ACL2

by Matt Kaufmann and J Strother Moore

ACL2 Version 1.8

December 1995

This is documentation for ACL2 Version 1.8

Published by Computational Logic, Inc.

1717 West Sixth Street, Suite 290,
Austin, TX 78703 USA

NOTE: This is a paper version of the documentation for ACL2 Version 1.8. Although this documentation is best read with an appropriate hypertext browser, the paper version should serve as a reference manual. If you are using the paper version as an introduction to ACL2, please start with the chapter entitled "ACL2-TUTORIAL."

In this paper version, "links" to topics will be implicitly indicated with the mark ^. For example, you may see books^ — this indicates that there is documentation present on books. The table of contents, as well as the index, will show you where to find the topic named "BOOKS".

1 ACL2-TUTORIAL

ACL2-TUTORIAL tutorial introduction to ACL2

This section contains a tutorial introduction^ to ACL2, some examples of the use of ACL2, and pointers to additional information.

* INTRODUCTION:: introduction to ACL2
* STARTUP:: How to start using ACL2; the ACL2 command^ loop
* TIDBITS:: some basic hints for using ACL2
* TIPS:: some hints for using the ACL2 prover
* TUTORIAL-EXAMPLES:: examples of ACL2 usage

You might also find CLI Technical Report 101 helpful for a high-level view of the design goals of ACL2.

If you are already familiar with Nqthm, see Section 5.12 [NQTHM-TO-ACL2], page 148 for help in making the transition from Nqthm to ACL2.

If you would like more familiarity with Nqthm, we suggest CLI Technical Report 100, which works through a non-trivial example. A short version of that paper, which is entitled "Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem," is to appear in the Journal of Automated Reasoning's special issue on induction, probably in 1995 or 1996. Readers may well find that this paper indirectly imparts useful information about the effective use of ACL2.

1.1 INTRODUCTION

INTRODUCTION introduction to ACL2

This section contains introductory material on ACL2 including what ACL2 is, how to get started using the system, how to read the output, and other introductory topics. It was written almost entirely by Bill Young of Computational Logic, Inc.

You might also find CLI Technical Report 101 helpful, especially if you are familiar with Nqthm. If you would like more familiarity with Nqthm, we suggest CLI Technical Report 100.

OVERVIEW

ACL2 is an automated reasoning system under development at Computational Logic, Inc. It is the successor to the Nqthm (or Boyer-Moore) logic and proof system and its Pc-Nqthm interactive enhancement. The acronym ACL2 actually stands for “A Computational Logic for Applicative Common Lisp”. This title suggests several distinct but related aspects of ACL2.

We assume that readers of the ACL2 documentation^ have at least a very slight familiarity with some Lisp-like language. We will address the issue of prerequisites further, in “ABOUT THIS TUTORIAL” below.

As a **logic**, ACL2 is a formal system with rigorously defined syntax and semantics. In mathematical parlance, the ACL2 logic is a first-order logic of total recursive functions providing mathematical induction on the ordinals up to epsilon-0 and two extension principles: one for recursive definition and one for constrained introduction of new function symbols, here called encapsulation. The syntax of ACL2 is that of Common Lisp; ACL2 specifications are “also” Common Lisp programs in a way that we will make clear later. In less formal language, the ACL2 logic is an integrated collection of rules for defining (or axiomatizing) recursive functions, stating properties of those functions, and rigorously establishing those properties. Each of these activities is mechanically supported.

As a **specification language**, ACL2 supports modeling of systems of various kinds. An ACL2 function can equally be used to express purely formal relationships among mathematical entities, to describe algorithms, or to capture the intended behavior of digital systems. For digital systems, an ACL2 specification is a mathematical **model** that is intended to formalize relevant aspects of system behavior. Just as physics allows us to model the behavior of continuous physical systems, ACL2 allows us to model digital systems, including many with physical realizations such as computer hardware. As early as the 1930's Church, Kleene, Turing and others established that recursive functions provide an expressive formalism for modeling digital computation. Digital computation should be understood in a broad sense, covering a wide variety of activities including almost any systematic or algorithmic activity, or activity that can be reasonably approximated in that way. This ranges from the behavior of a digital circuit to the behavior of a programming language compiler to the behavior of a controller for a physical system (as long as the system can be adequately modeled discretely). All of these have been modeled using ACL2 or its predecessor Nqthm.

ACL2 is a **computational** logic in at least three distinct senses. First, the theory of recursive functions is often considered the mathematics of computation. Church conjectured that any “effective computation” can be modeled as a recursive function. Thus, ACL2 provides an expressive language for modeling digital systems. Second, many ACL2 specifications are executable. In fact, recursive functions written in ACL2 **are** Common Lisp functions that can be submitted to any compliant Common Lisp compiler and executed (in an environment where suitable ACL2-specific macros and functions are defined). Third, ACL2 is computational in the sense that calculation is heavily integrated into the reasoning process. Thus, an expression with explicit constant values but no free variables can be simplified by calculation rather than by complex logical manipulations.

ACL2 is a powerful, automated **theorem prover** or proof checker. This means that a competent user can utilize the ACL2 system to discover proofs of theorems stated in the ACL2 logic or to check previously discovered proofs. The basic deductive steps in an ACL2-checked proof are often quite large, due to the sophisticated combination of decision procedures, conditional rewriting, mathematical and structural induction, propositional simplification, and complex heuristics to orchestrate the interactions of these capabilities. Unlike some automated proof systems, ACL2 does not produce a formal proof. However, we believe that if ACL2 certifies the "theoremhood" of a given conjecture, then such a formal proof exists and, therefore, the theorem is valid. The ultimate result of an ACL2 proof session is a collection of "events^," possibly grouped into "books^," that can be replayed in ACL2. Therefore, a proof can be independently validated by any ACL2 user.

ACL2 may be used in purely automated mode in the shallow sense that conjectures are submitted to the prover and the user does not interact with the proof attempt (except possibly to stop it) until the proof succeeds or fails. However, any non-trivial proof attempt is actually interactive, since successful proof "events^" influence the subsequent behavior of the prover. For example, proving a lemma may introduce a rule that subsequently is used automatically by the prover. Thus, any realistic proof attempt, even in "automatic" mode, is really an interactive dialogue with the prover to craft a sequence of events^ building an appropriate theory and proof rules leading up to the proof of the desired result. Also, ACL2 supports annotating a theorem with "hints^" designed to guide the proof attempt. By supplying appropriate hints^, the user can suggest proof strategies that the prover would not discover automatically. There is a "proof-tree^" facility (see Chapter 12 [PROOF-TREE], page 579) that allows the user to monitor^ the progress and structure of a proof attempt in real-time. Exploring failed proof attempts is actually where heavy-duty ACL2 users spend most of their time.

ACL2 can also be used in a more explicitly interactive mode. The "proof-checker^" subsystem of ACL2 allows exploration of a proof on a fairly low level including expanding calls of selected function symbols, invoking specific rewrite^ rules, and selectively navigating around the proof. This facility can be used to gain sufficient insight into the proof to construct an automatic version, or to generate a detailed interactive-style proof that can be replayed in batch mode.

Because ACL2 is all of these things — computational logic, specification language, programming^ system, and theorem prover — it is more than the sum of its parts. The careful integration of these diverse aspects has produced a versatile automated reasoning system suitable for building highly reliable digital systems. In the remainder of this tutorial, we will illustrate some simple uses of this automated reasoning system.

ABOUT THIS TUTORIAL

ACL2 is a complex system with a vast array of features, bells and whistles. However, it is possible to perform productive work with the system using only a small portion of the available functionality. The goals of this tutorial are to:

> familiarize the new user with the most basic features of and modes of interaction with ACL2;
>
> familiarize her with the form of output of the system; and
>
> work through a graduated series of examples.

The more knowledge the user brings to this system, the easier it will be to become proficient. On one extreme: the **ideal** user of ACL2 is an expert Common Lisp programmer, has deep understanding of automated reasoning, and is intimately familiar with the earlier Nqthm system. Such ideal users are unlikely to need this tutorial. However, without some background knowledge, the beginning user is likely to become extremely confused and frustrated by this system. We suggest that a new user of ACL2 should:

> (a) have a little familiarity with Lisp, including basic Lisp programming and prefix notation (a Lisp reference manual such as Guy Steele's “Common Lisp: The Language” is also helpful);
>
> (b) be convinced of the utility of formal modeling; and
>
> (c) be willing to gain familiarity with basic automated theorem proving topics such as rewriting and algebraic simplification.

We will not assume any deep familiarity with Nqthm (the so-called “Boyer-Moore Theorem Prover”), though the book “A Computational Logic Handbook” by Boyer and Moore (Academic Press, 1988) is an extremely useful reference for many of the topics required to become a competent ACL2 user. We'll refer to it as ACLH below.

As we said in the introduction, ACL2 has various facets. For example, it can be used as a Common Lisp programming^ system to construct application programs. In fact, the ACL2 system itself is a large Common Lisp program constructed almost entirely within ACL2. Another use of ACL2 is as a specification and modeling tool. That is the aspect we will concentrate on in the remainder of this tutorial.

GETTING STARTED

This section is an abridged version of what's available elsewhere; feel free to see Section 1.2 [STARTUP], page 16 for more details.

How you start ACL2 will be system dependent, but you'll probably type something like "acl2" at your operating system prompt. Consult your system administrator for details.

When you start up ACL2, you'll probably find yourself inside the ACL2 command^ loop, as indicated by the following prompt^.

```
ACL2 !>
```

If not, you should type `(LP)`. See Section 8.63 [LP], page 327, which has a lot more information about the ACL2 command^ loop.

There are two "modes" for using ACL2, `:logic`^ and `:program`^. When you begin ACL2, you will ordinarily be in the `:logic`^ mode. This means that any new function defined is not only executable but also is axiomatically defined in the ACL2 logic. (See Section 8.18 [DEFUN-MODE], page 256 and see Section 8.16 [DEFAULT-DEFUN-MODE], page 254.) Roughly speaking, `:program`^ mode is available for using ACL2 as a programming^ language without some of the logical burdens necessary for formal reasoning. In this tutorial we will assume that we always remain in `:logic`^ mode and that our purpose is to write formal models of digital systems and to reason about them.

Now, within the ACL2 command^ loop you can carry out various kinds of activities, including the folllowing. (We'll see examples later of many of these.)

define new functions (see Section 6.16 [DEFUN], page 169);

execute functions on concrete data;

pose and attempt to prove conjectures about previously defined functions (see Section 6.15 [DEFTHM], page 168);

query the ACL2 "world^" or database (e.g., see Section 7.7 [PE], page 209); and

numerous other things.

In addition, there is extensive on-line documentation^, of which this tutorial introduction is a part.

INTERACTING WITH ACL2

The standard means of interacting with ACL2 is to submit a sequence of forms for processing by the ACL2 system. These forms are checked for syntactic and semantic acceptability and appropriately processed by the system. These forms can be typed directly at the ACL2 prompt^. However, most successful ACL2 users prefer to do their work using the Emacs text editor, maintaining an Emacs "working" buffer in which forms are edited. Those forms are then copied to the ACL2 interaction buffer, which is often the `"*shell*"` buffer.

In some cases, processing succeeds and makes some change to the ACL2 "logical world^," which affects the processing of subsequent forms. How can this processing fail? For example, a proposed theorem will be rejected unless all function symbols mentioned have been previously defined. Also the ability of ACL2 to discover the proof of a theorem may depend on the user previously having proved other theorems. Thus, the order in which forms are submitted to ACL2 is quite important. Maintaining forms in an appropriate order in your working buffer will be helpful for re-playing the proof later.

One of the most common events^ in constructing a model is introducing new functions. New functions are usually introduced using the `defun^` form; we'll encounter some exceptions later. Proposed function definitions are checked to make sure that they are syntactically and semantically acceptable (e.g., that all mentioned functions have been previously defined) and, for recursive functions, that their recursive calls **terminate**. A recursive function definition is guaranteed to terminate if there is some some "measure" of the arguments and a "well-founded" ordering such that the arguments to the function get smaller in each recursive call. See Section 14.17 [WELL-FOUNDED-RELATION], page 662.

For example, suppose that we need a function that will append two lists together. (We already have one in the ACL2 `append^` function; but suppose perversely that we decide to define our own.) Suppose we submit the following definition (you should do so as well and study the system output):

```
(defun my-app (x y)
  (if (atom x)
      y
    (cons (car x) (my-app x y))))
```

The system responds with the following message:

```
ACL2 Error in ( DEFUN MY-APP ...):  No :MEASURE was supplied with
the definition of MY-APP.  Our heuristics for guessing one have not
made any suggestions.  No argument of the function is tested along
every branch and occurs as a proper subterm at the same argument
position in every recursive call.  You must specify a :MEASURE.  See
:DOC defun.
```

This means that the system could not find an expression involving the formal parameters x and y that decreases under some well-founded order in every recursive call (there is only one such call). It should be clear that there is no such measure in this case because the only recursive call doesn't change the arguments at all. The definition is obviously flawed; if it were accepted and executed it would loop forever. Notice that a definition that is rejected is not stored in the system database; there is no need to take any action to have it "thrown away." Let's try again with the correct definition. The interaction now looks like (we're also putting in the ACL2 prompt^; you don't type that):

```
ACL2 !>(defun my-app (x y)
         (if (atom x)
             y
           (cons (car x) (my-app (cdr x) y))))

The admission of MY-APP is trivial, using the relation E0-ORD-<
(which is known to be well-founded on the domain recognized by
E0-ORDINALP) and the measure (ACL2-COUNT X).  We observe that the
type of MY-APP is described by the theorem
(OR (CONSP (MY-APP X Y)) (EQUAL (MY-APP X Y) Y)).
We used primitive type reasoning.

Summary
Form:  ( DEFUN MY-APP ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.07 seconds (prove: 0.00, print: 0.00, other: 0.07)
MY-APP
```

Notice that this time the function definition was accepted. We didn't have to supply a measure explicitly; the system inferred one from the form of the definition. On complex functions it may be necessary to supply a measure explicitly. (See Section 8.91 [XARGS], page 369.)

The system output provides several pieces of information.

> The revised definition is acceptable. The system realized that there is a particular measure (namely, `(acl2-count x)`) and a well-founded relation (`e0-ord-<`) under which the arguments of `my-app` get smaller in recursion. Actually, the theorem prover proved several theorems to admit `my-app`. The main one was that when `(atom x)` is false the `acl2-count` of `(cdr x)` is less than (in the `e0-ord-<` sense) the `acl2-count` of `x`. `Acl2-count^` is the most commonly used measure of the "size" of an ACL2 object. `E0-ord-<^` is the ordering relation on ordinals less than epsilon-0. On the natural numbers it is just ordinary "<".
>
> The observation printed about "the type of MY-APP" means that calls of the function `my-app` will always return a value that is either a cons^ pair or is equal to the second parameter.
>
> The summary provides information about which previously introduced definitions and lemmas were used in this proof, about some notable things to watch out for (the Warnings), and about how long this event took to process.

Usually, it's not important to read this information. However, it is a good habit to scan it briefly to see if the type information is surprising to you or if there are Warnings. We'll see an example of them later.

After a function is accepted, it is stored in the database and available for use in other function definitions or lemmas. To see the definition of any function use the `:pe^` command (see Section 7.7 [PE], page 209). For example,

```
ACL2 !>:pe my-app
 L       73:x(DEFUN MY-APP (X Y)
                    (IF (ATOM X)
                        Y (CONS (CAR X) (MY-APP (CDR X) Y))))
```

This displays the definition along with some other relevant information. In this case, we know that this definition was processed in `:logic^` mode (the "L") and was the 73rd command^ processed in the current session.

We can also try out our newly defined function on some sample data. To do that, just submit a form to be evaluated to ACL2. For example,

```
ACL2 !>(my-app '(0 1 2) '(3 4 5))
(0 1 2 3 4 5)
ACL2 !>(my-app nil nil)
```

```
NIL
ACL2 !>
```

Now suppose we want to prove something about the function just introduced. We conjecture, for example, that the length of the append^ of two lists is the sum of their lengths. We can formulate this conjecture in the form of the following ACL2 `defthm^` form.

```
(defthm my-app-length
  (equal (len (my-app x y))
         (+ (len x) (len y))))
```

First of all, how did we know about the functions `len` and `+^`, etc.? The answer to that is somewhat unsatisfying — we know them from our past experience in using Common Lisp and ACL2. It's hard to know that a function such as `len` exists without first knowing some Common Lisp. If we'd guessed that the appropriate function was called `length^` (say, from our knowledge of Lisp) and tried `:pe length`, we would have seen that `length^` is defined in terms of `len`, and we could have explored from there. Luckily, you can write a lot of ACL2 functions without knowing too many of the primitive functions.

Secondly, why don't we need some "type" hypotheses? Does it make sense to append things that are not lists? Well, yes. ACL2 and Lisp are both quite weakly typed. For example, inspection of the definition of `my-app` shows that if `x` is not a cons^ pair, then `(my-app x y)` always returns `y`, no matter what `y` is.

Thirdly, would it matter if we rewrote the lemma with the equality reversed, as follows?

```
(defthm my-app-length2
  (equal (+ (len x) (len y))
         (len (my-app x y)))).
```

The two are **logically** equivalent, but...yes, it would make a big difference. Recall our remark that a lemma is not only a "fact" to be proved; it also is used by the system to prove other later lemmas. The current lemma would be stored as a rewrite^ rule. (See Chapter 14 [RULE-CLASSES], page 617.) For a rewrite^ rule, a conclusion of the form `(EQUAL LHS RHS)` means to replace instances of the `LHS` by the appropriate instance of the `RHS`. Presumably, it's better to rewrite^ (len (my-app x y)) to (+ (len x) (len y)) than the other way around. The reason is that the system "knows" more about +^ than it does about the new function symbol my-app.

So let's see if we can prove this lemma. Submitting our preferred defthm^ to ACL2 (do it!), we get the following interaction:

```
          --------------------------------------------------
ACL2 !>(defthm my-app-length
  (equal (len (my-app x y))
         (+ (len x) (len y))))

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  These merge into two derived
induction schemes.  However, one of these is flawed and so we are
left with one viable candidate.

We will induct according to a scheme suggested by (LEN X), but
modified to accommodate (MY-APP X Y).  If we let (:P X Y) denote *1
above then the induction scheme we'll use is
(AND (IMPLIES (NOT (CONSP X)) (:P X Y))
     (IMPLIES (AND (CONSP X) (:P (CDR X) Y))
              (:P X Y))).
This induction is justified by the same argument used to admit LEN,
namely, the measure (ACL2-COUNT X) is decreasing according to the
relation E0-ORD-< (which is known to be well-founded on the domain
recognized by E0-ORDINALP).  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 (LEN (MY-APP X Y))
                (+ (LEN X) (LEN Y)))).

But simplification reduces this to T, using the :definitions of FIX,
LEN and MY-APP, the :type-prescription rule LEN, the :rewrite rule
UNICITY-OF-0 and primitive type reasoning.

Subgoal *1/1
(IMPLIES (AND (CONSP X)
              (EQUAL (LEN (MY-APP (CDR X) Y))
                     (+ (LEN (CDR X)) (LEN Y))))
         (EQUAL (LEN (MY-APP X Y))
                (+ (LEN X) (LEN Y)))).
```

```
This simplifies, using the :definitions of LEN and MY-APP, primitive
type reasoning and the :rewrite rules COMMUTATIVITY-OF-+ and
CDR-CONS, to

Subgoal *1/1'
(IMPLIES (AND (CONSP X)
              (EQUAL (LEN (MY-APP (CDR X) Y))
                     (+ (LEN Y) (LEN (CDR X)))))
         (EQUAL (+ 1 (LEN (MY-APP (CDR X) Y)))
                (+ (LEN Y) 1 (LEN (CDR X))))).

But simplification reduces this to T, using linear arithmetic,
primitive type reasoning and the :type-prescription rule LEN.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM MY-APP-LENGTH ...)
Rules: ((:REWRITE UNICITY-OF-0)
        (:DEFINITION FIX)
        (:REWRITE COMMUTATIVITY-OF-+)
        (:DEFINITION LEN)
        (:REWRITE CDR-CONS)
        (:DEFINITION MY-APP)
        (:TYPE-PRESCRIPTION LEN)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:FAKE-RUNE-FOR-LINEAR NIL))
Warnings:  None
Time:  0.30 seconds (prove: 0.13, print: 0.05, other: 0.12)
 MY-APP-LENGTH
          --------------------------------------------------
```

Wow, it worked! In brief, the system first tried to rewrite^ and simplify as much as possible. Nothing changed; we know that because it said "Name the formula above *1." Whenever the system decides to name a formula in this way, we know that it has run out of techniques to use other than proof by induction.

The induction performed by ACL2 is structural or "Noetherian" induction. You don't need to know much about that except that it is induction based on the structure of some object. The heuristics infer the structure of the object from the way the object is recursively decomposed by the functions used in the conjecture. The heuristics of ACL2 are reasonably good at selecting an induction scheme in simple cases. It is possible to override the heuristic choice by providing an `:induction` hint (see Section 8.41 [HINTS], page 297). In the case of the theorem above, the system inducts on the structure of `x` as suggested by the decomposition of `x` in both `(my-app x y)` and `(len x)`. In the base case, we assume that `x` is not a `consp^`. In the inductive case, we assume that it is a `consp^` and assume that the conjecture holds for `(cdr x)`.

There is a close connection between the analysis that goes on when a function like `my-app` is accepted and when we try to prove something inductively about it. That connection is spelled out well in Boyer and Moore's book "A Computational Logic," if you'd like to look it up. But it's pretty intuitive. We accepted `my-app` because the "size" of the first argument `x` decreases in the recursive call. That tells us that when we need to prove something inductively about `my-app`, it's a good idea to try an induction on the size of the first argument. Of course, when you have a theorem involving several functions, it may be necessary to concoct a more complicated induction^ schema, taking several of them into account. That's what's meant by "merging" the induction schemas.

The proof involves two cases: the base case, and the inductive case. You'll notice that the subgoal numbers go **down** rather than up, so you always know how many subgoals are left to process. The base case (`Subgoal *1/2`) is handled by opening up the function definitions, simplifying, doing a little rewriting, and performing some reasoning based on the types of the arguments. You'll often encounter references to system defined lemmas (like `unicity-of-0`). You can always look at those with `:pe^`; but, in general, assume that there's a lot of simplification power under the hood that's not too important to understand fully.

The inductive case (`Subgoal *1/1`) is also dispatched pretty easily. Here we assume the conjecture true for the `cdr^` of the list and try to prove it for the entire list. Notice that the prover does some simplification and then prints out an updated version of the goal (`Subgoal *1/1'`). Examining these gives you a pretty good idea of what's going on in the proof.

Sometimes one goal is split into a number of subgoals, as happened with the induction above. Sometimes after some initial processing the prover decides it needs to prove a subgoal by induction; this subgoal is given a name and pushed onto a stack of goals. Some steps, like generalization (see ACLH), are not necessarily validity preserving; that is, the system may adopt a false subgoal while trying to prove a true one. (Note that this is ok in the sense that it is not "unsound." The system will fail in its attempt to establish the false subgoal and the main proof attempt will fail.) As you gain facility with using the prover, you'll get pretty good at recognizing what to look for when reading a proof script. The prover's proof-tree^ utility helps with monitoring an ongoing proof and jumping to designated locations in the proof (see Chapter 12 [PROOF-TREE], page 579). See Section 1.4 [TIPS], page 19 for a number of useful pointers on using the theorem prover effectively.

When the prover has successfully proved all subgoals, the proof is finished. As with a `defun^`, a summary of the proof is printed. This was an extremely simple proof, needing no additional guidance. More realistic examples typically require the user to look carefully at the failed proof log to find ways to influence the prover to do better on its next attempt. This means either: proving some rules that will then be available to the prover, changing the global state in ways that will affect the proof, or providing some hints^ locally that will influence the prover's behavior. Proving this lemma (`my-app-length`) is an example of the first. Since this is a rewrite^ rule, whenever in a later proof an instance of the form `(LEN (MY-APP X Y))` is encountered, it will be rewritten to the corresponding instance of `(+ (LEN X) (LEN Y))`. Disabling the rule by executing the command^

```
(in-theory (disable my-app-length)),
```

is an example of a global change to the behavior of the prover since this rewrite^ will not be performed subsequently (unless the rule is again enable^d). Finally, we can add a (local) disable^ "hint" to a `defthm^`, meaning to disable^ the lemma only in the proof of one or more subgoals. For example:

```
(defthm my-app-length-commutativity
  (equal (len (my-app x y))
         (len (my-app y x)))
  :hints (("Goal" :in-theory (disable my-app-length))))
```

In this case, the hint supplied is a bad idea since the proof is much harder with the hint than without it. Try it both ways.

By the way, to undo the previous event use `:u` (see Section 7.15 [U], page 216). To undo back to some earlier event use `:ubt` (see Section 7.16 [UBT], page 217). To view the current event use `:pe :here`. To list several events^ use `:pbt` (see Section 7.2 [PBT], page 205).

Notice the form of the hint in the previous example (see Section 8.41 [HINTS], page 297). It specifies a goal to which the hint applies. `"Goal"` refers to the top-level goal of the theorem. Subgoals are given unique names as they are generated. It may be useful to suggest that a function symbol be disable^d only for Subgoal 1.3.9, say, and a different function enable^d only on Subgoal 5.2.8. Overuse of such hints^ often suggests a poor global proof strategy.

We now recommend that you visit documentation^ on additional examples. See Section 1.5 [TUTORIAL-EXAMPLES], page 30.

1.2 STARTUP

STARTUP How to start using ACL2; the ACL2 command^ loop

When you start up ACL2, you'll probably find yourself inside the ACL2 command^ loop, as indicated by the following prompt^.

```
ACL2 !>
```

If not, you should type `(LP)`. See Section 8.63 [LP], page 327, which has a lot more information about the ACL2 command^ loop.

You should now be in ACL2. The current "default-defun-mode^" is `:logic^`; the other mode is `:program^`, which would cause the letter `p` to be printed in the prompt^. `:Logic^` means that any function we define is not only executable but also is axiomatically defined in the ACL2 logic. See Section 8.18 [DEFUN-MODE], page 256 and see Section 8.16 [DEFAULT-DEFUN-MODE], page 254. For example we can define a function `my-cons` as follows. (You may find it useful to start up ACL2 and submit this and other command^s below to the ACL2 command^ loop, as we won't include output below.)

```
ACL2 !>(defun my-cons (x y) (cons x y))
```

An easy theorem may then be proved: the `car^` of `(my-cons a b)` is A.

```
ACL2 !>(defthm car-my-cons (equal (car (my-cons a b)) a))
```

Notice that unlike Nqthm, the theorem command^ is `defthm^` rather than `prove-lemma`. See Section 6.15 [DEFTHM], page 168, which explains (among other things) that the default is to turn theorems into rewrite^ rules.

Various keyword commands are available to query the ACL2 "world^", or database. For example, we may view the definition of `my-cons` by invoking a command to print events^, as follows.

```
ACL2 !>:pe my-cons
```

Also see Section 7.7 [PE], page 209. We may also view all the lemmas that rewrite^ term^s whose top function symbol is `car^` by using the following command, whose output will refer to the lemma `car-my-cons` proved above.

```
ACL2 !>:pl car
```

Also see Section 7.10 [PL], page 211. Finally, we may print all the command^s back through the initial world^ as follows.

```
ACL2 !>:pbt 0
```

See Chapter 7 [HISTORY], page 203 for a list of commands, including these, for viewing the current ACL2 world^.

Continue with the documentation^ for tutorial-examples^ to see a simple but illustrative example in the use of ACL2 for reasoning about functions.

1.3 TIDBITS

TIDBITS some basic hints for using ACL2

See Chapter 3 [BOOKS], page 89 for a discussion of books. Briefly, a book is a file whose name ends in ".lisp" that contains ACL2 events^; see Chapter 6 [EVENTS], page 153.

See Chapter 7 [HISTORY], page 203 for a list of useful commands. Some examples:

```
:pbt :here      ; print the current event
:pbt (:here -3) ; print the last four events
:u              ; undo the last event
:pe append      ; print the definition of append
```

See Chapter 5 [DOCUMENTATION], page 129 to learn how to print documentation to the terminal. There are also versions of the documentation^ for Mosaic, Emacs Info, and hardcopy.

There are quite a few kinds of rules allowed in ACL2 besides `:rewrite^` rules, though we hope that beginners won't usually need to be aware of them. See Chapter 14 [RULE-CLASSES], page 617 for details. In particular, there is support for congruence^ rewriting. See Section 15.10 [RUNE], page 675 ("RUle NamE") for a description of the various kinds of rules in the system. Also see Chapter 15 [THEORIES], page 667 for a description of how to build theories^ of rune^s, which are often used in hints; see Section 8.41 [HINTS], page 297.

A "programming^ mode" is supported; see Section 6.25 [PROGRAM], page 186, see Section 8.18 [DEFUN-MODE], page 256, and see Section 8.16 [DEFAULT-DEFUN-MODE], page 254. It can be useful to prototype functions after executing the command `:program^`, which will cause definitions to be syntaxed-checked only.

ACL2 supports mutual recursion, though this feature is not tied into the automatic discovery of induction^ schemas and is often not the best way to proceed when you expect to be reasoning about the functions. See Section 8.20 [DEFUNS], page 261; also see Section 6.24 [MUTUAL-RECURSION], page 184.

See Section 9.7 [LD], page 380 for discussion of how to load files of events^. There are many options to `ld^`, including ones to suppress proofs and to control output.

The `:otf-flg^` (Onward Thru the Fog FLaG) is a useful feature that Nqthm users have often wished for. It prevents the prover from aborting a proof attempt and inducting on the original conjecture. See Section 8.67 [OTF-FLG], page 330.

ACL2 supports redefinition and redundancy in events^; see Section 8.56 [LD-REDEFINITION-ACTION], page 314 and see Section 8.76 [REDUNDANT-EVENTS], page 337.

A proof-tree^ display feature is available for use with Emacs. This feature provides a view of ACL2 proofs that can be much more useful than reading the stream of characters^ output by the theorem prover as its "proof." See Chapter 12 [PROOF-TREE], page 579.

An interactive feature similar to Pc-Nqthm is supported in ACL2. See Section 11.101 [VERIFY], page 577 and see Chapter 11 [PROOF-CHECKER], page 515.

ACL2 allows you to monitor^ the use of rewrite^ rules. See Chapter 4 [BREAK-REWRITE], page 113.

See Section 8.6 [ARRAYS], page 226 to read about applicative, fast arrays^ in ACL2.

To quit the ACL2 command^ loop, or (in akcl) to return to the ACL2 command^ loop after an interrupt, type `:q^`. To continue (resume) after an interrupt (in akcl), type `:r`. To cause an interrupt (in akcl under Unix (trademark of AT&T)), hit control-C (twice, if inside Emacs). To exit ACL2 altogether, first type `:q^` to exit the ACL2 command^ loop, and then exit Lisp (by typing `(user::bye)` in akcl).

See Section 8.81 [STATE], page 344 to read about the von Neumannesque ACL2 state^ object. Also see Section 9.1 [@], page 373, and see Section 9.3 [ASSIGN], page 378, to learn about reading and setting global state^ variables.

1.4 TIPS

TIPS some hints for using the ACL2 prover

We present here some tips for using ACL2 effectively. Though this collection is somewhat *ad hoc*, we try to provide some organization, albeit somewhat artificial: for example, the sections overlap, and no particular order is intended. This material has been adapted by Bill Young from a very similar list for Nqthm that appeared in the conclusion of: "Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem," by Matt Kaufmann and Paolo Pecchiari, CLI Technical Report 100, June, 1995. We also draw from a similar list in Chapter 13 of "A Computational Logic Handbook" by R.S. Boyer and J S. Moore (Academic Press, 1988). We'll refer to this as "ACLH" below.

These tips are organized roughly as follows.

A. ACL2 Basics

B. Strategies for creating events

C. Dealing with failed proofs

D. Performance tips

E. Miscellaneous tips and knowledge

F. Some things you DON'T need to know

ACL2 BASICS

A1. The ACL2 logic.

This is a logic of total functions. For example, if `A` and `B` are less than or equal to each other, then we need to know something more in order to conclude that they are equal (e.g., that they are numbers). This kind of twist is important in writing definitions; for example, if you expect a function to return a number, you may want to apply the function `fix^` or some variant (e.g., `nfix^` or `ifix^`) in case one of the formals is to be returned as the value.

ACL2's notion of ordinals is important on occasion in supplying "measure hints^" for the acceptance of recursive definitions. Be sure that your measure is really an ordinal. Consider the following example, which ACL2 fails to admit (as explained below).

```
(defun cnt (name a i x)
  (declare (xargs :measure (+ 1 i)))
  (cond ((zp (+ 1 i))
         0)
        ((equal x (aref1 name a i))
         (1+ (cnt name a (1- i) x)))
        (t (cnt name a (1- i) x))))
```

One might think that `(+ 1 i)` is a reasonable measure, since we know that `(+ 1 i)` is a positive integer in any recursive call of `cnt`, and positive integers are ACL2 ordinals (see Section 8.24 [E0-ORDINALP], page 263). However, the ACL2 logic requires that the measure be an ordinal unconditionally, not just under the governing assumptions that lead to recursive calls. An appropriate fix is to apply `nfix^` to `(+ 1 i)`, i.e., to use

```
(declare (xargs :measure (nfix (+ 1 i))))
```

in order to guarantee that the measure will always be an ordinal (in fact, a positive integer).

A2. Simplification.

The ACL2 simplifier is basically a rewriter, with some "linear^ arithmetic" thrown in. One needs to understand the notion of conditional rewriting. See Section 14.14 [REWRITE], page 657.

A3. Parsing of rewrite rules.

ACL2 parses rewrite^ rules roughly as explained in ACLH, *except* that it never creates "unusual" rule classes. In ACL2, if you want a `:linear^` rule, for example, you must specify `:linear^` in the `:rule-classes^`. See Chapter 14 [RULE-CLASSES], page 617, and also see Section 14.14 [REWRITE], page 657 and see Section 14.10 [LINEAR], page 643.

A4. Linear arithmetic.

On this subject, it should suffice to know that the prover can handle truths about `+^` and `-^`, and that linear^ rules (see above) are somehow "thrown in the pot" when the prover is doing such reasoning. Perhaps it's also useful to know that linear^ rules can have hypotheses, and that conditional rewriting is used to relieve those hypotheses.

A5. Events.

Over time, the expert ACL2 user will know some subtleties of its events^. For example, `in-theory^` events^ and hints^ are important, and they distinguish between a function and its executable counterpart.

B. STRATEGIES FOR CREATING EVENTS

In this section, we concentrate on the use of definitions and rewrite^ rules. There are quite a few kinds of rules allowed in ACL2 besides rewrite^ rules, though most beginning users probably won't usually need to be aware of them. See Chapter 14 [RULE-CLASSES], page 617 for details. In particular, there is support for congruence^ rewriting. Also see Section 15.10 [RUNE], page 675 ("RUle NamE") for a description of the various kinds of rules in the system.

B1. Use high-level strategy.

Decompose theorems into "manageable" lemmas (admittedly, experience helps here) that yield the main result "easily." It's important to be able to outline non-trivial proofs by hand (or in your head). In particular, avoid submitting goals to the prover when there's no reason to believe that the goal will be proved and there's no "sense" of how an induction argument would apply. It is often a good idea to avoid induction in complicated theorems unless you have a reason to believe that it is appropriate.

B2. Write elegant definitions.

Try to write definitions in a reasonably modular style, especially recursive ones. Think of ACL2 as a programming^ language whose procedures are definitions and lemmas, hence we are really suggesting that one follow good programming^ style (in order to avoid duplication of "code," for example).

When possible, complex functions are best written as compositions of simpler functions. The theorem prover generally performs better on primitive recursive functions than on more complicated recursions (such as those using accumulating parameters).

Avoid large non-recursive definitions which tend to lead to large case explosions. If such definitions are necessary, try to prove all relevant facts about the definitions and then disable^ them.

Whenever possible, avoid mutual recursion if you care to prove anything about your functions. The induction heuristics provide essentially no help with reasoning about mutually defined functions. Mutually recursive functions can usually be combined into a single function with a "flag" argument. (However, see Section 1.5.6.3 [MUTUAL-RECURSION-PROOF-EXAMPLE], page 65 for a small example of proof involving mutually recursive functions.)

B3. Look for analogies.

Sometimes you can easily edit sequences of lemmas into sequences of lemmas about analogous functions.

B4. Write useful rewrite rules.

As explained in A3 above, every rewrite^ rule is a directive to the theorem prover, usually to replace one term^ by another. The directive generated is determined by the syntax of the `defthm^`

submitted. Never submit a rewrite^ rule unless you have considered its interpretation as a proof directive.

B4a. Rewrite rules should simplify.

Try to write rewrite^ rules whose right-hand sides are in some sense "simpler than" (or at worst, are variants of) the left-hand sides. This will help to avoid infinite loops in the rewriter.

B4b. Avoid needlessly expensive rules.

Consider a rule whose conclusion's left-hand side (or, the entire conclusion) is a term^ such as `(consp x)` that matches many term^s encountered by the prover. If in addition the rule has complicated hypotheses, this rule could slow down the prover greatly. Consider switching the conclusion and a complicated hypothesis (negating each) in that case.

B4c. The "Knuth-Bendix problem".

Be aware that left sides of rewrite^ rules should match the "normalized forms", where "normalization" (rewriting) is inside out. Be sure to avoid the use of nonrecursive function symbols on left sides of rewrite^ rules, except when those function symbols are disable^d, because they tend to be expanded away before the rewriter would encounter an instance of the left side of the rule. Also assure that subexpressions on the left hand side of a rule are in simplified form.

B4d. Avoid proving useless rules.

Sometimes it's tempting to prove a rewrite^ rule even before you see how it might find application. If the rule seems clean and important, and not unduly expensive, that's probably fine, especially if it's not too hard to prove. But unless it's either part of the high-level strategy or, on the other hand, intended to get the prover past a particular unproved goal, it may simply waste your time to prove the rule, and then clutter the database of rules if you are successful.

B4e. State rules as strongly as possible, usually.

It's usually a good idea to state a rule in the strongest way possible, both by eliminating unnecessary hypotheses and by generalizing subexpressions to variables.

Advanced users may choose to violate this policy on occasion, for example in order to avoid slowing down the prover by excessive attempted application of the rule. However, it's a good rule of thumb to make the strongest rule possible, not only because it will then apply more often, but also because the rule will often be easier to prove (see also B6 below). New users are sometimes tempted to put in extra hypotheses that have a "type restriction" appearance, without realizing that the way ACL2 handles (total) functions generally lets it handle trivial cases easily.

B4f. Avoid circularity.

A stack overflow in a proof attempt almost always results from circular rewriting. Use `brr^` to investigate the stack; see Section 4.1 [BREAK-LEMMA], page 117. Because of the complex heuristics, it is not always easy to define just when a rewrite^ will cause circularity. See the very good discussion of this topic in ACLH.

See Section 4.1 [BREAK-LEMMA], page 117 for a trick involving use of the forms `brr t` and `(cw-gstack *deep-gstack* state)` for inspecting loops in the rewriter.

B4g. Remember restrictions on permutative rules.

Any rule that permutes the variables in its left hand side could cause circularity. For example, the following axiom is automatically supplied by the system:

```
(defaxiom commutativity-of-+
          (equal (+ x y) (+ y x))).
```

This would obviously lead to dangerous circular rewriting if such "permutative" rules were not governed by a further restriction. The restriction is that such rules will not produce a term^ that is "lexicographically larger than" the original term^ (see Section 8.62 [LOOP-STOPPER], page 324). However, this sometimes prevents intended rewrites. See Chapter 13 of ACLH for a discussion of this problem.

B5. Conditional vs. unconditional rewrite rules.

It's generally preferable to form unconditional rewrite^ rules unless there is a danger of case explosion. That is, rather than pairs of rules such as

```
(implies p
         (equal term1 term2))
```

and

```
(implies (not p)
         (equal term1 term3))
```

consider:

```
(equal term1
       (if p term2 term3))
```

However, sometimes this strategy can lead to case explosions: `IF^` term^s introduce cases in ACL2. Use your judgment. (On the subject of `IF^`: `COND^`, `CASE^`, `AND^`, and `OR^` are macros that abbreviate `IF^` forms, and propositional functions such as `IMPLIES^` quickly expand into `IF^` term^s.)

B6. Create elegant theorems.

Try to formulate lemmas that are as simple and general as possible. For example, sometimes properties about several functions can be "factored" into lemmas about one function at a time. Sometimes the elimination of unnecessary hypotheses makes the theorem easier to prove, as does generalizing first by hand.

B7. Use `defaxiom^`s temporarily to explore possibilities.

When there is a difficult goal that seems to follow immediately (by a `:use` hint or by rewriting) from some other lemmas, you can create those lemmas as `defaxiom`^ events^ (or, the application of `skip-proofs`^ to `defthm`^ events^) and then double-check that the difficult goal really does follow from them. Then you can go back and try to turn each `defaxiom`^ into a `defthm`^. When you do that, it's often useful to disable^ any additional rewrite^ rules that you prove in the process, so that the "difficult goal" will still be proved from its lemmas when the process is complete.

Better yet, rather than disabling rewrite^ rules, use the `local`^ mechanism offered by `encapsulate`^ to make temporary rules completely local^ to the problem at hand. See Section 6.18 [ENCAPSULATE], page 176 and see Section 6.21 [LOCAL], page 181.

B9. Use books.

Consider using previously certified books^, especially for arithmetic reasoning. This cuts down the duplication of effort and starts your specification and proof effort from a richer foundation. See the file `"doc/README"` in the ACL2 distribution for information on books^ that come with the system.

C. DEALING WITH FAILED PROOFS

C1. Look in proof output for goals that can't be further simplified.

Use the "proof-tree^" utility to explore the proof space. However, you don't need to use that tool to use the "checkpoint" strategy. The idea is to think of ACL2 as a "simplifier" that either proves the theorem or generates some goal to consider. That goal is the first "checkpoint," i.e., the first goal that does not further simplify. Exception: it's also important to look at the induction scheme in a proof by induction, and if induction seems appropriate, then look at the first checkpoint *after* the induction has begun.

Consider whether the goal on which you focus is even a theorem. Sometimes you can execute it for particular values to find a counterexample.

When looking at checkpoints, remember that you are looking for any reason at all to believe the goal is a theorem. So for example, sometimes there may be a contradiction in the hypotheses.

Don't be afraid to skip the first checkpoint if it doesn't seem very helpful. Also, be willing to look a few lines up or down from the checkpoint if you are stuck, bearing in mind however that this practice can be more distracting than helpful.

C2. Use the "break rewrite" facility.

`Brr^` and related utilities let you inspect the "rewrite stack." These can be valuable tools in large proof efforts. See Section 4.1 [BREAK-LEMMA], page 117 for an introduction to these tools, and see Chapter 4 [BREAK-REWRITE], page 113 for more complete information.

The break facility is especially helpful in showing you why a particular rewrite rule is not being applied.

C3. Use induction hints when necessary. Of course, if you can define your functions so that they suggest the correct inductions to ACL2, so much the better! But for complicated inductions, induction hints^ are crucial. See Section 8.41 [HINTS], page 297 for a description of `:induct` hints^.

C4. Use the "Proof Checker" to explore.

The `verify^` command supplied by ACL2 allows one to explore problem areas "by hand." However, even if you succeed in proving a conjecture with `verify^`, it is useful to prove it without using it, an activity that will often require the discovery of rewrite^ rules that will be useful in later proofs as well.

C5. Don't have too much patience.

Interrupt the prover fairly quickly when simplification isn't succeeding.

C6. Simplify rewrite rules.

When it looks difficult to relieve the hypotheses of an existing rewrite^ rule that "should" apply in a given setting, ask yourself if you can eliminate a hypothesis from the existing rewrite^ rule. If so, it may be easier to prove the new version from the old version (and some additional lemmas), rather than to start from scratch.

C7. Deal with base cases first.

Try getting past the base case(s) first in a difficult proof by induction. Usually they're easier than the inductive step(s), and rules developed in proving them can be useful in the inductive step(s) too. Moreover, it's pretty common that mistakes in the statement of a theorem show up in the base case(s) of its proof by induction.

C8. Use `:expand` hints. Consider giving `:expand` hints^. These are especially useful when a proof by induction is failing. It's almost always helpful to open up a recursively defined function that is supplying the induction scheme, but sometimes ACL2 is too timid to do so; or perhaps the function in question is disable^d.

D. PERFORMANCE TIPS

D1. Disable rules.

There are a number of instances when it is crucial to disable^ rules, including (often) those named explicitly in `:use` hints^. Also, disable^ recursively defined functions for which you can prove what seem to be all the relevant properties. The prover can spend significant time "behind the scenes" trying to open up recursively defined functions, where the only visible effect is slowness.

D2. Turn off the "break rewrite" facility. Remember to execute `:brr nil` after you've finished with the "break rewrite" utility (see Chapter 4 [BREAK-REWRITE], page 113), in order to bring the prover back up to full speed.

E. MISCELLANEOUS TIPS AND KNOWLEDGE

E1. Order of application of rewrite rules.

Keep in mind that the most recent rewrite^ rules in the history^ are tried first.

E2. Relieving hypotheses is not full-blown theorem proving.

Relieving hypotheses on rewrite^ rules is done by rewriting and linear^ arithmetic alone, not by case splitting or by other prover processes "below" simplification.

