt{(consp x)} = \texttt{nil} \vee \texttt{(consp x)} = \texttt{t}$

Axiom 7. $\texttt{(consp (cons x y))} = \texttt{t}$

Axiom 8. $\texttt{(consp '}\alpha\texttt{)} = \texttt{nil}$,
where α is an integer, string, or symbol

Axiom 9. $\texttt{(car (cons x y))} = \texttt{x}$

Axiom 10. $\texttt{(cdr (cons x y))} = \texttt{y}$

Axiom 11. $\texttt{(consp x)} = \texttt{t} \rightarrow \texttt{(cons (car x) (cdr x))} = \texttt{x}$

Axiom 12. $\texttt{(consp x)} = \texttt{nil} \rightarrow \texttt{(car x)} = \texttt{nil}$

Axiom 13. $\texttt{(consp x)} = \texttt{nil} \rightarrow \texttt{(cdr x)} = \texttt{nil}$

Axioms, continued

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 propositional tautologies, reason by cases, substitute equals for equals.

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

Example Proof Using Axioms and the Deduction Law

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.

$$\texttt{(consp x)} = \texttt{t} \wedge \texttt{x} = \texttt{(car z)} \rightarrow \texttt{(consp (car z))} \neq \texttt{nil}$$

Proof.

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

Use of the Rule of Instantiation

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)
```

Function Definition Adds An Axiom

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))
```

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.

Terms as Formulas, continued

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`.

Abuse of Terminology

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
$\texttt{(and p q)} \neq \texttt{nil} \leftrightarrow (\texttt{p} \neq \texttt{nil}) \wedge (\texttt{q} \neq \texttt{nil})$.

Problem 19.
Prove
$\texttt{(or p q)} \neq \texttt{nil} \leftrightarrow (\texttt{p} \neq \texttt{nil}) \vee (\texttt{q} \neq \texttt{nil})$.

Problem 20.
Prove
$\texttt{(implies p q)} \neq \texttt{nil} \leftrightarrow (\texttt{p} \neq \texttt{nil}) \rightarrow (\texttt{q} \neq \texttt{nil})$.

Problem 21.
Prove
$\texttt{(iff p q)} \neq \texttt{nil} \leftrightarrow (\texttt{p} \neq \texttt{nil}) \leftrightarrow (\texttt{q} \neq \texttt{nil})$.

Problem 22.
Prove
$\texttt{(equal x y)} \neq \texttt{nil} \leftrightarrow (\texttt{x} = \texttt{y})$

Use of Only Terms

These theorems allow changing the propositional functions into their logical counterparts as we move the "$\neq$ `nil`" into the term. We can always drop a "$\neq$ `nil`" anywhere it occurs in a formula.

Problem 23.
Using the theorems above, prove that

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

is equivalent to

$$(\texttt{p} \wedge (\texttt{q} \rightarrow \texttt{r})) \rightarrow \texttt{s}$$

which is equivalent to

$$((\texttt{p} \wedge \neg\ \texttt{q}) \rightarrow \texttt{s}) \wedge ((\texttt{p} \wedge \texttt{q} \wedge \texttt{r}) \rightarrow \texttt{s})$$

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)))
```

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)`.[1]

Problem 29.
Suppose we define

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

Prove `(equal t nil)`.

[1] 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.

Definitions, Revisited

The problems on the previous slides 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!

The restrictions we'll impose will prevent us from defining many useful functions but guarantee that we don't *ruin* the logic with "definitions" like those shown above.

ACL2 is much more generous in its restrictions but they are spiritually similar: both here and in ACL2 the restrictions on definitions will guarantee that every defined function terminates.

We do not explain in this document why these restrictions suffice.

Definitions, Revisited again

One way to make sure a function terminates is to insist that there is an argument that is being car'd and/or cdr'd at least once every time the function recurs and that before it recurs the definition tests that the argument is a cons-pair.

For example, `(defun f (x) (not (f x)))` is disallowed by this restriction.

And, `(defun f (x) (not (f (cdr x))))` is also disallowed.

But

```
(defun f (x)
  (if (consp x)
      (and (not (f (car x)))
           (not (f (cdr (cdr x)))))
      t))
```

is allowed because `x` is car'd and/or cdr'd in every recursion and the function tests `(consp x)` before recurring.

We say `(consp x)` "rules" the two recursive calls above.

Simplified Definitional Principle

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`.

The idea in this next definition is to take a term β and a particular occurrence r of some subterm in β and define the set of tests that rule r.

Then, if you have a function definition like `(defun` f `(`$v_1 \ldots v_k$`)` β`)` you can let r be a particular recursive call of f in β and then determine what tests rule that call.

The *rulers* of an occurrence of a term r in another term β is the set defined as follows:

1. if β is `(if` p x y`)` and r is in x, then the rulers of r in β is the set obtained by adding p to the set of rulers of r in x;
2. if β is `(if` p x y`)` and r is in y, then the rulers of r in β is the set obtained by adding `(NOT` p`)` to the set of rulers of r in y;
3. otherwise, the rulers of r in β is the empty set.

Simplified Definitional Principle, continued

Thus, in the term `(if a (if b (h c) (h d)) (g c))`, both `a` and `b` rule the first occurrence of `c` and the occurrence of `(h c)`.

In addition, `a` and `(not b)` rule the occurrences of `d` and `(h d)`. Finally, `(not a)` rules the second occurrence of `c` and `(g c)`.

Note that our definition of "rulers" does not include every test that has to be true in order to reach the occurrence in question.

For example, `p` does not rule the occurrence of `a` in `(car (if p a b))` even though the only way evaluation can reach `a` is if `p` is true.

The rulers of the occurrence of `a` in that term is the empty set, because that term is not a call of `if`.

However, `p` does rule the occurrence of `a` in the equivalent term `(if p (car a) (car b))`.

The reason we've defined rulers this way has to do with heuristics in the ACL2 theorem prover.

Simplified Definitional Principle, continued

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) rules 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))
```