E3. "Free variables" in rewrite rules.
The set of "free variables" of a rewrite^ rule is defined to contain those variables occurring in the rule that do not occur in the left-hand side of the rule. It's often a good idea to avoid rules containing free variables because they are "weak," in the sense that hypotheses containing such variables can generally only be proved when they are "obviously" present in the current context. This weakness suggests that it's important to put the most "interesting" (specific) hypotheses about free variables first, so that the right instances are considered. For example, suppose you put a very general hypothesis such as `(consp x)` first. If the context has several term^s around that are known to be `consp^`s, then `x` may be bound to the wrong one of them.

E4. Obtaining information Use `:pl^ foo` to inspect rewrite^ rules whose left hand sides are applications of the function `foo`. Another approach to seeing which rewrite^ rules apply is to enter the proof-checker^ with `verify^`, and use the `show-rewrites` or `sr` command.

E5. Consider esoteric rules with care.

If you care to see Chapter 14 [RULE-CLASSES], page 617 and peruse the list of subtopics (which will be listed right there in most versions of this documentation^), you'll see that ACL2 supports a wide variety of rules in addition to :rewrite^ rules. Should you use them? This is a complex question that we are not ready to answer with any generality. Our general advice is to avoid relying on such rules as long as you doubt their utility. More specifically: be careful not to use conditional type prescription rules, as these have been known to bring ACL2 to its knees, unless you are conscious that you are doing so and have reason to believe that they are working well.

F. SOME THINGS YOU DON'T NEED TO KNOW

Most generally: you shouldn't usually need to be able to predict too much about ACL2's behavior. You should mainly just need to be able to react to it.

F1. Induction heuristics.

Although it is often important to read the part of the prover's output that gives the induction scheme chosen by the prover, it is not necessary to understand how the prover made that choice. (Granted, advanced users may occasionally gain minor insight from such knowledge. But it's truly minor in many cases.) What *is* important is to be able to tell it an appropriate induction when it doesn't pick the right one (after noticing that it doesn't). See C3 above.

F2. Heuristics for expanding calls of recursively defined functions.

As with the previous topic, the important thing isn't to understand these heuristics but, rather, to deal with cases where they don't seem to be working. That amounts to supplying `:expand` hints^ for those calls that you want opened up, which aren't. See also C8 above.

F3. The "waterfall".

As discussed many times already, a good strategy for using ACL2 is to look for checkpoints (goals stable under simplification) when a proof fails, perhaps using the proof-tree^ facility. Thus, it is reasonable to ignore almost all the prover output, and to avoid pondering the meaning of the other "processes" that ACL2 uses besides simplification (such as elimination, cross-fertilization, generalization, and elimination of irrelevance). For example, you don't need to worry about prover output that mentions "type reasoning" or "abbreviations," for example.

1.5 TUTORIAL-EXAMPLES

TUTORIAL-EXAMPLES examples of ACL2 usage

Beginning users may find these examples at least as useful as the extensive documentation^ on particular topics. We suggest that you read these in the following order:

Tutorial1-Towers-of-Hanoi^
Tutorial2-Eights-Problem^
Tutorial3-Phonebook-Example^
Tutorial4-Defun-Sk-Example^
Tutorial5-Miscellaneous-Examples^

You may also wish to visit the other introductory sections, startup^ and tidbits^. These contain further information related to the use of ACL2.

* SOLUTION-TO-SIMPLE-EXAMPLE:: solution to a simple example
* TUTORIAL1-TOWERS-OF-HANOI:: The Towers of Hanoi Example
* TUTORIAL2-EIGHTS-PROBLEM:: The Eights Problem Example
* TUTORIAL3-PHONEBOOK-EXAMPLE:: A Phonebook Specification
* TUTORIAL4-DEFUN-SK-EXAMPLE:: example of quantified notions
* TUTORIAL5-MISCELLANEOUS-EXAMPLES:: miscellaneous ACL2 examples

When you feel you have read enough examples, you might want to try the following very simple example on your own. First define the notion of the "fringe" of a tree, where we identify trees simply as cons^ structures, with atom^s at the leaves. For example:

```
ACL2 !>(fringe '((a . b) c . d))
(A B C D)
```

Next, define the notion of a "leaf" of a tree, i.e., a predicate `leaf-p` that is true of an atom if and only if that atom appears at the tip of the tree. Define this notion without referencing the function `fringe`. Finally, prove the following theorem, whose proof may well be automatic (i.e., not require any lemmas).

```
(defthm leaf-p-iff-member-fringe
  (iff (leaf-p atm x)
       (member-equal atm (fringe x))))
```

For a solution, see Section 1.5.1 [SOLUTION-TO-SIMPLE-EXAMPLE], page 31.

1.5.1 SOLUTION-TO-SIMPLE-EXAMPLE

SOLUTION-TO-SIMPLE-EXAMPLE solution to a simple example

To see a statement of the problem solved below, see Section 1.5 [TUTORIAL-EXAMPLES], page 30.

Here is a sequence of ACL2 events^ that illustrates the use of ACL2 to make definitions and prove theorems. We will introduce the notion of the fringe of a tree, as well as the notion of a leaf of a tree, and then prove that the members of the fringe are exactly the leaves.

We begin by defining the fringe of a tree, where we identify trees simply as cons^ structures, with atom^s at the leaves. The definition is recursive, breaking into two cases. If `x` is a cons^, then the `fringe` of `x` is obtained by appending together the `fringe`s of the `car^` and `cdr^` (left and right child) of `x`. Otherwise, `x` is an atom^ and its `fringe` is the one-element list containing only `x`.

```
(defun fringe (x)
  (if (consp x)
```

```
          (append (fringe (car x))
                  (fringe (cdr x)))
        (list x)))
```

Now that `fringe` has been defined, let us proceed by defining the notion of an atom appearing as a "leaf", with the goal of proving that the leaves of a tree are exactly the members of its `fringe`.

```
(defun leaf-p (atm x)
  (if (consp x)
      (or (leaf-p atm (car x))
          (leaf-p atm (cdr x)))
    (equal atm x)))
```

The main theorem is now as follows. Note that the rewrite rule below uses the equivalence relation `iff^` (see Section 14.6 [EQUIVALENCE], page 632) rather than `equal^`, since `member^` returns the tail of the given list that begins with the indicated member, rather than returning a Boolean. (Use `:pe member` to see the definition of `member^`.)

```
(defthm leaf-p-iff-member-fringe
  (iff (leaf-p atm x)
       (member-equal atm (fringe x))))
```

1.5.2 TUTORIAL1-TOWERS-OF-HANOI

TUTORIAL1-TOWERS-OF-HANOI   The Towers of Hanoi Example

This example was written almost entirely by Bill Young of Computational Logic, Inc.

We will tackle the famous "Towers of Hanoi" problem. This problem is illustrated by the following picture.

```
          |        |        |
          |        |        |
         ---       |        |
        -----      |        |
```

```
   -------      |       |

      A         B       C
```

We have three pegs — `a`, `b`, and `c` — and `n` disks of different sizes. The disks are all initially on peg `a`. The goal is to move all disks to peg `c` while observing the following two rules.

1. Only one disk may be moved at a time, and it must start and finish the move as the topmost disk on some peg;

2. A disk can never be placed on top of a smaller disk.

Let's consider some simple instances of this problem. If `n` = 1, i.e., only one disk is to be moved, simply move it from `a` to `c`. If `n` = 2, i.e., two disks are to be moved, the following sequence of moves suffices: move from `a` to `b`, move from `a` to `c`, move from `b` to `c`.

In general, this problem has a straightforward recursive solution. Suppose that we desire to move `n` disks from `a` to `c`, using `b` as the intermediate peg. For the basis, we saw above that we can always move a single disk from `a` to `c`. Now if we have `n` disks and assume that we can solve the problem for `n-1` disks, we can move `n` disks as follows. First, move `n-1` disks from `a` to `b` using `c` as the intermediate peg; move the single disk from `a` to `c`; then move `n-1` disks from `b` to `c` using `a` as the intermediate peg.

In ACL2, we can write a function that will return the sequence of moves. One such function is as follows. Notice that we have two base cases. If `(zp n)` then `n` is not a positive integer; we treat that case as if `n` were 0 and return an empty list of moves. If `n` is 1, then we return a list containing the single appropriate move. Otherwise, we return the list containing exactly the moves dictated by our recursive analysis above.

```
(defun move (a b)
  (list 'move a 'to b))

(defun hanoi (a b c n)
  (if (zp n)
      nil
    (if (equal n 1)
        (list (move a c))
      (append (hanoi a c b (1- n))
              (cons (move a c)
                    (hanoi b a c (1- n)))))))
```

Notice that we give `hanoi` four arguments: the three pegs, and the number of disks to move. It is necessary to supply the pegs because, in recursive calls, the roles of the pegs differ. In any execution of the algorithm, a given peg will sometimes be the source of a move, sometimes the destination, and sometimes the intermediate peg.

After submitting these functions to ACL2, we can execute the `hanoi` function on various specific arguments. For example:

```
ACL2 !>(hanoi 'a 'b 'c 1)
((MOVE A TO C))

ACL2 !>(hanoi 'a 'b 'c 2)
((MOVE A TO B)
 (MOVE A TO C)
 (MOVE B TO C))

ACL2 !>(hanoi 'a 'b 'c 3)
((MOVE A TO C)
 (MOVE A TO B)
 (MOVE C TO B)
 (MOVE A TO C)
 (MOVE B TO A)
 (MOVE B TO C)
 (MOVE A TO C))
```

From the algorithm it is clear that if it takes `m` moves to transfer `n` disks, it will take `(m + 1 + m) = 2m + 1` moves for `n+1` disks. From some simple calculations, we see that we need the following number of moves in specific cases:

Disks	0	1	2	3	4	5	6	7	...
Moves	0	1	3	7	15	31	63	127	...

The pattern is fairly clear. To move `n` disks requires `(2^n - 1)` moves. Let's attempt to use ACL2 to prove that fact.

First of all, how do we state our conjecture? Recall that `hanoi` returns a list of moves. The length of the list (given by the function `len`) is the number of moves required. Thus, we can state the following conjecture.

```
(defthm hanoi-moves-required-first-try
  (equal (len (hanoi a b c n))
         (1- (expt 2 n))))
```

When we submit this to ACL2, the proof attempt fails. Along the way we notice subgoals such as:

```
Subgoal *1/1'
(IMPLIES (NOT (< 0 N))
         (EQUAL 0 (+ -1 (EXPT 2 N)))).
```

This tells us that the prover is considering cases that are uninteresting to us, namely, cases in which `n` might be negative. The only cases that are really of interest are those in which `n` is a non-negative natural number. Therefore, we revise our theorem as follows:

```
(defthm hanoi-moves-required
  (implies (and (integerp n)
                (<= 0 n))    ;; n is at least 0
           (equal (len (hanoi a b c n))
                  (1- (expt 2 n)))))
```

and submit it to ACL2 again.

Again the proof fails. Examining the proof script we encounter the following text. (How did we decide to focus on this goal? Some information is provided in ACLH, and the ACL2 documentation on tips^ may be helpful. But the simplest answer is: this was the first goal suggested by the "proof-tree^" tool below the start of the proof by induction. See Chapter 12 [PROOF-TREE], page 579.)

```
Subgoal *1/5''
(IMPLIES (AND (INTEGERP N)
              (< 0 N)
              (NOT (EQUAL N 1))
              (EQUAL (LEN (HANOI A C B (+ -1 N)))
```

```
                 (+ -1 (EXPT 2 (+ -1 N))))
          (EQUAL (LEN (HANOI B A C (+ -1 N)))
                 (+ -1 (EXPT 2 (+ -1 N)))))
     (EQUAL (LEN (APPEND (HANOI A C B (+ -1 N))
                         (CONS (LIST 'MOVE A 'TO C)
                               (HANOI B A C (+ -1 N)))))
            (+ -1 (* 2 (EXPT 2 (+ -1 N))))))
```

It is difficult to make much sense of such a complicated goal. However, we do notice something interesting. In the conclusion is a term^ of the following shape.

```
(LEN (APPEND ... ...))
```

We conjecture that the length of the `append^` of two lists should be the sum of the lengths of the lists. If the prover knew that, it could possibly have simplified this term^ earlier and made more progress in the proof. Therefore, we need a rewrite^ rule that will suggest such a simplification to the prover. The appropriate rule is:

```
(defthm len-append
  (equal (len (append x y))
         (+ (len x) (len y))))
```

We submit this to the prover, which proves it by a straightforward induction. The prover stores this lemma as a rewrite^ rule and will subsequently (unless we disable^ the rule) replace term^s matching the left hand side of the rule with the appropriate instance of the term^ on the right hand side.

We now resubmit our lemma `hanoi-moves-required` to ACL2. On this attempt, the proof succeeds and we are done.

One bit of cleaning up is useful. We needed the hypotheses that:

```
(and (integerp n) (<= 0 n)).
```

This is an awkward way of saying that `n` is a natural number; natural is not a primitive data type in ACL2. We could define a function `naturalp`, but it is somewhat more convenient to define a macro as follows:

```
(defmacro naturalp (x)
  (list 'and (list 'integerp x)
             (list '<= 0 x)))
```

Subsequently, we can use `(naturalp n)` wherever we need to note that a quantity is a natural number. See Section 6.10 [DEFMACRO], page 163 for more information about ACL2 macros. With this macro, we can reformulate our theorem as follows:

```
(defthm hanoi-moves-required
  (implies (naturalp n)
           (equal (len (hanoi a b c n))
                  (1- (expt 2 n))))).
```

Another interesting (but much harder) theorem asserts that the list of moves generated by our `hanoi` function actually accomplishes the desired goal while following the rules. When you can state and prove that theorem, you'll be a very competent ACL2 user.

By the way, the name "Towers of Hanoi" derives from a legend that a group of Vietnamese monks works day and night to move a stack of 64 gold disks from one diamond peg to another, following the rules set out above. We're told that the world will end when they complete this task. From the theorem above, we know that this requires 18,446,744,073,709,551,615 moves:

```
ACL2 !>(1- (expt 2 64))
18446744073709551615
ACL2 !>
```

We're guessing they won't finish any time soon.

1.5.3 TUTORIAL2-EIGHTS-PROBLEM

TUTORIAL2-EIGHTS-PROBLEM The Eights Problem Example

This example was written almost entirely by Bill Young of Computational Logic, Inc.

This simple example was brought to our attention as one that Paul Jackson has solved using the NuPrl system. The challenge is to prove the theorem:

```
for all n > 7, there exist naturals i and j such that: n = 3i + 5j.
```

In ACL2, we could phrase this theorem using quantification. However we will start with a constructive approach, i.e., we will show that values of `i` and `j` exist by writing a function that will construct such values for given `n`. Suppose we had a function `(split n)` that returns an appropriate pair `(i . j)`. Our theorem would be as follows:

```
(defthm split-splits
  (let ((i (car (split n)))
        (j (cdr (split n))))
    (implies (and (integerp n)
                  (< 7 n))
             (and (integerp i)
                  (<= 0 i)
                  (integerp j)
                  (<= 0 j)
                  (equal (+ (* 3 i) (* 5 j))
                         n)))))
```

That is, assuming that `n` is a natural number greater than 7, `(split n)` returns values `i` and `j` that are in the appropriate relation to `n`.

Let's look at a few cases:

```
 8 = 3x1 + 5x1;    11 = 3x2 + 5x1;     14 = 3x3 + 5x1;    ...
 9 = 3x3 + 5x0;    12 = 3x4 + 5x0;     15 = 3x5 + 5x0;    ...
10 = 3x0 + 5x2;    13 = 3x1 + 5x2;     16 = 3x2 + 5x2;    ...
```

Maybe you will have observed a pattern here; any natural number larger than 10 can be obtained by adding some multiple of 3 to 8, 9, or 10. This gives us the clue to constructing a proof. It is clear that we can write split as follows:

```
(defun bump-i (x)
  ;; Bump the i component of the pair
  ;; (i . j) by 1.
  (cons (1+ (car x)) (cdr x)))

(defun split (n)
  ;; Find a pair (i . j) such that
  ;; n = 3i + 5j.
  (if (or (zp n) (< n 8))
      nil ;; any value is really reasonable here
    (if (equal n 8)
        (cons 1 1)
      (if (equal n 9)
          (cons 3 0)
        (if (equal n 10)
            (cons 0 2)
          (bump-i (split (- n 3))))))))
```

Notice that we explicitly compute the values of `i` and `j` for the cases of 8, 9, and 10, and for the degenerate case when `n` is not a natural or is less than 8. For all naturals greater than `n`, we decrement `n` by 3 and bump the number of 3's (the value of i) by 1. We know that the recursion must terminate because any integer value greater than 10 can eventually be reduced to 8, 9, or 10 by successively subtracting 3.

Let's try it on some examples:

```
ACL2 !>(split 28)
(6 . 2)

ACL2 !>(split 45)
(15 . 0)

ACL2 !>(split 335)
(110 . 1)
```

Finally, we submit our theorem `split-splits`, and the proof succeeds. In this case, the prover is "smart" enough to induct according to the pattern indicated by the function split.

For completeness, we'll note that we can state and prove a quantified version of this theorem. We introduce the notion `split-able` to mean that appropriate `i` and `j` exist for `n`.

```
(defun-sk split-able (n)
  (exists (i j)
          (equal n (+ (* 3 i) (* 5 j)))))
```

Then our theorem is given below. Notice that we prove it by observing that our previous function `split` delivers just such an `i` and `j` (as we proved above).

```
(defthm split-splits2
  (implies (and (integerp n)
                (< 7 n))
           (split-able n))
  :hints (("Goal" :use (:instance split-able-suff
                                  (i (car (split n)))
                                  (j (cdr (split n)))))))
```

Unfortunately, understanding the mechanics of the proof requires knowing something about the way `defun-sk^` works. See Section 6.17 [DEFUN-SK], page 173 or see Section 1.5.5 [TUTORIAL4-DEFUN-SK-EXAMPLE], page 56 for more on that subject.

1.5.4 TUTORIAL3-PHONEBOOK-EXAMPLE

TUTORIAL3-PHONEBOOK-EXAMPLE A Phonebook Specification

The other tutorial examples are rather small and entirely self contained. The present example is rather more elaborate, and makes use of a feature that really adds great power and versatility to ACL2, namely: the use of previously defined collections of lemmas, in the form of "books^."

This example was written almost entirely by Bill Young of Computational Logic, Inc.

This example is based on one developed by Ricky Butler and Sally Johnson of NASA Langley for the PVS system, and subsequently revised by Judy Crow, *et al*, at SRI. It is a simple phone book specification. We will not bother to follow their versions closely, but will instead present a style of specification natural for ACL2.

The idea is to model an electronic phone book with the following properties.

Our phone book will store the phone numbers of a city.

It must be possible to retrieve a phone number, given a name.

It must be possible to add and delete entries.

Of course, there are numerous ways to construct such a model. A natural approach within the Lisp/ACL2 context is to use "association lists" or "alists." Briefly, an alist is a list of pairs `(key . value)` associating a value with a key. A phone book could be an alist of pairs `(name . pnum)`. To find the phone number associated with a given name, we merely search the alist until we find the appropriate pair. For a large city, such a linear list would not be efficient, but at this point we are interested only in **modeling** the problem, not in deriving an efficient implementation. We could address that question later by proving our alist model equivalent, in some desired sense, to a more efficient data structure.

We could build a theory of alists from scratch, or we can use a previously constructed theory (book) of alist definitions and facts. By using an existing book, we build upon the work of others, start our specification and proof effort from a much richer foundation, and hopefully devote more of our time to the problem at hand. Unfortunately, it is not completely simple for the new user to know what books^ are available and what they contain. We hope later to improve the documentation of the growing collection of books^ available with the ACL2 distribution; for now, the reader is encouraged to look in the README file in the `books` subdirectory. For present purposes, the beginning user can simply take our word that a book exists containing useful alist definitions and facts. On our local machine, these definitions and lemmas can be introduced into the current theory using the command^:

```
(include-book "/slocal/src/acl2/v1-8/books/public/alist-defthms")
```

This book has been "certified," which means that the definitions and lemmas have been mechanically checked and stored in a safe manner. (See Chapter 3 [BOOKS], page 89 and see Section 6.20 [INCLUDE-BOOK], page 180 for details.)

Including this book makes available a collection of functions including the following:

```
(ALISTP A)    ; is A an alist (actually a primitive ACL2 function)

(BIND X V A)  ; associate the key X with value V in alist A
```

```
(BINDING X A) ; return the value associated with key X in alist A

(BOUND? X A)  ; is key X associated with any value in alist A

(DOMAIN A)    ; return the list of keys bound in alist A

(RANGE A)     ; return the list of values bound to keys in alist A

(REMBIND X A) ; remove the binding of key X in alist A
```

Along with these function definitions, the book also provides a number of proved lemmas that aid in simplifying expressions involving these functions. (See Chapter 14 [RULE-CLASSES], page 617 for the way in which lemmas are used in simplification and rewriting.) For example,

```
(defthm bound?-bind
  (equal (bound? x (bind y v a))
         (or (equal x y)
             (bound? x a))))
```

asserts that `x` will be bound in `(bind y v a)` if and only if: either `x = y` or `x` was already bound in `a`. Also,

```
(defthm binding-bind
  (equal (binding x (bind y v a))
         (if (equal x y)
             v
           (binding x a))))
```

asserts that the resulting binding will be `v`, if `x = y`, or the binding that `x` had in `a` already, if not.

Thus, the inclusion of this book essentially extends our specification and reasoning capabilities by the addition of new operations and facts about these operations that allow us to build further specifications on a richer and possibly more intuitive foundation.

However, it must be admitted that the use of a book such as this has two potential limitations:

the definitions available in a book may not be ideal for your particular problem;

it is (extremely) likely that some useful facts (especially, rewrite^ rules) are not available in the book and will have to be proved.

For example, what is the value of `binding` when given a key that is not bound in the alist? We can find out by examining the function definition. Look at the definition of the `binding` function (or any other defined function), using the `:pe^` command:

```
ACL2 !>:pe binding
   d     33  (INCLUDE-BOOK
                  "/slocal/src/acl2/v1-8/books/public/alist-defthms")

>V d          (DEFUN BINDING (X A)
                     "The value bound to X in alist A."
                     (DECLARE (XARGS :GUARD (ALISTP A)))
                     (CDR (ASSOC-EQUAL X A)))
```

This tells us that `binding` was introduced by the given `include-book^` form, is currently disable^d in the current theory, and has the definition given by the displayed `defun^` form. We see that `binding` is actually defined in terms of the primitive `assoc-equal^` function. If we look at the definition of `assoc-equal^`:

```
ACL2 !>:pe assoc-equal
 V     -489  (DEFUN ASSOC-EQUAL (X ALIST)
                    (DECLARE (XARGS :GUARD (ALISTP ALIST)))
                    (COND ((ENDP ALIST) NIL)
                          ((EQUAL X (CAR (CAR ALIST)))
                           (CAR ALIST))
                          (T (ASSOC-EQUAL X (CDR ALIST)))))
```

we can see that `assoc-equal^` returns `nil` upon reaching the end of an unsuccessful search down the alist. So `binding` returns `(cdr nil)` in that case, which is `nil`. Notice that we could also have investigated this question by trying some simple examples.

```
ACL2 !>(binding 'a nil)
NIL

ACL2 !>(binding 'a (list (cons 'b 2)))
NIL
```

These definitions aren't ideal for all purposes. For one thing, there's nothing that keeps us from having `nil` as a value bound to some key in the alist. Thus, if `binding` returns `nil` we don't always know if that is the value associated with the key in the alist, or if that key is not bound. We'll have to keep that ambiguity in mind whenever we use `binding` in our specification. Suppose instead that we wanted `binding` to return some error string on unbound keys. Well, then we'd just have to write our own version of `binding`. But then we'd lose much of the value of using a previously defined book. As with any specification technique, certain tradeoffs are necessary.

Why not take a look at the definitions of other alist functions and see how they work together to provide the ability to construct and search alists? We'll be using them rather heavily in what follows so it will be good if you understand basically how they work. Simply start up ACL2 and execute the form shown earlier, but substituting our directory name for the top-level ACL2 directory with yours. Alternatively, the following should work if you start up ACL2 in the directory of the ACL2 sources:

```
(include-book (:relative "books" "public" "alist-defthms"))
```

Then, you can use :pe^ to look at function definitions. You'll soon discover that almost all of the definitions are built on definitions of other, more primitive functions, as `binding` is built on `assoc-equal^`. You can look at those as well, of course, or in many cases visit their documentation.

The other problem with using a predefined book is that it will seldom be "sufficiently complete," in the sense that the collection of rewrite^ rules supplied won't be adequate to prove everything we'd like to know about the interactions of the various functions. If it were, there'd be no real reason to know that `binding` is built on top of `assoc-equal^`, because everything we'd need to know about `binding` would be nicely expressed in the collection of theorems supplied with the book. However, that's very seldom the case. Developing such a collection of rules is currently more art than science and requires considerable experience. We'll encounter examples later of "missing" facts about `binding` and our other alist functions. So, let's get on with the example.

Notice that alists are mappings of keys to values; but, there is no notion of a "type" associated with the keys or with the values. Our phone book example, however, does have such a notion of types; we map names to phone numbers. We can introduce these "types" by explicitly defining them, e.g., names are strings and phone numbers are integers. Alternatively, we can **partially define** or axiomatize a recognizer for names without giving a full definition. A way to safely introduce such "constrained" function symbols in ACL2 is with the `encapsulate^` form. For example, consider the following form.

```
(encapsulate
  ;; Introduce a recognizer for names with a ``type'' lemma.
  ((namep (x) t))
  ;;
  (local (defun namep (x)
           ;; This declare is needed to tell
           ;; ACL2 that we're aware that the
           ;; argument x is not used in the body
           ;; of the function.
           (declare (ignore x))
           t))
  ;;
  (defthm namep-booleanp
    (booleanp (namep x))))
```

This `encapsulate^` form introduces the new function `namep` of one argument and one result and constrains `(namep x)` to be Boolean, for all inputs `x`. More generally, an encapsulation establishes an environment in which functions can be defined and theorems and rules added without necessarily introducing those functions, theorems, and rules into the environment outside the encapsulation. To be admissible, all the events in the body of an encapsulate must be admissible. But the effect of an encapsulate is to assume only the non-local events.

The first "argument" to `encapsulate`, `((namep (x) t))` above, declares the intended signature^s of new function symbols that will be "exported" from the encapsulation without definition. The `local^ defun^` of `name` defines name within the encapsulation always to return `t`. The `defthm` event establishes that `namep` is Boolean. By making the `defun` local but the `defthm` non-`local` this encapsulate constrains the undefined function `namep` to be Boolean; the admissibility of the encapsulation establishes that there exists a Boolean function (namely the constant function returning `t`).

We can subsequently use `namep` as we use any other Boolean function, with the proviso that we know nothing about it except that it always returns either `t` or `nil`. We use `namep` to "recognize" legal keys for our phonebook alist.

We wish to do something similar to define what it means to be a legal phone number. We submit the following form to ACL2:

```
(encapsulate
  ;; Introduce a recognizer for phone numbers.
  ((pnump (x) t))
```

```
;;
(local (defun pnump (x)
         (not (equal x nil))))
;;
(defthm pnump-booleanp
  (booleanp (pnump x)))
;;
(defthm nil-not-pnump
  (not (pnump nil)))).
```

This introduces a Boolean-valued recognizer `pnump`, with the additional proviso that the constant `nil` is not a `pnump`. We impose this restriction to guarantee that we'll never bind a name to `nil` in our phone book and thereby introduce the kind of ambiguity described above regarding the use of `binding`.

Now a legal phone book is an alist mapping from `nameps` to `pnumps`. We can define this as follows:

```
(defun name-phonenum-pairp (x)
  ;; Recognizes a pair of (name . pnum).
  (and (consp x)
       (namep (car x))
       (pnump (cdr x))))

(defun phonebookp (l)
  ;; Recognizes a list of such pairs.
  (if (not (consp l))
      (null l)
    (and (name-phonenum-pairp (car l))
         (phonebookp (cdr l)))))
```

Thus, a phone book is really a list of pairs `(name . pnum)`. Notice that we have not assumed that the keys of the phone book are distinct. We'll worry about that question later. (It is not always desirable to insist that the keys of an alist be distinct. But it may be a useful requirement for our specific example.)

Now we are ready to define some of the functions necessary for our phonebook example. The functions we need are:

```
(IN-BOOK? NM BK)         ; does NM have a phone number in BK
```

```
(FIND-PHONE NM BK)         ; find NM's phone number in phonebook BK

(ADD-PHONE NM PNUM BK)     ; give NM the phone number PNUM in BK

(CHANGE-PHONE NM PNUM BK) ; change NM's phone number to PNUM in BK

(DEL-PHONE NM PNUM)        ; remove NM's phone number from BK
```

Given our underlying theory of alists, it is easy to write these functions. But we must take care to specify appropriate "boundary" behavior. Thus, what behavior do we want when, say, we try to change the phone number of a client who is not currently in the book? As usual, there are numerous possibilities; here we'll assume that we return the phone book unchanged if we try anything "illegal."

Possible definitions of our phone book functions are as follows. (Remember, an `include-book` form such as the ones shown earlier must be executed in order to provide definitions for functions such as `bound?`.)

```
(defun in-book? (nm bk)
  (bound? nm bk))

(defun find-phone (nm bk)
  (binding nm bk))

(defun add-phone (nm pnum bk)
  ;; If nm already in-book?, make no change.
  (if (in-book? nm bk)
      bk
    (bind nm pnum bk)))

(defun change-phone (nm pnum bk)
  ;; Make a change only if nm already has a phone number.
  (if (in-book? nm bk)
      (bind nm pnum bk)
    bk))

(defun del-phone (nm bk)
  ;; Remove the binding from bk, if there is one.
  (rembind nm bk))
```

Notice that we don't have to check whether a name is in the book before deleting, because `rembind` is essentially a no-op if `nm` is not bound in `bk`.

In some sense, this completes our specification. But we can't have any real confidence in its correctness without validating our specification in some way. One way to do so is by proving some properties of our specification. Some candidate properties are:

1. A name will be in the book after we add it.
2. We will find the most recently added phone number for a client.
3. If we change a number, we'll find the change.
4. Changing and then deleting a number is the same as just deleting.
5. A name will not be in the book after we delete it.

Let's formulate some of these properties. The first one, for example, is:

```
(defthm add-in-book
  (in-book? nm (add-phone nm pnum bk))).
```

You may wonder why we didn't need any hypotheses about the "types" of the arguments. In fact, `add-in-book` is really expressing a property that is true of alists in general, not just of the particular variety of alists we are dealing with. Of course, we could have added some extraneous hypotheses and proved:

```
(defthm add-in-book
  (implies (and (namep nm)
                (pnump pnum)
                (phonebookp bk))
           (in-book? nm (add-phone nm pnum bk)))),
```

but that would have yielded a weaker and less useful lemma because it would apply to fewer situations. In general, it is best to state lemmas in the most general form possible and to eliminate unnecessary hypotheses whenever possible. The reason for that is simple: lemmas are usually stored as rules and used in later proofs. For a lemma to be used, its hypotheses must be relieved (proved to hold in that instance); extra hypotheses require extra work. So we avoid them whenever possible.

There is another, more important observation to make about our lemma. Even in its simpler form (without the extraneous hypotheses), the lemma `add-in-book` may be useless as a rewrite^ rule. Notice that it is stated in terms of the non-recursive functions `in-book?` and `add-phone`. If such functions appear in the left hand side of the conclusion of a lemma, the lemma may not ever be used. Suppose in a later proof, the theorem prover encountered a term^ of the form:

```
(in-book? nm (add-phone nm pnum bk)).
```

Since we've already proved `add-in-book`, you'd expect that this would be immediately reduced to true. However, the theorem prover will often "expand" the non-recursive definitions of `in-book?` and `add-phone` using their definitions **before** it attempts rewriting with lemmas. After this expansion, lemma `add-in-book` won't "match" the term^ and so won't be applied. Look back at the proof script for `add-in-proof` and you'll notice that at the very end the prover warned you of this potential difficulty when it printed:

```
Warnings:  Non-rec
Time:  0.18 seconds (prove: 0.05, print: 0.00, other: 0.13)
ADD-IN-BOOK
```

The "Warnings" line notifies you that there are non-recursive function calls in the left hand side of the conclusion and that this problem might arise. Of course, it may be that you don't ever plan to use the lemma for rewriting or that your intention is to disable^ these functions. Disable^d functions are not expanded and the lemma should apply. However, you should always take note of such warnings and consider an appropriate response. By the way, we noted above that `binding` is disable^d. If it were not, none of the lemmas about `binding` in the book we included would likely be of much use for exactly the reason we just gave.

For our current example, let's assume that we're just investigating the properties of our specifications and not concerned about using our lemmas for rewriting. So let's go on. If we really want to avoid the warnings, we can add `:rule-classes nil` to each `defthm` event; see Chapter 14 [RULE-CLASSES], page 617.

Property 2 is: we always find the most recently added phone number for a client. Try the following formalization:

```
(defthm find-add-first-cut
```

```
(equal (find-phone nm (add-phone nm pnum bk))
       pnum))
```

and you'll find that the proof attempt fails. Examining the proof attempt and our function definitions, we see that the lemma is false if `nm` is already in the book. We can remedy this situation by reformulating our lemma in at least two different ways:

```
(defthm find-add1
  (implies (not (in-book? nm bk))
           (equal (find-phone nm (add-phone nm pnum bk))
                  pnum)))

(defthm find-add2
  (equal (find-phone nm (add-phone nm pnum bk))
         (if (in-book? nm bk)
             (find-phone nm bk)
             pnum)))
```

For technical reasons, lemmas such as `find-add2`, i.e., which do not have hypotheses, are usually slightly preferable. This lemma is stored as an "unconditional" rewrite^ rule (i.e., has no hypotheses) and, therefore, will apply more often than `find-add1`. However, for our current purposes either version is all right.

Property 3 says: If we change a number, we'll find the change. This is very similar to the previous example. The formalization is as follows.

```
(defthm find-change
  (equal (find-phone nm (change-phone nm pnum bk))
         (if (in-book? nm bk)
             pnum
           (find-phone nm bk))))
```

Property 4 says: changing and then deleting a number is the same as just deleting. We can model this as follows.

```
(defthm del-change
  (equal (del-phone nm (change-phone nm pnum bk))
```

```
                (del-phone nm bk)))
```

Unfortunately, when we try to prove this, we encounter subgoals that seem to be true, but for which the prover is stumped. For example, consider the following goal. (Note: `endp` holds of lists that are empty.)

```
Subgoal *1/4
(IMPLIES (AND (NOT (ENDP BK))
              (NOT (EQUAL NM (CAAR BK)))
              (NOT (BOUND? NM (CDR BK)))
              (BOUND? NM BK))
         (EQUAL (REMBIND NM (BIND NM PNUM BK))
                (REMBIND NM BK))).
```

Our intuition about `rembind` and `bind` tells us that this goal should be true even without the hypotheses. We attempt to prove the following lemma.

```
(defthm rembind-bind
  (equal (rembind nm (bind nm pnum bk))
         (rembind nm bk)))
```

The prover proves this by induction, and stores it as a rewrite rule. After that, the prover has no difficulty in proving `del-change`.

The need to prove lemma `rembind-bind` illustrates a point we made early in this example: the collection of rewrite^ rules supplied by a previously certified book will almost never be everything you'll need. It would be nice if we could operate purely in the realm of names, phone numbers, and phone books without ever having to prove any new facts about alists. Unfortunately, we needed a fact about the relation between `rembind` and `bind` that wasn't supplied with the alists theory. Hopefully, such omissions will be rare.

Finally, let's consider our property 5 above: a name will not be in the book after we delete it. We formalize this as follows:

```
(defthm in-book-del
  (not (in-book? nm (del-phone nm bk))))
```

This proves easily. But notice that it's only true because `del-phone` (actually `rembind`) removes **all** occurrences of a name from the phone book. If it only removed, say, the first one it encountered, we'd need a hypothesis that said that `nm` occurs at most once in `bk`. Ah, maybe that's a property you hadn't considered. Maybe you want to ensure that **any** name occurs at most once in any valid phonebook.

To complete this example, let's consider adding an **invariant** to our specification. In particular, suppose we want to assure that no client has more than one associated phone number. One way to ensure this is to require that the domain of the alist is a "set" (has no duplicates).

```
(defun setp (l)
  (if (atom l)
      (null l)
    (and (not (member-equal (car l) (cdr l)))
         (setp (cdr l)))))

(defun valid-phonebookp (bk)
  (and (phonebookp bk)
       (setp (domain bk))))
```

Now, we want to show under what conditions our operations preserve the property of being a `valid-phonebookp`. The operations `in-book?` and `find-phone` don't return a phone book, so we don't really need to worry about them. Since we're really interested in the "types" of values preserved by our phonebook functions, let's look at the types of those operations as well.

```
(defthm in-book-booleanp
  (booleanp (in-book? nm bk)))

(defthm in-book-namep
  (implies (and (phonebookp bk)
                (in-book? nm bk))
           (namep nm))
  :hints (("Goal" :in-theory (enable bound?))))

(defthm find-phone-pnump
  (implies (and (phonebookp bk)
                (in-book? nm bk))
           (pnump (find-phone nm bk)))
  :hints (("Goal" :in-theory (enable bound? binding))))
```

Note the "`:hints^`" on the last two lemmas. Neither of these would prove without these hints^, because once again there are some facts about `bound?` and `binding` not available in our current context. Now, we could figure out what those facts are and try to prove them. Alternatively, we can enable^ `bound?` and `binding` and hope that by opening up these functions, the conjectures will reduce to versions that the prover does know enough about or can prove by induction. In this case, this strategy works. The hints tell the prover to enable^ the functions in question when considering the designated goal.

Below we develop the theorems showing that `add-phone`, `change-phone`, and `del-phone` preserve our proposed invariant. Notice that along the way we have to prove some subsidiary facts, some of which are pretty ugly. It would be a good idea for you to try, say, `add-phone-preserves-invariant` without introducing the following four lemmas first. See if you can develop the proof and only add these lemmas as you need assistance. Then try `change-phone-preserves-invariant` and `del-phone-preserves-invariant`. They will be easier. It is illuminating to think about why `del-phone-preserves-invariant` does not need any "type" hypotheses.

```
(defthm bind-preserves-phonebookp
  (implies (and (phonebookp bk)
                (namep nm)
                (pnump num))
           (phonebookp (bind nm num bk))))

(defthm member-equal-strip-cars-bind
  (implies (and (not (equal x y))
                (not (member-equal x (strip-cars a))))
           (not (member-equal x (strip-cars (bind y z a))))))

(defthm bind-preserves-domain-setp
  (implies (and (alistp bk)
                (setp (domain bk)))
           (setp (domain (bind nm num bk))))
  :hints (("Goal" :in-theory (enable domain))))

(defthm phonebookp-alistp
  (implies (phonebookp bk)
           (alistp bk)))

(defthm ADD-PHONE-PRESERVES-INVARIANT
  (implies (and (valid-phonebookp bk)
                (namep nm)
                (pnump num))
```

```
            (valid-phonebookp (add-phone nm num bk)))
  :hints (("Goal" :in-theory (disable domain-bind))))

(defthm CHANGE-PHONE-PRESERVES-INVARIANT
  (implies (and (valid-phonebookp bk)
                (namep nm)
                (pnump num))
           (valid-phonebookp (change-phone nm num bk)))
  :hints (("Goal" :in-theory (disable domain-bind))))

(defthm remove-equal-preserves-setp
  (implies (setp l)
           (setp (remove-equal x l))))

(defthm rembind-preserves-phonebookp
  (implies (phonebookp bk)
           (phonebookp (rembind nm bk))))

(defthm DEL-PHONE-PRESERVES-INVARIANT
  (implies (valid-phonebookp bk)
           (valid-phonebookp (del-phone nm bk))))
```

As a final test of your understanding, try to formulate and prove an invariant that says that no phone number is assigned to more than one name. The following hints may help.

1. Define the appropriate invariant. (Hint: remember the function `range`.)

2. Do our current definitions of `add-phone` and `change-phone` necessarily preserve this property? If not, consider what hypotheses are necessary in order to guarantee that they do preserve this property.

3. Study the definition of the function `range` and notice that it is defined in terms of the function `strip-cdrs^`. Understand how this defines the range of an alist.

4. Formulate the correctness theorems and attempt to prove them. You'll probably benefit from studying the invariant proof above. In particular, you may need some fact about the function `strip-cdrs^` analogous to the lemma `member-equal-strip-cars-bind` above.

Below is one solution to this exercise. Don't look at the solution, however, until you've struggled a bit with it. Notice that we didn't actually change the definitions of `add-phone` and `change-phone`, but added a hypothesis saying that the number is "new." We could have changed the definitions to check this and return the phonebook unchanged if the number was already in use.

```
(defun pnums-in-use (bk)
  (range bk))

(defun phonenums-unique (bk)
  (setp (pnums-in-use bk)))

(defun new-pnump (pnum bk)
  (not (member-equal pnum (pnums-in-use bk))))

(defthm member-equal-strip-cdrs-rembind
  (implies (not (member-equal x (strip-cdrs y)))
           (not (member-equal x (strip-cdrs (rembind z y))))))

(defthm DEL-PHONE-PRESERVES-PHONENUMS-UNIQUE
  (implies (phonenums-unique bk)
           (phonenums-unique (del-phone nm bk)))
  :hints (("Goal" :in-theory (enable range))))

(defthm strip-cdrs-bind-non-member
  (implies (and (not (bound? x a))
                (alistp a))
           (equal (strip-cdrs (bind x y a))
                  (append (strip-cdrs a) (list y))))
  :hints (("Goal" :in-theory (enable bound?))))

(defthm setp-append-list
  (implies (setp l)
           (equal (setp (append l (list x)))
                  (not (member-equal x l)))))

(defthm ADD-PHONE-PRESERVES-PHONENUMS-UNIQUE
  (implies (and (phonenums-unique bk)
                (new-pnump pnum bk)
                (alistp bk))
           (phonenums-unique (add-phone nm pnum bk)))
  :hints (("Goal" :in-theory (enable range))))

(defthm member-equal-strip-cdrs-bind
  (implies (and (not (member-equal z (strip-cdrs a)))
                (not (equal z y)))
           (not (member-equal z (strip-cdrs (bind x y a))))))

(defthm CHANGE-PHONE-PRESERVES-PHONENUMS-UNIQUE
  (implies (and (phonenums-unique bk)
                (new-pnump pnum bk)
                (alistp bk))
           (phonenums-unique (change-phone nm pnum bk)))
```

```
:hints (("Goal" :in-theory (enable range))))
```

1.5.5 TUTORIAL4-DEFUN-SK-EXAMPLE

TUTORIAL4-DEFUN-SK-EXAMPLE example of quantified notions

This example illustrates the use of `defun-sk^` and `defthm^` events^ to reason about quantifiers. See Section 6.17 [DEFUN-SK], page 173.

Many users prefer to avoid the use of quantifiers, since ACL2 provides only very limited support for reasoning about quantifiers.

Here is a list of events^ that proves that if there are arbitrarily large numbers satisfying the disjunction `(OR P R)`, then either there are arbitrarily large numbers satisfying `P` or there are arbitrarily large numbers satisfying `R`.

```
; Introduce undefined predicates p and r.
(defstub p (x) t)
(defstub r (x) t)

; Define the notion that something bigger than x satisfies p.
(defun-sk some-bigger-p (x)
  (exists y (and (< x y) (p y))))

; Define the notion that something bigger than x satisfies r.
(defun-sk some-bigger-r (x)
  (exists y (and (< x y) (r y))))

; Define the notion that arbitrarily large x satisfy p.
(defun-sk arb-lg-p ()
  (forall x (some-bigger-p x)))

; Define the notion that arbitrarily large x satisfy r.
(defun-sk arb-lg-r ()
  (forall x (some-bigger-r x)))

; Define the notion that something bigger than x satisfies p or r.
(defun-sk some-bigger-p-or-r (x)
  (exists y (and (< x y) (or (p y) (r y)))))

; Define the notion that arbitrarily large x satisfy p or r.
(defun-sk arb-lg-p-or-r ()
```

```
  (forall x (some-bigger-p-or-r x)))

; Prove the theorem promised above.  Notice that the functions open
; automatically, but that we have to provide help for some rewrite
; rules because they have free variables in the hypotheses.  The
; ``witness functions'' mentioned below were introduced by DEFUN-SK.

(thm
 (implies (arb-lg-p-or-r)
          (or (arb-lg-p)
              (arb-lg-r)))
 :hints (("Goal"
          :use
          ((:instance some-bigger-p-suff
                      (x (arb-lg-p-witness))
                      (y (some-bigger-p-or-r-witness
                          (max (arb-lg-p-witness)
                               (arb-lg-r-witness)))))
           (:instance some-bigger-r-suff
                      (x (arb-lg-r-witness))
                      (y (some-bigger-p-or-r-witness
                          (max (arb-lg-p-witness)
                               (arb-lg-r-witness)))))
           (:instance arb-lg-p-or-r-necc
                      (x (max (arb-lg-p-witness)
                              (arb-lg-r-witness))))))))

; And finally, here's a cute little example.  We have already
; defined above the notion (some-bigger-p x), which says that
; something bigger than x satisfies p.  Let us introduce a notion
; that asserts that there exists both y and z bigger than x which
; satisfy p.  On first glance this new notion may appear to be
; stronger than the old one, but careful inspection shows that y and
; z do not have to be distinct.  In fact ACL2 realizes this, and
; proves the theorem below automatically.

(defun-sk two-bigger-p (x)
  (exists (y z) (and (< x y) (p y) (< x z) (p z))))

(thm (implies (some-bigger-p x) (two-bigger-p x)))

; A technical point:  ACL2 fails to prove the theorem above
; automatically if we take its contrapositive, unless we disable
; two-bigger-p as shown below.  That is because ACL2 needs to expand
; some-bigger-p before applying the rewrite rule introduced for
; two-bigger-p, which contains free variables.  The moral of the
; story is:  Don't expect too much automatic support from ACL2 for
; reasoning about quantified notions.
```

```
(thm (implies (not (two-bigger-p x)) (not (some-bigger-p x)))
     :hints (("Goal" :in-theory (disable two-bigger-p))))
```

1.5.6 TUTORIAL5-MISCELLANEOUS-EXAMPLES

TUTORIAL5-MISCELLANEOUS-EXAMPLES miscellaneous ACL2 examples

The following examples are more advanced examples of usage of ACL2. They are included largely for reference, in case someone finds them useful.

1.5.6.1 FILE-READING-EXAMPLE

FILE-READING-EXAMPLE example of reading files in ACL2

This example illustrates the use of ACL2's IO^ primitives to read the forms in a file. See Section 10.135 [IO], page 456.

This example provides a solution to the following problem. Let's say that you have a file that contains s-expressions. Suppose that you want to build a list by starting with `nil`, and updating it "appropriately" upon encountering each successive s-expression in the file. That is, suppose that you have written a function `update-list` such that `(update-list obj current-list)` returns the list obtained by "updating" `current-list` with the next object, `obj`, encountered in the file. The top-level function for processing such a file, returning the final list, could be defined as follows. Notice that because it opens a channel to the given file, this function modifies state^ and hence must return state^. Thus it actually returns two values: the final list and the new state^.

```
(defun process-file (filename state)
  (mv-let
   (channel state)
   (open-input-channel filename :object state)
   (mv-let (result state)
           (process-file1 nil channel state) ;see below
           (let ((state (close-input-channel channel state)))
```

```
                    (mv result state)))))
```

The function `process-file1` referred to above takes the currently constructed list (initially, `nil`), together with a channel to the file being read and the state^, and returns the final updated list. Notice that this function is tail recursive. This is important because many Lisp compilers will remove tail recursion, thus avoiding the potential for stack overflows when the file contains a large number of forms.

```
(defun process-file1 (current-list channel state)
  (mv-let (eofp obj state)
          (read-object channel state)
          (cond
           (eofp (mv current-list state))
           (t (process-file1 (update-list obj current-list)
                             channel state)))))
```

1.5.6.2 GUARD-EXAMPLE

GUARD-EXAMPLE   a brief transcript illustrating guard^s in ACL2

This note addresses the question: what is the use of guard^s in ACL2? Although we recommend that beginners try to avoid guard^s for a while, we hope that the summary here is reasonably self-contained and will provide a reasonable introduction to guards in ACL2. For a more systematic discussion, see Section 8.39 [GUARD], page 285. For a summary of that topic, see Section 8.39.3 [GUARD-QUICK-REFERENCE], page 288.

Before we get into the issue of guard^s, let us note that there are two important "modes":

defun-mode^ — "Does this defun^ add an axiom (':logic mode') or not (':program mode')?" (See Section 8.18 [DEFUN-MODE], page 256.) Only `:logic^` mode functions can have their "guard^s verified" via mechanized proof; see Section 6.38 [VERIFY-GUARDS], page 199.

`set-guard-checking^` — "Should runtime guard^ violations signal an error (t) or go undetected (nil)? Equivalently, are expressions evaluated in Common Lisp or in the logic?" (See Section 9.12 [SET-GUARD-CHECKING], page 386.)

Prompt examples

Here some examples of the relation between the ACL2 prompt^ and the "modes" discussed above. Also see Section 8.17 [DEFAULT-PRINT-PROMPT], page 255. The first examples all have `ld-skip-proofsp nil`; that is, proofs are *not* skipped.

```
ACL2 !>    ; logic mode with guard checking on
ACL2 >     ; logic mode with guard checking off
ACL2 p!>   ; program mode with guard checking on
ACL2 p>    ; program mode with guard checking off
```

Here are some examples with `default-defun-mode`^ of `:logic`^.

```
ACL2 >     ; guard checking off, ld-skip-proofsp nil
ACL2 s>    ; guard checking off, ld-skip-proofsp t
ACL2 !>    ; guard checking on, ld-skip-proofsp nil
ACL2 !s>   ; guard checking on, ld-skip-proofsp t
```

Sample session

```
ACL2 !>(+ 'abc 3)

ACL2 Error in TOP-LEVEL: The guard for the function symbol
BINARY-+, which is (AND (ACL2-NUMBERP X) (ACL2-NUMBERP Y)),
is violated by the arguments in the call (+ 'ABC 3).

ACL2 !>:set-guard-checking nil
;;;; verbose output omitted here
ACL2 >(+ 'abc 3)
3
ACL2 >(< 'abc 3)
T
ACL2 >(< 3 'abc)
NIL
ACL2 >(< -3 'abc)
T
ACL2 >:set-guard-checking t

Turning guard checking on.

ACL2 !>(defun sum-list (x)
        (declare (xargs :guard (integer-listp x)
                        :verify-guards nil))
```

```
        (cond ((endp x) 0)
              (t (+ (car x) (sum-list (cdr x))))))

The admission of SUM-LIST is trivial, using the relation
E0-ORD-< (which is known to be well-founded on the domain
recognized by E0-ORDINALP) and the measure (ACL2-COUNT X).
We observe that the type of SUM-LIST is described by the
theorem (ACL2-NUMBERP (SUM-LIST X)).  We used primitive type
reasoning.

Summary
Form:  ( DEFUN SUM-LIST ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.03 seconds
   (prove: 0.00, print: 0.00, proof tree: 0.00, other: 0.03)
 SUM-LIST
ACL2 !>(sum-list '(1 2 3))
6
ACL2 !>(sum-list '(1 2 abc 3))

ACL2 Error in TOP-LEVEL: The guard for the function symbol
BINARY-+, which is (AND (ACL2-NUMBERP X) (ACL2-NUMBERP Y)),
is violated by the arguments in the call (+ 'ABC 3).

ACL2 !>:set-guard-checking nil
;;;; verbose output omitted here
ACL2 >(sum-list '(1 2 abc 3))
6
ACL2 >(defthm sum-list-append
       (equal (sum-list (append a b))
              (+ (sum-list a) (sum-list b))))

<< Starting proof tree logging >>

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction
schemes are suggested by this conjecture.  Subsumption
reduces that number to two.  However, one of these is flawed
and so we are left with one viable candidate.

...

That completes the proof of *1.

Q.E.D.
```

Guard verification vs. defun

```
      Declare Form                        Guards Verified?

  (declare (xargs :mode :program ...))          no
  (declare (xargs :guard g))                    yes
  (declare (xargs :guard g :verify-guards nil)) no
  (declare (xargs ...<no :guard>...))           no

ACL2 >:pe sum-list
 l       8  (DEFUN SUM-LIST (X)
              (DECLARE (XARGS :GUARD (INTEGER-LISTP X)
                              :VERIFY-GUARDS NIL))
              (COND ((ENDP X) 0)
                    (T (+ (CAR X) (SUM-LIST (CDR X))))))
ACL2 >(verify-guards sum-list)
The non-trivial part of the guard conjecture for SUM-LIST,
given the :type-prescription rule SUM-LIST, is

Goal
(AND (IMPLIES (AND (INTEGER-LISTP X) (NOT (CONSP X)))
              (EQUAL X NIL))
     (IMPLIES (AND (INTEGER-LISTP X) (NOT (ENDP X)))
              (INTEGER-LISTP (CDR X)))
     (IMPLIES (AND (INTEGER-LISTP X) (NOT (ENDP X)))
              (ACL2-NUMBERP (CAR X)))).

...

ACL2 >:pe sum-list
 lv      8  (DEFUN SUM-LIST (X)
              (DECLARE (XARGS :GUARD (INTEGER-LISTP X)
                              :VERIFY-GUARDS NIL))
ACL2 >:set-guard-checking t

ACL2 !>(sum-list '(1 2 abc 3))

ACL2 Error in TOP-LEVEL: The guard for the function symbol
SUM-LIST, which is (INTEGER-LISTP X), is violated by the
arguments in the call (SUM-LIST '(1 2 ABC ...)).

ACL2 !>:set-guard-checking nil

ACL2 >(sum-list '(1 2 abc 3))
6
ACL2 >:comp sum-list
Compiling gazonk0.lsp.
```

```
End of Pass 1.
End of Pass 2.
Finished compiling gazonk0.lsp.
Loading gazonk0.o
start address -T 1bbf0b4 Finished loading gazonk0.o
Compiling gazonk0.lsp.
End of Pass 1.
End of Pass 2.
Finished compiling gazonk0.lsp.
Loading gazonk0.o
start address -T 1bc4408 Finished loading gazonk0.o
 SUM-LIST
ACL2 >:q

Exiting the ACL2 read-eval-print loop.
ACL2>(trace sum-list)
(SUM-LIST)

ACL2>(lp)

ACL2 Version 1.8.  Level 1.  Cbd "/slocal/src/acl2/v1-8/".
Type :help for help.
ACL2 >(sum-list '(1 2 abc 3))
6
ACL2 >(sum-list '(1 2 3))
  1> (SUM-LIST (1 2 3))>
    2> (SUM-LIST (2 3))>
      3> (SUM-LIST (3))>
        4> (SUM-LIST NIL)>
        <4 (SUM-LIST 0)>
      <3 (SUM-LIST 3)>
    <2 (SUM-LIST 5)>
  <1 (SUM-LIST 6)>
6
ACL2 >:pe sum-list-append
          9  (DEFTHM SUM-LIST-APPEND
                     (EQUAL (SUM-LIST (APPEND A B))
                            (+ (SUM-LIST A) (SUM-LIST B))))
ACL2 >(verify-guards sum-list-append)

The non-trivial part of the guard conjecture for
SUM-LIST-APPEND, given the :type-prescription rule SUM-LIST,
is

Goal
(AND (TRUE-LISTP A)
     (INTEGER-LISTP (APPEND A B))
     (INTEGER-LISTP A)
```

```
     (INTEGER-LISTP B)).

...

****** FAILED ******* See :DOC failure ****** FAILED ******
ACL2 >(defthm common-lisp-sum-list-append
         (if (and (integer-listp a)
                  (integer-listp b))
             (equal (sum-list (append a b))
                    (+ (sum-list a) (sum-list b)))
             t)
         :rule-classes nil)

<< Starting proof tree logging >>

By the simple :rewrite rule SUM-LIST-APPEND we reduce the
conjecture to

Goal'
(IMPLIES (AND (INTEGER-LISTP A)
              (INTEGER-LISTP B))
         (EQUAL (+ (SUM-LIST A) (SUM-LIST B))
                (+ (SUM-LIST A) (SUM-LIST B)))).

But we reduce the conjecture to T, by primitive type
reasoning.

Q.E.D.
;;;; summary omitted here
ACL2 >(verify-guards common-lisp-sum-list-append)

The non-trivial part of the guard conjecture for
COMMON-LISP-SUM-LIST-APPEND, given the :type-prescription
rule SUM-LIST, is

Goal
(AND (IMPLIES (AND (INTEGER-LISTP A)
                   (INTEGER-LISTP B))
              (TRUE-LISTP A))
     (IMPLIES (AND (INTEGER-LISTP A)
                   (INTEGER-LISTP B))
              (INTEGER-LISTP (APPEND A B)))).

...

Q.E.D.

That completes the proof of the guard theorem for
```

```
COMMON-LISP-SUM-LIST-APPEND.  COMMON-LISP-SUM-LIST-APPEND
is compliant with Common Lisp.
;;;; Summary omitted here.
ACL2 >(defthm foo (consp (mv x y)))

...

Q.E.D.


ACL2 >(verify-guards foo)

ACL2 Error in (VERIFY-GUARDS FOO): The number of values we
need to return is 1 but the number of values returned by the
call (MV X Y) is 2.

> (CONSP (MV X Y))


ACL2 Error in (VERIFY-GUARDS FOO): The guards for FOO cannot
be verified because the theorem has the wrong syntactic
form.  See :DOC verify-guards.
```

1.5.6.3 MUTUAL-RECURSION-PROOF-EXAMPLE

MUTUAL-RECURSION-PROOF-EXAMPLE   a small proof about mutually recursive functions

Sometimes one wants to reason about mutually recursive functions. Although this is possible in ACL2, it can be a bit awkward. This example is intended to give some ideas about how one can go about such proofs.

For an introduction to mutual recursion in ACL2, see Section 6.24 [MUTUAL-RECURSION], page 184.

We begin by defining two mutually recursive functions: one that collects the variables from a term^, the other that collects the variables from a list of term^s. We actually imagine the term^ argument to be a `pseudo-termp`^; see Section 8.72 [PSEUDO-TERMP], page 335.

```
(mutual-recursion

(defun free-vars1 (term ans)
  (cond ((atom term)
```

```
           (add-to-set-eq term ans))
          ((fquotep term) ans)
          (t (free-vars1-lst (fargs term) ans))))

(defun free-vars1-lst (lst ans)
  (cond ((atom lst) ans)
        (t (free-vars1-lst (cdr lst)
                           (free-vars1 (car lst) ans)))))

)
```

The idea of the following function is that it suggests a proof by induction in two cases, according to the top-level `if^` structure of the body. In one case, `(atom x)` is true, and the theorem to be proved should be proved under no additional hypotheses except for `(atom x)`. In the other case, `(not (atom x))` is assumed together with three instances of the theorem to be proved, one for each recursive call in this case. So, one instance substitutes `(car x)` for `x`; one substitutes `(cdr x)` for `x`; and the third substitutes `(cdr x)` for `x` and `(free-vars1 (car x) ans)` for `ans`. If you think about how you would go about a hand proof of the theorem to follow, you'll come up with a similar scheme.

```
(defun symbol-listp-free-vars1-induction (x ans)
  (if (atom x)
; then we just make sure x and ans aren't considered irrelevant
      (list x ans)
    (list (symbol-listp-free-vars1-induction (car x) ans)
          (symbol-listp-free-vars1-induction (cdr x) ans)
          (symbol-listp-free-vars1-induction
           (cdr x)
           (free-vars1 (car x) ans)))))
```

We now prove the two theorems together as a conjunction, because the inductive hypotheses for one are sometimes needed in the proof of the other (even when you do this proof on paper!).

```
(defthm symbol-listp-free-vars1
  (and (implies (and (pseudo-termp x)
                     (symbol-listp ans))
                (symbol-listp (free-vars1 x ans)))
       (implies (and (pseudo-term-listp x)
                     (symbol-listp ans))
                (symbol-listp (free-vars1-lst x ans))))
  :hints
  (("Goal" :induct (symbol-listp-free-vars1-induction x ans))))
```

The above works, but let's try for a more efficient proof, which avoids cluttering the proof with irrelevant (false) inductive hypotheses.

```
(ubt 'symbol-listp-free-vars1-induction)

(defun symbol-listp-free-vars1-induction (flg x ans)

; Flg is non-nil (or t) if we are ``thinking'' of a single term.

  (if (atom x)
      (list x ans)
    (if flg
        (symbol-listp-free-vars1-induction nil (cdr x) ans)
      (list (symbol-listp-free-vars1-induction t (car x) ans)
            (symbol-listp-free-vars1-induction
             nil
             (cdr x)
             (free-vars1 (car x) ans))))))
```

We now state the theorem as a conditional, so that it can be proved nicely using the induction^ scheme that we have just coded. The prover will not store an `if^` term^ as a rewrite^ rule, but that's OK (as long as we tell it not to try), because we're going to derive the corollaries of interest later and make **them** into rewrite^ rules.

```
(defthm symbol-listp-free-vars1-flg
  (if flg
      (implies (and (pseudo-termp x)
                    (symbol-listp ans))
               (symbol-listp (free-vars1 x ans)))
    (implies (and (pseudo-term-listp x)
                  (symbol-listp ans))
             (symbol-listp (free-vars1-lst x ans))))
  :hints
  (("Goal" :induct (symbol-listp-free-vars1-induction flg x ans)))
  :rule-classes nil)
```

And finally, we may derive the theorems we are interested in as immediate corollaries.

```
(defthm symbol-listp-free-vars1
  (implies (and (pseudo-termp x)
                (symbol-listp ans))
           (symbol-listp (free-vars1 x ans)))
  :hints (("Goal" :by (:instance symbol-listp-free-vars1-flg
                                 (flg t)))))
```

```
(defthm symbol-listp-free-vars1-lst
  (implies (and (pseudo-term-listp x)
                (symbol-listp ans))
           (symbol-listp (free-vars1-lst x ans)))
  :hints (("Goal" :by (:instance symbol-listp-free-vars1-flg
                                 (flg nil)))))
```

2 BDD

BDD ordered binary decision diagrams with rewriting

Ordered binary decision diagrams (OBDDs, often simply called BDDs) are a technique, originally published by Randy Bryant, for the efficient simplification of Boolean expressions. In ACL2 we combine this technique with rewriting to handle arbitrary ACL2 terms that can represent not only Boolean values, but non-Boolean values as well. In particular, we provide a setting for deciding equality of bit vectors (lists of Boolean values).

* BDD-ALGORITHM:: summary of the BDD algorithm in ACL2
* BDD-INTRODUCTION:: examples illustrating the use of BDDs in ACL2
* IF*:: for conditional rewriting with BDDs
* SHOW-BDD:: inspect failed BDD proof attempts

An introduction to BDDs for the automated reasoning community may be found in "Introduction to the OBDD Algorithm for the ATP Community" by J Moore, *Journal of Automated Reasoning* (1994), pp. 33-45. (This paper also appears as Technical Report #84 from Computational Logic, Inc.)

Further information about BDDs in ACL2 can be found in the subtopics of this documentation^ section. In particular, see Section 2.2 [BDD-INTRODUCTION], page 79 for a good starting place that provides a number of examples.

See Section 8.41 [HINTS], page 297 for a description of `:bdd` hints. For quick reference, here is an example; but only the `:vars` part of the hint is required, as explained in the documentation for hints^. The values shown are the defaults.

```
(:vars nil :bdd-constructors (cons) :prove t :literal :all)
```

We suggest that you next visit the documentation topic BDD-INTRODUCTION^.

2.1 BDD-ALGORITHM

BDD-ALGORITHM summary of the BDD algorithm in ACL2

The BDD algorithm in ACL2 uses a combination of manipulation of `IF` terms and unconditional rewriting. In this discussion we begin with some relevant mathematical theory. This is followed by a description of how ACL2 does BDDs, including concluding discussions of soundness, completeness, and efficiency.

We recommend that you read the other documentation about BDDs in ACL2 before reading the rather technical material that follows. See Chapter 2 [BDD], page 69.

Here is an outline of our presentation. Readers who want a user perspective, without undue mathematical theory, may wish to skip to Part (B), referring to Part (A) only on occasion if necessary.

(A) **Mathematical Considerations**

> (A1) BDD term order
>
> (A2) BDD-constructors and BDD terms, and their connection with aborting the BDD algorithm
>
> (A3) Canonical BDD terms
>
> (A4) A theorem stating the equivalence of provable and syntactic equality for canonical BDD terms

(B) **Algorithmic Considerations**

> (B1) BDD rules (rules used by the rewriting portion of the ACL2 BDD algorithm)
>
> (B2) Terms "known to be Boolean"
>
> (B3) An "IF-lifting" operation used by the algorithm, as well as an iterative version of that operation
>
> (B4) The ACL2 BDD algorithm
>
> (B5) Soundness and Completeness of the ACL2 BDD algorithm
>
> (B6) Efficiency considerations

(A) **Mathematical Considerations**

(A1) *BDD term order*

Our BDD algorithm creates a total "BDD term order" on ACL2 terms, on the fly. We use this order in our discussions below of IF-lifting and of canonical BDD terms, and in the algorithm's use of commutativity. The particular order is unimportant, except that we guarantee (for purposes of commutative functions) that constants are smaller in this order than non-constants.

(A2) *BDD-constructors* (assumed to be `'(cons)`) and *BDD terms*

We take as given a list of function symbols that we call the "BDD-constructors." By default, the only BDD-constructor is `cons^`, although it is legal to specify any list of function symbols as the BDD-constructors, either by using the acl2-defaults-table^ (see Section 9.2 [ACL2-DEFAULTS-TABLE], page 374) or by supplying a `:BDD-CONSTRUCTORS` hint (see Section 8.41 [HINTS], page 297). Warning: this capability is largely untested and may produce undesirable results. Henceforth, except when explicitly stated to the contrary, we assume that BDD-constructors is `'(cons)`.

Roughly speaking, a BDD^ term is the sort of term^ produced by our BDD algorithm, namely a tree with all `cons^` nodes lying above all non-`CONS` nodes. More formally, a term^ is said to be a BDD^ term if it contains **no** subterm of either of the following forms, where `f` is not `CONS`.

```
(f ... (CONS ...) ...)

(f ... 'x ...)  ; where (consp x) = t
```

We will see that whenever the BDD algorithm attempts to create a term^ that is not a BDD^ term, it aborts instead. Thus, whenever the algorithm completes without aborting, it creates a BDD^ term.

(A3) *Canonical BDD terms*

We can strengthen the notion of "BDD term" to a notion of "canonical BDD term" by imposing the following additional requirements, for every subterm of the form `(IF x y z)`:

(a) `x` is a variable, and it precedes (in the BDD term order) every variable occurring in `y` or `z`;

(b) `y` and `z` are syntactically distinct; and,

(c) it is not the case that `y` is `t` and `z` is `nil`.

We claim that it follows easily from our description of the BDD algorithm that every term it creates is a canonical BDD term, assuming that the variables occurring in all such terms are treated by the algorithm as being Boolean (see (B2) below) and that the terms contain no function symbols other than `IF` and `CONS`. Thus, under those assumptions the following theorem shows that the BDD algorithm never creates distinct terms that are provably equal, a property that is useful for completeness and efficiency (as we explain in (B5) and (B6) below).

(A4) *Provably equal canonical BDD terms are identical*

We believe that the following theorem and proof are routine extensions of a standard result and proof to terms that allow calls of `CONS`.

Theorem. Suppose that `t1` and `t2` are canonical BDD terms that contain no function symbols other than `IF` and `CONS`. Also suppose that `(EQUAL t1 t2)` is a theorem. Then `t1` and `t2` are syntactically identical.

Proof of theorem: By induction on the total number of symbols occurring in these two terms. First suppose that at least one term is a variable; without loss of generality let it be `t1`. We must prove that `t2` is syntactically the same as `t1`. Now it is clearly consistent that `(EQUAL t1 t2)` is false if `t2` is a call of `CONS` (to see this, simply let `t1` be an value that is not a `CONSP`). Similarly, `t2` cannot be a constant or a variable other than `t1`. The remaining possibility to rule out is that `t2` is of the form `(IF t3 t4 t5)`, since by assumption its function symbol must be `IF` or `CONS` and we have already handled the latter case. Since `t2` is canonical, we know that `t3` is a variable. Since `(EQUAL t1 t2)` is provable, i.e.,

```
(EQUAL t1 (if t3 t4 t5))
```

is provable, it follows that we may substitute either `t` or `nil` for `t3` into this equality to obtain two new provable equalities. First, suppose that `t1` and `t3` are distinct variables. Then these substitutions show that `t1` is provably equal to both `t4` and `t5` (since `t3` does not occur in `t4` or `t5` by property (a) above, as `t2` is canonical), and hence `t4` and `t5` are provably equal to each other, which implies by the inductive hypothesis that they are the same term — and this contradicts the assumption that `t2` is canonical (property (b)). Therefore `t1` and `t3` are the same variable, i.e., the equality displayed above is actually `(EQUAL t1 (if t1 t4 t5))`. Substituting `t` and then `nil` for `t1` into this provable equality lets us prove `(EQUAL t t4)` and `(EQUAL nil t5)`, which by the inductive hypothesis implies that `t4` is (syntactically) the term `t` and `t5` is `nil`. That is, `t2` is `(IF t1 t nil)`, which contradicts the assumption that `t2` is canonical (property (c)).

Next, suppose that at least one term is a call of `IF`. Our first observation is that the other term is also a call of `IF`. For if the other is a call of `CONS`, then they cannot be provably equal, because the former has no function symbols other than `IF` and hence is Boolean when all its variables are assigned Boolean values. Also, if the other is a constant, then both branches of the `IF` term are provably equal to that constant and hence these branches are syntactically identical by the inductive hypothesis, contradicting property (b). Hence, we may assume for this case that both terms are calls of `IF`; let us write them as follows.

```
t0:  (IF t1 t2 t3)
u0:  (IF u1 u2 u3)
```

Note that `t1` and `u1` are variables, by property (a) of canonical BDD terms. First we claim that `t1` does not strictly precede `u1` in the BDD term order. For suppose `t1` does strictly precede `u1`. Then property (a) of canonical BDD terms guarantees that `t1` does not occur in `u0`. Hence, an argument much like one used above shows that `u0` is provably equal to both `t2` (substituting `t` for `t1`) and `t3` (substituting `nil` for `t1`), and hence `t2` and `t3` are provably equal. That implies that they are identical terms, by the inductive hypothesis, which then contradicts property (b) for `u0`. Similarly, `u1` does not strictly precede `t1` in the BDD term order. Therefore, `t1` and `u1` are the same variable. By substituting `t` for this variable we see that `t2` and `u2` are provably equal, and hence they are equal by the inductive hypothesis. Similarly, by substituting `nil` for `t1` (and `u1`) we see that `t3` and `u3` are provably, hence syntactically, equal.

We have covered all cases in which at least one term is a variable or at least one term is a call of `IF`. If both terms are constants, then provable and syntactic equality are clearly equivalent. Finally, then, we may assume that one term is a call of `CONS` and the other is a constant or a call of `CONS`. The constant case is similar to the `CONS` case if the constant is a `CONSP`, so we omit it; while if the constant is not a `CONSP` then it is not provably equal to a call of `CONS`; in fact it is provably *not* equal!

So, we are left with a final case, in which canonical BDD terms `(CONS t1 t2)` and `(CONS u1 u2)` are provably equal, and we want to show that `t1` and `u1` are syntactically equal as are `t2` and `u2`. These conclusions are easy consequences of the inductive hypothesis, since the ACL2 axiom `CONS-EQUAL` (which you can inspect using `:PE^`) shows that equality of the given terms is equivalent to the conjunction of `(EQUAL t1 t2)` and `(EQUAL u1 u2)`. Q.E.D.

(B) **Algorithmic Considerations**

(B1) *BDD rules*

A rule of class `:rewrite^` (see Chapter 14 [RULE-CLASSES], page 617) is said to be a "BDD^ rewrite rule" if and only if it satisfies the following criteria. (1) The rule is enable^d. (2) Its equivalence^ relation is `equal^`. (3) It has no hypotheses. (4) Its `:loop-stopper^` field is `nil`, i.e., it is not a permutative rule. (5) All variables occurring in the rule occur in its left-hand side (i.e., there are no "free variables"; see Section 14.14 [REWRITE], page 657). A rule of class `:definition^` (see Chapter 14 [RULE-CLASSES], page 617) is said to be a "BDD^ definition rule" if it satisfies all the criteria above (except (4), which does not apply), and moreover the top function symbol of the left-hand side was not recursively (or mutually recursively) defined. Technical point: Note that this additional criterion is independent of whether or not the indicated function symbol actually occurs in the right-hand side of the rule.

Both BDD rewrite rules and BDD definition rules are said to be "BDD rules."

(B2) *Terms "known to be Boolean"*

We apply the BDD algorithm in the context of a top-level goal to prove, namely, the goal at which the `:BDD` hint is attached. As we run the BDD algorithm, we allow ourselves to say that a set of term^s is "known to be Boolean" if we can verify that the goal is provable from the assumption that at least one of the terms is not Boolean. Equivalently, we allow ourselves to say that a set of terms is "known to be Boolean" if we can verify that the original goal is provably equivalent to the assertion that if all terms in the set are Boolean, then the goal holds. The notion "known to be Boolean" is conservative in the sense that there are generally sets of terms for which the above equivalent criteria hold and yet the sets of terms are not noted as as being "known to be Boolean." However, ACL2 uses a number of tricks, including type-set^ reasoning and analysis of the structure of the top-level goal, to attempt to establish that a sufficiently inclusive set of terms is known to be Boolean.

From a practical standpoint, the algorithm determines a set of terms known to be Boolean; we allow ourselves to say that each term in this set if "known to be Boolean." The algorithm assumes that these terms are indeed Boolean, and can make use of that assumption. For example, if `t1` is known to be Boolean then the algorithm simplifies `(IF t1 t nil)` to `t1`; see (iv) in the discussion immediately below.

(B3) *IF-lifting* and the *IF-lifting-for-IF loop*

Suppose that one has a term^ of the form `(f ... (IF test x y) ...)`, where `f` is a function symbol other than `CONS`. Then we say that "IF-lifting" `test` "from" this term produces the following term, which is provably equal to the given term.

```
(if test
    (f ... x ...)  ; resulting true branch
    (f ... y ...)) ; resulting false branch
```

Here, we replace each argument of `f` of the form `(IF test .. ..)`, for the same `test`, in the same way. In this case we say that "IF-lifting applies to" the given term, "yielding the test" `test` and with the "resulting two branches" displayed above. Whenever we apply IF-lifting, we do so for the available `test` that is least in the BDD term order (see (A1) above).

We consider arguments `v` of `f` that are "known to be Boolean" (see above) to be replaced by `(IF v t nil)` for the purposes of IF-lifting, i.e., before IF-lifting is applied.

There is one special case, however, for IF-lifting. Suppose that the given term is of the form `(IF v y z)` where `v` is a variable and is the test to be lifted out (i.e., it is least in the BDD term order among the potential tests). Moroever, suppose that neither `y` nor `z` is of the form `(IF v W1 W2)` for that same `v`. Then IF-lifting does not apply to the given term.

We may now describe the IF-lifting-for-IF loop, which applies to terms of the form `(IF test tbr fbr)` where the algorithm has already produced `test`, `tbr`, and `fbr`. First, if `test` is `nil` then we return `fbr`, while if `test` is a non-`nil` constant or a call of `CONS` then we return `tbr`. Otherwise, we see if IF-lifting applies. If IF-lifting does not apply, then we return `(IF test tbr fbr)`. Otherwise, we apply IF-lifting to obtain a term of the form `(IF x y z)`, by lifting out the appropriate test. Now we recursively apply the IF-lifting-for-IF loop to the term `(IF x y z)`, unless any of the following special cases apply.

(i) If `y` and `z` are the same term, then return `y`.

(ii) Otherwise, if `x` and `z` are the same term, then replace `z` by `nil` before recursively applying IF-lifting-for-IF.

(iii) Otherwise, if `x` and `y` are the same term and `y` is known to be Boolean, then replace `y` by `t` before recursively applying IF-lifting-for-IF.

(iv) If `z` is `nil` and either `x` and `y` are the same term or `x` is "known to be Boolean" and `y` is `t`, then return `x`.

NOTE: When a variable `x` is known to be Boolean, it is easy to see that the form `(IF x t nil)` is always reduced to `v` by this algorithm.

(B4) *The ACL2 BDD algorithm*

We are now ready to present the BDD algorithm for ACL2. It is given an ACL2 term^, `x`, as well as an association list `va` that maps variables to terms, including all variables occurring in `x`. We maintain the invariant that whenever a variable is mapped by `va` to a term, that term has already been constructed by the algorithm, except: initially `va` maps every variable occurring in the top-level term to itself. The algorithm proceeds as follows. We implicitly ordain that whenever the BDD algorithm attempts to create a term^ that is not a BDD^ term (as defined above in (A2)), it aborts instead. Thus, whenever the algorithm completes without aborting, it creates a BDD^ term.

If `x` is a variable, return the result of looking it up in `va`.

If `x` is a constant, return `x`.

If `x` is of the form `(IF test tbr fbr)`, then first run the algorithm on `test` with the given `va` to obtain `test'`. If `test'` is `nil`, then return the result `fbr'` of running the algorithm on `fbr` with the given `va`. If `test'` is a constant other than `nil`, or is a call of `CONS`, then return the result `tbr'` of running the algorithm on `tbr` with the given `va`. If `tbr` is identical to `fbr`, return `tbr`. Otherwise, return the result of applying the IF-lifting-for-IF loop (described above) to the term `(IF test' tbr' fbr')`.

If `x` is of the form `(IF* test tbr fbr)`, then compute the result exactly as though `IF`^ were used rather than `IF*`^, except that if `test'` is not a constant or a call of `CONS` (see paragraph above), then abort the BDD computation. Informally, the tests of `IF*`^ terms are expected to "resolve." NOTE: This description shows how `IF*`^ can be used to implement conditional rewriting in the BDD algorithm.

If `x` is a `LAMBDA` expression `((LAMBDA vars body) . args)` (which often corresponds to a `LET`^ term; see Section 10.141 [LET], page 460), then first form an alist `va'` by binding each `v` in `vars` to the result of running the algorithm on the corresponding member of `args`, with the current alist `va`. Then, return the result of the algorithm on `body` in the alist `va'`.

Otherwise, `x` is of the form `(f x1 x2 ... xn)`, where `f` is a function symbol other than `IF`^ or `IF*`^. In that case, let `xi'` be the result of running the algorithm on `xi`, for `i` from 1 to `n`, using the given alist `va`. First there are a few special cases. If `f` is `EQUAL`^ then we return `t` if `x1'` is syntactically identical to `x2'` (where this test is very fast; see (B6) below); we return `x1'` if it is known to be Boolean and `x2'` is `t`; and similarly, we return `x2'` if it is known to be Boolean and `x1'` is `t`. Next, if each `xi'` is a constant and the `:executable-counterpart`^ of `f` is enabled, then the result is obtained by computation. Next, if `f` is `BOOLEANP`^ and `x1'` is known to be Boolean, `t` is returned. Otherwise, we proceed as follows, first possibly swapping the arguments if they are out of (the BDD term) order and if `f` is known to be commutative (see below). If a BDD rewrite rule (as defined above) matches the term `(f x1'... xn')`, then the most recently stored such rule is applied. If there is no such match and `f` is a

> BDD-constructor, then we return `(f x1'... xn')`. Otherwise, if a BDD definition rule matches this term, then the most recently stored such rule (which will usually be the original definition for most users) is applied. If none of the above applies and neither does IF-lifting, then we return `(f x1'... xn')`. Otherwise we apply IF-lifting to `(f x1'... xn')` to obtain a term `(IF test tbr fbr)`; but we aren't done yet. Rather, we run the BDD algorithm (using the same alist) on `tbr` and `fbr` to obtain terms `tbr'` and `fbr'`, and we return `(IF test tbr' fbr')` unless `tbr'` is syntactically identical to `fbr'`, in which case we return `tbr'`.

When is it the case that, as said above, "`f` is known to be commutative"? This happens when an enabled rewrite rule is of the form `(EQUAL (f X Y) (f Y X))`. Regarding swapping the arguments in that case: recall that we may assume very little about the BDD term order, essentially only that we swap the two arguments when the second is a constant and the first is not, for example, in `(+ x 1)`. Other than that situation, one cannot expect to predict accurately when the arguments of commutative operators will be swapped.

(B5) Soundness and Completeness of the ACL2 BDD algorithm

Roughly speaking, "soundness" means that the BDD algorithm should give correct answers, and "completeness" means that it should be powerful enough to prove all true facts. Let us make the soundness claim a little more precise, and then we'll address completeness under suitable hypotheses.

Claim (Soundness). If the ACL2 BDD algorithm runs to completion on an input term `t0`, then it produces a result that is provably equal to `t0`.

We leave the proof of this claim to the reader. The basic idea is simply to check that each step of the algorithm preserves the meaning of the term under the bindings in the given alist.

Let us start our discussion of completeness by recalling the theorem proved above in (A4).

Theorem. Suppose that `t1` and `t2` are canonical BDD terms that contain no function symbols other than `IF` and `CONS`. Also suppose that `(EQUAL t1 t2)` is a theorem. Then `t1` and `t2` are syntactically identical.

Below we show how this theorem implies the following completeness property of the ACL2 BDD algorithm. We continue to assume that `CONS` is the only BDD-constructor.

Claim (Completeness). Suppose that `t1` and `t2` are provably equal terms, under the assumption that all their variables are known to be Boolean. Assume further that under this same assumption, top-level runs of the ACL2 BDD algorithm on these terms return terms that contain only the function symbols `IF` and `CONS`. Then the algorithm returns the same term for both `t1` and `t2`, and the algorithm reduces `(EQUAL t1 t2)` to `t`.

Why is this claim true? First, notice that the second part of the conclusion follows immediately from the first, by definition of the algorithm. Next, notice that the terms `u1` and `u2` obtained by running the algorithm on `t1` and `t2`, respectively, are provably equal to `t1` and `t2`, respectively, by the Soundness Claim. It follows that `u1` and `u2` are provably equal to each other. Since these terms contain no function symbols other than `IF` or `CONS`, by hypothesis, the Claim now follows from the Theorem above together with the following lemma.

Lemma. Suppose that the result of running the ACL2 BDD algorithm on a top-level term `t0` is a term `u0` that contains only the function symbols `IF` and `CONS`, where all variables of `t0` are known to be Boolean. Then `u0` is a canonical BDD term.

Proof: left to the reader. Simply follow the definition of the algorithm, with a separate argument for the IF-lifting-for-IF loop.

Finally, let us remark on the assumptions of the Completeness Claim above. The assumption that all variables are known to be Boolean is often true; in fact, the system uses the forward-chaining rule `boolean-listp-forward` (you can see it using `:pe^`) to try to establish this assumption, if your theorem has a form such as the following.

```
(let ((x (list x0 x1 ...))
      (y (list y0 y1 ...)))
  (implies (and (boolean-listp x)
                (boolean-listp y))
           ...))
```

Moreover, the `:BDD` hint can be used to force the prover to abort if it cannot check that the indicated variables are known to be Boolean; see Section 8.41 [HINTS], page 297.

Finally, consider the effect in practice of the assumption that the terms resulting from application of the algorithm contain calls of `IF` and `CONS` only. Typical use of BDDs in ACL2 takes place in a theory (see Chapter 15 [THEORIES], page 667) in which all relevant non-recursive function symbols are enabled and all recursive function symbols possess enabled BDD rewrite rules that tell them how open up. For example, such a rule may say how to expand on a given function call's argument that has the form `(CONS a x)`, while another may say how to expand when that argument is `nil`). (See for example the rules `append-cons` and `append-nil` in the documentation for `IF*^`.) We leave it to future work to formulate a theorem that guarantees that the BDD algorithm produces terms containing calls only of `IF` and `CONS` assuming a suitably "complete" collection of rewrite rules.

(B6) *Efficiency considerations*

Following Bryant's algorithm, we use a graph representation of term^s created by the BDD algorithm's computation. This representation enjoys some important properties.

> (Time efficiency) The test for syntactic equality of BDD terms is very fast.
>
> (Space efficiency) Equal BDD data structures are stored identically in memory.

Implementation note. The representation actually uses a sort of hash table for BDD terms that is implemented as an ACL2 1-dimensional array. See Section 8.6 [ARRAYS], page 226. In addition, we use a second such hash table to avoid recomputing the result of applying a function symbol to the result of running the algorithm on its arguments. We believe that these uses of hash tables are standard. They are also discussed in Moore's paper on BDDs; see Chapter 2 [BDD], page 69 for the reference.

2.2 BDD-INTRODUCTION

BDD-INTRODUCTION examples illustrating the use of BDDs in ACL2

See Chapter 2 [BDD], page 69 for a brief introduction to BDDs in ACL2 and for pointers to other documentation on BDDs in ACL2. Here, we illustrate the use of BDDs in ACL2 by way of some examples. For a further example, see Section 2.3 [IF*], page 84.

Let us begin with a really simple example. (We will explain the `:bdd` hint `(:vars nil)` below.)

```
ACL2 !>(thm (equal (if a b c) (if (not a) c b))
            :hints (("Goal" :bdd (:vars nil)))) ; Prove with BDDs

[Note:  A hint was supplied for our processing of the goal above.
Thanks!]

But simplification with BDDs (7 nodes) reduces this to T, using the
:definitions EQUAL and NOT.

Q.E.D.
```

```
Summary
Form:  ( THM ...)
Rules: ((:DEFINITION EQUAL) (:DEFINITION NOT))
Warnings:  None
Time:  0.18 seconds (prove: 0.05, print: 0.02, other: 0.12)

Proof succeeded.
ACL2 !>
```

The `:bdd` hint `(:vars nil)` indicates that BDDs are to be used on the indicated goal, and that any so-called "variable ordering" may be used: ACL2 may use a convenient order that is far from optimal. It is beyond the scope of the present documentation to address the issue of how the user may choose good variable orderings. Someday our implementation of BDDs may be improved to include heuristically-chosen variable orderings rather than rather random ones.

Here is a more interesting example.

```
(defun v-not (x)
; Complement every element of a list of Booleans.
  (if (consp x)
      (cons (not (car x)) (v-not (cdr x)))
    nil))

; Now we prove a rewrite rule that explains how to open up v-not on
; a consp.
(defthm v-not-cons
  (equal (v-not (cons x y))
         (cons (not x) (v-not y))))

; Finally, we prove for 7-bit lists that v-not is self-inverting.
(thm
 (let ((x (list x0 x1 x2 x3 x4 x5 x6)))
   (implies (boolean-listp x)
            (equal (v-not (v-not x)) x)))
 :hints (("Goal" :bdd
                 ;; Note that this time we specify a variable order.
                 (:vars (x0 x1 x2 x3 x4 x5 x6)))))
```

It turns out the the variable order doesn't seem to matter in this example; using several orders we found that 30 nodes were created, and the proof time was about 1/10 of a second on a (somewhat enhanced) Sparc 2. The same proof took about a minute and a half without any `:bdd` hint! This observation is a bit misleading perhaps, since the theorem for arbitrary `x`,

```
(thm
 (implies (boolean-listp x)
          (equal (v-not (v-not x)) x)))
```

only takes about 1.5 times as long as the `:bdd` proof for 7 bits, above! Nevertheless, BDDs can be very useful in reducing proof time, especially when there is no regular structure to facilitate proof by induction, or when the induction scheme is so complicated to construct that significant user effort is required to get the proof by induction to go through.

Finally, consider the preceding example, with a `:bdd` hint of (say) `(:vars nil)`, but with the rewrite rule `v-not-cons` above disabled. In that case, the proof fails, as we see below. That is because the BDD algorithm in ACL2 uses hypothesis-free :rewrite^ rules, `:executable-counterpart^s`, and nonrecursive definitions, but it does not use recursive definitions.

Notice that when we issue the `(show-bdd)` command, the system's response clearly shows that we need a rewrite rule for simplifying terms of the form `(v-not (cons ...))`.

```
ACL2 !>(thm
        (let ((x (list x0 x1 x2 x3 x4 x5 x6)))
          (implies (boolean-listp x)
                   (equal (v-not (v-not x)) x)))
        :hints (("Goal" :bdd (:vars nil)
                 :in-theory (disable v-not-cons))))

[Note:  A hint was supplied for our processing of the goal above.
Thanks!]


ACL2 Error in ( THM ...):  Attempted to create V-NOT node during BDD
processing with an argument that is a call of a bdd-constructor,
which would produce a non-BDD term (as defined in :DOC
bdd-algorithm).  See :DOC show-bdd.


Summary
Form:  ( THM ...)
Rules: NIL
Warnings:  None
Time:  0.58 seconds (prove: 0.13, print: 0.00, other: 0.45)

******** FAILED ********  See :DOC failure  ******** FAILED ********
ACL2 !>(show-bdd)

BDD computation on Goal yielded 17 nodes.
==============================
```

```
BDD computation was aborted on Goal, and hence there is no
falsifying assignment that can be constructed.  Here is a backtrace
of calls, starting with the top-level call and ending with the one
that led to the abort.  See :DOC show-bdd.

(LET ((X (LIST X0 X1 X2 X3 X4 X5 ...)))
     (IMPLIES (BOOLEAN-LISTP X)
              (EQUAL (V-NOT (V-NOT X)) X)))
  alist: ((X6 X6) (X5 X5) (X4 X4) (X3 X3) (X2 X2) (X1 X1) (X0 X0))

(EQUAL (V-NOT (V-NOT X)) X)
  alist: ((X (LIST X0 X1 X2 X3 X4 X5 ...)))

(V-NOT (V-NOT X))
  alist: ((X (LIST X0 X1 X2 X3 X4 X5 ...)))

(V-NOT X)
  alist: ((X (LIST X0 X1 X2 X3 X4 X5 ...)))
ACL2 !>
```

The term that has caused the BDD algorithm to abort is thus `(V-NOT X)`, where `X` has the value `(LIST X0 X1 X2 X3 X4 X5 ...)`, i.e., `(CONS X0 (LIST X1 X2 X3 X4 X5 ...))`. Thus, we see the utility of introducing a rewrite rule to simplify terms of the form `(V-NOT (CONS ...))`. The moral of this story is that if you get an error of the sort shown above, you may find it useful to execute the command `(show-bdd)` and use the result as advice that suggests the left hand side of a rewrite rule.