Problems

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))
```

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.

For now, we willl use a much simpler principle.

A substitution σ is a `car`/`cdr` *substitution* on `x` if the binding (image) of `x` under σ is a `car`/`cdr` nest around `x`.

The other bindings of σ are unrestricted. For example, σ = $\{$`x` $\leftarrow$ `(car x)`, `y` $\leftarrow$ `(cons (cdr x) y)`$\}$ is a `car`/`cdr` substitution on `x`.

Principle of Structural Induction

Principle of Structural Induction: Let ψ be the term representing a conjecture. ψ may be proved by selecting an "induction" variable `x`, selecting a set of `car`/`cdr` substitutions on `x` $\sigma_1, \ldots, \sigma_n$, and by proving the following subgoals:

Base Case:

```
(implies (not (consp x))
         ψ)
```

and

Induction Step:

```
(implies (and (consp x)              ; test
              ψ/σ1                   ; induction hypothesis 1
              ⋮
              ψ/σn)                  ; induction hypothesis n
         ψ)                          ; induction conclusion
```

Here is an example Induction Step.

```
(implies (and (consp x)
              ψ/{x ← (car x), y ← (app x y)}
              ψ/{x ← (cdr (cdr x)), y ← (cons x y)}
              ψ/{x ← (cdr (cdr x)), y ← y})
         ψ)
```

Principle of Structural Induction

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)`.

Example Structural Induction 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.

Example Structural Induction Proof, continued

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.

Example Structural Induction Proof, a closer look

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}
```

Example Structural Induction Proof, a closer look

```
(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`.

Problems

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)
```

Problems, continued

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 `nils`, `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))
```

Total Ordering

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))
```

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.

Peano Arithmetic Problems, continued

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)))
```

Peano Arithmetic Problems, continued

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)))
```

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.

ACL2 Arithmetic, continued

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`

ACL2 Arithmetic, continued

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 n^{th} element of a list, treating n as a natural.

Note: this definition is unacceptable under our current Principle of Structural Recursion because (consp x) does not rule the recursive call. We will return to this point.)

```
(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.)

ACL2 Arithmetic, continued

The numbers are axiomatized with the standard axioms for rational fields.

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.

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.

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.

Problems

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 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.

Discussion

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.

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.

$$\omega \times 2 \qquad ||||| \ldots ||||| \ldots$$

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.

The Ordinals, continued

ω^2 $\quad$ ||||| ... ||||| ... ||||| ... ||||| ... ||||| ... ...

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.

The Ordinals, continued

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

The Ordinals, continued

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

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.

The Ordinals, Recognizer

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 Ordinals, Less Than

The function o< is the "less than" relation on ordinals. We show its definition on the next slides.

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.

The Ordinals, Less Than

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.

Problems about Ordinals, continued

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.

The theorem that `o<` is well-founded is a "meta-theorem", it can be proved about the ACL2 system but it cannot be proved within the ACL2 system.

A Note About the Ordinals

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 the online documentation topics `ordinals` and `o-p` in the ACL2 documentation; this documentation can be found starting at the ACL2 homepage from which some of the examples above have been chosen.

The Definitional Principle

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

The definition

$$\texttt{(defun } f \texttt{ } (v_1 \ldots v_n) \texttt{ } \beta\texttt{)}$$

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 (the *measure*) such that the following are theorems:
   - *Ordinal Conjecture*
     `(o-p` m`)`
   - *Measure Conjecture(s)* For each recursive call of $(f a_1 \ldots a_n)$ in β and the conjunction q of tests ruling 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:

$(f v_1 \ldots v_n)$ = β.

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

Problems

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.

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, $q_1, \ldots q_k$, 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`))` .

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.

Relations Between Recursion and Induction, continued

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.

Relations Between Recursion and Induction: the Key Insight

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.

Relations Between Recursion and Induction, continued

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
```

Relations Between Recursion and Induction, continued

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)))
```

Relations Between Recursion and Induction, continued

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))
```

Relations Between Recursion and Induction, continued

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` $\leftarrow$ `(d1 x a), a` $\leftarrow$ `(a1 x a)`$\}$
$\sigma_{1,2}$	$\{$`x` $\leftarrow$ `(d2 x a), a` $\leftarrow$ `(a2 x a)`$\}$
$\sigma_{2,1}$	$\{$`x` $\leftarrow$ `(d3 x a), a` $\leftarrow$ `(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.

Relations Between Recursion and Induction, continued

The measure conjectures required by the Induction Principle for this choice of parameters are the exactly 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.

> *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))`.

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)`.

Relations Between Recursion and Induction, continued

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.

Relations Between Recursion and Induction, continued

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.

Relations Between Recursion and Induction, continued

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.

Relations Between Recursion and Induction, continued

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)
           (< 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)`.

Exercises That Relate Recursion to Induction

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.