Here is another sort of failed proof. In this version we have omitted the hypothesis that the input is a bit vector. Below we use `show-bdd` to see what went wrong, and use the resulting information to construct a counterexample. This failed proof corresponds to a slightly modified input theorem, in which `x` is bound to the 4-bit list `(list x0 x1 x2 x3)`.

```
ACL2 !>(thm
        (let ((x (list x0 x1 x2 x3)))
          (equal (v-not (v-not x)) x))
        :hints (("Goal" :bdd
                 ;; This time we do not specify a variable order.
                 (:vars nil))))

[Note:  A hint was supplied for our processing of the goal above.
Thanks!]


ACL2 Error in ( THM ...):  The :BDD hint for the current goal has
```

```
successfully simplified this goal, but has failed to prove it.
Consider using (SHOW-BDD) to suggest a counterexample; see :DOC
show-bdd.


Summary
Form:  ( THM ...)
Rules: NIL
Warnings:  None
Time:  0.18 seconds (prove: 0.07, print: 0.00, other: 0.12)

******** FAILED ********  See :DOC failure  ******** FAILED ********
ACL2 !>(show-bdd)

BDD computation on Goal yielded 73 nodes.
==============================

Falsifying constraints:
((X0 "Some non-nil value")
 (X1 "Some non-nil value")
 (X2 "Some non-nil value")
 (X3 "Some non-nil value")
 ((EQUAL 'T X0) T)
 ((EQUAL 'T X1) T)
 ((EQUAL 'T X2) T)
 ((EQUAL 'T X3) NIL))

==============================

Term obtained from BDD computation on Goal:

(IF X0
    (IF X1
        (IF X2 (IF X3 (IF # # #) (IF X3 # #))
            (IF X2 'NIL (IF X3 # #)))
        (IF X1 'NIL
            (IF X2 (IF X3 # #) (IF X2 # #))))
    (IF X0 'NIL
        (IF X1 (IF X2 (IF X3 # #) (IF X2 # #))
            (IF X1 'NIL (IF X2 # #)))))

ACL2 Query (:SHOW-BDD):  Print the term in full?  (N, Y, W or ?):
n ; I've seen enough.  The assignment shown above suggests
  ; that if we bind x3 to a non-nil value other than T,
  ; and bind x0, x1, and x2 to t, then we expect to get a
  ; counterexample.
ACL2 !>(let ((x0 t) (x1 t) (x2 t) (x3 7))
         (let ((x (list x0 x1 x2 x3)))
```

```
              ;; Let's use LIST instead of EQUAL to see how the two
              ;; lists differ.
              (list (v-not (v-not x)) x)))
((T T T T) (T T T 7))
ACL2 !>
```

See Section 2.3 [IF*], page 84 for another example.

2.3 IF*

IF*   for conditional rewriting with BDDs

The function `IF*` is defined to be `IF^`, but it is used in a special way by ACL2's BDD^ package.

As explained elsewhere (see Section 2.1 [BDD-ALGORITHM], page 69), ACL2's BDD^ algorithm gives special treatment to term^s of the form `(IF* TEST TBR FBR)`. In such cases, the algorithm simplifies `TEST` first, and the result of that simplification must be a constant (normally `t` or `nil`, but any non-`nil` explicit value is treated like `t` here). Otherwise, the algorithm aborts.

Thus, `IF*` may be used to implement a sort of conditional rewriting for ACL2's BDD^ package, even though this package only nominally supports unconditional rewriting. The following contrived example should make this point clear.

Suppose that we want to prove that `(nthcdr (length x) (append x y))` is equal to `y`, but that we would be happy to prove this only for lists having length 4. We can state such a theorem as follows.

```
(let ((x (list x0 x1 x2 x3)))
  (equal (nthcdr (length x) (append x y))
         y))
```

If we want to prove this formula with a `:BDD^` hint, then we need to have appropriate rewrite rules around. First, note that `LENGTH` is defined as follows (try `:PE^ LENGTH^`):

```
(length x)
 =
(if (stringp x)
    (len (coerce x 'list))
    (len x))
```

Since BDD^-based rewriting is merely very simple unconditional rewriting (see Section 2.1 [BDD-ALGORITHM], page 69), we expect to have to prove a rule reducing `STRINGP^` of a `CONS^`:

```
(defthm stringp-cons
  (equal (stringp (cons x y))
         nil))
```

Now we need a rule to compute the `LEN` of `X`, because the definition of `LEN` is recursive and hence not used by the BDD^ package.

```
(defthm len-cons
  (equal (len (cons a x))
         (1+ (len x))))
```

We imagine this rule simplifying `(LEN (LIST X0 X1 X2 X3))` in terms of `(LEN (LIST X1 X2 X3))`, and so on, and then finally `(LEN nil)` should be computed by execution (see Section 2.1 [BDD-ALGORITHM], page 69).

We also need to imagine simplifying `(APPEND X Y)`, where still `X` is bound to `(LIST X0 X1 X2 X3)`. The following two rules suffice for this purpose (but are needed, since `APPEND^`, actually `BINARY-APPEND^`, is recursive).

```
(defthm append-cons
  (equal (append (cons a x) y)
         (cons a (append x y))))

(defthm append-nil
  (equal (append nil x)
         x))
```

Finally, we imagine needing to simplify calls of `NTHCDR^`, where the the first argument is a number (initially, the length of `(LIST X0 X1 X2 X3)`, which is 4). The second lemma below is the traditional way to accomplish that goal (when not using BDDs), by proving a conditional rewrite rule. (The first lemma is only proved in order to assist in the proof of the second lemma.)

```
(defthm fold-constants-in-+
  (implies (and (syntaxp (quotep x))
                (syntaxp (quotep y)))
           (equal (+ x y z)
                  (+ (+ x y) z))))

(defthm nthcdr-add1-conditional
```

```
  (implies (not (zp (1+ n)))
           (equal (nthcdr (1+ n) x)
                  (nthcdr n (cdr x)))))
```

The problem with this rule is that its hypothesis makes it a conditional rewrite^ rule, and conditional rewrite rules are not used by the BDD^ package. (See Section 2.1 [BDD-ALGORITHM], page 69 for a discussion of "BDD rules.") (Note that the hypothesis cannot simply be removed; the resulting formula would be false for `n = -1` and `x = '(a)`, for example.) We can solve this problem by using `IF*`, as follows; comments follow.

```
(defthm nthcdr-add1
  (equal (nthcdr (+ 1 n) x)
         (if* (zp (1+ n))
              x
              (nthcdr n (cdr x)))))
```

How is `nthcdr-add1` applied by the BDD^ package? Suppose that the BDD^ computation encounters a term^ of the form `(NTHCDR (+ 1 N) X)`. Then the BDD^ package will apply the rewrite^ rule `nthcdr-add1`. The first thing it will do when attempting to simplify the right hand side of that rule is to attempt to simplify the term `(ZP (1+ N))`. If `N` is an explicit number (which is the case in the scenario we envision), this test will reduce (assuming the executable counterparts of `ZP`^ and `BINARY-+`^ are enable^d) to `t` or to `nil`. In fact, the lemmas above (not including the lemma `nthcdr-add1-conditional`) suffice to prove our goal:

```
(thm (let ((x (list x0 x1 x2 x3)))
       (equal (nthcdr (length x) (append x y))
              y))
     :hints (("Goal" :bdd (:vars nil))))
```

If we execute the following form that disables the definition and executable counterpart of the function `ZP`^

```
(in-theory (disable zp (zp)))
```

before attempting the proof of the theorem above, we can see more clearly the point of using `IF*`. In this case, the prover makes the following report.

```
ACL2 Error in ( THM ...):  Unable to resolve test of IF* for term

(IF* (ZP (+ 1 N)) X (NTHCDR N (CDR X)))

under the bindings
```

```
((X (CONS X0 (CONS X1 (CONS X2 #)))) (N '3))

-- use SHOW-BDD to see a backtrace.
```

If we follow the advice above, we can see rather clearly what happened. See Section 2.4 [SHOW-BDD], page 88.

```
ACL2 !>(show-bdd)

BDD computation on Goal yielded 21 nodes.
==============================

BDD computation was aborted on Goal, and hence there is no
falsifying assignment that can be constructed.  Here is a backtrace
of calls, starting with the top-level call and ending with the one
that led to the abort.  See :DOC show-bdd.

(LET ((X (LIST X0 X1 X2 X3)))
     (EQUAL (NTHCDR (LENGTH X) (APPEND X Y)) Y))
  alist: ((Y Y) (X3 X3) (X2 X2) (X1 X1) (X0 X0))

(NTHCDR (LENGTH X) (APPEND X Y))
  alist: ((X (LIST X0 X1 X2 X3)) (Y Y))

(IF* (ZP (+ 1 N)) X (NTHCDR N (CDR X)))
  alist: ((X (LIST* X0 X1 X2 X3 Y)) (N 3))
ACL2 !>
```

Each of these term-alist pairs led to the next, and the test of the last one, namely `(ZP (+ 1 N))` where `N` is bound to `3`, was not simplified to `t` or to `nil`.

What would have happened if we had used `IF^` in place of `IF*` in the rule `nthcdr-add1`? In that case, if `ZP` and its executable counterpart were disabled then we would be put into an infinite loop! For, each time a term of the form `(NTHCDR k V)` is encountered by the BDD package (where k is an explicit number), it will be rewritten in terms of `(NTHCDR k-1 (CDR V))`. We would prefer that if for some reason the term `(ZP (+ 1 N))` cannot be decided to be `t` or to be `nil`, then the BDD computation should simply abort.

Even if there were no infinite loop, this kind of use of `IF*` is useful in order to provide feedback of the form shown above whenever the test of an `IF` term fails to simplify to `t` or to `nil`.

2.4 SHOW-BDD

SHOW-BDD inspect failed BDD proof attempts

Attempts to use BDDs (see Chapter 2 [BDD], page 69), using `:bdd^` hints^, can fail for various reasons. Sometimes it is useful to explore such failures. To do so, one may simply execute the form

```
(show-bdd)
```

inside the ACL2 loop. The system's response is generally self-explanatory. Perhaps you have already seen `show-bdd` used in some examples (see Section 2.2 [BDD-INTRODUCTION], page 79 and see Section 2.3 [IF*], page 84). Here we give some details about `show-bdd`.

`(Show-bdd)` reports the number of nodes created during the BDD^ computation, followed by additional information depending on whether or not the computation ran to completion or aborted (for reasons explained elsewhere; see Section 2.1 [BDD-ALGORITHM], page 69). If the computation did abort, a backtrace is printed that should be useful in understanding where the problem lies. Otherwise, `(show-bdd)` prints out "falsifying constraints." This list of pairs associates term^s with values and suggests how to construct a binding list for the variables in the conjecture that will falsify the conjecture. It also prints out the term^ that is the result of simplifying the input term^. In each of these cases, parts of the object may be hidden during printing, in order to avoid creating reams of uninteresting output. If so, the user will be queried about whether he wishes to see the entire object (alist or term^), which may be quite large. The following responses are legal:

`:w` — Walk around the object with a structure editor

`t` — Print the object in full

`nil` — Do not print any more of the object

`Show-bdd` actually has three optional arguments, probably rarely used. The general form is

```
(show-bdd goal-name falsifying-ans term-ans)
```

where `goal-name` is the name of the goal on which the `:bdd^` hint was used (or, `nil` if the system should find such a goal), `falsifying-ans` is the answer to be used in place of the query for whether to print the falsifying constraints in full, and `term-ans` is the answer to be used in place of the query for whether to print the resulting term^ in full.

3 BOOKS

BOOKS files of ACL2 event forms

A "book" is a file of ACL2 events^ that have been certified as admissible. Using `include-book^` you can construct a new logical world^ by assuming the events^ in any number of mutually compatible books. Relevant documented topics are listed below.

* BOOK-CONTENTS:: restrictions on the forms inside books^
* BOOK-EXAMPLE:: how to create, certify, and use a simple book
* BOOK-NAME:: conventions associated with book names
* CBD:: connected book directory string
* CERTIFICATE:: how a book is known to be admissible and where its `defpkg^`s reside
* CERTIFY-BOOK:: how to produce a certificate^ for a book
* FULL-BOOK-NAME:: book naming conventions assumed by ACL2
* KEEP:: how we know if `include-book^` read the correct files
* PATHNAME:: introduction to filename conventions in ACL2
* PORTCULLIS:: the gate guarding the entrance to a certified book
* SET-CBD:: to set the connected book directory
* UNCERTIFIED-BOOKS:: invalid certificate^s and uncertified books^

Related topics other than immediate subtopics:
* INCLUDE-BOOK:: load the events^ in a file

Introduction.

A "book" is a file of ACL2 forms. Books are prepared entirely by the user of the system, i.e., they are "source" files not "object" files. Some of the forms in a book are marked `local^` and the others are considered "non-local."

`Include-book^` lets you load a book into any ACL2 world^. If completed without error, the inclusion of a book extends the logic of the host world^ by the addition of just the non-local events^ in the book. You may extend the world^ by successively including a variety of books to obtain the desired collection of definitions and rules. Unless name conflicts occur (which are detected and signalled) inclusion of a book is consistency preserving provided the book itself is consistent as discussed later. However, `include-book^` merely assumes the validity of the events^ in a book; if you include a book that contains an inconsistency (e.g., an inadmissible definition) then the resulting theory is inconsistent.

It is possible to "certify" a book, with `certify-book^`, guaranteeing that the error-free inclusion of the certified forms will produce a consistent extension of a consistent logic. Certification processes both the `local^` and non-local forms, so you can mark as `local^` those events^ you need for certification that you want to hide from users of the book (e.g., hacks, crocks, and kludges on the way to a good set of `:rewrite^` rules). Certification can also "compile" a book, thereby speeding up the execution of the functions defined within it. The desire to compile books is largely responsible for the restrictions we put on the forms allowed in books.

Extensive documentation^ is available on the various aspects of books. We recommend that you read it all before using books. It has been written so as to make sense when read in a certain linear sequence, called the "guided tour", though in general you may browse through it randomly. If you are on the guided tour, you should next read the documentation^ on book-example (see Section 3.2 [BOOK-EXAMPLE], page 91).

3.1 BOOK-CONTENTS

BOOK-CONTENTS restrictions on the forms inside books^

```
Example Book:

; This book defines my app function and the theorem that it is
; associative.  One irrelevant help lemma is proved first but
; it is local and so not seen by include-book.  I depend on the
; inferior book "weird-list-primitives" from which I get
; definitions of hd and tl.

(in-package "MY-PKG")

(include-book "weird-list-primitives")

(defun app (x y) (if (consp x) (cons (hd x) (app (tl x) y)) y))

(local
 (defthm help-lemma
   (implies (true-listp x) (equal (app x nil) x))))

(defthm app-is-associative
  (equal (app (app a b) c) (app a (app b c))))
```

The first form in a book must be `(in-package "pkg")` where `"pkg"` is some package name known to ACL2 whenever the book is certified. The rest of the forms in a book are embedded event forms, i.e., `defun^`s, `defthm^`s, etc., some of which may be marked `local^`. See Section 8.25 [EMBEDDED-EVENT-FORM], page 267. The usual Common Lisp commenting conventions are provided. Note that since a book consists of embedded event forms, we can talk about the "local^" and "non-local" events^ of a book.

Because `in-package^` is not an embedded event form, the only `in-package^` in a book is the initial one. Because `defpkg^` is not an embedded event form, a book can never contain a `defpkg^` form. Because `include-book^` is an embedded event form, books^ may contain references to other books^. This makes books^ structured objects.

When the forms in a book are read from the file, they are read with `current-package^` set to the package named in the `in-package^` form at the top of the file. The effect of this is that all symbols are intern^ed in that package, except those whose packages are given explicitly with the "::" notation. For example, if a book begins with `(in-package "ACL2-X")` and then contains the form

```
(defun fn (x)
  (acl2::list 'car x))
```

then `defun^`, `fn`, `x`, and `car^` are all intern^ed in the `"ACL2-X"` package. I.e., it is as though the following form were read instead:

```
(acl2-x::defun acl2-x::fn (acl2-x::x)
    (acl2::list 'acl2-x::car acl2-x::x)).
```

Of course, `acl2-x::defun` would be the same symbol as `acl2::defun` if the `"ACL2-X"` package imported `acl2::defun`.

If each book has its own unique package name and all the names defined within the book are in that package, then name clashes between books^ are completely avoided. This permits the construction of useful logical world^s by the successive inclusion of many books^. Although it is often too much trouble to manage multiple packages, their judicious use is a way to minimize name clashes. Often, a better way is to use `local`; see Section 6.21 [LOCAL], page 181.

How does `include-book^` know the definitions of the packages used in a book, since `defpkg^`s cannot be among the forms? More generally, how do we know that the forms in a book will be admissible in the host logical world^ of an `include-book^`? See Section 3.5 [CERTIFICATE], page 99 for answers to these questions.

3.2 BOOK-EXAMPLE

BOOK-EXAMPLE how to create, certify, and use a simple book

Suppose you have developed a sequence of admissible events^ which you want to turn into a book. We call this "publishing" the book. This note explains how to do that.

A key idea of books^ is that they are "incremental" in the sense that when you include a book in a host logical world^, the world^ is incrementally extended by the results established in that book. This is allowed only if every name defined by the incoming book is either new or is already identically defined. See Section 8.76 [REDUNDANT-EVENTS], page 337. This is exactly the same problem faced by a programmer who wishes to provide a utility to other people: how can he make sure he doesn't create name conflicts? The solution, in Common Lisp, is also the same: use packages. While books^ and packages have a very tenuous formal connection (every book must start with an `in-package`^), the creation of a book is intimately concerned with the package issue. Having motivated what would otherwise appear as an unnecessary fascination with packages below, we now proceed with a description of how to publish a book.

Just to be concrete, let's suppose you have already gotten ACL2 to accept the following sequence of command^s, starting in the ACL2 initial state^.

```
(defpkg "ACL2-MY-BOOK"
        (union-eq *common-lisp-symbols-from-main-lisp-package*
                  *acl2-exports*))
(in-package "ACL2-MY-BOOK")
(defun app (x y)
  (if (consp x) (cons (car x) (app (cdr x) y)) y))
(defun rev (x)
  (if (consp x) (app (rev (cdr x)) (list (car x))) nil))
(defthm rev-app-hack
  (equal (rev (app a (list x))) (cons x (rev a))))
(defthm rev-rev
  (implies (acl2::true-listp x) (equal (rev (rev x)) x)))
```

Observe that the first form above defines a package (which imports the symbols defined in CLTL such as `if`^ and `cons`^ and the symbols used to command^ ACL2 such as `defun`^ and `defthm`^). The second form selects that package as the current one. All subsequent forms are read into that package. The remaining forms are just event forms: `defun`^s and `defthm`^s in this case.

Typically you would have created a file with Emacs containing these forms and you will have submitted each of them interactively to ACL2 to confirm that they are all admissible. That interactive verification should start in ACL2's initial world^ — although you might, of course, start your sequence of events^ with some `include-book`^s to build a more elaborate world^.

The first step towards publishing a book containing the results above is to create a file that starts with the `in-package^` and then contains the rest of the forms. Let's call that file `"my-book.lisp"`. The name is unimportant, except it must end with `".lisp"`. If there are events^ that you do not wish to be available to the user of the book — e.g., lemmas you proved on your way toward proving the main ones — you may so mark them by enclosing them in `local^` forms. See Section 6.21 [LOCAL], page 181. Let us suppose you wish to hide `rev-app-hack` above. You may also add standard Lisp comments to the file. The final content of `"my-book.lisp"` might be:

```
; This book contains my app and rev functions and the theorem
; that rev is its own inverse.

  (in-package "ACL2-MY-BOOK")
  (defun app (x y)
    (if (consp x) (cons (car x) (app (cdr x) y)) y))
  (defun rev (x)
    (if (consp x) (app (rev (cdr x)) (list (car x))) nil))

; The following hack is not exported.
  (local (defthm rev-app-hack
    (equal (rev (app a (list x))) (cons x (rev a)))))

  (defthm rev-rev
    (implies (acl2::true-listp x) (equal (rev (rev x)) x)))
```

The file shown above **is** the book. By the time this note is done you will have seen how to certify that the book is correct, how to compile it, and how to use it in other host world^s. Observe that the `defpkg^` is not in the book. It cannot be: Common Lisp compilers disagree on how to treat new package definitions appearing in files to be compiled.

Since a book is just a source file typed by the user, ACL2 provides a mechanism for checking that the events^ are all admissible and then marking the file as checked. This is called certification. To certify `"my-book.lisp"` you should first get into ACL2 with an initial world^. Then, define the package needed by the book, by typing the following `defpkg^` to the ACL2 prompt^:

```
ACL2 !>(defpkg "ACL2-MY-BOOK"
               (union-eq *common-lisp-symbols-from-main-lisp-package*
                         *acl2-exports*))
```

Then execute the command^:

```
ACL2 !>(certify-book "my-book" 1 t) ; the `t' is in fact the default
```

Observe that you do not type the ".lisp" part of the file name. For purposes of books^, the book's name is "my-book" and by the time all is said and done, there will be several extensions in addition to the ".lisp" extension associated with it.

The 1 tells certify-book^ that you acknowledge that there is one command in this "certification world^" (namely the defpkg^). To use the book, any prospective host world^ must be extended by the addition of whatever command^s occurred before certification. It would be a pity to certify a book in a world^ containing junk because that junk will become the "portcullis^" guarding entrance to the book. The t above tells certify-book^ that you wish to compile "my-book.lisp" also. Certify-book^ makes many checks but by far the most important and time-consuming one is that it "proves" every event in the file.

When certify-book^ is done it will have created two new files. The first will be called "my-book.cert" and contains the "certificate^" attesting to the admissibility of the events^ in "my-book.lisp". The certificate^ contains the defpkg^ and any other forms necessary to construct the certification world^. It also contains various check sums used to help you keep track of which version of "my-book.lisp" was certified.

The second file created by certify-book^ is the compiled version of "my-book.lisp" and will have a name that is assigned by the host compiler (e.g., "my-book.o" in AKCL, "my-book.lbin" or "my-book.sbin" in Lucid). Certify-book^ will also load this object file. When certify-book^ is done, you may throw away the logical world^ it created, for example by executing the command^ :u.

To use the book later in any ACL2 session, just execute the event (include-book "my-book"). This will do the necessary defpkg^, load the non-local^ events^ in "my-book.lisp" and then load the compiled code for the non-local functions defined in that file. Checks are made to insure that the certificate^ file exists and describes the version of "my-book.lisp" that is read. The compiled code is loaded if and only if it exists and has a later write date than the source file.

Since include-book^ is itself an event, you may put such forms into other books^. Thus it is possible for the inclusion of a single book to lead to the inclusion of many others. The check sum information maintained in certificate^s helps deal with the version control problem of the referenced books^. I.e., if this version of "my-book" is used during the certification of "your-book", then the certificate^ for "your-book" includes the check sum of this version of "my-book". If a later (include-book "your-book") finds a version of "my-book" with a different check sum, an error is signalled. But check sums are not perfect and the insecurity of the host file system prevents ACL2 from guaranteeing the logical soundness of an include-book^ event, even for a book that appears to have a valid certificate^ (they can be forged, after all). (See Section 3.5 [CERTIFICATE], page 99 for further discussion.)

This concludes the example of how to create, certify and use a book. If you wish, you could now review the documentation^ for book-related topics (see Chapter 3 [BOOKS], page 89) and browse through them. They'll probably make sense in this context. Alternatively, you could continue the "guided tour" through the rest of the documentation^ of books^. See Section 3.3 [BOOK-NAME], page 95, following the pointer given at the conclusion.

3.3 BOOK-NAME

BOOK-NAME conventions associated with book names

```
Examples:
"list-processing"
"/usr/home/smith/my-arith"
```

Book names are strings and lists that can be elaborated into file names. We elaborate book names by concatenating the "connected book directory" (see Section 3.4 [CBD], page 97) string on the left and some "extension," such as `".lisp"`, on the right. However, the connected book directory is not added if the book name itself already represents an absolute file name. Furthermore, `include-book^` and `certify-book^` temporarily reset the connected book directory to be the directory of the book being processed. This allows `include-book^` forms to use file names without explicit mention of the enclosing book's directory. This in turn allows books^ (together with those that they include, using `include-book^`) to be moved between directories while maintaining their certification and utility. (One does need to be careful about using absolute filenames (see Section 3.9 [PATHNAME], page 105) when of certifying books in worlds that have already included other books; see Section 3.10 [PORTCULLIS], page 108.)

You may wish to read elsewhere for details of ACL2 file name conventions (see Section 3.9 [PATHNAME], page 105), for a discussion of the filename that is the result of the elaboration described here (see Section 3.7 [FULL-BOOK-NAME], page 103), and for details of the concept of the connected book directory (see Section 3.4 [CBD], page 97). For details of how `include-book^` (see Section 6.20 [INCLUDE-BOOK], page 180) and `certify-book^` (see Section 3.6 [CERTIFY-BOOK], page 101) use these concepts, see below.

Often a book name is simply the familiar name of the file. (See Section 3.7 [FULL-BOOK-NAME], page 103 for discussion of the notions of "directory string," "familiar name," and "extension". These concepts are not on the guided tour through books^ and you should read them separately.)

However, it is permitted for book names to include a directory or part of a directory name. Book names never include the extension, since ACL2 must routinely tack several different extensions onto the name during `include-book^`. For example, `include-book^` uses the `".lisp"`, `".cert"` and possibly the `".o"` or `".lbin"` extensions of the book name.

Book names are elaborated into full file names by `include-book^` and `certify-book^`. This elaboration is sensitive to the "connected book directory." The connected book directory is an absolute filename string (see Section 3.9 [PATHNAME], page 105) that is part of the ACL2 `state^`. (You may wish to see Section 3.4 [CBD], page 97 and to see Section 3.11 [SET-CBD], page 110 — note that these are not on the guided tour).

If a book name is an absolute filename string, ACL2 elaborates it simply by appending the desired extension to the right. If a book name is an absolute structured pathname, then it is first converted to an absolute pathname string (see Section 3.9 [PATHNAME], page 105) and ACL2 elaborates the result.

If a book name is a relative filename string, ACL2 appends the connected book directory on the left (eliminating the excess colon (`:`) character in the case of the Macintosh file system) and the desired extension on the right. If a book name is a relative structured pathname^, it is first converted to a relative pathname^ string and then ACL2 elaborates the result.

Note that it is possible that the book name includes some partial specification of the directory. For example, if the connected book directory is `"/usr/home/smith/"` then the book name `"project/task-1/arith"` is a book name that will be elaborated to

```
"/usr/home/smith/project/task-1/arith.lisp".
```

Also, the structured book name

```
(:relative "project" "task-1" "arith")
```

will be elaborated the same way. However, the directory separator (which is the character slash (`/`) in Unix (trademark of AT&T) and colon (`:`) on the Macintosh) may not appear in the final component of a structured book name; for example, `(:relative "project" "task-1/arith")` is illegal.

Observe that while the events^ in this `"arith"` book are being processed the connected book directory will temporarily be set to

```
"/usr/home/smith/project/task-1/".
```

Thus, if the book requires other books^, e.g.,

```
(include-book "naturals")
```

then it is not necessary to specify the directory on which they reside provided that directory is the same as the superior book.

This inheritance of the connected book directory and its use to elaborate the names of inferior books^ makes it possible to move books^ and their inferiors to new directories, provided they maintain the same relative relationship. Moreover, if all arguments to `include-book^` command^s are structured directories, then it is even possible to move with ease whole collections of books^ to different filesystems that use a different operating system than the one under which the original certification was performed.

The `".cert"` extension of a book, if it exists, is presumed to contain the most recent certificate^ for the book. See Section 3.5 [CERTIFICATE], page 99 (or, if you are on the guided tour, wait until the tour gets there).

See Section 3.1 [BOOK-CONTENTS], page 90 to continue the guided tour.

3.4 CBD

CBD connected book directory string

```
Example:
ACL2 !>:cbd
"/usr/home/smith/"
```

The connected book directory is a nonempty string that specifies a directory as an absolute pathname. (See Section 3.9 [PATHNAME], page 105 for a discussion of file naming conventions. Note however that the `cbd^` is always a string, even though `:set-cbd^` accepts structured directories.) When `include-book^` is given a relative book name it elaborates it into a full book name, essentially by appending the connected book directory string to the left and `".lisp"` to the right. (For details, see Section 3.3 [BOOK-NAME], page 95 and also see Section 3.7 [FULL-BOOK-NAME], page 103.) Furthermore, `include-book^` temporarily sets the connected book directory to the directory string of the resulting full book name so that references to inferior books^ in the same directory may omit the directory. See Section 3.4 [CBD], page 97 for how to set the connected book directory string.

```
General Form:
(cbd)
```

This is a macro that expands into a term involving the single free variable `state^`. It returns the connected book directory string.

The connected book directory (henceforth called the "`cbd`") is used by `include-book^` to elaborate the supplied book name into a full book name (see Section 3.7 [FULL-BOOK-NAME], page 103). For example, in the Unix (trademark of AT&T) operating system, if the `cbd` is `"/usr/home/smith/"` then the elaboration of the book-name^ `"project/task-1/arith"` (to the `".lisp"` extension) is `"/usr/home/smith/project/task-1/arith.lisp"`. That full-book-name^ is what include-book^ opens to read the source text for the book.

The `cbd` may be changed using `set-cbd^` (see Section 3.11 [SET-CBD], page 110). Furthermore, during the processing of the events^ in a book, `include-book^` sets the `cbd` to be the directory string of the full-book-name^ of the book. Thus, if the `cbd` is `"/usr/home/smith/"` then during the processing of events^ by

```
(include-book "project/task-1/arith")
```

the `cbd` will be set to `"/usr/home/smith/project/task-1/"`. Note that if `"arith"` recursively includes a subbook, say `"naturals"`, that resides on the same directory, the `include-book^` event for it may omit the specification of that directory. For example, `"arith"` might contain the event

```
(include-book "naturals").
```

In general, suppose we have a superior book and several inferior books^ which are included by events^ in the superior book. Any inferior book residing on the same directory as the superior book may be referenced in the superior without specification of the directory.

We call this a "relative" as opposed to "absolute" naming. The use of relative naming is preferred because it permits books^ (and their accompanying inferiors) to be moved between directories while maintaining their certificate^s and utility. Certified books^ that reference inferiors by absolute file names are unusable (and rendered uncertified) if the inferiors are moved to new directories.

Technical Note and a Challenge to Users:

After elaborating the book name to a full book name, `include-book^` opens a channel to the file to process the events^ in it. In some host Common Lisps, the actual file opened depends upon a notion of "connected directory" similar to our connected book directory. Our intention in always elaborating book names into absolute filename strings (see Section 3.9 [PATHNAME], page 105 for terminology) is to circumvent the sensitivity to the connected directory. But we may have insufficient control over this since the ultimate file naming conventions are determined by the host operating system rather than Common Lisp (though, we do check that the operating system "appears" to be one that we "know" about). Here is a question, which we'll pose assuming that we have an operating system that calls itself "Unix." Suppose we have a file name, filename, that begins with a slash, e.g., `"/usr/home/smith/..."`. Consider two successive invocations of CLTL's

```
(open filename :direction :input)
```

separated only by a change to the operating system's notion of connected directory. Must these two invocations produce streams to the same file? A candidate string might be something like `"/usr/home/smith/*/usr/local/src/foo.lisp"` which includes some operating system-specific special character to mean "here insert the connected directory" or, more generally, "here make the name dependent on some non-ACL2 aspect of the host's state." If such "tricky" name strings beginning with a slash exist, then we have failed to isolate ACL2 adequately from the operating system's file naming conventions. Once upon a time, ACL2 did not insist that the `cbd` begin with a slash and that allowed the string `"foo.lisp"` to be tricky because if one were connected to `"/usr/home/smith/"` then with the empty `cbd` `"foo.lisp"` is a full book name that names the same file as `"/usr/home/smith/foo.lisp"`. If the actual file one reads is determined by the operating system's state then it is possible for ACL2 to have two distinct "full book names" for the same file, the "real" name and the "tricky" name. This can cause ACL2 to include the same book twice, not recognizing the second one as redundant.

3.5 CERTIFICATE

CERTIFICATE how a book is known to be admissible and where its `defpkg^`s reside

A book, say `"arith"`, is said to have a "certificate" if there is a file named `"arith.cert"`. Certificates are created by the function `certify-book^` and inspected by `include-book^`. Certificates contain two parts: a portcullis^ and a keep^. The portcullis^ must be raised before the book can be entered. After a book has been entered, the keep^ is used to verify that the entrants were properly identified. Because the file system is insecure it is possible for the inclusion of a book to cause inconsistency even though the book carries an impeccable certificate.

The presence of a "valid" certificate file for a book attests to two things: all of the events^ of the book are admissible in a certain extension of the initial ACL2 logic, and the non-`local`^ events^ of the book are independent of the `local`^ ones (see Section 8.60 [LOCAL-INCOMPATIBILITY], page 321). In addition, the certificate contains the command^s used to construct the world^ in which certification occurred. Among those command^s, of course, are the `defpkg`^s defining the packages used in the book. When a book is included into a host world^, that world^ is first extended by the command^s listed in the certificate for the book. Unless that causes an error due to name conflicts, the extension insures that all the packages used by the book are identically defined in the host world^.

Security:

Because the host file system is insecure, there is no way ACL2 can guarantee that the current contents of a book is the same as it was when its certificate was written. That is, between the time a book is certified and the time it is used, it may be modified. Furthermore, certificates can be counterfeited. Check sums (see Section 8.9 [CHECK-SUM], page 243) are used to help detect such problems. But check sums provide imperfect security: two different files can have the same check sum.

Therefore, from the strictly logical point of view, one must consider even the inclusion of certified books^ as placing a burden on the user:

> The non-erroneous inclusion of a certified book is consistency preserving provided (a) the objects read by `include-book`^ from the certificate were the objects written there by a `certify-book`^ and (b) the forms read by `include-book`^ from the book itself are the forms read by the corresponding `certify-book`^.

We say that a given execution of `include-book`^ is "certified" if a certificate file for the book is present and well-formed and the check sum information contained within it supports the conclusion that the events^ read by the `include-book`^ are the ones checked by `certify-book`^. When an uncertified `include-book`^ occurs, warnings are printed or errors are caused. But even if no warning is printed, you must accept burdens (a) and (b) if you use books^. These burdens are easier to live with if you protect your books^ so that other users cannot write to them, you abstain from running concurrent ACL2 jobs, and you abstain from counterfeiting certificates. But even on a single user uniprocessor, you can shoot yourself in the foot by using the ACL2 io^ primitives to fabricate an inconsistent book and the corresponding certificate.

Note that part (a) of the burden described above implies, in particular, that there are no guarantees when a certificate is copied. When books^ are renamed (as by copying them), it is recommended that their certificates be removed and the books^ be recertified. The expectation is that recertification will go through without a hitch if relative pathname^s are used, though if the books^ are moved to a file system with a different underlying operating system, things will go more easily if structured pathname^s are used as arguments to `include-book^`. See Section 3.9 [PATHNAME], page 105, which is not on the guided tour.

As noted above, certificates are divided into two parts, a portcullis^ and a keep^. See Section 3.10 [PORTCULLIS], page 108 to continue the guided tour through books^.

3.6 CERTIFY-BOOK

CERTIFY-BOOK how to produce a certificate^ for a book

```
Examples:
(certify-book "my-arith" 3)   ;certify in a world with 3 commands
(certify-book "my-arith")     ;certify in a world with 0 commands
(certify-book "my-arith" 0 nil) ;as above, but do not compile
(certify-book (:relative "my-arith") 0 nil) ;as above
(certify-book "my-arith" t)   ;certify from world of existing certificate

General Form:
(certify-book book-name k compile-flg)
```

where `book-name` is a book name (see Section 3.3 [BOOK-NAME], page 95), `k` is either `t` or an integer used to indicate your approval of the "certification world^," and `compile-flg` indicates whether you wish to compile the (functions in the) book. `Compile-flg` defaults to `t`, meaning to compile. The second argument `k` is actually optional as well; it defaults to `0`.

For a general discussion of books, see Chapter 3 [BOOKS], page 89. `Certify-book` is akin to what we have historically called a "proveall": all the forms in the book are "proved" to guarantee their admissibility. More precisely, `certify-book` (1) reads the forms in the book, confirming that the appropriate packages are defined in the certification world^; (2) does the full admissibility checks on each form (proving termination of recursive functions, proving theorems, etc.), checking as it goes that each form is an embedded event form (see Section 8.25 [EMBEDDED-EVENT-FORM], page 267); (3) rolls the world^ back to the initial certification world^ and does an `include-book^` of the book to check for local^ incompatibilities (see Section 8.60 [LOCAL-INCOMPATIBILITY], page 321); (4) writes a certificate^ recording not only that the book was certified but also recording the command^s necessary to recreate the certification world^ (so the appropriate packages can be defined when the book is included in other world^s) and the check sums of all the books^ involved (see Section 3.5 [CERTIFICATE], page 99); (5) compiles the book if so directed (and then loads the object file in that case). The result of executing a `certify-book` command^ is the creation of a single new event, which is actually an `include-book^` event. If you don't want its included events^ in your present world^, simply execute `:ubt^ :here` afterwards.

`Certify-book` requires that the default defun-mode^ (see Section 8.16 [DEFAULT-DEFUN-MODE], page 254) be `:logic^` when certification is attempted. If the mode is not `:logic^`, an error is signalled.

An error will occur if `certify-book` has to deal with any uncertified book other than the one on which it was called. For example, if the book being certified includes another book, that subbook must already have been certified.

Certification occurs in some logical world^, called the "certification world^." That world^ must contain the `defpkg^`s needed to read and execute the forms in the book. The command^s necessary to recreate that world^ from the ACL2 initial world^ will be copied into the certificate^ created for the book. Those command^s will be re-executed whenever the book is included, to insure that the appropriate packages (and all other names used in the certification world^) are correctly defined. The certified book will be more often usable if the certification world^ is kept to a minimal extension of the ACL2 initial world^. Thus, before you call `certify-book` for the first time on a book, you should get into the initial ACL2 world^ (e.g., with `:ubt 1` or just starting a new version of ACL2), `defpkg^` the desired packages, and then invoke `certify-book`.

The `k` argument to `certify-book` must be either a nonnegative integer or else the symbol `t`. If `k` is an integer, then it must be the number of command^s that have been executed to create the world^ in which `certify-book` was called. One way to obtain this number is by doing `:pbt 1` to see all the command^s back to the first one. The last command^ number printed in the `:pbt^` display is the appropriate `k`. This number is just the maximum command^ number, `:max^` — see Section 8.11 [COMMAND-DESCRIPTOR], page 245 — but unless `:max^` is `0`, `certify-book` requires that you actually input the number as a way of reminding you to inspect the world^ before calling `certify-book`.

If `k` is `t` it means that `certify-book` should use the same world^ used in the last certification of this book. `K` may be `t` only if you call `certify-book` in the initial ACL2 world^ (`:max = 0`) and there is a certificate^ on file for the book being certified. (Of course, the certificate^ is probably invalid.) In this case, `certify-book` reads the old certificate^ to obtain the portcullis^ command^s and executes them to recreate the certification world^.

If you have a certified book that has remained unchanged for some time you are unlikely even to remember the appropriate `defpkg^`s for it. If you begin to change the book, don't throw away its certificate^ file just because it has become invalid! It is an important historical document until the book is re-certified.

When `certify-book` is directed to produce a compiled file, it calls the Common Lisp function `compile-file` on the original source file. This creates a compiled file with an extension known to ACL2, e.g., if the book is named `"my-book"` then the source file is `"my-book.lisp"` and the compiled file under AKCL will be `"my-book.o"` while under Lucid it will be `"my-book.lbin"` or `"my-book.sbin"`. The compiled file is then loaded. When `include-book^` is used later on `"my-book"` it will automatically load the compiled file, provided the compiled file has a later write date than the source file. The only effect of such compilation^ and loading is that the functions defined in the book execute faster. See Section 8.39 [GUARD], page 285 for a discussion of the issues.

When `certify-book` is directed not to produce a compiled file, it will delete any existing compiled file for the book, so as not to mislead `include-book^` into loading the now outdated compiled file.

After execution of a `certify-book` form, the value of `acl2-defaults-table^` is restored to what it was immediately before that `certify-book` form was executed. See Section 9.2 [ACL2-DEFAULTS-TABLE], page 374.

This completes the tour through the documentation^ of books^.

3.7 FULL-BOOK-NAME

FULL-BOOK-NAME book naming conventions assumed by ACL2

For this discussion we assume that the resident operating system is Unix (trademark of AT&T), but analogous remarks apply to other operating systems supported by ACL2, in particular, the Macintosh operating system where '`:`' plays roughly the role of '`/`' in Unix; see Section 3.9 [PATHNAME], page 105.

ACL2 defines a "full book name" to be an "absolute filename string," that may be divided into contiguous sections: a "directory string", a "familiar name" and an "extension". See Section 3.9 [PATHNAME], page 105 for the definitions of "absolute," "filename string," and other notions pertaining to naming files. Below we exhibit the three sections of one such string:

```
"/usr/home/smith/project/arith.lisp"

"/usr/home/smith/project/"          ; directory string
```

```
                           "arith"      ; familiar name
                                ".lisp" ; extension
```

The sections are marked by the rightmost slash and rightmost dot, as shown below.

```
"/usr/home/smith/project/arith.lisp"
                        |     |
                        slash dot
                        |     |
"/usr/home/smith/project/"           ; directory string
                        "arith"      ; familiar name
                             ".lisp" ; extension
```

The directory string includes (and terminates with) the rightmost slash. The extension includes (and starts with) the rightmost dot. The dot must be strictly to the right of the slash so that the familiar name is well-defined and nonempty.

If you are using ACL2 on a system in which file names do not have this form, please contact the authors and we'll see what we can do about generalizing ACL2's conventions.

3.8 KEEP

KEEP how we know if `include-book^` read the correct files

The certificate (see Section 3.5 [CERTIFICATE], page 99 for general information) of a certified file is divided into two parts, a portcullis^ and a keep. These names come from castle lore. The keep is the strongest and usually tallest tower of a castle from which the entire courtyard can be surveyed by the defenders. The keep of a book is a list of file names and check sums used after the book has been included, to determine if the files read were (up to check sum) those certified.

Once the portcullis^ is open, `include-book^` can enter the book and read the event forms therein. The non-`local^` event forms are in fact executed, extending the host theory. That may read in other books^. When that has been finished, the keep of the certificate^ is inspected. The keep is a list of the book names which are included (heritarily through all subbooks) in the certified book (including the certified book itself) together with the check sums of the objects in those books^ at the time of certification. We compare the check sums of the books^ just included to the check sums of the books^ stored in the keep. If differences are found then we know that the book or one of its subbooks has been changed since certification.

See Section 6.20 [INCLUDE-BOOK], page 180 to continue the guided tour through books^.

3.9 PATHNAME

PATHNAME introduction to filename conventions in ACL2

Although the notion of pathname objects from Common Lisp is not supported in ACL2, nor is the function `pathname`, nevertheless ACL2 adopts conventions for naming files that are not so different from Common Lisp's conventions, using strings that we'll call "filename strings" and (for ACL2 books^) certain lists that we'll call "structured pathnames." Some of these strings and lists correspond to "absolute," i.e. complete, descriptions of a file or directory; others are "relative" to the current working directory or to the connected book directory (see Section 3.4 [CBD], page 97). The reason we support structured pathnames is to allow books^ to be transferred between machines running different operating systems (see Section 3.3 [BOOK-NAME], page 95).

Consider by way of introduction the following examples in the Unix (trademark of AT&T) operating system. The filename string

```
"/home/smith/acl2/book-1.lisp"
```

corresponds to the structured pathname

```
(:absolute "home" "smith" "acl2" "book-1.lisp")
```

and, relative to the connected book directory `"/home/smith/"` (see Section 3.4 [CBD], page 97), corresponds to the relative filename string

```
"acl2/book1.lisp"
```

and to the relative structured pathname

```
(:relative "acl2" "book-1.lisp")
```

The rest of the discussion focuses on filename strings vs. structured pathnames, with attention to directories vs. files and to absolute vs. relative pathnames.

Filename strings. One way to specify a file in ACL2 is by way of a nonempty string. The form of this string must conform to the requirements of the resident operating system. Thus in the Unix environment, slash (`/`) is the "directory separator": a filename such as `"/home/smith/acl2/book-1.lisp"` may be read as "Inside the directory `"/home"`, then inside the directory `"smith"`, then inside the directory `"acl2"`, consider the file `"book-1.lisp"`." On the Macintosh, the directory separator is colon (`:`). Such strings represent a directory when they terminate with the directory separator, and a file otherwise. The directory separator may be obtained as:

```
(directory-separator (@ operating-system) 'top).
```

Filename strings may be referred to either as absolute pathnames, which are complete descriptions of the location of a file or directory, or as relative pathnames, which give the location of a file or directory relative to a given (implicit) directory. In Unix, filename strings starting with the directory separator (`/`) are absolute pathnames, such as the example in the preceding paragraph. All other filename strings are relative pathnames.

For example, the string

```
"examples/my-book"
```

is a relative pathname. If the implicit directory (which for books^ is the connected book directory; see Section 3.4 [CBD], page 97) is `"/home/smith/"`, then the corresponding absolute pathname is the filename string obtained by concatenating the implicit directory with the relative pathname,

```
"/home/smith/examples/my-book"
```

On the Macintosh the convention is exactly reversed: a string represents an absolute pathname exactly when its first character is **not** the directory separator (`:`). Thus, the absolute pathname corresponding to the one displayed just above is

```
"home:smith:examples:my-book"
```

which may be derived from the relative pathname

```
":examples:my-book"
```

and implicit directory

```
"home:smith:"
```

Notice that on the Macintosh, when we concatenate^ an implicit directory string to a relative pathname string, we must drop the excess colon, since a colon terminates the implicit directory string and also starts the relative pathname.

Structured pathnames. In ACL2, books^ may be named with filename strings or with structured pathnames, which we now describe. (Note that structured pathnames are only used in the realm of books^: for example, although `:set-cbd^` accepts structured pathnames, `ld^` currently only accepts filename strings.) A structured pathname is a list beginning with one of the keywords `:absolute` or `:relative` and then continuing with strings, except that `:back` may appear any number of times between `:relative` and the first string in the list. Each of the strings represents a subdirectory, except that the last may represent a file. For example, the filename string (in Unix) `"/home/smith/acl2/book-1.lisp"` represents the same file as does the following structured pathname.

```
(:absolute "home" "smith" "acl2" "book-1.lisp")
```

This is an absolute structured pathname, because it begins with the keyword `:absolute`. The other type of structured pathname is the relative structured pathname, which begins with the keyword `:relative`. For example, the following structured pathname corresponds to the relative filename string (in Unix) `"examples/my-book"`:

```
(:relative "examples" "my-book")
```

As mentioned above, relative structured pathnames may also contain any number of occurrences of the keyword `:back` after the (initial element) `:relative` and before the first string. This keyword represents the notion of going up one level in the file system (just like `".."` in Unix). For example, suppose that the connected book directory (see Section 3.4 [CBD], page 97) can be represented by the following structured pathname.

```
(:absolute "home" "smith" "acl2" "examples")
```

Then the structured pathname

```
(:relative :back :back "numbers" "book-2.lisp")
```

represents the same file as does the following absolute structured pathname.

```
(:absolute "home" "smith" "numbers" "book-2.lisp")
```

Here, the first `:back` may be thought of as "cancelling" the subdirectory `"examples"` and the second `:back` may be thought of as "cancelling" its parent `"acl2"`, after which we are to continue as specified by the given relative structured pathname.

There is no distinction between files and directories for structured pathnames, but the intention is clear from context. For example, the argument to `:set-cbd^` is a directory, while the argument to `include-book^` is a file (but without the `".lisp"` extension (see Section 3.3 [BOOK-NAME], page 95).

Finally, we note that our notion of structured pathname corresponds to a similar notion of structured directory described in the second edition of Guy Steele's book "Common Lisp the Language." The main difference is that we allow the trivial structured pathname (`:relative`), but Steele does not.

3.10 PORTCULLIS

PORTCULLIS the gate guarding the entrance to a certified book

The certificate (see Section 3.5 [CERTIFICATE], page 99 for general information) of a certified file is divided into two parts, a portcullis and a keep^. These names come from castle lore. The portcullis of a castle is an iron grate that slides up through the ceiling of the tunnel-like entrance. The portcullis of a book insures that `include-book^` does not start to read the book until the appropriate context has been created.

Technically, the portcullis is a pair containing a list of command^s used to create the "certification world^" and an alist specifying the check sums of all the books^ included in that world^. The portcullis is constructed automatically by `certify-book^` from the world^ in which `certify-book^` is called, but that world^ must have certain properties described below. After listing the properties we discuss the issues more leisurely.

Each command^ in the portcullis must be either a `defpkg^` form or an embedded event form (see Section 8.25 [EMBEDDED-EVENT-FORM], page 267). In addition, we exclude `include-book^` events^ unless the name of the included book is an absolute filename (see Section 3.9 [PATHNAME], page 105). For example, under Unix (trademark of AT&T) any string supplied to `include-book^` in the portcullis must begin with a slash (`/`). Since the portcullis command^s are recovered automatically by `certify-book^`, this restriction on the form of portcullis command^s naturally becomes a restriction on the world^ in which `certify-book^` may be called. We explain the exclusion after discussing the general issues.

Consider a book to be certified. The book is a file containing event forms. Suppose the file contains references to such symbols as `my-pkg::fn` and `acl2-arith::cancel`, but that the book itself does not create the packages. Then a hard Lisp error would be caused merely by the attempt to read the expressions in the book. The corresponding `defpkg^`s cannot be written into the book itself because the book must be compilable and Common Lisp compilers differ on the rules concerning the inline definition of new packages. The only safe course is to make all `defpkg^`s occur outside of compiled files.

More generally, when a book is certified it is certified within some logical world^. That "certification world^" contains not only the necessary `defpkg^`s but also, perhaps, function and constant definitions and maybe even references to other books^. When `certify-book^` creates the certificate^ for a file it recovers from the certification world^ the command^s used to create that world^ from the initial ACL2 world^. Those command^s become part of the portcullis for the certified book. In addition, `certify-book^` records in the portcullis the check sums (see Section 8.9 [CHECK-SUM], page 243) of all the books^ included in the certification world^.

`Include-book^` presumes that it is impossible even to read the contents of a certified book unless the portcullis can be "raised." To raise the portcullis we must be able to execute (possibly redundantly, but certainly without error), all of the command^s in the portcullis and then verify that the books^ thus included were identical to those used to build the certification world^ (up to check sum). This raising of the portcullis must be done delicately since `defpkg^`s are present: we cannot even read a command^ in the portcullis until we have successfully executed the previous ones, since packages are being defined.

Clearly, a book is most useful if it is certified in the most elementary extension possible of the initial logic. If, for example, your certification world^ happens to contain a `defpkg^` for `"MY-PKG"` and the function `foo`, then those definitions become part of the portcullis for the book. Every time the book is included, those names will be defined and will have to be either new or redundant (see Section 8.76 [REDUNDANT-EVENTS], page 337). But if those names were not necessary to the certification of the book, their presence would unnecessarily restrict the utility of the book.

Recall that we disallow `include-book^` events^ from the portcullis unless the included book's name is an absolute filename (See Section 3.9 [PATHNAME], page 105). Thus, for example, under the Unix operating system it is impossible to certify a book if the certification world^ was created with

```
ACL2 !>(include-book^ "arith")
```

The problem here is that the file actually read on behalf of such an `include-book^` depends upon the then current setting of the connected book directory (see Section 3.4 [CBD], page 97).

That setting could be changed before the certification occurs. If we were to copy `(include-book "arith")` into the portcullis of the book being certified, there is no assurance that the `"arith"` book included would come from the correct directory. However, by requiring that the `include-book^`s in the certification world^ give book names that begin with slash we effectively require you to specify the full file name of each book involved in creating your certification world^. Observe that the execution of

```
(include-book^ "/usr/local/src/acl2/library/arith")
```

or equivalently (using structured pathname^s; see Section 3.9 [PATHNAME], page 105)

```
(include-book^
  (:absolute "usr" "local" "src" "acl2" "library" "arith"))
```

does not depend on the current book directory. On the other hand, this requirement — effectively that absolute file names be used in the certification world^ — means that a book that requires another book in its certification world^ will be rendered uncertified if the required book is removed to another directory. If possible, any `include-book^` command^ required for a book ought to be placed in the book itself and not in the certification world^. The only time this cannot be done is if the required book is necessary to some `defpkg^` required by your book. Of course, this is just the same advice we have been giving: keep the certification world^ as elementary as possible.

See Section 3.8 [KEEP], page 104 to continue the guided tour of books^.

3.11 SET-CBD

SET-CBD to set the connected book directory

```
Example Forms:
ACL2 !>:set-cbd "/usr/home/smith/"
ACL2 !>:set-cbd (:absolute "usr" "home" "smith")
ACL2 !>:set-cbd (:relative :back "smith")
```

See Section 3.4 [CBD], page 97 for a description of the connected book directory.

```
General Form:
(set-cbd str)
```

where `str` is either a nonempty string that represents the absolute pathname for a directory, or a structured pathname that represents a directory (see Section 3.9 [PATHNAME], page 105). This command sets the connected book directory (see Section 3.4 [CBD], page 97) to the string representing the indicated directory. Thus, this command may determine which files are processed by `include-book^` and `certify-book^` command^s typed at the top-level. However, the `cbd^` is also set by those two book processing command^s; thus, the setting of the `cbd^` is irrelevant to `include-book^` command^s in other books^.

Note that if a structured pathname^ is used for `:set-cbd`, then it may be absolute or relative. (See Section 3.9 [PATHNAME], page 105 for relevant definitions.) However, if a string is used for `:set-cbd`, it must represent an absolute pathname^. We do not support syntax for relative pathname^s for use with `:set-cbd` that is peculiar to particular operating systems, such as `".."` in Unix (trademark of AT&T), because we want to be sure to be able to correctly identify the resulting absolute pathname of the `:cbd^`.

3.12 UNCERTIFIED-BOOKS

UNCERTIFIED-BOOKS invalid certificate^s and uncertified books^

`Include-book^` has a special provision for dealing with uncertified books^: If the file has no certificate^ or an invalid certificate^ (i.e., one whose check sums describe files other than the ones actually read), a warning is printed and the book is otherwise processed as though it were certified and had an open portcullis^. (For details see Chapter 3 [BOOKS], page 89, see Section 3.5 [CERTIFICATE], page 99, and see Section 3.10 [PORTCULLIS], page 108.)

This can be handy, but it can have disastrous consequences.

The provision allowing uncertified books^ to be included can have disastrous consequences, ranging from hard lisp errors, to damaged memory, to quiet logical inconsistency.

It is possible for the inclusion of an uncertified book to render the logic inconsistent. For example, one of its non-`local^` events^ might be `(defthm t-is-nil (equal t nil))`. It is also possible for the inclusion of an uncertified book to cause hard errors or breaks^ into raw Common Lisp. For example, if the file has been edited since it was certified, it may contain too many open parentheses, causing Lisp to read past “end of file.” Similarly, it might contain non-ACL2 objects such as `3.1415` or ill-formed event forms that cause ACL2 code to break.

Even if a book is perfectly well formed and could be certified (in a suitable extension of ACL2's initial world^), its uncertified inclusion might cause Lisp errors or inconsistencies! For example, it might mention packages that do not exist in the host world^. The portcullis^ of a certified book insures that the correct `defpkg`^s have been admitted, but if a book is read without actually raising its portcullis^, symbols in the file, e.g., `acl2-arithmetic::fn`, could cause "unknown package" errors in Common Lisp. Perhaps the most subtle disaster occurs if the host world^ does have a `defpkg`^ for each package used in the book but the host `defpkg`^ imports different symbols than those required by the portcullis^. In this case, it is possible that formulas which were theorems in the certified book are non-theorems in the host world^, but those formulas can be read without error and will then be quietly assumed.

In short, if you include an uncertified book, **all bets are off** regarding the validity of the future behavior of ACL2.

That said, it should be noted that ACL2 is pretty tough and if errors don't occur, the chances are that deductions after the inclusion of an uncertified book are probably justified in the (possibly inconsistent) logical extension obtained by assuming the admissibility and validity of the definitions and conjectures in the book.

4 BREAK-REWRITE

BREAK-REWRITE   the read-eval-print loop entered to monitor^ rewrite rules

ACL2 allows the user to monitor^ the application of rewrite rules. When monitor^ed rules are about to be tried by the rewriter, an interactive break occurs and the user is allowed to watch and, in a limited sense, control the attempt to apply the rule. This interactive loop, which is technically just a call of the standard top-level ACL2 read-eval-print loop, `ld^`, on a "wormhole^ state^" (see Section 8.90 [WORMHOLE], page 364), is called "break-rewrite." While in break-rewrite, certain keyword commands are available for accessing information about the context in which the lemma is being tried. These keywords are called break-rewrite "commands."

To abort from inside break-rewrite at any time, execute sharpsign-period (`#.`).

For further information, see the related `:doc^` topics listed below.

* BREAK-LEMMA:: a quick introduction to breaking rewrite rules in ACL2
* BRR:: to enable or disable the breaking of rewrite rules
* BRR-COMMANDS:: Break-Rewrite^ Commands
* BRRatsign:: (BRR@) to access context sensitive information within `break-rewrite^`
* MONITOR:: to monitor the attempted application of a rule name
* MONITORED-RUNES:: print the monitor^ed rune^s and their break conditions
* OK-IF:: conditional exit from `break-rewrite`
* UNMONITOR:: to stop monitoring a rule name

As explained in the documentation for `monitor^`, it is possible to cause the ACL2 rewriter to monitor^ the attempted application of selected rules. When such a rule is about to be tried, the rewriter evaluates its break condition and if the result is non-`nil`, break-rewrite is entered.

Break-rewrite permits the user to inspect the current state^ by evaluating break-rewrite commands. Type `:help^` in break-rewrite to see what the break-rewrite commands are. However, break-rewrite is actually just a call of the general ACL2 read-eval-print loop, `ld^`, on a certain state^ and the break-rewrite commands are simply aliases provided by the `ld^`-special `ld-keyword-aliases^`. See Section 9.7 [LD], page 380 for details about this read-eval-print loop. Thus, with a few exceptions, anything you can do at the ACL2 top-level can be done within break-rewrite. For example, you can evaluate arbitrary expressions, use the keyword command hack, access documentation^, print events^, and even define functions and prove theorems. However, the "certain state^" upon which `ld^` was called is a "wormhole^ state^" (see Section 8.90 [WORMHOLE], page 364) because break-rewrite is not allowed to have any effect upon the behavior of rewrite. What this means, very roughly but understandably, is that break-rewrite operates on a copy of the state^ being used by rewrite and when break-rewrite exits the wormhole^ closes and the state^ "produced" by break-rewrite disappears. Thus, break-rewrite lets you query the state of the rewriter and even do experiments involving proofs, etc., but these experiments have no effect on the ongoing proof attempt.

When you first enter break-rewrite a simple herald is printed such as:

```
(3 Breaking (:rewrite lemma12) on (delta a (+ 1 j)):
```

The integer after the open parenthesis indicates the depth of nested break-rewrite calls. In this discussion we use 3 consistently for this integer. Unless you abort or somehow enter unbalanced parentheses into the script, the entire session at a given depth will be enclosed in balanced parentheses, making it easy to skip over them in Emacs.

You then will see the break-rewrite prompt^:

```
3 ACL2 !>
```

The leading integer is, again, the depth. Because breaks often occur recursively it is convenient always to know the level with which you are interacting.

You may type arbitrary commands as in the top-level ACL2 loop. For example, you might type:

```
3 ACL2 !>:help
```

or

```
3 ACL2 !>:pe lemma12
```

More likely, upon entering break-rewrite you will determine the context of the attempted application. Here are some useful commands:

```
3 ACL2 >:target           ; the term being rewritten
3 ACL2 >:unify-subst      ; the unifying substitution
3 ACL2 >:path             ; the stack of goals pursued by the rewriter
                          ; starting at the top-level clause being simplified
                          ; and ending with the current application
```

At this point in the interaction the system has not yet tried to apply the monitor^ed rule. That is, it has not tried to establish the hypotheses, considered the heuristic cost of backchaining, rewritten the right-hand side of the conclusion, etc. When you are ready for it to try the rule you can type one of several different "proceed" commands. The basic proceed commands are `:ok`, `:go`, and `:eval`.

```
:ok
```

exits break-rewrite without further interaction. When break-rewrite exits it prints "`3)`", closing the parenthesis that opened the level `3` interaction.

```
:go
```

exits break-rewrite without further interaction, but prints out the result of the application attempt, i.e., whether the application succeeded, if so, what the `:target` term was rewritten to, and if not why the rule was not applicable.

```
:eval
```

causes break-rewrite to attempt to apply the rule but interaction at this level of break-rewrite resumes when the attempt is complete. When control returns to this level of break-rewrite a message indicating the result of the application attempt (just as in `:go`) is printed, followed by the prompt^ for additional user input.

Generally speaking, `:ok` and `:go` are used when the break in question is routine or uninteresting and `:eval` is used when the break is one that the user anticipates is causing trouble. For example, if you are trying to determine why a lemma isn't being applied to a given term and the `:target` of the current break-rewrite is the term in question, you would usually `:eval` the rule and if break-rewrite reports that the rule failed then you are in a position to determine why, for example by carefully inspecting the `:type-alist` of governing assumptions or why some hypothesis of the rule could not be established.

It is often the case that when you are in break-rewrite you wish to change the set of monitor^ed rune^s. This can be done by using `:monitor^` and `:unmonitor^` as noted above. For example, you might want to monitor^ a certain rule, say `hyp-reliever`, just when it is being used while attempting to apply another rule, say `main-lemma`. Typically then you would monitor^ `main-lemma` at the ACL2 top-level, start the proof-attempt, and then in the break-rewrite in which `main-lemma` is about to be tried, you would install a monitor^ on `hyp-reliever`. If during the ensuing `:eval` `hyp-reliever` is broken you will know it is being used under the attempt to apply `main-lemma`.

However, once `hyp-reliever` is being monitor^ed it will be monitor^ed even after `main-lemma` has been tried. That is, if you let the proof attempt proceed then you may see many other breaks on `hyp-reliever`, breaks that are not "under" the attempt to apply `main-lemma`. One way to prevent this is to `:eval` the application of `main-lemma` and then `:unmonitor^ hyp-reliever` before exiting. But this case arises so often that ACL2 supports several additional "flavors" of proceed commands.

`:Ok!`, `:go!`, and `:eval!` are just like their counterparts (`:ok`, `:go`, and `:eval`, respectively), except that while processing the rule that is currently broken no rune^s are monitor^ed. When consideration of the current rule is complete, the set of monitor^ed rune^s is restored to its original setting.

`:Ok$`, `:go$`, and `:eval$` are similar but take an additional argument which must be a list of rune^s. An example usage of `:eval$` is

```
3 ACL2 !>:eval$ ((:rewrite hyp-reliever))
```

These three commands temporarily install unconditional breaks on the rune^s listed, proceed with the consideration of the currently broken rule, and then restore the set of monitor^ed rules to its original setting.

Thus, there are nine ways to proceed from the initial entry into break-rewrite although we often speak as though there are two, `:ok` and `:eval`, and leave the others implicit. We group `:go` with `:ok` because in all their flavors they exit break-rewrite without further interaction (at the current level). All the flavors of `:eval` require further interaction after the rule has been tried.

To abort a proof attempt and return to the top-level of ACL2 you may at any time type `#.` (that is, number-sign followed by a period) followed by a carriage return.

We now address ourselves to the post-`:eval` interaction with break-rewrite. As noted, that interaction begins with break-rewrite's report on the results of applying the rule: whether it worked and either what it produced or why it failed. This information is also printed by certain keyword commands available after `:eval`, namely `:wonp`, `:rewritten-rhs`, and `:failure-reason`. In addition, by using `brr@^` (see Section 4.4 [BRR@], page 121) you can obtain this information in the form of ACL2 data objects. This allows the development of more sophisticated "break conditions"; see Section 4.5 [MONITOR], page 123 for examples. In this connection we point out the macro form `(ok-if term)`. See Section 4.7 [OK-IF], page 127. This command exits break-rewrite if `term` evaluates to non-`nil` and otherwise does not exit. Thus it is possible to define macros that provide other kinds of exits from break-rewrite. The only way to exit break-rewrite after `:eval` is `:ok` (or, equivalently, the use of `ok-if^`).

ACL2 users who wish to know more about break-rewrite so that they can develop more convenient ways to monitor^ rules are encouraged to speak to J Moore.

The rest of this documentation^ discusses a few implementation details of break-rewrite and may not be interesting to the typical user.

There is no ACL2 function named break-rewrite. It is an illusion created by appropriate calls to two functions named `brkpt1` and `brkpt2`. As previously noted, break-rewrite is `ld^` operating on a wormhole^ state^. One might therefore wonder how break-rewrite can apply a rule and then communicate the results back to the rewriter running in the external state^. The answer is that it cannot. Nothing can be communicated through a wormhole^. In fact, `brkpt1` and `brkpt2` are each calls of `ld^` running on wormhole^ state^s. `Brkpt1` implements the pre-`:eval` break-rewrite and `brkpt2` implements the post-`:eval` break-rewrite. The rewriter actually calls `brkpt1` before attempting to apply a rule and calls `brkpt2` afterwards. In both cases, the rewriter passes into the wormhole^ the relevant information about the current context. Logically `brkpt1` and `brkpt2` are no-ops and `rewrite^` ignores the `nil` they return. But while control is in them the execution of `rewrite^` is suspended and cannot proceed until the break-rewrite interactions complete.

This design causes a certain anomoly that might be troubling. Suppose that inside break-rewrite before `:evaling` a rule (i.e., in the `brkpt1` wormhole^ state^) you define some function, `foo`. Suppose then you `:eval` the rule and eventually control returns to break-rewrite (i.e., to `brkpt2` on a wormhole^ state^ with the results of the application in it). You will discover that `foo` is no longer defined! That is because the wormhole^ state^ created during your `pre-:eval` interaction is lost when we exit the wormhole^ to resume the proof attempt. The post-`:eval` wormhole^ state^ is in fact identical to the initial pre-`:eval` state^ (except for the results of the application) because `rewrite^` did not change the external state^ and both wormhole^ state^s are copies of it.

There is a lot more to know about break-rewrite, most of which is fairly easy to learn from looking at the code, since it is all expressed in ACL2. Feel free to ask questions of J Moore.

4.1 BREAK-LEMMA

BREAK-LEMMA a quick introduction to breaking rewrite rules in ACL2

```
Example:
:brr t                          ; if you haven't done that yet
:monitor (:rewrite lemma12) t   ; to install a break point on the
                                ; rule named (:rewrite lemma12)
```

ACL2 does not support Nqthm's `break-lemma` but supports a very similar and more powerful break facility. Suppose some proof is failing; apparently some particular rule is not being used and you wish to learn why. Then you need the ACL2 break-rewrite^ facility. See Chapter 4 [BREAK-REWRITE], page 113 and all of its associated `:doc^` topics for details. The following basic steps are required.

(1) To enable the "break rewrite" feature, you must first execute

```
ACL2 !>:brr t
```

at the top-level of ACL2. Equivalently, evaluate `(brr t)`. Break-rewrite^ stays enabled until you disable it with `(brr nil)`. When break-rewrite^ is enabled the ACL2 rewriter will run slower than normal but you will be able to monitor^ the attempts to apply specified rules.

(2) Decide what rune^s (see Section 15.10 [RUNE], page 675) you wish to monitor^. For example, you might want to know why `(:rewrite lemma12 . 2)` is not being used in the attempted proof. That, by the way, is the name of the second rewrite rule generated from the event named `lemma12`.

The command

```
ACL2 !>:monitor (:rewrite lemma12 . 2) t
```

will install an "unconditional" break point on that rule. The "`t`" at the end of the command means it is unconditional, i.e., a break will occur every time the rule is tried. ACL2 supports conditional breaks also, in which case the `t` is replaced by an expression that evaluates to non-`nil` when you wish for a break to occur. See Section 4.5 [MONITOR], page 123. The above keyword command is, of course, equivalent to

```
ACL2 !>(monitor '(:rewrite lemma12 . 2) t)
```

which you may also type. You may install breaks on as many rules as you wish. You must use `monitor^` on each rule. You may also change the break condition on a rule with `monitor^`. Use `unmonitor^` (see Section 4.8 [UNMONITOR], page 127) to remove a rule from the list of monitor^ed rules.

(3) Then try the proof again. When a monitor^ed rule is tried by the rewriter you will enter an interactive break, called break-rewrite^. See Chapter 4 [BREAK-REWRITE], page 113 for a detailed description. Very simply, break-rewrite^ lets you inspect the context of the attempted application both before and after the attempt. When break-rewrite^ is entered it will print out

the "target" term being rewritten. If you type `:go` break-rewrite^ will try the rule and then exit, telling you (a) whether the rule was applied, (b) if so, how the target was rewritten, and (c) if not, why the rule failed. There are many other commands. See Section 4.3 [BRR-COMMANDS], page 120.

(4) When you have finished using the break-rewrite^ feature you should disable it to speed up the rewriter. You can disable it with

```
ACL2 !>:brr nil
```

The list of monitor^ed rules and their break conditions persists but is ignored. If you enable break-rewrite^ later, the list of monitor^ed rules will be displayed and will be used again by rewrite.

You should disable the break-rewrite^ feature whenever you are not intending to use it, even if the list of monitor^ed rules is empty, because the rewriter is slowed down as long as break-rewrite^ is enabled.

Finally, a wonderful trick is the following. When there is a stack overflow, abort the proof and then try it again after turning on rewrite stack monitoring with `:brr t`. Then, provoke the stack overflow again. Quit the ACL2 top-level loop and execute the following form in raw Lisp:

```
  (cw-gstack *deep-gstack* state)
```

The loop in the rewriter will probably be evident!

4.2 BRR

BRR   to enable or disable the breaking of rewrite rules

```
Example:
:brr t       ; enable
:brr nil     ; disable

General Form:
(brr flg)
```

where `flg` evaluates to `t` or `nil`. This function modifies `state^` so that the attempted application of certain rewrite rules are "broken." "`Brr`" stands for "break-rewrite" and can be thought of as a mode with two settings. The normal mode is "disabled."

When `brr` mode is "enabled" the ACL2 rewriter monitors the attempts to apply certain rules and advises the user of those attempts by entering an interactive wormhole break. From within this break the user can watch selected application attempts. See Chapter 4 [BREAK-REWRITE], page 113.

The rules monitored are selected by using the `monitor^` and `unmonitor^` commands. It is possible to break a rune "conditionally" in the sense that an interactive break will occur only if a specified predicate is true of the environment at the time of the attempted application. See Section 4.5 [MONITOR], page 123 and see Section 4.8 [UNMONITOR], page 127.

Even if a non-empty set of rules has been selected, no breaks will occur unless `brr` mode is enabled. Thus, the first time in a session that you wish to monitor a rewrite rule, use `:brr t` to enable `brr` mode. Thereafter you may select runes to be monitored with `monitor^` and `unmonitor^` with the effect that whenever monitored rules are tried (and their break conditions are met) an interactive break will occur. Be advised that when `brr` mode is enabled the rewriter is somewhat slower than normal. Furthermore, that sluggishness persists even if no runes are monitored. You may regain normal performance — regardless of what runes are monitored — by disabling `brr` mode with `:brr nil`.

Why isn't `brr` mode disabled automatically when no runes are monitored? More generally, why does ACL2 have `brr` mode at all? Why not just test whether there are monitored runes? If you care about the answers, see Section 8.88 [WHY-BRR], page 361.

4.3 BRR-COMMANDS

BRR-COMMANDS Break-Rewrite^ Commands

```
#.              abort to ACL2 top-level
:target         term being rewritten
:unify-subst    substitution making :lhs equal :target
:hyps           hypotheses of the rule
:hyp i          ith hypothesis of the rule
:lhs            left-hand side of rule's conclusion
:rhs            right-hand side of rule's conclusion
:type-alist     type assumptions governing :target
```

```
:initial-ttree  ttree before :eval
:ancestors      negations of backchaining hypotheses being pursued
:wonp           indicates if application succeed (after :eval)
:rewritten-rhs  rewritten :rhs (after :eval)
:final-ttree    ttree after :eval
:failure-reason reason rule failed (after :eval)
:path           rewrite's path from top clause to :target
:frame i        ith frame in :path
:top            top-most frame in :path
:ok             exit break
:go             exit break, printing result
:eval           try rule and re-enter break afterwards
:ok!            :ok but no recursive breaks
:go!            :go but no recursive breaks
:eval!          :eval but no recursive breaks
:ok$ runes      :ok with runes monitored during recursion
:go$ runes      :go with runes monitored during recursion
:eval$ runes    :eval with runes monitored during recursion
:help           this message
:standard-help  :help message from ACL2 top-level
```

Break-rewrite^ is just a call of the standard ACL2 read-eval-print loop, `ld^`, on a "wormhole^" state^. Thus, you may execute any command^ you might normally execute at the top-level of ACL2. However, all state^ changes you cause from within break-rewrite^ are lost when you exit or `:eval` the rule. See Chapter 4 [BREAK-REWRITE], page 113 for more details and see Section 9.7 [LD], page 380 for general information about the standard ACL2 read-eval-print loop.

4.4 BRR@

BRR@ to access context sensitive information within `break-rewrite^`

```
Example:
(brr@ :target)      ; the term being rewritten
(brr@ :unify-subst) ; the unifying substitution

General Form:
(brr@ :symbol)
```

where `:symbol` is one of the following keywords. Those marked with `*` probably require an implementor's knowledge of the system to use effectively. They are supported but not well documented. More is said on this topic following the table.

```
:symbol             (brr@ :symbol)
```

```
-------           --------------------

:target           the term to be rewritten.  This term is an
                  instantiation of the left-hand side of the
                  conclusion of the rewrite-rule being broken.
                  This term is in translated form!  Thus, if
                  you are expecting (equal x nil) -- and your
                  expectation is almost right -- you will see
                  (equal x 'nil); similarly, instead of (cadr a)
                  you will see (car (cdr a)).  In translated
                  forms, all constants are quoted (even nil, t,
                  strings and numbers) and all macros are
                  expanded.

:unify-subst      the substitution that, when applied to :target,
                  produces the left-hand side of the rule being
                  broken.  This substitution is an alist pairing
                  variable symbols to translated (!) terms.

:wonp             t or nil indicating whether the rune was
                  successfully applied.  (brr@ :wonp) returns
                  nil if evaluated before :EVALing the rule.

:rewritten-rhs    the result of successfully applying the rule
                  or else nil if (brr@ :wonp) is nil.  The result
                  of successfully applying the rule is always a
                  translated (!) term and is never nil.

:failure-reason   some non-nil lisp object indicating why the rule
                  was not applied or else nil.  Before the rule is
                  :EVALed, (brr@ :failure-reason) is nil.  After
                  :EVALing the rule, (brr@ :failure-reason) is nil
                  if (brr@ :wonp) is t.  Rather than document the
                  various non-nil objects returned as the failure
                  reason, we encourage you simply to evaluate
                  (brr@ :failure-reason) in the contexts of interest.
                  Alternatively, study the ACL2 function tilde-@-
                  failure-reason-phrase.

:lemma          * the rewrite rule being broken.  For example,
                  (access rewrite-rule (brr@ :lemma) :lhs) will
                  return the left-hand side of the conclusion
                  of the rule.

:type-alist     * a display of the type-alist governing :target.
                  Elements on the displayed list are of the form
                  (term type), where term is a term and type
                  describes information about term assumed to hold
```

```
                      in the current context.  The type-alist may be
                      used to determine the current assumptions, e.g.,
                      whether A is a CONSP.

:ancestors         *  a list of translated terms each of which may be
                      assumed to be true because the rewriter is currently
                      trying to prove each nil.  When the rewriter
                      recursively rewrites a hypothesis of a rule it
                      adds the negation of the hypothesis to ancestors.
                      Ancestors is primarily used to prevent infinite
                      backchaining.

:gstack            *  the current goal stack.  The gstack is maintained
                      by rewrite and is the data structure printed as the
                      current ``path.''  Thus, any information derivable
                      from the :path brr command is derivable from gstack.
                      For example, from gstack one might determine that
                      the current term is the second hypothesis of a
                      certain rewrite rule.
```

In general `brr@-expressions` are used in break conditions, the expressions that determine whether interactive breaks occur when monitor^ed rune^s are applied. See Section 4.5 [MONITOR], page 123. For example, you might want to break only those attempts in which one particular term is being rewritten or only those attempts in which the binding for the variable `a` is known to be a `consp^`. Such conditions can be expressed using ACL2 system functions and the information provided by `brr@`. Unfortunately, digging some of this information out of the internal data structures may be awkward or may, at least, require intimate knowledge of the system functions. But since conditional expressions may employ arbitrary functions and macros, we anticipate that a set of convenient primitives will gradually evolve within the ACL2 community. It is to encourage this evolution that `brr@` provides access to the `*`'d data.

4.5 MONITOR

MONITOR   to monitor the attempted application of a rule name

```
Example:
(monitor '(:rewrite assoc-of-app) 't)
:monitor (:rewrite assoc-of-app) t
:monitor (:definition app) (equal (brr@ :target) '(app c d))

General Form:
(monitor rune term)
```

where `rune` is a rune^ and `term` is a term, called the “break condition.” `Rune` must be either a `:rewrite` rune^ (of subclass `backchain`) or a `:definition` rune^. (Note: Some `:rewrite` rules are considered “simple abbreviations” — see Section 8.79 [SIMPLE], page 340 — and cannot be monitor^ed. Monitoring is carried out by code inside the rewriter but abbreviation rules are applied by a special purpose simplifier.) When a rune^ is monitor^ed any attempt to apply it may result in an interactive break in an ACL2 “wormhole^ state^.” There you will get a chance to see how the application proceeds. See Chapter 4 [BREAK-REWRITE], page 113 for a description of the interactive loop entered. Whether an interactive break occurs depends on the value of the break condition expression associated with the monitor^ed rune^.

To remove a rune^ from the list of monitor^ed rune^s, use `unmonitor`. To see which rune^s are monitor^ed and what their break conditions are, evaluate `(monitored-runes)`.

`Monitor`, `unmonitor` and `monitored-runes` are macros that expand into expressions involving `state`. While these macros appear to return the list of monitor^ed rune^s this is an illusion. They all print monitor^ed rune^ information to the comment window and then return error triples instructing `ld` to print nothing. It is impossible to return the list of monitor^ed rune^s because it exists only in the wormhole^ state^ with which you interact when a break occurs. This allows you to change the monitor^ed rune^s and their conditions during the course of a proof attempt without changing the state^ in which the the proof is being constructed.

Unconditional break points are obtained by using the break condition `t`. We now discuss conditional break points. The break condition, `expr`, must be a term that contains no free variables other than `state` and that returns a single non-`state` result. In fact, the result should be `nil`, `t`, or a true list of commands to be fed to the resulting interactive break. Whenever the system attempts to use the associated rule, `expr` is evaluated in the wormhole^ interaction state^. A break occurs only if the result of evaluating `expr` is non-`nil`. If the result is a true list, that list is appended to the front of `standard-oi` and hence is taken as the initial user commands issued to the interactive break.

In order to develop effective break conditions it must be possible to access context sensitive information, i.e., information about the context in which the monitor^ed rune^ is being tried. The `brr@` macro may be used in break conditions to access such information as the term being rewritten and the current governing assumptions. This information is not stored in the proof state^ but is transferred into the wormhole^ state^ when breaks occur. The macro form is `(brr@ :sym)` where `:sym` is one of several keyword symbols, including `:target` (the term being rewritten), `:unify-subst` (the substitution that instantiates the left-hand side of the conclusion of the rule so that it is the target term), and `:type-alist` (the governing assumptions). See Section 4.4 [BRR@], page 121.

For example,

```
ACL2 !>:monitor (:rewrite assoc-of-app)
                (equal (brr@ :target) '(app a (app b c)))
```

will monitor `(:rewrite assoc-of-app)` but will cause an interactive break only when the target term, the term being rewritten, is `(app a (app b c))`.

Because break conditions are evaluated in the interaction environment, the user developing a break condition for a given rune^ can test candidate break conditions before installing them. For example, suppose an unconditional break has been installed on a rune^, that an interactive break has occurred and that the user has determined both that this particular application is uninteresting and that many more such applications will likely occur. An appropriate response would be to develop an expression that recognizes such applications and returns `nil`. Of course, the hard task is figuring out what makes the current application uninteresting. But once a candidate expression is developed, the user can evaluate it in the current context simply to confirm that it returns `nil`.

Recall that when a break condition returns a non-`nil` true list that list is appended to the front of `standard-oi`. For example,

```
ACL2 !>:monitor (:rewrite assoc-of-app) '(:go)
```

will cause `(:rewrite assoc-of-app)` to be monitor^ed and will make the break condition be `'(:go)`. This break condition always evaluates the non-`nil` true list `(:go)`. Thus, an interactive break will occur every time `(:rewrite assoc-of-app)` is tried. The break is fed the command `:go`. Now the command `:go` causes `break-rewrite` to (a) evaluate the attempt to apply the lemma, (b) print the result of that attempt, and (c) exit from the interactive break and let the proof attempt continue. Thus, in effect, the above `:monitor` merely "traces" the attempted applications of the rune^ but never causes an interactive break requiring input from the user.

It is possible to use this feature to cause a conditional break where the effective break condition is tested **after** the lemma has been tried. For example:

```
ACL2 !>:monitor (:rewrite lemma12)
                '(:unify-subst
                  :eval$ nil
                  :ok-if (or (not (brr@ :wonp))
                             (not (equal (brr@ :rewritten-rhs) '(foo a))))
                  :rewritten-rhs)
```

causes the following behavior when `(:rewrite lemma12)` is tried. A break always occurs, but it is fed the commands above. The first, `:unify-subst`, causes `break-rewrite` to print out the unifying substitution. Then in response to `:eval$ nil` the lemma is tried but with all rune^s temporarily unmonitor^ed. Thus no breaks will occur during the rewriting of the hypotheses of the lemma. When the attempt has been made, control returns to `break-rewrite` (which will print the results of the attempt, i.e., whether the lemma was applied, if so what the result is, if not why it failed). The next command, the `:ok-if` with its following expression, is a conditional exit command. It means exit `break-rewrite` if either the attempt was unsuccessful, `(not (brr@ :wonp))`, or if the result of the rewrite is any term other than `(foo a)`. If this condition is met, the break is exited and the remaining break commands are irrelevant. If this condition is not met then the next command, `:rewritten-rhs`, prints the result of the application (which in this contrived example is known to be `(foo a)`). Finally, the list of supplied commands is exhausted but `break-rewrite` expects more input. Therefore, it begins prompting the user for input. The end result, then, of the above `:monitor` command is that the rune^ in question is elaborately traced and interactive breaks occur whenever it rewrites its target to `(foo a)`.

We recognize that the above break condition is fairly arcane. We suspect that with experience we will develop some useful idioms. For example, it is straightforward now to define macros that monitor rune^s in the ways suggested by the following names: `trace-rune`, `break-if-target-is`, and `break-if-result-is`. For example, the last could be defined as

```
(defmacro break-if-result-is (rune term)
  `(monitor ',rune
            ''(:eval :ok-if
                     (not (equal (brr@ :rewritten-rhs) ',term)))))
```

(Note however that the submitted term must be in translated form.)

Since we don't have any experience with this kind of control on lemmas we thought it best to provide a general (if arcane) mechanism and hope that the ACL2 community will develop the special cases that we find most convenient.

4.6 MONITORED-RUNES

MONITORED-RUNES   print the monitor^ed rune^s and their break conditions

```
Example and General Form:
:monitored-runes
```

This macro prints a list, each element of which is of the form `(rune expr)`, showing each monitor^ed rune^ and its current break condition.

4.7 OK-IF

OK-IF conditional exit from `break-rewrite`

```
Example Form:
:ok-if (null (brr@ :wonp))

General Form:
:ok-if expr
```

where `expr` is a term involving no free variables other than `state` and returning one non-`state` result which is treated as Boolean. This form is intended to be executed from within `break-rewrite` (see Chapter 4 [BREAK-REWRITE], page 113).

Consider first the simple situation that the `(ok-if term)` is a command read by `break-rewrite`. Then, if the term is non-`nil`, `break-rewrite` exits and otherwise it does not.

More generally, `ok-if` returns an ACL2 error triple `(mv erp val state)`. (See Section 9.7 [LD], page 380 for more on error triples.) If any form being evaluated as a command by `break-rewrite` returns the triple returned by `(ok-if term)` then the effect of that form is to exit break-rewrite^ if term is non-`nil`. Thus, one might define a function or macro that returns the value of `ok-if` expressions on all outputs and thus create a convenient new way to exit `break-rewrite`.

The exit test, `term`, generally uses `brr@` to access context sensitive information about the attempted rule application. See Section 4.4 [BRR@], page 121. `Ok-if` is useful inside of command sequences produced by break conditions. See Section 4.5 [MONITOR], page 123. `:ok-if` is most useful after an `:eval` command has caused `break-rewrite` to try to apply the rule because in the resulting break environment `expr` can access such things as whether the rule succeeded, if so, what term it produced, and if not, why. There is no need to use `:ok-if` before `:eval`ing the rule since the same effects could be achieved with the break condition on the rule itself. Perhaps we should replace this concept with `:eval-and-break-if`? Time will tell.

4.8 UNMONITOR

UNMONITOR to stop monitoring a rule name

```
Examples:
(unmonitor '(:rewrite assoc-of-app))
:unmonitor (:rewrite assoc-of-app)
```

```
:unmonitor :all

General Forms:
(unmonitor rune)
(unmonitor :all)
```

Here, `rune` is a rune^ that is currently among those with break points installed. This function removes the break.

Subtle point: Because you may want to unmonitor a "rune^" that is no longer a rune^ in the current ACL2 world^, we don't actually check this about rune^. Instead, we simply check that rune^ is a `consp` beginning with a `keywordp`. That way, you'll know you've made a mistake if you try to `:unmonitor binary-append` instead of `:unmonitor (:definition binary-append)`, for example.

5 DOCUMENTATION

DOCUMENTATION functions that display documentation at the terminal

This section explains the ACL2 online documentation system. Thus, it assumes that you are typing at the terminal, inside an ACL2 session. If you are reading this description in another setting (for example, on paper), simply ignore the parts of this description that involve typing at the terminal.

For an introduction to the ACL2 online documentation system, type `:more`^ below. Whenever the documentation system concludes with "(cont'd)" you may type `:more`^ to see the next block of documentation.

Topics related to documentation are documented individually:

* *TERMINAL-MARKUP-TABLE*:: a markup^ table used for printing to the terminal
* ARGS:: `args`, `guard`^, `type`, `constraint`^, etc., of a function symbol
* DOC:: brief documentation^ (type `:doc name`)
* DOC!:: all the documentation^ for a name (type `:doc! name`)
* DOC-STRING:: formatted documentation^ strings
* DOCS:: available documentation^ topics (by section)
* HELP:: brief survey of ACL2 features
* MARKUP:: the markup language for ACL2 documentation^ strings
* MORE:: your response to `:doc`^ or `:more-doc`^'s "`(cont'd)`"
* MORE!:: another response to "(cont'd)"
* MORE-DOC:: a continuation of the `:doc`^ documentation^
* NQTHM-TO-ACL2:: ACL2 analogues of Nqthm functions and commands

The ACL2 online documentation feature allows you to see extensive documentation on many ACL2 functions and ideas. You may use the documentation facilities to document your own ACL2 functions and theorems.

If there is some name you wish to know more about, then type

```
ACL2 !>:doc name
```

in the top-level loop. If the name is documented, a brief blurb will be printed. If the name is not documented, but is "similar" to some documented names, they will be listed. Otherwise, `nil` is returned.

Every name that is documented contains a one-line description, a few notes, and some details. `:Doc^` will print the one-liner and the notes. When `:doc^` has finished it stops with the message "(cont'd)" to remind you that details are available. If you then type

```
ACL2 !>:more
```

a block of the continued text will be printed, again concluding with "(cont'd)" if the text continues further, or concluding with "`*-`" if the text has been exhausted. By continuing to type `:more^` until exhausting the text you can read successive blocks. Alternatively, you can type `:more!^` to get all the remaining blocks.

If you want to get the details and don't want to see the elementary stuff typed by `:doc^` name, type:

```
ACL2 !>:MORE-DOC name
```

We have documented not just function names but names of certain important ideas too. For example, see Section 14.14 [REWRITE], page 657 and see Section 14.12 [META], page 652 to learn about `:rewrite^` rules and `:meta^` rules, respectively. See Section 8.41 [HINTS], page 297 to learn about the structure of the `:hints^` argument to the prover. The `deflabel^` event (see Section 6.9 [DEFLABEL], page 162) is a way to introduce a logical name for no reason other than to attach documentation to it; also see Section 6.6 [DEFDOC], page 158.

How do you know what names are documented? There is a documentation data base which is querried with the `:docs^` command.

The documentation data base is divided into sections. The sections are listed by

```
ACL2 !>:docs *
```

Each section has a name, `sect`, and by typing

```
ACL2 !>:docs sect
```

or equivalently

```
ACL2 !>:doc sect
```

you will get an enumeration of the topics within that section. Those topics can be further explored by using :doc^ (and :more^) on them. In fact the section name itself is just a documented name. :more^ generally gives an informal overview of the general subject of the section.

```
ACL2 !>:docs **
```

will list all documented topics, by section. This fills several pages but might be a good place to start.

If you want documentation on some topic, but none of our names or brief descriptions seem to deal with that topic, you can invoke a command to search the text in the data base for a given string. This is like the GNU Emacs "apropos^" command.

```
ACL2 !>:docs "functional inst"
```

will list every documented topic whose :doc^ or :more-doc^ text includes the substring "functional inst", where case and the exact number of spaces are irrelevant.

If you want documentation on an ACL2 function or macro and the documentation data base does not contain any entries for it, there are still several alternatives.

```
ACL2 !>:args fn
```

will print the arguments and some other relevant information about the named function or macro. This information is all gleaned from the definition (not from the documentation data base) and hence this is a definitive way to determine if fn is defined as a function or macro.

You might also want to type:

```
ACL2 !>:pc fn
```

which will print the command^ which introduced fn. You should see Section 8.11 [COMMAND-DESCRIPTOR], page 245 for details on the kinds of input you can give the :pc^ command.

The entire ACL2 documentation data base is user extensible. That is, if you document your function definitions or theorems, then that documentation is made available via the data base and its query commands.

The implementation of our online documentation system makes use of Common Lisp's "documentation strings." While Common Lisp permits a documentation string to be attached to any defined concept, Common Lisp assigns no interpretation to these strings. ACL2 attaches special significance to documentation strings that begin with the characters "`:doc-section`". When such a documentation string is seen, it is stored in the data base and may be displayed via `:doc^`, `:more^`, `:docs^`, etc. Such documentation strings must follow rigid syntactic rules to permit their processing by our commands. These are spelled out elsewhere; see Section 5.5 [DOC-STRING], page 134.

A description of the structure of the documentation data base may also be found; see Section 5.5 [DOC-STRING], page 134.

5.1 *TERMINAL-MARKUP-TABLE*

TERMINAL-MARKUP-TABLE   a markup^ table used for printing to the terminal

The value of the ACL2 constant `*terminal-markup-table*` is an association list pairing markup^ keys with strings, to be used for printing to the terminal. See Section 5.8 [MARKUP], page 140 for a description of the ACL2 markup^ language.

The entries in `*terminal-markup-table*` are of the form

```
(key . fmt-string)
```

where `key` is one of the doc-string^ tilde directives (see Section 5.8 [MARKUP], page 140), and `fmt-string` is a string as expected by the ACL2 printing function `fmt^`. The system arranges that for any `arg`, when an expression ~key[arg] is encountered by the documentation^ printer, `fmt^` will print `fmt-string` in the association list where `#\a` is bound to `arg` and `#\A` is bound to the result of applying the function `string-upcase^` to `arg`.

It is possible to implement tools in ACL2 for printing documentation^ to other than the terminal. In fact, such tools exist for Texinfo and for HTML. For now, discussion of this capability is beyond the scope of the present topic.

5.2 ARGS

ARGS   `args`, `guard^`, `type`, `constraint^`, etc., of a function symbol

```
Example:
:args assoc-eq
```

`Args` takes one argument, a symbol which must be the name of a function or macro, and prints out the formal parameters, the guard^ expression, the output signature^, the deduced type, the constraint^ (if any), and whether documentation^ about the symbol is available via `:doc`^. The output signature^ is either `&` indicating that the function returns one value; `state`^, indicating that it returns a state^, or an expression `(mv & & ... state ... &)` indicating how many multiple values it returns and which if any is a state^.

5.3 DOC

DOC brief documentation^ (type `:doc name`)

NOTE: The `:doc` command only makes sense at the terminal. Furthermore it only works at the terminal when a “full-size” image has been built. Most users will probably access the ACL2 documentation in other ways, as explained in the file `"doc/README"` that comes with the ACL2 distribution.

```
Examples:
ACL2 !>:doc DEFTHM          ; print documentation of DEFTHM
ACL2 !>:doc logical-name    ; print documentation of LOGICAL-NAME
ACL2 !>:doc "MY-PKG"       ; print documentation of "MY-PKG"

Related Topics:
:more                      ; continues last :doc or :more-doc text
:more-doc name             ; prints more documentation for name
:docs **                   ; lists all documented symbols
:docs "compil"             ; documented symbols apropos "compil"
:DOC documentation         ; describes how documentation works

General Form:
ACL2>:doc logical-name
```

where `logical-name`^ is a logical name (see Section 8.61 [LOGICAL-NAME], page 322) for which you hope there is documentation^. Chances are there is no documentation^ at the moment, but we are working on adding documentation^ strings for all the user level ACL2 functions.

For a general discussion of our treatment of documentation strings, see Chapter 5 [DOCUMENTATION], page 129.

This is the first cut at online documentation^. Users can be particularly helpful by sending mail on the inadequacies of the system. Address it just to Moore and put Documentation^ in the subject line. There are several things that trouble me about what I've done here.

First, many concepts aren't documented. Ultimately, I'd like to . document (a) every CLTL primitive (e.g., `case^` and `coerce^`) and (b) every ACL2 extension (e.g., `aref1^` and `getprop`). But so far I have focussed on documenting (c) the ACL2 system primitives (e.g., `defthm^` and what `hints^` look like). My priorities are (c), then (b), and then (a), following the philosophy that the most unstable features should get online documentation^ in these early releases. Having gotten the basic documentation^ in place, I'll document new things as they are added, and in response to your pleas I'll try to add documentation^ to old things that are widely regarded as important.

Second, I worry that the existing documentation^ is unhelpful because it provides too much or too little detail, or it provides the detail too far away from where it is needed. Please be on the lookout for this. Did you get what you needed when you appealed to `:doc` or `:more-doc^`? If not, what was it you needed? Would more cross-references help^? Did you get lost in maze of cross-references?

5.4 DOC!

DOC! all the documentation^ for a name (type `:doc! name`)

NOTE: The `:doc!` command only makes sense at the terminal.

```
Examples:
ACL2 !>:doc! defthm
ACL2 !>:doc! certificate
```

This command is like `:doc name` followed by `:more!^`. It prints all the documentation^ of `name`.

5.5 DOC-STRING

DOC-STRING formatted documentation^ strings

```
Examples:
":Doc-Section name
one-liner~/notes~/details"

":Doc-Section name
```

```
one-liner~/
notes~/
details~/
:cite old-name1
:cited-by old-name2"
```

Use `(get-doc-string 'name state)` to see other examples.

Documentation^ strings not beginning with "`:Doc-Section`" (case is irrelevant) are ignored. See Section 5.8 [MARKUP], page 140 for how to supply formatting information (such as fonts and displayed text) in documentation^ strings.

ACL2 attaches special importance to documentation^ strings beginning with the header "`:Doc-Section`" (or any variant thereof obtained by changing case). Any documentation^ string that does not begin with such a header is considered unformatted and is ignored. For the rest of this discussion, we use the phrase "documentation^ string" as though it read "formatted documentation^ string."

Documentation^ strings are always processed in the context of some symbol, `name`, being defined. (Indeed, if an event defines no symbol, e.g., `verify-guards^` or `in-theory^`, then it is not permitted to have a formatted documentation^ string.) The string will be associated with name in the "documentation^ data base." The data base is divided into "sections" and each section is named by a symbol. Among the sections are `events^`, `documentation^`, `history^`, `other^`, and `miscellaneous^`. A complete list of the sections may be obtained by typing `:docs *` at the terminal. You can create new sections. The main purpose of sections is simply to partition the large set of names into smaller subsets whose contents can be enumerated separately. The idea is that the user may remember (or recognize) the relevant section name and then read its contents to find interesting items.

Within a section are "documentation^ tuples" which associate with each documented name its documentation^ string and a list of related documented names, called the "related names" of the name. When `:doc^` prints the documentation^ for name, it always lists the related names.

When a formatted documentation^ string is submitted with the defining event of some name, the section name and an initial set of related names are parsed from the string. In addition, the formatted string contains various "levels" of detail that are printed out at different times. Finally, it is possible for a string to cause the newly documented name to be added to the related names of any previously documented name. Thus, as new names are introduced they can be grouped with old ones.

The general form of an ACL2 formatted documentation^ string is

```
":DOC-SECTION <section-name>
  <one-liner>~/
  <notes>~/
  <details>~/
  :CITE <n1>
  ...
  :CITE <nn>
  :CITED-BY <m1>
  ...
  :CITED-BY <mm>"
```

Before we explain this, let it be noted that (get-doc-string name state) will return the documentation^ string associated with name in the documentation^ data base. You may want to call get-doc-string on 'pe^ and 'union-theories^ just to see some concrete documentation^ strings. This documentation^ string, which is rather long, is under 'doc-string.

A formatted documentation^ string has five parts: the header and section-name (terminating in the first #\Newline), the <one-liner>, <notes>, and <details> (each terminating in a tilde-slash ("~/") pair), and a citation part. These five parts are parsed into six components. <section-name> is read as the name of a symbol, section-name. <one-liner>, <notes>, and <details> are arbitrary sequences of characters^ (ignoring initial white space and not including the tilde-slash pairs which terminate them). The <ni> are read as symbols and assembled into a list called the "cite" symbols. The <mi> are read as symbols and assembled into a list called the "cited-by" symbols. See the warning below regarding the hackish nature of our symbol reader.

Section-name must either be a previously documented symbol or else be name, the symbol being documented. To open a new section of the data base, named section-name, you should define the logical name section-name (as by deflabel^ or any other event; also see Section 6.6 [DEFDOC], page 158) and attach to it a documentation^ string for section section-name. You might wish to print out the documentation^ string we use for some of our section names, e.g., (get-doc-string 'events state). By forcing section names to be documented symbols, we permit sections themselves to have one line descriptions and discussions, presented by the standard documentation^ facilities like the facilities :doc^ and :more-doc^ that may be used at the terminal.

Each of the ni's and mi's must be previously documented symbols.

Both <one-liner> and <details> must be non-empty, i.e., must contain some non-whitespace characters^. <notes> may be empty. The :cites and :cited-bys pairs may be intermingled and may be separated by either newlines or spaces. The citation part may be empty. When the citation part is empty, the tilde-slash pair terminating the <details> part may be omitted. Thus, the simplest form of a formatted documentation^ string is:

```
":Doc-Section <section-name>
 <one-liner>~/~/
 <details>"
```

Since white space at the front of <one-liner>, <notes> and <details> is ignored, we often precede those parts by #\Newlines to make the strings easier to read in our source files. We also typically indent all of the text in the string by starting each line with a few spaces. (The Emacs commands for formatting Lisp get confused if you have arbitrary characters^ on the left margin.) We assume that every line in <one-liner>, <notes>, and <details> starts with at least as many spaces as <one-liner> does, i.e., we assume they are all indented the same amount (or more). Let d be the number of spaces separating <one-liner> from the #\Newline preceding it. When the various parts are printed, we "de-indent" by stripping out the first d spaces following each #\Newline.

However, we find that when documentation^ is printed flush against the left margin it is difficult to distinguish the documentation^ text from previous output. We therefore prefix each line we print by a special pad of characters^. By default, this pad is "| " so that documentation^ text has a vertical bar running down the left margin. But the pad is just the value of the global variable doc-prefix and you may assign^ it any string you wish.

To add such a string to the data base under the symbol name we make a new entry in the section-name section of the data base. The entry associates name with the string and uses the string's cites list as the initial value of the related names field. In addition, we add name to the related names field of each of the names listed in the string's cited-by list. We also add name to the related names field of its section-name. Observe that the cites list in a string is only the initial value of the related names of the names. Future documentation^ strings may add to it via :cited-by or :doc-section. Indeed, this is generally the case. We discuss this further below.

When a brief description of name is required (as by :docs **), name and <one-liner> are printed. <one-liner> is usually printed starting in column 15 (however see Section 8.69 [PRINT-DOC-START-COLUMN], page 331). Despite its name, <one-liner> need not be one line. It usually is one line, however.

When you type :doc^ name at the terminal, the first response will be to print name and <one-liner>. Then :doc^ prints <notes>, if any. Then, if name is the name of a section, it prints the <one-liner>s for each of its related names. For example, try :doc events. If name is not a section name but does have some related names, they are merely listed but not explained. Try :doc theory-functions. :more-doc name prints <details>.

Our style is to let each new concept add itself to the related names of old concepts. To do otherwise increases the chances that documentation^ gets outdated because one often forgets to update supposedly complete lists of the relevant topics when new topics are invented. For example, `:doc theory-functions` lists each available theory function. But `get-doc-string` of `'theory-functions^` just shows a few examples and has an empty cites list. From where do we get the names of the theory functions listed by `:doc^`? The answer is that each theory function has its own documentation^ string and those strings each specify `:cited-by` theory-functions^. See for example `get-doc-string` of `'union-theories^`. So by the time the entire system is assembled, the related names of `'theory-functions^` contains all the (documented) theory functions. This makes it easy to add new theory functions without changing the general discussion in `'theory-functions^`.

When an event or command^ form is printed, as by `:pe^` or `:pc^`, that contains a formatted documentation^ string, we do not print the actual documentation^ string (since they are usually large and distracting). Instead we print the string:

```
"Documentation available via :doc"
```

inviting you to use `:doc^` and `:more-doc^` (or `get-doc-string`) if you wish to see the documentation^ at the terminal.

Warning on Reading Symbols from Strings: When we read a symbol, such as the section-symbol, from a documentation^ string, we use a quick and dirty imitation of the much more powerful CLTL `read` program. In particular, we scan past any whitespace, collect all the characters^ we see until we get to more whitespace or the end of the string, convert the characters^ to upper case, make a string out of them, and `intern^` that string. Thus, if you typed `":Doc-Section 123 ..."` we would read the `123` as the symbol `|123|`. Observe that special characters^, such as parentheses and escape characters^, are not afforded their usual reverence by our hack. Furthermore, the question arises: in which package do we `intern^` the symbol? The answer is, usually, the package containing the name being defined. I.e., if you are documenting `my-pkg::name` and you attach a documentation^ string that begins `":Doc-Section: Machines ..."` then the section-symbol will be `my-pkg::machines`. We recognize two special cases. If the first character read is a colon, we use the `keyword` package. If the first five characters^ read are `acl2::` then we intern^ in the `"ACL2"` package. Our own section names, e.g., `events^`, are in the `"ACL2"` package.

In a related area, when you ask for the documentation^ of a name, e.g., when you type `:doc name` at the terminal, that name is read with the full ACL2 reader, not the hack just described. That name is read into the current package. Thus, if you are operating `in-package^` `"MY-PKG"` and type `:doc events`, what is read is `my-pkg::events`. The data base may not contain an entry for this symbol. Before reporting that no documentation^ exists, we try `acl2::events`.

One last note: **defpkg^** permits a formatted documentation^ string, which is associated in the data base with the name of the package. But the name of the package is a string, not a symbol. It is permitted to access the documentation^ of a string (i.e., package name). But there are no facilities for getting such a **stringp^** name into the related names of another name nor of making such stringp^ names be section names. That is because we always read symbols from strings and never read strings from strings. I.e., if you did write `"Doc-Section \"MY-PKG\" ..."` it would read in as a weird symbol.

5.6 DOCS

DOCS available documentation^ topics (by section)

NOTE: The `:docs` command only makes sense at the terminal.

```
Examples:
ACL2 !>:docs *          ; lists documentation sections
ACL2 !>:docs **         ; lists all documented topics within all sections
ACL2 !>:docs events     ; lists all topics in section EVENTS
ACL2 !>:docs "compil"   ; lists topics ``apropos''
```

The data base of formatted documentation^ strings is structured into sections. Within a section are topics. Each topic has a one-liner, some notes, and some detailed discussions. The `:docs` command provides a view of the entire data base.

`:docs` takes one argument, as described below:

```
arg             effect

*               list all section headings in the data base
**              list all section headings and all topics within
                each section
name            list all topics in the section named name (where
                name is some symbol other than * and **).  This
                is always the same as :doc name.
pattern         list all topics whose :doc or :more-doc text
                mentions the string pattern.  For purposes of this
                string matching we ignore distinctions of case and
                the amount and kind (but not presence) of white
                space.  We also treat hyphen as whitespace.
```

5.7 HELP

HELP brief survey of ACL2 features

```
Example:
ACL2 !>:help
```

See Section 8.63 [LP], page 327 for general information about the top-level command^ environment for ACL2.

5.8 MARKUP

MARKUP the markup language for ACL2 documentation^ strings

ACL2 documentation^ strings make special use of the tilde character (~). In particular, we describe here a "markup language" for which the tilde character plays a special role. The markup language is valuable if you want to write documentation^ that is to be displayed outside your ACL2 session. If you are not writing such documentation^, and if also you do not use the character '~', then there is no need to read on.

Three uses of the tilde character (~) in documentation^ strings are as follows. Below we explain the uses that constitute the ACL2 markup language.

`~/`

Indicates the end of a documentation **section**; see Section 5.5 [DOC-STRING], page 134.

`~~`

Indicates the literal insertion of a tilde character (~).

`~]`

This directive in a documentation string is effective only during the processing of part 2, the details (see Section 5.5 [DOC-STRING], page 134), and controls how much is shown on each round of `:more`^ processing when printing to the terminal. If the system is not doing `:more^` processing, then it acts as though the `~]` is not present. Otherwise, the system put out a newline and halts documentation printing on the present topic, which can be resumed if the user types `:more^` at the terminal.

The other uses of the tilde character are of the following form.

```
~key[arg]
```

Before launching into an explanation of how this works in detail, let us consider some small examples.

Here is a word that is code:

```
~c[function-name].
```

Here is a phrase with an "emphasized" word, "not":

```
Do ~em[not] do that.
```

Here is the same phrase, but where "not" receives stronger emphasis (presumably boldface in a printed version):

```
Do ~st[not] do that.
```

Here is a passage that is set off as a display, in a fixed-width font:

```
~bv[]
This passage has been set off as ``verbatim''.
The present line starts just after a line break.  Normally, printed
text is formatted, but inside ~bv[]...~ev[], line breaks are taken
literally.
~ev[]
```

In general, the idea is to provide a "markup language" that can be reasonably interpreted not only at the terminal (via `:doc^`), but also via translators into other languages. In fact, translators have been written into Texinfo and HTML.

Let us turn to a more systematic consideration of how to mark text in documentation^ strings using expressions of the form `~key[arg]`, which we will call "doc-string^ tilde directives." The idea is that `key` informs the documentation^ printer (which could be the terminal, a hardcopy printer, or some hypertext tool) about the "style" used to display `arg`. The intention is that each such printer should do the best it can. For example, we have seen above that `~em[arg]` tells the printer to *emphasize* `arg` if possible, using an appropriate display to indicate emphasis (italics, or perhaps surrounding `arg` with some character like `_`, or ...). For another example, the directive for bold font, `~b[arg]`, says that printed text for `arg` should be in bold if possible, but if there is no bold font available (such as at the terminal), then the argument should be printed in some other reasonable manner (for example, as ordinary text). The `key` part is case-insensitive; for example, you can use ~BV[] or ~Bv[] or ~bV[] in place of ~bv[].

Every form below may have any string as the argument (inside `[..]`), as long as it does not contain a newline (more on that below). However, when an argument does not make much sense to us, we show it below as the empty string, e.g., "`~bv[]`" rather than "`~bv[arg]`".

```
~-[]       Print the equivalent of a dash

~b[arg]    Print the argument in bold font, if available

~bid[arg]   ``Begin implementation dependent'' -- Ignores argument at
           terminal.

~bf[]      Begin formatted text (respecting spaces and line breaks),
           but in ordinary font (rather than, say, fixed-width font)
           if possible

~bq[]      Begin quotation (indented text, if possible)

~bv[]      Begin verbatim (print in fixed-width font, respecting
           spaces and line breaks)

~c[arg]    Print arg as ``code'', such as in a fixed-width font

~ef[]      End format; balances ~bf[]

~eid[arg]   ``End implementation dependent'' -- Ignores argument at
           terminal.

~em[arg]   Emphasize arg, perhaps using italics

~eq[]      End quotation; balances ~bq[]

~ev[]      End verbatim; balances ~bv[]

~i[arg]    Print arg in italics font

~id[arg]    ``Implementation dependent'' -- Ignores argument at
           terminal.
```

```
~il[arg]  Print argument as is, but make it a link (for true
          hypertext environments)

~ilc[arg] Same as ~il[arg], except that arg should be printed as
          with ~c[arg]

~l[arg]   Ordinary link; prints as ``See :DOC arg'' at the terminal
          (but also see ~pl below, which puts ``see'' in lower case)

~nl[]     Print a newline

~par[]    Paragraph mark, of no significance at the terminal
          (can be safely ignored; see also notes below)

~pl[arg]  Parenthetical link (borrowing from Texinfo):  same as
          ~l[arg], except that ``see'' is in lower case.  This is
          typically used at other than the beginning of a sentence.

~sc[arg]  Print arg in (small, if possible) capital letters

~st[arg]  Strongly emphasize arg, perhaps using a bold font

~t[arg]   Typewriter font; similar to ~c[arg], but leaves less
          doubt about the font that will be used.

~terminal[arg]  Terminal only; arg is to be ignored except when
          reading documentation at the terminal, using :DOC.
```

Style notes and further details

It is not a good idea to put doc-string^ tilde directives inside verbatim environments, `~bv[] ... ~ev[]`.

Do not nest doc-string^ tilde directives; that is, do not write

```
The ~c[~il[append] function ...
```

but note that the "equivalent" expression

```
The ~ilc[append] function ...
```

is fine. The following phrase is also acceptable:

```
~bf[]This is
~em[formatted] text.
~ef[]
```

because the nesting is only conceptual, not literal.

We recommend that for displayed text, `~bv[]` and `~ev[]` should usually each be on lines by themselves. That way, printed text may be less encumbered with excessive blank lines. Here is an example.

```
Here is some normal text.  Now start a display:
~bv[]
  2 + 2 = 4
~ev[]
And here is the end of that paragraph.

Here is the start of the next paragraph.
```

The analogous consideration applies to `~bf[]` and `~ef[]` as well as `~bq[]` and `~eq[]`.

You may "quote" characters^ inside the `arg` part of `~key[arg]`, by preceding them with `~`. This is, in fact, the only legal way to use a newline character or a right bracket (]) inside the argument to a doc-string^ tilde directive.

Write your documentation^ strings without hyphens. Otherwise, you may find your text printed on paper (via TeX, for example) like this —

```
Here is a hyphe- nated word.
```

even if what you had in mind was:

```
Here is a hyphe-
nated word.
```

When you want to use a dash (as opposed to a hyphen), consider using ~-[], which is intended to be interpreted as a "dash." For example:

```
This sentence ~-[] which is broken with dashes ~-[] is boring.
```

would be written to the terminal (using `:doc^`) by replacing `~-[]` with two hyphen characters^, but would presumably be printed on paper with a dash.

Be careful to balance the “begin” and “end” pairs, such as `~bv[]` and `~ev[]`. Also, do not use two “begin” directives (`~bf[]`, `~bq[]`, or `~bv[]`) without an intervening “end” directive. It is permissible (and perhaps this is not surprising) to use the doc-string^ part separator `~/` while between such a begin-end pair.

Because of a bug in texinfo (as of this writing), you may wish to avoid beginning a line with (any number of spaces followed by) the `-^` character or `~-[]`.

The “paragraph” directive, `~par[]`, is rarely if ever used. There is a low-level capability, not presently documented, that interprets two successive newlines as though they were `~par[]`. This is useful for the HTML driver. For further details, see the authors of ACL2.

Emacs code is available for manipulating documentation^ strings that contain doc-string^ tilde-directives (for example, for doing a reasonable job filling such documentation^ strings). See the authors if you are interested.

We tend to use `~em[arg]` for “section headers,” such as “Style notes and further details” above. We tend to use `~st[arg]` for emphasis of words inside text. This division seems to work well for our Texinfo driver. Note that `~st[arg]` causes `arg` to be printed in upper-case at the terminal (using `:doc^`), while `~em[arg]` causes `arg` to be printed at the terminal as though `arg` were not marked for emphasis.

Our Texinfo and HTML drivers both take advantage of capabilities for indicating which characters^ need to be “escaped,” and how. Unless you intend to write your own driver, you probably do not need to know more about this issue; otherwise, contact the ACL2 authors. We should probably mention, however, that Texinfo makes the following requirement: when using `~l[arg]`, where `arg` contains one of the special characters^ `@^`, `{`, or `}`, you must immediately follow this use with a period or comma. Also, the Emacs “info” documentation^ that we generate by using our Texinfo driver has the property that in node names, `:` has been replaced by `|` (because of quirks in info); so for example, the “proof-checker^” simplification command, `s`, is documented under `acl2-pc||s` rather than under `acl2-pc::s`.

We have tried to keep this markup language fairly simple; in particular, there is no way to refer to a link by other than the actual name. So for example, when we want to make `:doc^` an invisible link in “code” font, we write the following form, which indicates that `:` should be in that font and then `doc^` should both be in that font and be an invisible link.

```
~c[:]~ilc[doc]
```

5.9 MORE

MORE   your response to `:doc^` or `:more-doc^`'s "`(cont'd)`"

NOTE: The command `:more` only makes sense at the terminal.

```
Example:
ACL2 !>:more
```

will continue printing whatever documentation^ was started by `:doc^` or `:more-doc^`.

When you type `:doc name`, for some documented `name`, the system responds by typing the one-liner and the notes sections of the documentation^ for `name`. It then types "`(cont'd)`" which stands for "continued". If you then type

```
ACL2 !>:more
```

the system will start to print the details section of `name`. The same thing could be achieved by typing `:more-doc name`, but that requires you to type name again.

Similarly, if you have typed `:more-doc^` name, the system will print the first "block" of the details section and then print "`(cont'd)`". Typing `:more` at that point will cause the next block of the details section to be printed. Eventually `:more` will conclude by printing "`*-`" which is the indicator that the text has been exhausted.

What is a "block" of text? `:More` looks for the end of a paragraph (two adjacent newlines) after printing `n` lines. If it doesn't find one before it has printed `k` lines, it just stops there. `N` and `k` here are the values of the two state^ global variables `'more-doc-min-lines` and `'more-doc-max-lines`. You may use `@^` and `assign^` to inspect and set these variables, e.g., `(@ more-doc-max-lines)` will return the current maximum number of lines printed by `:more` and `(assign more-doc-max-lines 19)` will set it to 19. On terminals having only 24 lines, we find min and max settings of 12 and 19 the most pleasant.

If you want `:more` to print all of the details instead of feeding them to you one block at a time, type `:more!^` instead.

5.10 MORE!

MORE! another response to "(cont'd)"

NOTE: The command `:more!` only makes sense at the terminal.

```
Example:
ACL2 !>:more!
```

will print all of the remaining documentation^ started by the last `:doc^` or `:more-doc^`.

See Section 5.9 [MORE], page 146 for some background. Typing `:more!` will print all remaining blocks of documentation^.

`:More!` is like `:more^` except that it prints all the text at once. For example, if you type `:doc^` name you will see some text followed by "`(cont'd)`". If you then type simply `:more!` you will see all of the details, while if you type `:more^` you will be fed the next block of details.

5.11 MORE-DOC

MORE-DOC a continuation of the `:doc^` documentation^

NOTE: The `:more-doc` command only makes sense at the terminal.

```
Examples:
ACL2 !>:more-doc DEFTHM
ACL2 !>:more-doc logical-name
```

Often it is assumed in the text provided by `:more-doc` name that you have read the text provided by `:doc name`.

`:More-doc` just continues spewing out at you the documentation^ string provided with a definition. If the user has done his job, `:doc^` will probably remind you of the basics and `:more-doc`, if read after `:doc^`, will address obscure details that are nevertheless worth noting.

When `:more-doc` types "`(cont'd)`" you can get the next block of the continuation by typing `:more^` or all of the remaining blocks by typing `:more!^`. See Section 5.9 [MORE], page 146.

5.12 NQTHM-TO-ACL2

NQTHM-TO-ACL2 ACL2 analogues of Nqthm functions and commands

```
Example Forms:
:nqthm-to-acl2 prove-lemma   ; Display ACL2 topic(s) and/or print
                             ; information corresponding to Nqthm
                             ; PROVE-LEMMA command.
(nqthm-to-acl2 'prove-lemma) ; Same as above.

General Form:
(nqthm-to-acl2 name)
```

where `name` is a notion documented for Nqthm: either a function in the Nqthm logic, or a command. If there is corresponding information available for ACL2, it will be printed in response to this command. This information is not intended to be completely precise, but rather, is intended to help those familiar with Nqthm to make the transition to ACL2.

We close with two tables that contain all the information used by this `nqthm-to-acl2` command. The first shows the correspondence between functions in the Nqthm logic and corresponding ACL2 functions (when possible); the second is similar, but for commands rather than functions.

```
Nqthm functions  -->     ACL2
----------------------------------------
ADD1            -->  1+
ADD-TO-SET      -->  ADD-TO-SET-EQUAL and ADD-TO-SET-EQ
AND             -->  AND
APPEND          -->  APPEND and BINARY-APPEND
APPLY-SUBR      -->  No correspondent, but see the documentation for
                     DEFEVALUATOR and META.
APPLY$          -->  No correspondent, but see the documentation for
                     DEFEVALUATOR and META.
ASSOC           -->  ASSOC-EQUAL, ASSOC and ASSOC-EQ
BODY            -->  No correspondent, but see the documentation for
                     DEFEVALUATOR and META.
CAR             -->  CAR
CDR             -->  CDR
CONS            -->  CONS
COUNT           -->  ACL2-COUNT
DIFFERENCE      -->  -
EQUAL           -->  EQUAL, EQ, EQL and =
EVAL$           -->  No correspondent, but see the documentation for
                     DEFEVALUATOR and META.
FALSE           -->  Nqthm's F corresponds to the ACL2 symbol NIL.
FALSEP          -->  NOT and NULL
```

```
FORMALS       -->  No correspondent, but see the documentation for
                   DEFEVALUATOR and META.
GEQ           -->  >=
GREATERP      -->  >
IDENTITY      -->  IDENTITY
IF            -->  IF
IFF           -->  IFF
IMPLIES       -->  IMPLIES
LEQ           -->  <=
LESSP         -->  <
LISTP         -->  CONSP
LITATOM       -->  SYMBOLP
MAX           -->  MAX
MEMBER        -->  MEMBER-EQUAL, MEMBER and MEMBER-EQ
MINUS         -->  - and UNARY--
NEGATIVEP     -->  MINUSP
NEGATIVE-GUTS -->  ABS
NLISTP        -->  ATOM
NOT           -->  NOT
NUMBERP       -->  ACL2-NUMBERP, INTEGERP and RATIONALP
OR            -->  OR
ORDINALP      -->  E0-ORDINALP
ORD-LESSP     -->  E0-ORD-<
PACK          -->  See intern and coerce.
PAIRLIST      -->  PAIRLIS$
PLUS          -->  + and BINARY-+
QUOTIENT      -->  /
REMAINDER     -->  REM and MOD
STRIP-CARS    -->  STRIP-CARS
SUB1          -->  1-
TIMES         -->  * and BINARY-*
TRUE          -->  The symbol T.
UNION         -->  UNION-EQUAL and UNION-EQ
UNPACK        -->  See symbol-name and coerce.
V&C$          -->  No correspondent, but see the documentation for
                   DEFEVALUATOR and META.
V&C-APPLY$    -->  No correspondent, but see the documentation for
                   DEFEVALUATOR and META.
ZERO          -->  The number 0.
ZEROP         -->  ZEROP

========================================

Nqthm commands   -->     ACL2
----------------------------------------
ACCUMULATED-PERSISTENCE
              -->  ACCUMULATED-PERSISTENCE
ADD-AXIOM     -->  DEFAXIOM
```

```
ADD-SHELL     -->  There is no shell principle in ACL2.
AXIOM         -->  DEFAXIOM
BACKQUOTE-SETTING
              -->  Backquote is supported in ACL2, but not
                   currently documented.
BOOT-STRAP    -->  GROUND-ZERO
BREAK-LEMMA   -->  MONITOR
BREAK-REWRITE -->  BREAK-REWRITE
CH            -->  PBT
                   See also :DOC history.
CHRONOLOGY    -->  PBT
                   See also :DOC history.
COMMENT       -->  DEFLABEL
COMPILE-UNCOMPILED-DEFNS
              -->  COMP
CONSTRAIN     -->  See :DOC encapsulate and :DOC local.
DATA-BASE     -->  Perhaps the closest ACL2 analogue of DATA-BASE
                   is PROPS.  But see :DOC history for a collection
                   of commands for querying the ACL2 database
                   (``world'').  Note that the notions of
                   supporters and dependents are not supported in
                   ACL2.
DCL           -->  DEFSTUB
DEFN          -->  DEFUN and DEFMACRO
DEFTHEORY     -->  DEFTHEORY
DISABLE       -->  DISABLE
DISABLE-THEORY
              -->  See :DOC theories.  The Nqthm command
                   (DISABLE-THEORY FOO) corresponds roughly to the
                   ACL2 command
                   (in-theory (set-difference-theories
                                (current-theory :here)
                                (theory 'foo))).
DO-EVENTS     -->  LD
DO-FILE       -->  LD
ELIM          -->  ELIM
ENABLE        -->  ENABLE
ENABLE-THEORY -->  See :DOC theories.  The Nqthm command
                   (ENABLE-THEORY FOO) corresponds roughly to the
                   ACL2 command
                   (in-theory (union-theories
                                (theory 'foo)
                                (current-theory :here))).
EVENTS-SINCE  -->  PBT
FUNCTIONALLY-INSTANTIATE
              -->  ACL2 provides a form of the :USE hint that
                   corresponds roughly to the
                   FUNCTIONALLY-INSTANTIATE event of Nqthm. See
```

```
                     :DOC lemma-instance.
GENERALIZE    -->  GENERALIZE
HINTS         -->  HINTS
LEMMA         -->  DEFTHM
MAINTAIN-REWRITE-PATH
              -->  BRR
MAKE-LIB      -->  There is no direct analogue of Nqthm's notion of
                   ``library.''  See :DOC books for a description
                   of ACL2's mechanism for creating and saving
                   collections of events.
META          -->  META
NAMES         -->  NAME
NOTE-LIB      -->  INCLUDE-BOOK
PPE           -->  PE
PROVE         -->  THM
PROVEALL      -->  See :DOC ld and :DOC certify-book.  The latter
                   corresponds to Nqthm's PROVE-FILE,which may be
                   what you're interested in,really.
PROVE-FILE    -->  CERTIFY-BOOK
PROVE-FILE-OUT
              -->  CERTIFY-BOOK
PROVE-LEMMA   -->  DEFTHM
                   See also :DOC hints.
R-LOOP        -->  The top-level ACL2 loop is an evaluation loop as
                   well, so no analogue of R-LOOP is necessary.
REWRITE       -->  REWRITE
RULE-CLASSES  -->  RULE-CLASSES
SET-STATUS    -->  IN-THEORY
SKIM-FILE     -->  LD-SKIP-PROOFSP
TOGGLE        -->  IN-THEORY
TOGGLE-DEFINED-FUNCTIONS
              -->  EXECUTABLE-COUNTERPART-THEORY
TRANSLATE     -->  TRANS and TRANS1
UBT           -->  UBT and U
UNBREAK-LEMMA -->  UNMONITOR
UNDO-BACK-THROUGH
              -->  UBT
UNDO-NAME     -->  See :DOC ubt.  There is no way to undo names in
                   ACL2 without undoing back through such names.
                   However, see :DOC ld-skip-proofsp for
                   information about how to quickly recover the
                   state.
```

6 EVENTS

EVENTS functions that extend the logic

* ADD-MACRO-ALIAS:: associate a function name with a macro name
* DEFAXIOM:: add an axiom
* DEFCHOOSE:: define a Skolem (witnessing) function
* DEFCONG:: prove that one equivalence^ relation preserves another in a given argument position of a given function
* DEFCONST:: define a constant
* DEFDOC:: add a documentation^ topic
* DEFEQUIV:: prove that a function is an equivalence^ relation
* DEFEVALUATOR:: introduce an evaluator function
* DEFLABEL:: build a landmark and/or add a documentation^ topic
* DEFMACRO:: define a macro
* DEFPKG:: define a new symbol package
* DEFREFINEMENT:: prove that `equiv1` refines `equiv2`
* DEFSTUB:: stub-out a function symbol
* DEFTHEORY:: define a theory (to enable^ or disable^ a set of rules)
* DEFTHM:: prove and name a theorem
* DEFUN:: define a function symbol
* DEFUN-SK:: define a function whose body has an outermost quantifier
* ENCAPSULATE:: constrain some functions and/or hide some events^
* IN-THEORY:: designate "current" theory (enabling its rules)
* INCLUDE-BOOK:: load the events^ in a file
* LOCAL:: hiding an event in an encapsulation or book
* LOGIC:: to set the default defun-mode^ to `:logic`
* MACRO-ALIASES-TABLE:: a table^ used to associate function names with macro names
* MUTUAL-RECURSION:: define some mutually recursive functions
* PROGRAM:: to set the default defun-mode^ to `:program`
* REMOVE-MACRO-ALIAS:: remove the association of a function name with a macro name
* SET-COMPILE-FNS:: have each function compiled as you go along.
* SET-IGNORE-OK:: allow unused formals and locals without an `ignore` declaration
* SET-INHIBIT-WARNINGS:: control warnings
* SET-INVISIBLE-FNS-ALIST:: set the invisible functions alist
* SET-IRRELEVANT-FORMALS-OK:: allow irrelevant formals in definitions
* SET-MEASURE-FUNCTION:: set the default measure function symbol
* SET-VERIFY-GUARDS-EAGERNESS:: the eagerness with which guard^ verification is tried.
* SET-WELL-FOUNDED-RELATION:: set the default well-founded relation
* TABLE:: user-managed tables
* TERM-TABLE:: a table used to validate meta rules
* THEORY-INVARIANT:: user-specified invariants on theories^
* VERIFY-GUARDS:: verify the guard^s of a function
* VERIFY-TERMINATION:: convert a function from :program mode to :logic mode

Any extension of the syntax of ACL2 (i.e., the definition of a new constant or macro), the axioms (i.e., the definition of a function), or the rule data base (i.e., the proof of a theorem), constitutes a logical "event." Events change the ACL2 logical world (see Section 8.89 [WORLD], page 362). Indeed, the only way to change the ACL2 world^ is via the successful evaluation of an event function. Every time the world^ is changed by an event, a landmark is left on the world^ and it is thus possible to identify the world^ "as of" the evaluation of a given event. An event may introduce new logical names. Some events introduce no new names (e.g., `verify-guards^`), some introduce exactly one (e.g., `defmacro^` and `defthm^`), and some may introduce many (e.g., `encapsulate^`).

6.1 ADD-MACRO-ALIAS

ADD-MACRO-ALIAS associate a function name with a macro name

```
Example:
(add-macro-alias append binary-append)
```

This example associates the function symbol `binary-append^` with the macro name `append^`. As a result, the name `append^` may be used as a runic designator (see Chapter 15 [THEORIES], page 667) by the various theory functions. See Section 6.23 [MACRO-ALIASES-TABLE], page 183 for more details.

```
General Form:
(add-macro-alias macro-name function-name)
```

This is a convenient way to add an entry to `macro-aliases-table^`. See Section 6.23 [MACRO-ALIASES-TABLE], page 183 and also see Section 6.26 [REMOVE-MACRO-ALIAS], page 187.

6.2 DEFAXIOM

DEFAXIOM add an axiom

WARNING: We strongly recommend that you not add axioms. If at all possible you should use `defun^` or `mutual-recursion^` to define new concepts recursively or use `encapsulate^` to constrain them constructively. Adding new axioms frequently renders the logic inconsistent.

```
Example:
(defaxiom sbar (equal t f)
```

```
            :rule-classes nil
            :doc ":Doc-Section ...")

General Form:
(defaxiom name term
          :rule-classes rule-classes
          :doc          doc-string)
```

where `name` is a new symbolic name (see Section 8.65 [NAME], page 329), `term` is a term intended to be a new axiom, and `rule-classes^` and `doc-string^` are as described in the corresponding documentation^ topics . The two keyword arguments are optional. If `:rule-classes^` is not supplied, the list `(:rewrite)` is used; if you wish the axiom to generate no rules, specify `:rule-classes^ nil`.

6.3 DEFCHOOSE

DEFCHOOSE define a Skolem (witnessing) function

```
Examples:
(defchoose choose-x-for-p-and-q (x) (y z)
  (and (p x y z)
       (q x y z)))

(defchoose choose-x-for-p-and-q x (y z) ; equivalent to the above
  (and (p x y z)
       (q x y z)))

(defchoose choose-x-and-y-for-p-and-q (x y) (z)
  (and (p x y z)
       (q x y z)))

General Form:
(defchoose fn
           (bound-var1 ... bound-varn)
           (free-var1 ... free-vark)
           doc-string ; optional
           body),
```

where `fn` is the symbol you wish to define and is a new symbolic name (see Section 8.65 [NAME], page 329), `(bound-var1 ... bound-varn)` is a list of distinct 'bound' variables (see below), `(free-var1 ... free-vark)` is the list of formal parameters of `fn` and is disjoint from the bound variables, and `body` is a term. The use of `lambda-list` keywords (such as `&optional`) is not allowed. The documentation^ string, `doc-string^`, is optional; for a description of its form, see Section 5.5 [DOC-STRING], page 134.

In the most common case, where there is only one bound variable, it is permissible to omit the enclosing parentheses on that variable. The effect is the same whether or not those parentheses are omitted. We describe this case first, where there is only one bound variable, and address the other case at the end.

The effect of the form

```
(defchoose fn bound-var (free-var1 ... free-vark)
  body)
```

is to introduce a new function symbol, `fn`, with formal parameters `(free-var1 ... free-vark)`, and the following axiom stating that `fn` picks a value of `bound-var` so that the body will be true, if such a value exists:

```
(implies body
         (let ((bound-var (fn free-var1 ... free-vark)))
           body))
```

This axiom is "clearly" conservative under the conditions expressed above: the function `fn` merely picks out a "witnessing" value of `bound-var` if there is one. The system in fact treats `fn` very much as though it were declared in the signature^ of an `encapsulate^` event, with the axiom above as the only axiom exported.

If there is more than one bound variable, as in

```
(defchoose fn
           (bound-var1 ... bound-varn)
           (free-var1 ... free-vark)
           body)
```

then `fn` returns `n` multiple values, and the defining axiom is expressed using `mv-let^` (see Section 10.171 [MV-LET], page 472) as follows:

```
(implies body
         (mv-let (bound-var1 ... bound-varn)
                 (fn free-var1 ... free-vark)
                 body))
```

It is illegal for the body to return multiple values. Also, note that the function introduced by a `defchoose` form will return 'state' if and only if the body returns 'state'; see Section 8.81 [STATE], page 344.

`Defchoose` is only executed in defun-mode^ `:logic^`; see Section 8.18 [DEFUN-MODE], page 256.

Also see Section 6.17 [DEFUN-SK], page 173.

Comment for logicians: As we point out in the documentation for `defun-sk^`, `defchoose` is "appropriate," by which we mean that it is conservative, even in the presence of `epsilon-0` induction. See Section 8.7 [BIBLIOGRAPHY], page 242 for a reference to the "story" that includes this argument.

6.4 DEFCONG

DEFCONG prove that one equivalence^ relation preserves another in a given argument position of a given function

```
Example:
(defcong set-equal iff (memb x y) 2)

is an abbreviation for
(defthm set-equal-implies-iff-memb-2
  (implies (set-equal y y-equiv)
           (iff (memb x y) (memb x y-equiv)))
  :rule-classes (:congruence))
```

See Section 14.3 [CONGRUENCE], page 627 and also see Section 14.6 [EQUIVALENCE], page 632.

```
General Form:
(defcong equiv1 equiv2 term k
  :rule-classes rule-classes
  :instructions instructions
  :hints hints
  :otf-flg otf-flg
  :event-name event-name
  :doc doc)
```

where `equiv1` and `equiv2` are known equivalence^ relations, `term` is a call of a function `fn` on the correct number of distinct variable arguments `(fn x1 ... xn)`, `k` is a positive integer less than or equal to the arity of `fn`, and other arguments are as specified in the documentation for `defthm^`.

The `defcong` macro expands into a call of `defthm^`. The name of the `defthm^` event is `equiv1-implies-equiv2-fn-k` unless an `:event-name` keyword argument is supplied for the name. The term of the theorem is

```
(implies (equiv1 xk yk)
         (equiv2 (fn x1... xk ...xn)
                 (fn x1... yk ...xn))).
```

The rule-class `:congruence^` is added to the `rule-classes^` specified, if it is not already there. All other arguments to the generated `defthm^` form are as specified by the keyword arguments above.

6.5 DEFCONST

DEFCONST define a constant

```
Examples:
(defconst *digits* '(0 1 2 3 4 5 6 7 8 9))
(defconst *n-digits* (the unsigned-byte (length *digits*)))

General Form:
(defconst name term doc-string)
```

where `name` is a symbol beginning and ending with the character `*`, `term` is a variable-free term that is evaluated to determine the value of the constant, and `doc-string^` is an optional documentation^ string (see Section 5.5 [DOC-STRING], page 134).

When a constant symbol is used as a term^, ACL2 replaces it by its value; see Section 8.85 [TERM], page 352.

It may be of interest to note that `defconst` is implemented at the lisp level using `defparameter`, as opposed to `defconstant`. (Implementation note: this is important for proper support of undoing and redefinition.)

6.6 DEFDOC

DEFDOC add a documentation^ topic

```
Examples:
(defdoc interp-section
   ":Doc-Section ...")

General Form:
(defdoc name doc-string)
```

where `name` is a symbol or string to be documented and `doc-string^` is a documentation^ string (see Section 5.5 [DOC-STRING], page 134). This event adds the documentation^ string for symbol `name` to the `:doc^` data base. It may also be used to change the documentation^ for `name` if `name` already has documentation^. The difference between this event and `deflabel^` is that, unlike `deflabel^` (but like `table^`), it does not mark the current history^ with the `name`. But like `deflabel^`, `defdoc` events^ are never considered redundant (see Section 8.76 [REDUNDANT-EVENTS], page 337).

See Section 6.9 [DEFLABEL], page 162 for a means of attaching a documentation^ string to a name that marks the current history^ with that name. We now elaborate further on how `defdoc` may be useful in place of `deflabel^`.

It is usually sufficient to use `deflabel^` when you might be tempted to use `defdoc`. However, unlike `deflabel^`, `defdoc` does not mark the current history^ with `name`. Thus, `defdoc` is useful for introducing the documentation^ for a `defun^` or `deftheory^` event, for example, several events^ before the function or theory is actually defined.

For example, suppose you want to define a theory (using `deftheory^`). You need to prove the lemmas in that theory before executing the `deftheory^` event. However, it is quite natural to define a `:doc-section` (see Section 5.5 [DOC-STRING], page 134) whose name is the name of the theory to be defined, and put the documentation^ for that theory's lemmas into that `:doc-section`. `Defdoc` is ideal for this purpose, since it can be used to introduce the `:doc-section`, followed by the lemmas referring to that `:doc-section`, and finally concluded with a `deftheory^` event of the same name. If `deflabel^` were used instead of `defdoc`, for example, then the `deftheory^` event would be disallowed because the name is already in use by the `deflabel^` event.

We also imagine that some users will want to use `defdoc` to insert the documentation^ for a function under development. This `defdoc` event would be followed by definitions of all the subroutines of that function, followed in turn by the function definition itself.

Any time `defdoc` is used to attach documentation^ to an already-documented name, the name must not be attached to a new `:doc-section`. We make this requirement as a way of avoiding loops in the documentation^ tree. When documentation^ is redefined, a warning will be printed to the terminal.

6.7 DEFEQUIV

DEFEQUIV prove that a function is an equivalence^ relation

```
Example:
(defequiv set-equal)

is an abbreviation for
(defthm set-equal-is-an-equivalence
  (and (booleanp (set-equal x y))
       (set-equal x x)
       (implies (set-equal x y) (set-equal y x))
       (implies (and (set-equal x y)
                     (set-equal y z))
                (set-equal x z)))
  :rule-classes (:equivalence))
```

See Section 14.6 [EQUIVALENCE], page 632.

```
General Form:
(defequiv fn
  :rule-classes rule-classes
  :instructions instructions
  :hints hints
  :otf-flg otf-flg
  :event-name event-name
  :doc doc)
```

where `fn` is a function symbol of arity 2, `event-name`, if supplied, is a symbol, and all other arguments are as specified in the documentation for `defthm^`. The `defequiv` macro expands into a call of `defthm`. The name of the `defthm` is `fn-is-an-equivalence`, unless `event-name` is supplied, in which case `event-name` is the name used. The term generated for the `defthm` event states that `fn` is Boolean, reflexive, symmetric, and transitive. The rule-class `:equivalence` is added to the rule-classes^ specified, if it is not already there. All other arguments to the generated `defthm` form are as specified by the other keyword arguments above.

6.8 DEFEVALUATOR

DEFEVALUATOR introduce an evaluator function

```
Example:
(defevaluator evl evl-list
```

```
  ((length x) (member x y)))
```

See Section 14.12 [META], page 652.

```
General Form:
(defevaluator ev ev-list
   ((g1 x1 ... xn_1)
    ...
    (gk x1 ... xn_k))
```

where `ev` and `ev`-list are new function symbols and `g1`, ..., `gk` are old function symbols with the indicated number of formals, i.e., each `gi` has `n_i` formals.

This function provides a convenient way to constrain `ev` and `ev-list` to be mutually-recursive evaluator functions for the symbols `g1`, ..., `gk`. Roughly speaking, an evaluator function for a fixed, finite set of function symbols is a restriction of the universal evaluator to terms composed of variables, constants, lambda expressions, and applications of the given functions. However, evaluator functions are constrained rather than defined, so that the proof that a given metafunction is correct vis-a-vis a particular evaluator function can be lifted (by functional instantiation) to a proof that it is correct for any larger evaluator function. See Section 14.12 [META], page 652 for a discussion of metafunctions.

`Defevaluator` executes an `encapsulate^` after generating the appropriate `defun^` and `defthm^` events. Perhaps the easiest way to understand what `defevaluator` does is to execute the keyword command

```
:trans1 (defevaluator evl evl-list ((length x) (member x y)))
```

and inspect the output.

Formally, `ev` is said to be an "evaluator function for `g1`, ..., `gk`, with mutually-recursive counterpart `ev-list`" iff `ev` and `ev-list` are constrained functions satisfying just the constraint^s discussed below.

`Ev` and `ev-list` must satisfy constraint^s (1)-(4) and (k):

```
(1) How to ev a variable symbol:
    (implies (symbolp x)
             (equal (ev x a) (cdr (assoc-eq x a))))
```

```
(2) How to ev a constant:
    (implies (and (consp x)
                  (equal (car x) 'quote))
             (equal (ev x a) (cadr x)))

(3) How to ev a lambda application:
    (implies (and (consp x)
                  (consp (car x)))
             (equal (ev x a)
                    (ev (caddar x)
                        (pairlis$ (cadar x)
                                  (ev-list (cdr x) a)))))

(4) How to ev an argument list:
    (implies (consp x-lst)
             (equal (ev-list x-lst a)
                    (cons (ev (car x-lst) a)
                          (ev-list (cdr x-lst) a))))
    (implies (not (consp x-lst))
             (equal (ev-list x-lst a)
                    nil))

(k) For each i from 1 to k, how to ev an application of gi,
    where gi is a function symbol of n arguments:
    (implies (and (consp x)
                  (equal (car x) 'gi))
             (equal (ev x a)
                    (gi (ev x1 a)
                        ...
                        (ev xn a)))),
    where xi is the (cad...dr x) expression equivalent to (nth i x).
```

`Defevaluator` defines suitable witnesses for `ev` and `ev-list`, proves the theorems about them, and constrains `ev` and `ev-list` appropriately. We expect `defevaluator` to work without assistance from you, though the proofs do take some time and generate a lot of output. The proofs are done in the context of a fixed theory, namely the result of applying `union-theories^` to two lists: the function symbols supplied, and the value of the constant `*defevaluator-form-base-theory*`.

6.9 DEFLABEL

DEFLABEL build a landmark and/or add a documentation^ topic

```
Examples:
(deflabel interp-section
```

```
   :doc
   ":Doc-Section ...")

General Form:
(deflabel name :doc doc-string)
```

where `name` is a new symbolic name (see Section 8.65 [NAME], page 329) and `doc-string^` is an optional documentation^ string (see Section 5.5 [DOC-STRING], page 134). This event adds the documentation^ string for symbol `name` to the `:doc^` data base. By virtue of the fact that `deflabel` is an event, it also marks the current history^ with the `name`. Thus, even undocumented labels are convenient as landmarks in a proof development. For example, you may wish to undo back through some label or compute a theory expression (see Chapter 15 [THEORIES], page 667) in terms of some labels. `Deflabel` events^ are never considered redundant. See Section 8.76 [REDUNDANT-EVENTS], page 337.

See Section 6.6 [DEFDOC], page 158 for a means of attaching a documentation^ string to a name without marking the current history^ with that name.

6.10 DEFMACRO

DEFMACRO define a macro

```
Example Defmacros:
(defmacro xor (x y)
  (list 'if x (list 'not y) y))

(defmacro git (sym key)
  (list 'getprop sym key nil
        ''current-acl2-world
        '(w state)))

(defmacro one-of (x &rest rst)
  (declare (xargs :guard (symbol-listp rst)))
  (cond ((null rst) nil)
        (t (list 'or
                 (list 'eq x (list 'quote (car rst)))
                 (list* 'one-of x (cdr rst))))))

Example Expansions:
term                    macroexpansion

(xor a b)              (if a (not b) b)
(xor a (foo b))        (if a (not (foo b)) (foo b))
```

```
(git 'car 'lemmas)     (getprop 'car 'lemmas nil
                                'current-acl2-world
                                (w state))

(one-of x a b c)       (or (eq x 'a)
                           (or (eq x 'b)
                               (or (eq x 'c) nil)))

(one-of x 1 2 3)       ill-formed (guard violation)

General Form:
(defmacro name macro-args doc-string dcl ... dcl body)
```

where `name` is a new symbolic name (see Section 8.65 [NAME], page 329), `macro-args^` specifies the formals of the macro (see Section 8.64 [MACRO-ARGS], page 328 for a description), and `body` is a term. `Doc-string^` is an optional documentation^ string; see Section 5.5 [DOC-STRING], page 134. Each `dcl` is an optional declaration (see Section 10.94 [DECLARE], page 432) except that the only `xargs^` keyword permitted by `defmacro` is `:guard^`.

Macroexpansion occurs when a form is read in, i.e., before the evaluation or proof of that form is undertaken. To experiment with macroexpansion, see Section 9.16 [TRANS], page 390. When a form whose `car^` is `name` arises as the form is read in, the arguments are bound as described in CLTL pp. 60 and 145, the guard^ is checked, and then the `body` is evaluated. The result is used in place of the original form.

In ACL2, macros do not have access to `state^`. That is, `state^` is not allowed among the formal parameters. This is in part a reflection of CLTL, pp. 143, “More generally, an implementation of Common Lisp has great latitude in deciding exactly when to expand macro calls with a program. ... Macros should be written in such a way as to depend as little as possible on the execution environment to produce a correct expansion.” In ACL2, the product of macroexpansion is independent of the current environment and is determined entirely by the macro body and the functions and constants it references. It is possible, however, to define macros that produce expansions that refer to `state^` or other variables not among the macro's arguments. See the `git` example above.

6.11 DEFPKG

DEFPKG define a new symbol package

```
Example:
(defpkg "MY-PKG"
```

```
          (union-eq *acl2-exports*
                    *common-lisp-symbols-from-main-lisp-package*))

General Form:
(defpkg "name" term doc-string)
```

where "name" is a (case-sensitive) string that names the package to be created, term is a variable-free expression that evaluates to a list of symbols (no two of which have the same symbol-name^) to be imported into the newly created package, and doc-string^ is an optional documentation^ string; see Section 5.5 [DOC-STRING], page 134. The name of the new package must be "new": the host lisp must not contain any package of that name. There are two exceptions to this newness rule, discussed at the end of this documentation.

defpkg forms can be entered at the top-level of the ACL2 command^ loop. They should occur in a file only if the file is not to be compiled and contains nothing besides defpkg and in-package^ forms.

After a successful defpkg it is possible to "intern" a string into the package using intern-in-package-of-symbol^. The result is a symbol that is in the indicated package, provided the imports allow it. For example, suppose 'my-pkg::abc is a symbol whose symbol-package-name^ is "MY-PKG". Suppose further that the imports specified in the defpkg for "MY-PKG" do not include a symbol whose symbol-name^ is "XYZ". Then

```
(intern-in-package-of-symbol "XYZ" 'my-pkg::abc)
```

returns a symbol whose symbol-name^ is "XYZ" and whose symbol-package-name^ is "MY-PKG". On the other hand, if the imports to the defpkg does include a symbol with the name "XYZ", say in the package "LISP", then

```
(intern-in-package-of-symbol "XYZ" 'my-pkg::abc)
```

returns that symbol (which is uniquely determined by the restriction on the imports list above). See Section 10.132 [INTERN-IN-PACKAGE-OF-SYMBOL], page 454.

defpkg is the only means by which an ACL2 user can create a new package or specify what it imports. That is, ACL2 does not support the Common Lisp functions make-package or import. Currently, ACL2 does not support exporting at all.

The Common Lisp function intern^ is weakly supported by ACL2. See Section 10.131 [INTERN], page 453.

We now explain the two exceptions to the newness rule for package names. The careful experimenter will note that if a package is created with a `defpkg` that is subsequently undone, the host lisp system will contain the created package even after the undo. Because ACL2 hangs onto world^s after they have been undone, e.g., to implement `:oops^` but, more importantly, to implement error recovery, we cannot actually destroy a package upon undoing it. Thus, the first exception to the newness rule is that `name` is allowed to be the name of an existing package if that package was created by an undone `defpkg` and the newly proposed imports list is identical to the old one. See Section 8.68 [PACKAGE-REINCARNATION-IMPORT-RESTRICTIONS], page 330. This exception does not violate the spirit of the newness rule, since one is disinclined to believe in the existence of undone packages. The second exception is that `name` is allowed to be the name of an existing package if the package was created by a `defpkg` with identical imports. That is, it is permissible to execute "redundant" `defpkg` command^s. The redundancy test is based on the values of the two import forms, not on the forms themselves.

6.12 DEFREFINEMENT

DEFREFINEMENT prove that `equiv1` refines `equiv2`

```
Example:
(defrefinement equiv1 equiv2)

is an abbreviation for
(defthm equiv1-refines-equiv2
  (implies (equiv1 x y) (equiv2 x y))
  :rule-classes (:refinement))
```

See Section 14.13 [REFINEMENT], page 656.

```
General Form:
(defrefinement equiv1 equiv2
  :rule-classes rule-classes
  :instructions instructions
  :hints hints
  :otf-flg otf-flg
  :event-name event-name
  :doc doc)
```

where `equiv1` and `equiv2` are known equivalence^ relations, `event-name`, if supplied, is a symbol and all other arguments are as specified in the documentation for `defthm^`. The `defrefinement` macro expands into a call of `defthm`. The name supplied is `equiv1-refines-equiv2`, unless `event-name` is supplied, in which case it is used as the name. The term supplied states that `equiv1` refines `equiv2`. The rule-class `:refinement` is added to the `rule-classes` specified, if it is not already there. All other arguments to the generated `defthm` form are as specified by the other keyword arguments above.

6.13 DEFSTUB

DEFSTUB   stub-out a function symbol

```
Examples:
ACL2 !>(defstub subr1 (x y state) (mv t state))
ACL2 !>(defstub subr2 (x y) t)

General Forms:
(defstub name formals output)
(defstub name formals output :doc doc-string)
```

where `name` is the name of the function to be stubbed out, `formals` is its list of formal parameters, and `output` is either a symbol (indicating that the function returns one result) or a term of the form `(mv s1 ... sn)`, where each `si` is a symbol (indicating that the function returns `n` results). Whether and where the symbol `state^` occurs in `formals` and `output` indicates how the function handles state^. It should be the case that `(name formals output)` is in fact a signature (see Section 8.78 [SIGNATURE], page 340).

`Defstub` actually generates an `encapsulate^` event (see Section 6.18 [ENCAPSULATE], page 176). Thus, no axioms are available about `name` but it may be used wherever a function of its arity, multiplicity, and state^ characteristics is permitted.

6.14 DEFTHEORY

DEFTHEORY   define a theory (to enable^ or disable^ a set of rules)

```
Example:
(deftheory interp-theory
           (set-difference-theories
             (universal-theory :here)
             (universal-theory 'interp-section)))

General Form:
(deftheory name term :doc doc-string)
```

where `name` is a new symbolic name (see Section 8.65 [NAME], page 329), `term` is a term that when evaluated will produce a theory (see Chapter 15 [THEORIES], page 667), and `doc-string^` is an optional documentation^ string (see Section 5.5 [DOC-STRING], page 134). Except for the variable `world^`, `term` must contain no free variables. `term` is evaluated with `world^` bound to the current world (see Section 8.89 [WORLD], page 362) and the resulting theory is then converted to a *runic theory* (see Chapter 15 [THEORIES], page 667) and associated with `name`. Henceforth, this runic theory is returned as the value of the theory expression `(theory name)`.

6.15 DEFTHM

DEFTHM prove and name a theorem

```
Examples:
(defthm assoc-of-app
        (equal (app (app a b) c)
               (app a (app b c))))
```

The following nonsensical example illustrates all the optional arguments but is illegal because not all combinations are permitted. See Section 8.41 [HINTS], page 297 for a complete list of hints^.

```
(defthm main
        (implies (hyps x y z) (concl x y z))
       :rule-classes (:REWRITE :GENERALIZE)
       :instructions (induct prove promote (dive 1) x
                             (dive 2) = top (drop 2) prove)
       :hints (("Goal"
                :do-not '(generalize fertilize)
                :in-theory (set-difference-theories
                             (current-theory :here)
                             '(assoc))
                :induct (and (nth n a) (nth n b))
                :use ((:instance assoc-of-append
                                (x a) (y b) (z c))
                      (:functional-instance
                        (:instance p-f (x a) (y b))
                        (p consp)
                        (f assoc)))))
       :otf-flg t
       :doc "#0[one-liner/example/details]")

General Form:
(defthm name term
        :rule-classes rule-classes
```

```
:instructions instructions
:hints        hints
:otf-flg      otf-flg
:doc          doc-string)
```

where `name` is a new symbolic name (see Section 8.65 [NAME], page 329), `term` is a term alleged to be a theorem, and `rule-classes^`, `instructions^`, `hints^`, `otf-flg^` and `doc-string^` are as described in their respective documentation^. The five keyword arguments above are all optional, however you may not supply both `:instructions^` and `:hints^`, since one drives the proof checker and the other drives the theorem prover. If `:rule-classes^` is not specified, the list `(:rewrite)` is used; if you wish the theorem to generate no rules, specify `:rule-classes^ nil`.

When ACL2 processes a `defthm` event, it first tries to prove the term^ using the indicated hints (see Section 8.41 [HINTS], page 297) or instructions^ (see Chapter 11 [PROOF-CHECKER], page 515). If it is successful, it stores the rules described by the rule-classes (see Chapter 14 [RULE-CLASSES], page 617), proving the necessary corollaries.

6.16 DEFUN

DEFUN define a function symbol

```
Examples:
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
      y))

(defun fact (n)
  (declare (xargs :guard (and (integerp n)
                              (>= n 0))))
  (if (zp n)
      1
      (* n (fact (1- n)))))

General Form:
(defun fn (var1 ... varn) doc-string dcl ... dcl body),
```

where `fn` is the symbol you wish to define and is a new symbolic name (see Section 8.65 [NAME], page 329), `(var1 ... varn)` is its list of formal parameters (see Section 8.65 [NAME], page 329), and `body` is its body. The definitional is logically admissible provided certain restrictions are met. These are sketched below.

Note that ACL2 does not support the use of `lambda-list` keywords (such as `&optional`) in the formals list of functions. We do support some such keywords in macros and often you can achieve the desired syntax by defining a macro in addition to the general version of your function. See Section 6.10 [DEFMACRO], page 163. The documentation^ string, `doc-string^`, is optional; for a description of its form, see Section 5.5 [DOC-STRING], page 134.

The *declarations* (see Section 10.94 [DECLARE], page 432), `dcl`, are also optional. If multiple `dcl` forms appear, they are effectively grouped together as one. Perhaps the most commonly used ACL2 specific declaration is of the form `(declare (xargs :guard g :measure m))`. This declaration in the `defun` of some function `fn` has the effect of making the "guard^" for `fn` be the term `g` and the "measure" be the term `m`. The notion of "measure" is crucial to ACL2's definitional principle.

We now briefly discuss the ACL2 definitional principle, using the following definition form which is offered as a more or less generic example.

```
(defun fn (x y)
  (declare (xargs :guard (g x y)
                  :measure (m x y)))
  (if (test x y)
      (stop x y)
      (step (fn (d x) y))))
```

Note that in our generic example, `fn` has just two arguments, `x` and `y`, the guard^ and measure terms involve both of them, and the body is a simple case split on `(test x y)` leading to a "non-recursive" branch, `(stop x y)`, and a "recursive" branch. In the recursive branch, `fn` is called after "decrementing" `x` to `(d x)` and some step function is applied to the result. Of course, this generic example is quite specific in form but is intended to illustrate the more general case.

Provided this definition is admissible under the logic, as outlined below, it adds the following axiom to the logic.

```
Defining Axiom:
(fn x y)
  =
(if (test x y)
    (stop x y)
  (step (fn (d x) y)))
```

Note that the guard^ of `fn` has no bearing on this logical axiom.

This defining axiom is actually implemented in the ACL2 system by a `:definition^` rule, namely

```
(equal (fn x y)
       (if (test a b)
           (stop a b)
         (step (fn (d a) b)))).
```

See Section 14.4 [DEFINITION], page 628 for a discussion of how definition rules are applied. Roughly speaking, the rule causes certain instances of `(fn x y)` to be replaced by the corresponding instances of the body above. This is called "opening up" `(fn x y)`. The instances of `(fn x y)` opened are chosen primarily by heuristics which determine that the recursive calls of `fn` in the opened body (after simplification) are more desirable than the unopened call of `fn`.

This discussion has assumed that the definition of `fn` was admissible. Exactly what does that mean? First, `fn` must be a previously unaxiomatized function symbol (however, see Section 8.56 [LD-REDEFINITION-ACTION], page 314). Second, the formal parameters must be distinct variable names. Third, the guard^, measure, and body should all be terms and should mention no free variables except the formal parameters. Thus, for example, body may not contain references to "global" or "special" variables; ACL2 constants or additional formals should be used instead.

The final conditions on admissibility concern the termination of the recursion. Roughly put, all applications of `fn` must terminate. In particular, there must exist a binary relation, `rel`, and some unary predicate `mp` such that `rel` is well-founded on objects satisfying `mp`, the measure term `m` must always produce something satisfying `mp`, and the measure term must decrease according to `rel` in each recursive call, under the hypothesis that all the tests governing the call are satisfied. By the meaning of well-foundedness, we know there are no infinitely descending chains of successively `rel`-smaller `mp`-objects. Thus, the recursion must terminate.

The only primitive well-founded relation in ACL2 is `e0-ord-<^` (see Section 8.23 [E0-ORD-<], page 262), which is known to be well-founded on the `e0-ordinalp^`s (see Section 8.24 [E0-ORDINALP], page 263). For the proof of well-foundedness, see Section 8.71 [PROOF-OF-WELL-FOUNDEDNESS], page 332. However it is possible to add new well-founded relations. For details, see Section 14.17 [WELL-FOUNDED-RELATION], page 662. We discuss later how to specify which well-founded relation is selected by `defun` and in the present discussion we assume, without loss of generality, that it is `e0-ord-<^` on the `e0-ordinalp^`s.

For example, for our generic definition of `fn` above, with measure term `(m x y)`, two theorems must be proved. The first establishes that `m` produces an ordinal:

```
(e0-ordinalp (m x y)).
```

The second shows that `m` decreases in the (only) recursive call of `fn`:

```
(implies (not (test x y))
         (e0-ord-< (m (d x) y) (m x y))).
```

Observe that in the latter formula we must show that the "m-size" of `(d x)` and `y` is "smaller than" the `m`-size of `x` and `y`, provided the test, `(test x y)`, in the body fails, thus leading to the recursive call `(fn (d x) y)`.

See Section 8.23 [E0-ORD-<], page 262 for a discussion of this notion of "smaller than." It should be noted that the most commonly used ordinals are the natural numbers and that on natural numbers, `e0-ord-<^` is just the familiar `<^` relation. Thus, it is very common to use a measure `m` that returns a nonnegative integer, for then `(e0-ordinalp (m x y))` becomes a simple conjecture about the type of `m` and the second formula above becomes a conjecture about the less-than relationship of nonnegative integer arithmetic.

The most commonly used measure function is `acl2-count^`, which computes a nonnegative integer size for all ACL2 objects. See Section 8.3 [ACL2-COUNT], page 224.

Probably the most common recursive scheme in Lisp programming^ is when some formal is supposed to be a list and in the recursive call it is replaced by its `cdr^`. For example, `(test x y)` might be simply `(atom x)` and `(d x)` might be `(cdr x)`. In that case, `(acl2-count x)` is a suitable measure because the `acl2-count^` of a `cons^` is strictly larger than the `acl2-count^`s of its `car^` and `cdr^`. Thus, "recursion by `car^`" and "recursion by `cdr^`" are trivially admitted if `acl2-count^` is used as the measure and the definition protects every recursive call by a test insuring that the decremented argument is a `consp^`. Similarly, "recursion by `1-^`" in which a positive integer formal is decremented by one in recursion, is also trivially admissible. See Section 14.1 [BUILT-IN-CLAUSES], page 621 to extend the class of trivially admissible recursive schemes.

We now turn to the question of which well-founded relation `defun` uses. It should first be observed that `defun` must actually select both a relation (e.g., `e0-ord-<^`) and a domain predicate (e.g., `e0-ordinalp^`) on which that relation is known to be well-founded. But, as noted elsewhere (see Section 14.17 [WELL-FOUNDED-RELATION], page 662), every known well-founded relation has a unique domain predicate associated with it and so it suffices to identify simply the relation here.

The `xargs^` field of a `declare^` permits the explicit specification of any known well-founded relation with the keyword `:well-founded-relation^`. An example is given below. If the `xargs^` for a `defun` specifies a well-founded relation, that relation and its associated domain predicate are used in generating the termination conditions for the definition.

If no `:well-founded-relation^` is specified, `defun` uses the `:well-founded-relation^` specified in the `acl2-defaults-table^`. See Section 6.34 [SET-WELL-FOUNDED-RELATION], page 192 to see how to set the default well-founded relation (and, implicitly, its domain predicate). The initial default well-founded relation is `e0-ord-<^` (with domain predicate `e0-ordinalp^`).

This completes the brief sketch of the ACL2 definitional principle.

The following example illustrates all of the available declarations, `xargs^`, and hints^, but is completely nonsensical.

```
(defun example (x y z a b c i j)
  (declare (ignore a b c)
           (type integer i j)
           (xargs :guard (symbolp x)
                  :measure (- i j)
                  :well-founded-relation my-wfr
                  :hints (("Goal"
                           :do-not-induct t
                           :do-not '(generalize fertilize)
                           :expand ((assoc x a) (member y z))
                           :restrict ((<-trans ((x x) (y (foo x)))))
                           :hands-off (length binary-append)
                           :in-theory (set-difference-theories
                                        (current-theory :here)
                                        '(assoc))
                           :induct (and (nth n a) (nth n b))
                           :use ((:instance assoc-of-append
                                            (x a) (y b) (z c))
                                 (:functional-instance
                                   (:instance p-f (x a) (y b))
                                   (p consp)
                                   (f assoc)))))
                  :guard-hints (("Subgoal *1/3'"
                                 :use ((:instance assoc-of-append
                                                  (x a) (y b) (z c)))))
                  :mode :logic
                  :otf-flg t))
  (example-body x y z i j))
```

6.17 DEFUN-SK

DEFUN-SK   define a function whose body has an outermost quantifier

```
Examples:
(defun-sk forall-x-p-and-q (y z)
  (forall x
          (and (p x y z)
               (q x y z))))

(defun-sk forall-x-p-and-q (y z) ; equivalent to the above
  (forall (x)
          (and (p x y z)
               (q x y z))))

(defun-sk some-x-y-p-and-q (z)
  (exists (x y)
          (and (p x y z)
               (q x y z))))
```

* EXISTS:: existential quantifier
* FORALL:: universal quantifier

General Form:

```
(defun-sk fn (var1 ... varn) body
  &key doc quant-ok skolem-name thm-name)
```

where `fn` is the symbol you wish to define and is a new symbolic name (see Section 8.65 [NAME], page 329), `(var1 ... varn)` is its list of formal parameters (see Section 8.65 [NAME], page 329), and `body` is its body, which must be quantified as described below. The `&key` argument `doc^` is an optional documentation^ string to be associated with `fn`; for a description of its form, see Section 5.5 [DOC-STRING], page 134. The other arguments are explained below. For a more elaborate example than those above, see Section 1.5.5 [TUTORIAL4-DEFUN-SK-EXAMPLE], page 56.

The third argument, `body`, must be of the form

```
(Q bound-vars term)
```

where: `Q` is the symbol `forall^` or `exists^` (in the "ACL2" package), `bound-vars` is a variable or true list of variables disjoint from `(var1 ... varn)` and not including `state^`, and `term` is a term. The case that `bound-vars` is a single variable `v` is treated exactly the same as the case that `bound-vars` is `(v)`.

The result of this event is to introduce a "Skolem function," whose name is the keyword argument `skolem-name` if that is supplied, and otherwise is the result of modifying `fn` by suffixing "-WITNESS" to its name. The following definition and and of the following two theorems are introduced for `skolem-name` and `fn` in the case that `bound-vars` (see above) is a single variable `v`. The name of the `defthm^` event may be supplied as the value of the keyword argument `:thm-name`; if it is not supplied, then it is the result of modifying `fn` by suffixing "-SUFF" to its name in the case that the quantifier is `exists^`, and "-NECC" in the case that the quantifier is `forall^`.

```
(defun fn (var1 ... varn)
  (let ((v (skolem-name var1 ... varn)))
    body))

(defthm fn-suff ;in case the quantifier is EXISTS
  (implies body
           (fn var1 ... varn)))

(defthm fn-necc ;in case the quantifier is FORALL
  (implies (not body)
           (not (fn var1 ... varn))))
```

In the case that `bound-vars` is a list of at least two variables, say `(bv1 ... bvk)`, the definition above is the following instead, but the theorem remains unchanged.

```
(defun fn (var1 ... varn)
  (mv-let (bv1 ... bvk)
          (skolem-name var1 ... varn)
          body))
```

In order to emphasize that the last element of the list `body` is a term, `defun-sk` checks that the symbols `forall^` and `exists^` do not appear anywhere in it. However, on rare occasions one might deliberately choose to violate this convention, presumably because `forall^` or `exists^` is being used as a variable or because a macro call will be eliminating "calls of" `forall^` and `exists^`. In these cases, the keyword argument `quant-ok` may be supplied a non-`nil` value. Then `defun-sk` will permit `forall^` and `exists^` in the body, but it will still cause an error if there is a real attempt to use these symbols as quantifiers.

Those who want a more flexible version of `defun-sk` that allows nested quantifiers, should contact the implementors. In the meantime, if you want to represent nested quantifiers, you have to manage that yourself. For example, in order to represent

```
(forall x (exists y (p x y z)))
```

you would use `defun-sk` twice, for example as follows.

```
(defun-sk exists-y-p (x z)
  (exists y (p x y z)))

(defun-sk forall-x-exists-y-p (z)
  (forall x (exists-y-p x z)))
```

Some distracting and unimportant warnings are inhibited during `defun-sk`.

`Defun-sk` is implemented using `defchoose^`, and hence should only be executed in defun-mode^ `:logic^`; see Section 8.18 [DEFUN-MODE], page 256 and see Section 6.3 [DEFCHOOSE], page 155.

Note that this way of implementing quantifiers is not a new idea. Hilbert was certainly aware of it 60 years ago! The "story" (see Section 8.7 [BIBLIOGRAPHY], page 242) explains why our use of `defchoose^` is appropriate, even in the presence of `epsilon-0` induction.

6.17.1 EXISTS

EXISTS existential quantifier

The symbol `exists` (in the ACL2 package) represents existential quantification in the context of a `defun-sk^` form. See Section 6.17 [DEFUN-SK], page 173 and see Section 6.17.2 [FORALL], page 176.

6.17.2 FORALL

FORALL universal quantifier

The symbol `forall` (in the ACL2 package) represents universal quantification in the context of a `defun-sk^` form. See Section 6.17 [DEFUN-SK], page 173 and see Section 6.17.1 [EXISTS], page 176.

6.18 ENCAPSULATE

ENCAPSULATE constrain some functions and/or hide some events^

```
Examples:
(encapsulate ((an-element (lst) t))
  (local (defun an-element (lst)
           (if (consp lst) (car lst) nil)))
  (local (defthm member-equal-car
            (implies (and lst (true-listp lst))
                     (member-equal (car lst) lst))))
  (defthm thm1
     (implies (null lst) (null (an-element lst))))
  (defthm thm2
     (implies (and (true-listp lst)
                   (not (null lst)))
              (member-equal (an-element lst) lst))))

(encapsulate
 ()

 (local (defthm hack
          (implies (and (syntaxp (quotep x))
                        (syntaxp (quotep y)))
                   (equal (+ x y z)
                          (+ (+ x y) z)))))

 (defthm nthcdr-add1-conditional
   (implies (not (zp (1+ n)))
            (equal (nthcdr (1+ n) x)
                   (nthcdr n (cdr x))))))

General Form:
(encapsulate (signature ... signature)
  ev1
  ...
  evn)
```

where each signature^ is as described in the documentation for signature^, each signature^ describes a different function symbol, and each `evi` is an embedded event form as described in the documentation for embedded-event-form^. There must be at least one `evi`. The `evi` inside `local^` special forms are called "local" events^ below. Events^ that are not `local^` are sometimes said to be "exported" by the encapsulation. We make the further restriction that no `defaxiom` event may be introduced in the scope of an `encapsulate` (not even by `encapsulate` or `include-book^` events that are among the `evi`). Furthermore, no non-`local^` `include-book^` event is permitted in the scope of any `encapsulate` with a non-empty list of signatures.

To be well-formed, an `encapsulate` event must have the properties that each event in the body (including the `local^` ones) can be successfully executed in sequence and that in the resulting theory, each function mentioned among the signature^s was introduced via a `local^` event and has the signature^ listed. In addition, the body may contain no "local incompatibilities" which, roughly stated, means that the events^ that are not `local^` must not syntactically require symbols defined by `local^` events^, except for the functions listed in the signature^s. See Section 8.60 [LOCAL-INCOMPATIBILITY], page 321. Finally, no non-`local^` recursive definition in the body may involve in its suggested induction scheme any function symbol listed among the signature^s. See Section 8.82 [SUBVERSIVE-INDUCTIONS], page 347.

The result of an `encapsulate` event is an extension of the logic in which, roughly speaking, the functions listed in the signature^s are constrained to have the signature^s listed and to satisfy the non-`local^` theorems proved about them. In fact, other functions introduced in the `encapsulate` event may be considered to have "constraint^s" as well. (See Section 8.12 [CONSTRAINT], page 247 for details, which are only relevant to functional instantiation.) Since the constraint^s were all theorems in the "ephemeral" or "local" theory, we are assured that the extension produced by `encapsulate` is sound. In essence, the `local^` definitions of the constrained functions are just "witness functions" that establish the consistency of the constraint^s. Because those definitions are `local^`, they are not present in the theory produced by encapsulation. `Encapsulate` also exports all rules generated by its non-`local^` events^, but rules generated by `local^` events^ are not exported.

The default-defun-mode^ for the first event in an encapsulation is the default defun-mode^ "outside" the encapsulation. But since events^ changing the defun-mode^ are permitted within the body of an `encapsulate`, the default defun-mode^ may be changed. However, defun-mode^ changes occurring within the body of the `encapsulate` are not exported. In particular, the `acl2-defaults-table^` after an `encapsulate` is always the same as it was before the `encapsulate`, even though the `encapsulate` body might contain defun-mode^ changing events^, `:program^` and `:logic^`. See Section 8.18 [DEFUN-MODE], page 256. More generally, after execution of an `encapsulate` event, the value of `acl2-defaults-table^` is restored to what it was immediately before that event was executed. See Section 9.2 [ACL2-DEFAULTS-TABLE], page 374.

Theorems about the constrained function symbols may then be proved — theorems whose proofs necessarily employ only the constraint^s. Thus, those theorems may be later functionally instantiated, as with the `:functional-instance` lemma instance (see Section 8.59 [LEMMA-INSTANCE], page 319), to derive analogous theorems about different functions, provided the constraints (see Section 8.12 [CONSTRAINT], page 247) can be proved about the new functions.

Observe that if the signature^s list is empty, `encapsulate` may still be useful for deriving theorems to be exported whose proofs require lemmas you prefer to hide (i.e., made `local^`).

The order of the events^ in the vicinity of an `encapsulate` is confusing. We discuss it in some detail here because when logical names are being used with theory functions to compute sets of rules, it is sometimes important to know the order in which events^ were executed. (See Section 8.61 [LOGICAL-NAME], page 322 and see Section 15.13 [THEORY-FUNCTIONS], page 678.) What, for example, is the set of function names extant in the middle of an encapsulation?

If the most recent event is `previous` and then you execute an `encapsulate` constraining `an-element` with two non-`local`^ events^ in its body, `thm1` and `thm2`, then the order of the events^ after the encapsulation is (reading chronologically forward): `previous`, `thm1`, `thm2`, `an-element` (the `encapsulate` itself). Actually, between `previous` and `thm1` certain extensions were made to the world^ by the superior `encapsulate`, to permit `an-element` to be used as a function symbol in `thm1`.

Finally, we note that an `encapsulate`^ event is redundant if and only if a syntactically identical `encapsulate`^ has already been executed under the same `default-defun-mode`^. See Section 8.76 [REDUNDANT-EVENTS], page 337.

6.19 IN-THEORY

IN-THEORY designate "current" theory (enabling its rules)

```
Example:
(in-theory (set-difference-theories
             (universal-theory :here)
             '(flatten (:executable-counterpart flatten))))

General Form:
(in-theory term :doc doc-string)
```

where `term` is a term that when evaluated will produce a theory (see Chapter 15 [THEORIES], page 667), and `doc-string`^ is an optional documentation^ string not beginning with "`:doc-section` ...". Except for the variable `world`^, `term` must contain no free variables. `Term` is evaluated with the variable `world`^ bound to the current world^ to obtain a theory and the corresponding runic theory (see Chapter 15 [THEORIES], page 667) is then made the current theory. Thus, immediately after the `in-theory`, a rule is enable^d iff its rule name is a member of the runic interpretation (see Chapter 15 [THEORIES], page 667) of some member of the value of `term`. See Section 15.13 [THEORY-FUNCTIONS], page 678 for a list of the commonly used theory manipulation functions.

Because no unique name is associated with an `in-theory` event, there is no way we can store the documentation^ string `doc-string^` in our documentation^ data base. Hence, we actually prohibit `doc-string^` from having the form of an ACL2 documentation^ string; see Section 5.5 [DOC-STRING], page 134.

6.20 INCLUDE-BOOK

INCLUDE-BOOK load the events^ in a file

```
Examples:
(include-book "my-arith")
(include-book (:RELATIVE "my-arith"))
(include-book "/home/smith/my-arith")
(include-book (:ABSOLUTE "home" "smith" "my-arith"))

General Form:
(include-book file :load-compiled-file action :doc doc-string)
```

where `file` is a book name. See Chapter 3 [BOOKS], page 89 for general information, see Section 3.3 [BOOK-NAME], page 95 for information about book names, and see Section 3.9 [PATHNAME], page 105 for information about file names (including structured pathname^s). `Action` is one of `t`, `nil`, `:warn` (the default), or `:try`; these values are explained below. `Doc-string` is an optional documentation^ string; see Section 5.5 [DOC-STRING], page 134. If the book has no `certificate^` or if its `certificate^` is invalid, a warning is printed and the book is included anyway; see Section 3.5 [CERTIFICATE], page 99. This can lead to serious errors; see Section 3.12 [UNCERTIFIED-BOOKS], page 111. If the portcullis of the certificate^ (see Section 3.10 [PORTCULLIS], page 108) cannot be raised in the host logical world^, an error is caused and no change occurs to the logic. Otherwise, the non-`local^` events^ in file are assumed. Then the keep^ of the certificate^ is checked to insure that the correct files were read; see Section 3.8 [KEEP], page 104. A warning is printed if uncertified books^ were included. Even if no warning is printed, `include-book` places a burden on you; see Section 3.5 [CERTIFICATE], page 99.

If there is a compiled file for the book that was created more recently than the book itself and the value `action` of the `:load-compiled-file` argument is not `nil`, or is omitted, then the compiled file is automatically loaded; otherwise it is not loaded. If `action` is `t` then the compiled file must be loaded or an error will occur, while if `action` is `:warn` (the default) then a warning will be printed. `Certify-book^` can be used to compile a book. The effect of compilation^ is to speed up the execution of the functions defined within the book when those functions are applied to specific values. The presence of compiled code for the functions in the book should not otherwise affect the performance of ACL2. See Section 8.39 [GUARD], page 285 for a discussion.

`Include-book` is similar in spirit to `encapsulate^` in that it is a single event that "contains" other events^, in this case the events^ listed in the file named. `Include-book` processes the non-`local^` event forms in the file, assuming that each is admissible. `Local^` events^ in the file are ignored. You may use `include-book` to load multiple books^, creating the logical world^ that contains the definitions and theorems of all of them.

If any non-`local^` event of the book attempts to define a name^ that has already been defined — and the book's definition is not syntactically identical to the existing definition — the attempt to include the book fails, an error message is printed, and no change to the logical world^ occurs. See Section 8.76 [REDUNDANT-EVENTS], page 337 for the details.

When a book is included, the default defun-mode^ (see Section 8.16 [DEFAULT-DEFUN-MODE], page 254) for the first event is always `:logic^`. That is, the default defun-mode^ "outside" the book — in the environment in which `include-book` was called — is irrelevant to the book. Events^ that change the defun-mode^ are permitted within a book (provided they are not in `local^` forms). However, such changes within a book are not exported, i.e., at the conclusion of an `include-book`, the "outside" default defun-mode^ is always the same as it was before the `include-book`.

Unlike every other event in ACL2, `include-book` puts a burden on you. Used improperly, `include-book` can be unsound in the sense that it can create an inconsistent extension of a consistent logical world^. A certification mechanism is available to help you carry this burden — but it must be understood up front that even certification is no guarantee against inconsistency here. The fundamental problem is one of file system security. See Section 3.5 [CERTIFICATE], page 99 for a discussion of the security issues.

After execution of an `include-book` form, the value of `acl2-defaults-table^` is restored to what it was immediately before that `include-book` form was executed. See Section 9.2 [ACL2-DEFAULTS-TABLE], page 374.

This concludes the guided tour through books^. See Section 6.27 [SET-COMPILE-FNS], page 187 for a subtle point about the interaction between `include-book` and on-the-fly compilation^. See Section 3.6 [CERTIFY-BOOK], page 101 for a discussion of how to certify a book.

6.21 LOCAL

LOCAL hiding an event in an encapsulation or book

```
Examples:
```

```
(local (defthm hack1
         (implies (and (acl2-numberp x)
                       (acl2-numberp y)
                       (equal (* x y) 1))
                  (equal y (/ x)))))

(local (defun double-naturals-induction (a b)
         (cond ((and (integerp a) (integerp b) (< 0 a) (< 0 b))
                (double-naturals-induction (1- a) (1- b)))
               (t (list a b)))))

General Form:
(local ev)
```

where `ev` is an event form. If the current default defun-mode^ (see Section 8.16 [DEFAULT-DEFUN-MODE], page 254) is `:logic^` and `ld-skip-proofsp^` is `nil` or `t`, then `(local ev)` is equivalent to `ev`. But if the current default defun-mode^ is `:program^` or if `ld-skip-proofsp^` is `'include-book^`, then `(local ev)` is a `no-op`. Thus, if such forms are in the event list of an `encapsulate^` event or in a book, they are processed when the encapsulation or book is checked for admissibility in `:logic^` mode but are skipped when extending the host world^. Such events^ are thus considered "local" to the verification of the encapsulation or book. The non-local events^ are the ones "exported" by the encapsulation or book. See Section 6.18 [ENCAPSULATE], page 176 for a thorough discussion. Also see Section 8.60 [LOCAL-INCOMPATIBILITY], page 321 for a discussion of a commonly encountered problem with such event hiding: you can't make an event local if its presence is required to make sense of a non-local one.

Note that events^ that change the default defun-mode^, and in fact any events^ that set the `acl2-defaults-table^`, are disallowed inside the scope of `local`. See Section 8.25 [EMBEDDED-EVENT-FORM], page 267.

6.22 LOGIC

LOGIC to set the default defun-mode^ to `:logic`

```
Example:
ACL2 p!>:logic
ACL2 !>
```

Typing the keyword `:logic` sets the default defun-mode^ to `:logic`.

Functions defined in `:logic` mode are logically defined. See Section 8.18 [DEFUN-MODE], page 256.

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

See Section 8.18 [DEFUN-MODE], page 256 for a discussion of the defun-mode^s available and what their effects on the logic are. See Section 8.16 [DEFAULT-DEFUN-MODE], page 254 for a discussion of how the default defun-mode^ is used. This event is equivalent to `(table acl2-defaults-table :defun-mode :logic)`. See Section 9.2 [ACL2-DEFAULTS-TABLE], page 374.

Recall that the top-level form `:logic` is equivalent to `(logic)`; see Section 8.46 [KEYWORD-COMMANDS], page 306. Thus, to change the default defun-mode^ to `:logic` in a book, use `(logic)`, which is an embedded event form, rather than `:logic`, which is not a legal form for books^. See Section 8.25 [EMBEDDED-EVENT-FORM], page 267.

6.23 MACRO-ALIASES-TABLE

MACRO-ALIASES-TABLE a table^ used to associate function names with macro names

```
Example:
(table macro-aliases-table 'append 'binary-append)
```

This example associates the function symbol `binary-append`^ with the macro name `append`^. As a result, the name `append`^ may be used as a runic designator (see Chapter 15 [THEORIES], page 667) by the various theory functions. Thus, for example, it will be legal to write

```
(in-theory (disable append))
```

as an abbreviation for

```
(in-theory (disable binary-append))
```

which in turn really abbreviates

```
(in-theory (set-difference-theories (current-theory :here)
                                    '(binary-append)))
```

```
General Form:

(table macro-aliases-table 'macro-name 'function-name)
```

or very generally

```
(table macro-aliases-table macro-name-form function-name-form)
```

where `macro-name-form` and `function-name-form` evaluate, respectively, to a macro name and a function name in the current ACL2 world^. See Section 6.35 [TABLE], page 193 for a general discussion of tables and the `table` event used to manipulate tables.

The `table^ macro-aliases-table^` is an alist that associates macro symbols with function symbols, so that macro names may be used as runic designators (see Chapter 15 [THEORIES], page 667). For a convenient way to add entries to this table^, see Section 6.1 [ADD-MACRO-ALIAS], page 154. To remove entries from the table^ with ease, see Section 6.26 [REMOVE-MACRO-ALIAS], page 187.

This table^ is used by the theory functions. For example, in order that `(disable append)` be interpreted as `(disable binary-append)`, it is necessary that the example form above has been executed. In fact, this table^ does indeed associate many of the macros provided by the ACL2 system, including `append^`, with function symbols. Loosely speaking, it only does so when the macro is "essentially the same thing as" a corresponding function; for example, `(append x y)` and `(binary-append x y)` represent the same term, for any expressions `x` and `y`.

6.24 MUTUAL-RECURSION

MUTUAL-RECURSION define some mutually recursive functions

```
Example:
(mutual-recursion
 (defun evenlp (x)
   (if (consp x) (oddlp (cdr x)) t))
 (defun oddlp (x)
   (if (consp x) (evenlp (cdr x)) nil)))

General Form:
(mutual-recursion def1 ... defn)
where each defi is a DEFUN form.
```

When mutually recursive functions are introduced it is necessary to do the termination analysis on the entire clique of definitions. Each `defun^` form specifies its own measure, either with the `:measure` keyword `xarg` (see Section 8.91 [XARGS], page 369) or by default to `acl2-count^`. When a function in the clique calls a function in the clique, the measure of the callee's actuals must be smaller than the measure of the caller's formals — just as in the case of a simply recursive function. But with mutual recursion, the callee's actuals are measured as specified by the callee's `defun^` while the caller's formals are measured as specified by the caller's `defun^`. These two measures may be different but must be comparable in the sense that `e0-ord-<^` decreases through calls.

The `guard^` analysis must also be done for all of the functions at the same time. If any one of the `defun^`s specifies the `:verify-guards^` `xarg` to be `nil`, then guard^ verification is omitted for all of the functions.

Technical Note: Each `defi` above must be of the form `(defun ...)`. In particular, it is not permitted for a `defi` to be a form that will macroexpand into a `defun^` form. This is because `mutual-recursion` is itself a macro, and since macroexpansion occurs from the outside in, at the time `(mutual-recursion def1 ... defk)` is expanded the `defi` have not yet been. But `mutual-recursion` must decompose the `defi`. We therefore insist that they be explicitly presented as `defun^`s.

Suppose you have defined your own `defun^`-like macro and wish to use it in a `mutual-recursion` expression. Well, you can't. (!) But you can define your own version of `mutual-recursion` that allows your `defun^`-like form. Here is an example. Suppose you define

```
(defmacro my-defun (&rest args) (my-defun-fn args))
```

where `my-defun-fn` takes the arguments of the `my-defun` form and produces from them a `defun^` form. As noted above, you are not allowed to write `(mutual-recursion (my-defun ...) ...)`. But you can define the macro `my-mutual-recursion` so that

```
(my-mutual-recursion (my-defun ...) ... (my-defun ...))
```

expands into `(mutual-recursion (defun ...) ... (defun ...))` by applying `my-defun-fn` to each of the arguments of `my-mutual-recursion`.

```
(defun my-mutual-recursion-fn (lst)
  (declare (xargs :guard (alistp lst)))

; Each element of lst must be a consp (whose car, we assume, is always
; MY-DEFUN).  We apply my-defun-fn to the arguments of each element and
```

```
; collect the resulting list of DEFUNs.

  (cond ((atom lst) nil)
        (t (cons (my-defun-fn (cdr (car lst)))
                 (my-mutual-recursion-fn (cdr lst))))))

(defmacro my-mutual-recursion (&rest lst)

; Each element of lst must be a consp (whose car, we assume, is always
; MY-DEFUN).  We obtain the DEFUN corresponding to each and list them
; all inside a MUTUAL-RECURSION form.

  (declare (xargs :guard (alistp lst)))
  (cons 'mutual-recursion (my-mutual-recursion-fn lst))).
```

6.25 PROGRAM

PROGRAM to set the default defun-mode^ to `:program`

```
Example:
ACL2 !>:program
ACL2 p!>
```

Typing the keyword `:program` sets the default defun-mode^ to `:program`.

Functions defined in `:program` mode are logically undefined but can be executed on constants outside of deductive contexts. See Section 8.18 [DEFUN-MODE], page 256.

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

See Section 8.18 [DEFUN-MODE], page 256 for a discussion of the defun-mode^s available and what their effects on the logic are. See Section 8.16 [DEFAULT-DEFUN-MODE], page 254 for a discussion of how the default defun-mode^ is used. This event is equivalent to `(table acl2-defaults-table :defun-mode :program)`. See Section 9.2 [ACL2-DEFAULTS-TABLE], page 374.

Recall that the top-level form `:program` is equivalent to `(program)`; see Section 8.46 [KEYWORD-COMMANDS], page 306. Thus, to change the default defun-mode^ to `:program` in a book, use `(program)`, which is an embedded event form, rather than `:program`, which is not a legal form for books^. See Section 8.25 [EMBEDDED-EVENT-FORM], page 267.

6.26 REMOVE-MACRO-ALIAS

REMOVE-MACRO-ALIAS   remove the association of a function name with a macro name

```
Example:
(remove-macro-alias append)

General Form:
(remove-macro-alias macro-name)
```

See Section 6.23 [MACRO-ALIASES-TABLE], page 183 for a discussion of macro aliases; also see Section 6.1 [ADD-MACRO-ALIAS], page 154. This form sets `macro-aliases-table^` to the result of deleting the key `macro-name` from that table^. If the name does not occur in the table^, then this form still generates an event, but the event has no real effect.

6.27 SET-COMPILE-FNS

SET-COMPILE-FNS   have each function compiled as you go along.

```
Example Forms:
(set-compile-fns t)    ; new functions compiled after DEFUN
(set-compile-fns nil)  ; new functions not compiled after DEFUN
```

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

Also see Section 9.5 [COMP], page 379, because it may be more efficient in some Common Lisps to compile many functions at once rather than to compile each one as you go along.

```
General Form:
(set-compile-fns term)
```

where `term` is a variable-free term that evaluates to `t` or `nil`. This macro is equivalent to

```
(table acl2-defaults-table :compile-fns term).
```

However, unlike that simple call of the `table^` event function (see Section 6.35 [TABLE], page 193), no output results from a `set-compile-fns` event.

`Set-compile-fns` may be thought of as an event that merely sets a flag to `t` or `nil`. The flag's effect is felt when functions are defined, as with `defun^`. If the flag is `t`, functions are automatically compiled after they are defined, as are their executable counterparts (see Section 8.30 [EXECUTABLE-COUNTERPART], page 271). Otherwise, functions are not automatically compiled. Because `set-compile-fns` is an event, the old value of the flag is restored when a `set-compile-fns` event is undone.

Even when `:set-compile-fns t` has been executed, functions are not individually compiled when processing an `include-book^` event. If you wish to include a book of compiled functions, we suggest that you first certify it with the compilation^ flag set; see Section 3.6 [CERTIFY-BOOK], page 101. More generally, compilation^ via `set-compile-fns` is suppressed when the state^ global variable `ld-skip-proofsp^` has value `'include-book^`.

6.28 SET-IGNORE-OK

SET-IGNORE-OK allow unused formals and locals without an `ignore` declaration

```
Examples:
(set-ignore-ok t)
(set-ignore-ok nil)
(set-ignore-ok :warn)
```

The first example above allows unused formals and locals, i.e., variables that would normally have to be declare^d `ignored`. The second example disallows unused formals and locals; this is the default. The third example allows them, but prints an appropriate warning.

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

```
General Form:
(set-ignore-ok flg)
```

where `flg` is either `t`, `nil`, or `:warn`.

One might find this event useful when one is generating function definitions by an automated procedure, when that procedure does not take care to make sure that all formals are actually used in the definitions that it generates.

6.29 SET-INHIBIT-WARNINGS

SET-INHIBIT-WARNINGS control warnings

```
Examples:
(set-inhibit-warnings "theory" "use")
```

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

```
General Form:
(set-inhibit-warnings string1 string2 ...)
```

where each string is considered without regard to case. This macro is equivalent to `(table acl2-defaults-table :inhibit-warnings lst)`, where `lst` is the list of strings supplied. This macro is an event (see Section 6.35 [TABLE], page 193), but no output results from a `set-inhibit-warnings` event.

The effect of this event is to suppress any warning whose label is a member of this list (where again, case is ignored). For example, the warning

```
  ACL2 Warning [Use] in ( THM ...):  It is unusual to :USE ....
```

will not be printed if `"use"` (or `"USE"`, etc.) is a member of the given list of strings.

Of course, if warnings are inhibited overall — see Section 9.13 [SET-INHIBIT-OUTPUT-LST], page 387 — then the value of `:inhibit-warnings` is entirely irrelevant.

6.30 SET-INVISIBLE-FNS-ALIST

SET-INVISIBLE-FNS-ALIST set the invisible functions alist

```
Examples:
(set-invisible-fns-alist ((binary-+ unary--)
                          (binary-* unary-/)
                          (unary-- unary--)
                          (unary-/ unary-/)))
```

Among other things, the setting above has the effect of making `unary--^` "invisible" for the purposes of applying permutative `:rewrite^` rules to `binary-+^` trees. Thus, `arg` and `(unary-- arg)` will be given the same weight and will be permuted so as to be adjacent. The form `(invisible-fns-alist (w state))` returns the current value of the invisible functions alist.

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

```
General Form:
(set-invisible-fns-alist alist)
```

where `alist` is either `t` or a true list of pairs, each element of which is of the form `(fn ufn1 ... ufnk)`, where `fn` is a function symbol and each `ufni` is a unary function symbol. When alist is `t`, the initial default alist is used in its place. Modulo the replacement of `alist` by the default setting when `alist` is `t`, this macro is equivalent to

```
(table acl2-defaults-table :invisible-fns-alist 'alist),
```

which is also an event (see Section 6.35 [TABLE], page 193), but no output results from a `set-invisible-fns-alist` event.

The "invisible functions alist" is an alist that associates with certain function symbols, e.g., `fn` above, a set of unary functions, e.g., the `ufni` above. The `ufni` associated with `fn` in the invisible functions alist are said to be "invisible with respect to `fn`." If `fn` is not the `car^` of any pair on the invisible functions `alist`, then no function is invisible for it. Thus, setting the invisible functions alist to `nil` completely eliminates the consideration of invisibility.

The notion of invisibility is involved in the use of the `:loop-stopper^` field of `:rewrite^` rules to prevent the indefinite application of permutative rewrite rules. Roughly speaking, if rewrite rules are being used to permute `arg` and (ufni arg) inside of a nest of `fn` calls, and `ufni` is invisible with respect to `fn`, then `arg` and `(ufni arg)` are considered to have the same "weight" and will be permuted so as to end up as adjacent tips in the `fn` nest. See Section 8.62 [LOOP-STOPPER], page 324.

6.31 SET-IRRELEVANT-FORMALS-OK

SET-IRRELEVANT-FORMALS-OK allow irrelevant formals in definitions

```
Examples:
(set-irrelevant-formals-ok t)
(set-irrelevant-formals-ok nil)
(set-irrelevant-formals-ok :warn)
```

The first example above allows irrelevant formals in definitions; see Section 10.136 [IRRELEVANT-FORMALS], page 458. The second example disallows irrelevant formals; this is the default. The third example allows irrelevant formals, but prints an appropriate warning.

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

```
General Form:
(set-irrelevant-formals-ok flg)
```

where `flg` is either `t`, `nil`, or `:warn`.

6.32 SET-MEASURE-FUNCTION

SET-MEASURE-FUNCTION set the default measure function symbol

```
Examples:
(set-measure-function nqthm::count)
```

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

```
General Form:
(set-measure-function name)
```

where `name` is a function symbol of one argument. This macro is equivalent to `(table acl2-defaults-table :measure-function 'name)`, which is also an event (see Section 6.35 [TABLE], page 193), but no output results from a `set-measure-function` event.

This event sets the default measure function to `name`. Subsequently, if a recursively defined function is submitted to `defun^` with no explicitly given `:measure` argument, `defun^` "guesses" the measure `(name var)`, where `name` is the then current default measure function and `var` is the first formal found to be tested along every branch and changed in every recursive call.

6.33 SET-VERIFY-GUARDS-EAGERNESS

SET-VERIFY-GUARDS-EAGERNESS the eagerness with which guard^ verification is tried.

```
Example Forms:                        try guard verification?
(set-verify-guards-eagerness 0) ; no, unless :verify-guards t
(set-verify-guards-eagerness 1) ; yes if a :guard is supplied
(set-verify-guards-eagerness 2) ; yes, unless :verify-guards nil
```

Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

```
General Form:
(set-verify-guards-eagerness n)
```

where `n` is a variable-free term that evaluates to `0`, `1`, or `2`. This macro is equivalent to

```
(table acl2-defaults-table :verify-guards-eagerness term).
```

However, unlike that simple call of the `table`^ event function (see Section 6.35 [TABLE], page 193), no output results from a `set-verify-guards-eagerness` event.

`Set-verify-guards-eagerness` may be thought of as an event that merely sets a flag to `0`, `1`, or `2`. The flag is used by certain `defun`^ events^ to determine whether guard^ verification is tried. The flag is irrelevant to those `defun`^ events^ in `:program`^ mode and to those `defun`^ events^ in which an explicit `:verify-guards`^ setting is provided among the `xargs`^. In the former case, guard^ verification is not done because it can only be done when logical functions are being defined. In the latter case, the explicit `:verify-guards`^ setting determines whether guard^ verification is tried. So consider a `:logic`^ mode `defun`^ in which no `:verify-guards`^ setting is provided. Is guard^ verification tried? The answer depends on the eagerness setting as follows. If the eagerness is `0`, guard^ verification is not tried. If the eagerness is `1`, it is tried iff a `:guard`^ is explicitly provided in the `defun`^. If the eagerness is `2`, guard^ verification is tried.

The default behavior of the system is as though the `:verify-guards-eagerness` is `1`.

6.34 SET-WELL-FOUNDED-RELATION

SET-WELL-FOUNDED-RELATION set the default well-founded relation

```
Examples:
(set-well-founded-relation lex2)
```

provided `lex2` has been proved to be a well-founded relation (see Section 14.17 [WELL-FOUNDED-RELATION], page 662). Note: This is an event! It does not print the usual event summary but nevertheless changes the ACL2 logical world^ and is so recorded.

```
General Form:
(set-well-founded-relation rel)
```

where `rel` has been proved to be a well-founded relation on objects satisfying some predicate, `mp`, as described in the documentation^ for well-founded-relation^. This macro is equivalent to `(table acl2-defaults-table :well-founded-relation 'rel)`.

This event sets the default well-founded relation to be that imposed on `mp`-measures by the relation `rel`. Subsequently, if a recursively defined function is submitted to `defun^` with no explicitly given `:well-founded-relation^` argument, `defun^` uses the default relation, `rel`, and the associated domain predicate `mp` used in its well-foundedness theorem. That is, the termination conditions generated will require proving that the measure used by the `defun^` is an `mp`-measure and that in every recursive call the measure of the arguments decreases according to `rel`.

6.35 TABLE

TABLE user-managed tables

```
Examples:
(table tests 1 '(...))                ; set contents of tests[1] to '(...)
(table tests 25)                      ; get contents of tests[25]
(table tests)                         ; return table tests as an alist
(table tests nil nil :clear)          ; clear table tests
(table tests nil (foo 7) :clear)      ; set table tests to (foo 7)
(table tests nil nil :guard)          ; fetch the table guard
(table tests nil nil :guard term)     ; set the table guard

General Form:
(table table-name key-term value-term op term)
```

where `table-name` is a symbol that is the name of a (possibly new) table, `key-term` and `value-term`, if present, are arbitrary terms involving (at most) the single variable `world^`, `op`, if present, is one of the table operations below, and `term`, if present, is a term. `Table` returns an acl2 “error triple." The effect of `table` on `state^` depends on `op` and how many arguments are presented. Some invocations actually have no effect on the ACL2 world^ and hence an invocation of `table` is not always an "event". We explain below, after giving some background information.

The ACL2 system provides "tables" by which the user can associate one object with another. Tables are in essence just conventional association lists — lists of pairs — but the ACL2 environment provides a means of storing these lists in the "ACL2 world" of the current `state^`. The ACL2 user could accomplish the same ends by using ACL2 "global variables;" however, limitations on global variable names are imposed to insure ACL2's soundness. By convention, no table is important to ACL2's soundness, even though some features of the system use tables, and the user is invited to make free use of tables. Because tables are stored in the ACL2 world^ they are restored by `include-book^` and undone by `:ubt^`. many users of Nqthm requested a facility by which user data could be saved in Nqthm "lib files" and tables are ACL2's answer to that request.

Abstractly, each table is an association list mapping "keys" to "values." In addition, each table has a "`:guard`," which is a term that must be true of any key and value used. By setting the `:guard` on a table you may enforce an invariant on the objects in the table, e.g., that all keys are positive integers and all values are symbols. Each table has a "name," which must be a symbol. Given a table name, there are six operations one might perform on the table.

`:put` — associate a value with a key (possibly changing the value currently associated with that key).

`:get` — retrieve the value associated with a key (or nil if no value has been associated with that key).

`:alist` — return an alist showing all keys and non-nil values in the table.

`:clear` — clear the table (so that every value is nil), or if val is supplied then set table to that value (which must be an alist).

`:guard` — fetch or set the :guard of the table.

When the operations above suggest that the table or its `:guard` are modified, what is actually meant is that the current state^ is redefined so that in it, the affected table name has the appropriate properties. in such cases, the table form is an event.

`Table` forms are commonly typed by the user while interacting with the system. `:Put` and `:get` forms are especially common. Therefore, we have adopted a positional syntax that is intended to be convenient for most applications. Essentially, some operations admit a "short form" of invocation.

```
(table name key-term value-term :put)   ; long form
(table name key-term value-term)        ; short form
```

evaluates the key- and value-terms, obtaining two objects that we call `key` and `value`, checks that the `key` and `value` satisfy the `:guard` on the named table and then "modifies" the named table so that the value associated with `key` is `value`. When used like this, `table` is actually an event in the sense that it changes the ACL2 world^. In general, the forms evaluated to obtain the `key` and `value` may involve the variable `world^`, which is bound to the then-current world^ during the evaluation of the forms. However, in the special case that the table in question is named `acl2-defaults-table^`, the `key` and `value` terms may not contain any variables. Essentially, the keys and values used in events^ setting the `acl2-defaults-table^` must be explicitly given constants. See Section 9.2 [ACL2-DEFAULTS-TABLE], page 374.

```
(table name key-term nil :get)          ; long form
(table name key-term)                   ; short form
```

evaluates the key-term (see note below), obtaining an object, `key`, and returns the value associated with `key` in the named table (or, `nil` if there is no value associated with `key`). When used like this, `table` is not an event; the value is simply returned.

```
(table name nil nil :alist)             ; long form
(table name)                            ; short form
```

returns an alist representing the named table; for every key in the table with a non-`nil` associated value, the alist pairs the key and its value. The order in which the keys are presented is unspecified. When used like this, `table` is not an event; the alist is simply returned.

```
(table name nil val :clear)
```

sets the named table to the alist `val`, making the checks that `:put` makes for each key and value of `val`. When used like this, `table` is an event because it changes the ACL2 world^.

```
(table name nil nil :guard)
```

returns the translated form of the guard of the named table.

```
(table name nil nil :guard term)
```

Provided the named table is empty and has not yet been assigned a `:guard` and `term` (which is not evaluated) is a term that mentions at most the variables `key`, `val` and `world^`, this event sets the `:guard` of the named table to `term`. Whenever a subsequent `:put` occurs, `term` will be evaluated with `key` bound to the key argument of the `:put`, `val` bound to the `val` argument of the `:put`, and `world^` bound to the then current world^. An error will be caused by the `:put` if the result of the evaluation is `nil`.

Note that it is not allowed to change the `:guard` on a table once it has been explicitly set. Before the `:guard` is explicitly set, it is effectively just `t`. After it is set it can be changed only by undoing the event that set it. The purpose of this restriction is to prevent the user from changing the `:guards` on tables provided by other people or the system.

The intuition behind the `:guard` mechanism on tables is to enforce invariants on the keys and values in a table, so that the values, say, can be used without run-time checking. But if the `:guard` of a table is sensitive to the ACL2 world^, it may be possible to cause some value in the table to cease satisfying the `:guard` without doing any operations on the table. Consider for example the `:guard` "no value in this table is the name of an event." As described, that is enforced each time a value is stored. Thus, `'bang` can be `:put` in the table provided there is no event named `bang`. But once it is in the table, there is nothing to prevent the user from defining `bang` as a function, causing the table to contain a value that could not be `:put` there anymore. Observe that not all state-sensitive `:guard`s suffer this problem. The `:guard` "every value is an event name" remains invariant, courtesy of the fact that undoing back through an event name in the table would necessarily undo the `:put` of the name into the table.

`Table` was designed primarily for convenient top-level use. Tables are not especially efficient. Each table is represented by an alist stored on the property list of the table name. `:Get` is just a `getprop` and `assoc-equal^`. `:Put` does a `getprop` to the get the table alist, a `put-assoc-equal` to record the new association, and a `putprop` to store the new table alist — plus the overhead associated with `:guard`s and undoable events^. Note that there are never duplicate keys in the resulting `alist`; in particular, when the operation `:clear` is used to install new `alist`, duplicate keys are removed from that alist.

A table name may be any symbol whatsoever. Symbols already in use as function or theorem names, for example, may be used as table names. Symbols in use only as table names may be defined with `defun^`, etc. Because there are no restrictions on the user's choice of table names, table names are not included among the logical names. Thus, `:pe name` will never display a table event (for a logical name other than `:here`). Either `:pe name` will display a "normal" event such as `(defun name ...)` or `(defthm name ...)` or else `:pe name` will cause an error indicating that `name` is not a logical name. This happens even if `name` is in use as a table name. Similarly, we do not permit table names to have documentation^ strings, since the same name might already have a documentation^ string. If you want to associate a documentation^ string with a table name that is being used no other way, define the name as a label and use the `:doc^` feature of `deflabel^` (see Section 6.9 [DEFLABEL], page 162); also see Section 6.6 [DEFDOC], page 158.

6.36 TERM-TABLE

TERM-TABLE a table used to validate meta rules

```
Example:
(table term-table t '((binary-+ x y) '3 'nil (car x)))
```

See Section 6.35 [TABLE], page 193 for a general discussion of tables and the `table` event used to manipulate tables.

The "`term-table`" is used at the time a meta rule is checked for syntactic correctness. Each proposed metafunction is run on each term in this table, and the result in each case is checked to make sure that it is a `termp` in the current world. In each case where this test fails, a warning is printed.

Whenever a metafunction is run in support of the application of a meta rule, the result must be a term in the current world. When the result is not a term, a hard error arises. The `term-table` is simply a means for providing feedback to the user at the time a meta rule is submitted, warning of the definite possibility that such a hard error will occur at some point in the future.

The key used in `term-table` is arbitrary. The top-most value is always the one that is used; it is the entire list of terms to be considered. Each must be a `termp` in the current ACL2 world.

6.37 THEORY-INVARIANT

THEORY-INVARIANT user-specified invariants on theories^

```
Example:
(theory-invariant (not (and (member-equal '(:rewrite left-to-right)
                                          theory)
```

```
                              (member-equal '(:rewrite right-to-left)
                                            theory))))

General Forms:
(theory-invariant term)     ; conjoin a new invariant term
or
(theory-invariant term key) ; change an existing invariant term
```

where `term` is a term that uses no variables other than `theory^` and `world^`, and `key` is an arbitrary "name" for this invariant. If `key` is omitted, an integer is generated and used. `Theory-invariant` is an event that adds to or modifies the table^ of user-supplied theory invariants that are checked each time a theory expression is evaluated.

The theory invariant mechanism is provided via a table (see Section 6.35 [TABLE], page 193) named `theory-invariant-table`. In fact, the `theory-invariant` "event" is just a macro that expands into a use of the `table^` event. More general access to the `theory-invariant` table^ is provided by `table^` itself. For example, the table^ can be inspected or cleared (setting the invariant to `t`) with `table^`.

`Theory-invariant-table` maps arbitrary keys to terms mentioning, at most, the variables `theory^` and `world^`. Every time an alleged theory expression is evaluated, e.g., in the `in-theory^` event or `:in-theory^` hint, each of the terms in `theory-invariant-table` is evaluated with `theory^` bound to the runic theory (see Chapter 15 [THEORIES], page 667) obtained from the theory expression and `world^` bound to the current ACL2 world^ (see Section 8.89 [WORLD], page 362). If the result is `nil`, a warning message is printed. Thus, the table^ can be thought of as a list of conjuncts. Each `term` in the table^ has a "name," which is just the key under which the term is stored. When a theory violates the restrictions specified by some term, both the name and the term are printed. By calling `theory-invariant` with a new term but the same name, you can overwrite that conjunct of the theory invariant.

Theory invariants are particularly useful in the context of large rule sets intended for re-use. Such sets often contain conflicting rules, e.g., rules that are to be enable^d when certain function symbols are disable^d, rules that rewrite in opposite directions and thus loop if simultaneously enable^d, groups of rules which should be enable^d in concert, etc. The developer of such rule sets understands these restrictions and probably documents them. The theory invariant mechanism allows the developer to codify his restrictions so that when they are violated the user is warned.

Since theory invariants are arbitrary terms, macros may be used to express commonly used restrictions. Because theory invariants are a new idea in ACL2, we have only defined one such macro for illustrative purposes. Executing the event

```
(theory-invariant (incompatible (:rewrite left-to-right)
                                (:rewrite right-to-left)))
```

would subsequently cause a warning message any time the current theory contained both of the two rune^s shown. Of course, incompatible^ is just defined as a macro. Its definition may be inspected with `:pe incompatible`.

Note: If the table^ event is used directly to `:put` a term into the theory invariant table^, be aware that the term must be in translated form. This is enforced by the `value` invariant for `theory-invariant-table`. But the upshot of this is that you will be unable to use macros in theory invariants stored directly with the `:put` table^ event.

6.38 VERIFY-GUARDS

VERIFY-GUARDS verify the guard^s of a function

```
Examples:
(verify-guards flatten)
(verify-guards flatten
               :hints (("Goal" :use (:instance assoc-of-app)))
               :otf-flg t
               :doc "string")

General Form:
(verify-guards name
        :hints        hints
        :otf-flg      otf-flg
        :doc          doc-string)
```

See Section 8.39 [GUARD], page 285 for a general discussion of guards. In the General Form above, `name` is the name of a `:logic`^ function (see Section 8.18 [DEFUN-MODE], page 256) that has not yet had its guard^s verified, each subroutine of which has had its guard^s verified; `hints`^ and `otf-flg`^ are as described in the corresponding `:doc`^ entries; and `doc-string`^, if supplied, is a string **not** beginning with "`:Doc-Section`". The three keyword arguments above are all optional. `Verify-guards` will attempt to prove that the guard^ on the named function implies the guard^s of all of the subroutines called in the body of the function. If successful, `name` is considered to have had its guard^s verified.

If `name` is one of several functions in a mutually recursive clique, `verify-guards` will attempt to verify the guard^s of all of the functions.

Because name is not uniquely associated with the verify-guards event (it necessarily names a previously defined function) the documentation^ string, doc-string^, is not stored in the documentation^ data base. Thus, we actually prohibit doc-string^ from having the form of an ACL2 documentation^ string; see Section 5.5 [DOC-STRING], page 134.

Verify-guards must often be used when the value of a recursive call of a defined function is given as an argument to a subroutine that is guard^ed. An example of such a situation is given below. Suppose app (read "append") has a guard^ requiring its first argument to be a true-listp^. Consider

```
(defun rev (x)
  (declare (xargs :guard (true-listp x)))
  (cond ((endp x) nil)
        (t (app (rev (cdr x)) (list (car x))))))
```

Observe that the value of a recursive call of rev is being passed into a guard^ed subroutine, app. In order to verify the guard^s of this definition we must show that (rev (cdr x)) produces a true-listp^, since that is what the guard^ of app requires. How do we know that (rev (cdr x)) is a true-listp^? The most elegant argument is a two-step one, appealing to the following two lemmas: (1) When x is a true-listp^, (cdr x) is a true-listp^. (2) When z is a true-listp^, (rev z) is a true-listp^. But the second lemma is a generalized property of rev, the function we are defining. This property could not be stated before rev is defined and so is not known to the theorem prover when rev is defined.

Therefore, we might break the admission of rev into three steps: define rev without addressing its guard^ verification, prove some general properties about rev, and then verify the guard^s. This can be done as follows:

```
(defun rev (x)
  (declare (xargs :guard (true-listp x)
                  :verify-guards nil))    ; Note this additional xarg.
  (cond ((endp x) nil)
        (t (app (rev (cdr x)) (list (car x))))))

(defthm true-listp-rev
  (implies (true-listp x2)
           (true-listp (rev x2))))

(verify-guards rev)
```

The ACL2 system can actually admit the original definition of rev, verifying the guard^s as part of the defun^ event. The reason is that, in this particular case, the system's heuristics just happen to hit upon the lemma `true-listp-rev`. But in many more complicated functions it is necessary for the user to formulate the inductively provable properties before guard^ verification is attempted.

6.39 VERIFY-TERMINATION

VERIFY-TERMINATION convert a function from :program mode to :logic mode

```
Examples:
(verify-termination fact)
(verify-termination
 fact
 (declare (xargs :guard (and (integerp x) (>= x 0)))))

General Forms:
(verify-termination fn dcl ... dcl)
(verify-termination (fn1 dcl ... dcl)
                    (fn2 dcl ... dcl)
                    ...)
```

where `fn` and the `fni` are function symbols having `:program`^ mode (see Section 8.18 [DEFUN-MODE], page 256) and all of the `dcl`s are either `declare`^ forms or documentation^ strings. The first form above is an abbreviation for

```
(verify-termination (fn dcl ... dcl))
```

so we limit our discussion to the second form. Each of the `fni` must be in the same clique of mutually recursively defined functions, but not every function in the clique need be among the `fni`.

`Verify-termination` attempts to establish the admissibility of the `fni`. `Verify-termination` retrieves their definitions, creates modified definitions using the `dcl`s supplied above, and resubmits these definitions. You could avoid using `verify-termination` by typing the new definitions yourself. So in that sense, `verify-termination` adds no new functionality. But if you have prototyped your system in `:program`^ mode and tested it, you can use `verify-termination` to resubmit your definitions and change their defun-mode^s to `:logic`^, addings hints^ without having to retype or recopy the code. You can also add or modify guard^s at this time.

The `defun`^ command^ executed by `verify-termination` is obtained by retrieving the `defun`^ (or `mutual-recursion`^) command^ that introduced the clique in question and then possibly modifying each definition as follows. Consider a function, `fn`, in the clique. If `fn` is not among the `fni` above, its definition is left unmodified. Otherwise, `fn` is some `fni` and we modify its definition by inserting into it the corresponding `dcls` listed with `fni` in the arguments to `verify-termination`. In addition, we throw out from the old declarations in `fn` the `:mode` specification and anything that is specified in the new `dcls`.

For example, suppose that `fact` was introduced with:

```
(defun fact (n)
  (declare (type integer n)
           (xargs :mode :program))
  (if (zp n) 1 (* n (fact (1- n))))).
```

Suppose later we do `(verify-termination fact)`. Then the following definition is submitted.

```
(defun fact (n)
  (declare (type integer n))
  (if (zp n) 1 (* n (fact (1- n))))).
```

Observe that this is the same definition as the original one, except the old specification of the `:mode` has been deleted so that the defun-mode^ now defaults to `:logic^`. Although the termination proof succeeds, ACL2 also tries to verify the guard^, because we have (implicitly) provided a guard^, namely `(integerp n)`, for this function. (See Section 8.39 [GUARD], page 285 for a general discussion of guards, and see Section 10.251 [TYPE-SPEC], page 504 for a discussion of how type declarations are used in guards.) Unfortunately, the guard^ verification fails, because the subterm `(zp n)` requires that `n` be nonnegative, as can be seen by invoking `:args zp`. (For a discussion of termination issues relating to recursion on the naturals, see Section 10.258 [ZERO-TEST-IDIOMS], page 508.) So we might want to submit

```
(verify-termination
 fact
 (declare (xargs :guard (and (integerp n) (<= 0 n)))))
```

which will submit the definition

```
(defun fact (n)
  (declare (xargs :guard (and (integerp n) (<= 0 n))))
  (if (zp n) 1 (* n (fact (1- n))))).
```

Observe that the declaration in the `verify-termination` command^ has been inserted into the definition. In fact, this command^ succeeds.

7 HISTORY

HISTORY functions that display or change history

* OOPS:: undo a `:u` or `:ubt^`
* PBT:: print the command^s back through a command^ descriptor
* PC:: print the command^ described by a command^ descriptor
* PCB:: print the command^ block described by a command^ descriptor
* PCB!:: print in full the command^ block described by a command^ descriptor
* PCS:: print the sequence of command^s between two command^ descriptors
* PE:: print the event named by a logical name
* PE!:: print all the events^ named by a logical name
* PF:: print the formula corresponding to the given name
* PL:: print the rules whose top function symbol is the given name
* PR:: print the rules stored by the event with the given name
* PR!:: print rules stored by the command with a given command descriptor
* PUFF:: replace a compound command^ by its immediate subevents
* PUFF*:: replace a compound command^ by its subevents
* U:: undo last command^, without a query
* UBT:: undo the command^s back through a command^ descriptor
* UBT!:: undo command^s, without a query or an error

ACL2 keeps track of the command^s that you have executed that have extended the logic or the rule data base, as by the definition of macros, functions, etc. Using the facilities in this section you can review the sequence of command^s executed so far. For example, you can ask to see the most recently executed command^, or the command^ `10` before that, or the command^ that introduced a given function symbol. You can also undo back through some previous command^, restoring the logical world^ to what it was before the given command^.

The annotations printed in the margin in response to some of these commands (such as 'P', 'L', and 'V') are explained in the documentation for `:pc^`.

Several technical terms are used in the documentation of the history command^s. You must understand these terms to use the command^s. These terms are documented via `:doc^` entries of their own. See Section 8.10 [COMMAND], page 244, see Chapter 6 [EVENTS], page 153, see Section 8.11 [COMMAND-DESCRIPTOR], page 245, and see Section 8.61 [LOGICAL-NAME], page 322.

7.1 OOPS

OOPS undo a `:u` or `:ubt^`

The keyword command^ `:oops` will undo the most recent `:ubt^` (or `:u`, which we here consider just another `:ubt^`). A second `:oops` will undo the next most recent `:ubt^`, a third will undo the `:ubt^` before that one, and a fourth `:oops` will return the logical world^ to its configuration before the first `:oops`.

Consider the logical world (see Section 8.89 [WORLD], page 362) that represents the current extension of the logic and ACL2's rules for dealing with it. The `:ubt^` and `:u` command^s "roll back" to some previous world^ (see Section 7.16 [UBT], page 217). Sometimes these command^s are used to inadvertently undo useful work and user's wish they could "undo the last undo." That is the function provided by `:oops`.

`:Oops` is best described in terms of an implementation. Imagine a ring of four world^s and a marker (∗) indicating the current ACL2 world^:

```
      *
     w0
   /    \
 w3      w1
   \    /
     w2
```

This is called the "kill ring" and it is maintained as follows. When you execute an event the current world^ is extended and the kill ring is not otherwise affected. When you execute `:ubt^` or `:u`, the current world^ marker is moved one step counterclockwise and that world^ in the ring is replaced by the result, say `w0'`, of the `:ubt^` or `:u`.

```
     w0
   /    \
*w0'     w1
   \    /
     w2
```

If you were to execute events^ at this point, `w0'` would be extended and no other changes would occur in the kill ring.

When you execute `:oops`, the marker is moved one step clockwise. Thus the kill ring becomes `* w0 / \ w0' w1 \ / w2` and the current ACL2 world^ is `w0` once again. That is, `:oops` "undoes" the `:ubt^` that produced `w0'` from `w0`. Similarly, a second `:oops` will move the marker to `w1`, undoing the undo that produced `w0` from `w1`. A third `:oops` makes `w2` the current world^. Note however that a fourth `:oops` restores us to the configuration above.

In general, the kill ring contains the current world^ and the three most recent world^s in which a `:ubt^` or `:u` were done.

While `:ubt^` may appear to discard the information in the events^ undone, we can see that the world^ in which the `:ubt^` occurred is still available. No information has been lost about that world^. But `:ubt^` does discard information! `:Ubt^` discards the information necessary to recover from the third most recent `ubt^`! An `:oops`, on the other hand, discards no information, it just selects the next available world^ on the kill ring and doing enough `:oops`es will return you to your starting point.

We can put this another way. You can freely type `:oops` and inspect the world^ that you thus obtain with `:pe^`, `:pc^`, and other history^ command^s. You can repeat this as often as you wish without risking the permanent loss of any information. But you must be more careful typing `:ubt^` or `:u`. While `:oops` makes `:ubt^` seem "safe" because the most recent `:ubt^` can always be undone, information is lost when you execute `:ubt^`.

7.2 PBT

PBT print the command^s back through a command^ descriptor

```
Examples:
:pbt :max      ; print back through the most recent command
:pbt :x        ; print back through the most recent command
:pbt fn        ; print back through the introduction of fn
:pbt 5         ; print back through the fifth command executed
:pbt (:x -4)   ; print back through the most recent five commands
```

See Section 8.11 [COMMAND-DESCRIPTOR], page 245.

`Pbt` takes one argument, a command^ descriptor, and prints the command^s from `:max^` (aka `:x`) through the one described. See Section 8.11 [COMMAND-DESCRIPTOR], page 245 for a description of what a command^ descriptor is. See Section 7.3 [PC], page 205 for a description of the format used to display command^s. `Pbt` will print the command^s that `ubt^` will undo.

7.3 PC

PC print the command^ described by a command^ descriptor

```
Examples:
:pc 3    ; print the third command executed
:pc :max ; print the most recent command
:pc :x   ; print the most recent command
:pc fn   ; print the command that introduced fn
```

See Section 8.11 [COMMAND-DESCRIPTOR], page 245.

`Pc` takes one argument, a command^ descriptor, and prints the command^ identified by that descriptor. See Section 8.11 [COMMAND-DESCRIPTOR], page 245. For example

```
ACL2 !>:pc foo
 LVd     52 (DEFUN FOO (X) X)
```

`Pc` always prints a space first, followed by three (possibly blank) characters^ ("LVd" above) explained below. Then `pc` prints the command^ number, a number uniquely identifying the command^'s position in the sequence of command^s since the beginning of the user's session. Finally, the command^ itself is printed.

While `pc` always prints a space first, some history^ command^s, for example `:pcs`^ and `:pe`^, use the first column of output to delimit a region of command^s or to point to a particular event within a command^.

For example, `:pcs 52 54` will print something like

```
/LVd     52 (DEFUN FOO (X) X)
 LV      53 (DEFUN BAR (X) (CONS X X))
\        54 (DEFTHM FOO-BAR (EQUAL (CAR (BAR X)) (FOO X)))
           : ...
        127 (DEFUN LATEST (X) X)
```

Here, the two slash characters^ in the first column are intended to suggest a bracket delimiting command^s 52 through 54. The last command^ printed by `pcs`^ is always the most recent command^, i.e., the command^ at `:here`, and is separated from the rest of the display by an elipsis if some command^s are omitted.

Similarly, the `:pe`^ command^ will print a particular event within a command^ block and will indicate that event by printing a "`>`^" in the first column. The symbol is intended to be an arrow pointing at the event in question.

For example, `:pe^ true-listp-app` might print:

```
          1 (INCLUDE-BOOK "list-book")
             \
>            (DEFTHM TRUE-LISTP-APP
                     (EQUAL (TRUE-LISTP (APP A B)) (TRUE-LISTP B)))
```

using the arrow to indicate the event itself. The slash printed to connect the command^, `include-book^`, with the event, `defthm^`, is intended to suggest a tree branch indicating that the event is inferior to (and part of) the command^.

The mysterious three characters^ sometimes preceding a command^ have the following interpretations. The first two have to do with the function symbols introduced by the command^ and are blank if no symbols were introduced.

At any time we can classify our function symbols into three disjoint sets, which we will here name with characters^. The "`P`" functions are those in `:program^` mode. The "`L`" functions are those in `:logic^` mode whose guard^s have not been verified. The "`V`" functions are those in `:logic^` mode whose guard^s have been verified. Note that `verify-termination^` and `verify-guards^` cause function symbols to be reclassified. If a command^ introduces function symbols then the first mysterious character indicates the class of the symbols at the time of introduction and the second character indicates the current class of the symbols (if the current class is different from the introductory class).

Thus, the display

```
PLd     52 (DEFUN FOO (X) X)
```

tells us that command^ 52 introduced a `:program^` function but that some command^ after 52 changed its mode to `:logic^` and that the guard^s of `foo` have not been verified. That is, `foo`'s termination has been verified even though it was not verified as part of the command^ that introduced `foo`. Had a subsequent command^ verified the guard^s of `foo`, the display would contain a `V` where the `L` is.

The display

```
P d     52 (DEFUN FOO (X) X)
```

indicates that `foo` was introduced in `:program^` mode and still is in that mode.

The third character indicates the enabled/disabled status of the rune^s introduced by the command^. If the status character is blank then all the rune^s (if any) introduced are enable^d. If the status character is "D" then some rune^s were introduced and they are all disable^d. If the status character is "d" then at least one, but not all, of the rune^s introduced is disable^d. Thus, in the display

```
L d     52 (DEFUN FOO (X) X)
```

we see that some rune^ introduced by command^ 52 is disable^d. As noted in the documentation for rune^, a `defun^` command^ introduces many rune^s, e.g., the axiomatic definition rule, `(:definition fn)`, the executable counterpart rule, `(:executable-counterpart fn)`, and type-prescription^s, `(:type-prescription fn)`. The display above does not say which of the rune^s based on `foo` is disable^d, but it does tell us one of them is.

7.4 PCB

PCB print the command^ block described by a command^ descriptor

```
Examples:
:pcb :max ; print the most recent command block
:pcb :x   ; print the most recent command block
:pcb fn   ; print the command block that introduced fn
:pcb 5    ; print the fifth command block
```

See Section 8.11 [COMMAND-DESCRIPTOR], page 245.

`Pcb` takes one argument, a command^ descriptor, and prints the command^ block of the command^ described. See Section 8.11 [COMMAND-DESCRIPTOR], page 245 for details of command^ descriptors. See Section 7.3 [PC], page 205 for description of the format in which command^s are displayed. The command^ block of a command^ consists of the command^ itself and all of the events^ it created. If the command^ created a single event and that event is in fact the command^ (i.e., if the command^ typed was just an event such as a `defun^` or `defthm^` rather than a macro that expanded to some event forms), then `pcb` just prints the command^. `Pcb` sketches command^ and all of the events^ it created, rather than printing them fully. If you wish to see just the command^, in its entirety, use `pc^`. If you wish to see one of the events^ within the block, in its entirety, use `pe^`. If you wish to see the command^ sketched and all of the events^ it created, in their entirety, use `pcb!^`.

7.5 PCB!

PCB! print in full the command^ block described by a command^ descriptor

```
Examples:
:pcb! :max ; print the most recent command block
:pcb! :x   ; print the most recent command block
:pcb! fn   ; print the command block that introduced fn
:pcb! 5    ; print the fifth command block
```

See Section 8.11 [COMMAND-DESCRIPTOR], page 245.

`Pcb!` takes one argument, a command^ descriptor, and prints the command^ block of the command^ described. Unlike `pcb^`, `pcb!` prints the event forms in full; see Section 7.4 [PCB], page 208 for details.

7.6 PCS

PCS print the sequence of command^s between two command^ descriptors

```
Examples:
:pcs 1 5              ; print commands 1 through 5
:pcs 5 1              ; same as above
:pcs :x (:x -3)       ; print the 3 most recently executed commands
:pcs fn assoc-of-fn   ; print the commands between the one that introduced
                      ; fn and the one that introduced assoc-of-fn
```

`Pcs` takes two arguments, both of which are command^ descriptors, and prints the command^s between them with `pc^`. The order of the two descriptors is irrelevant. See Section 8.11 [COMMAND-DESCRIPTOR], page 245 for a description of command^ descriptors. See Section 7.3 [PC], page 205 for a description of the format in which command^s are displayed.

7.7 PE

PE print the event named by a logical name

```
Example:
:pe fn   ; sketches the command that introduced fn and
```

```
             ; prints in full the event within it that created fn.
```

See Section 8.61 [LOGICAL-NAME], page 322.

`Pe` takes one argument, a logical name, and prints in full the most recent event corresponding to the name. `Pe` also sketches the command^ responsible for that event if the command^ is different from the event itself. See Section 7.3 [PC], page 205 for a description of the format used to display a command^. To remind you that the event is inferior to the command^, i.e., you can only undo the entire command^, not just the event, the event is indented slightly from the command^ and a slash (meant to suggest a tree branch) connects them.

Use `pe!^` if you want all events^ corresponding to a given name; see Section 7.8 [PE!], page 210.

7.8 PE!

PE! print all the events^ named by a logical name

```
Examples:
:pe! name ; prints each event that corresponds to name.
```

`Pe!` takes one argument, a logical name other than `:here`, and prints in full all the events^ for that name. `Pe!` also sketches the command^ responsible for that event in each case that the command^ is different from the event itself. See Section 7.3 [PC], page 205 for a description of the format used to display a command^. To remind you that the event is inferior to the command^, i.e., you can only undo the entire command^, not just the event, the event is indented slightly from the command^ and a slash (meant to suggest a tree branch) connects them.

Use `pe^` if you want only the latest event corresponding to a given name, or if you want to use the logical name `:here`; see Section 7.7 [PE], page 209.

7.9 PF

PF print the formula corresponding to the given name

```
Examples:
:pf (:definition fn) ; prints the definition of fn as an equality
:pf fn               ; same as above
```

```
:pf (:rewrite foo)    ; prints the statement of the rewrite rule foo
:pf foo               ; same as above
```

pf takes one argument, an event name or a rune^, and prints the formula associated with name.

7.10 PL

PL print the rules whose top function symbol is the given name

```
Examples:
:pl foo ; prints the rewrite rules that rewrite some call of foo
```

Pl takes one argument, a function symbol, and displays the lemmas that rewrite some term whose top function symbol is the given one. Note that names of macros are not relevant here; for example, for the rules about `+^` you should give the command `:pl binary-+^`.

In fact the kinds of rules printed by `:pl` are rewrite rules, definition rules, and meta^ rules.

7.11 PR

PR print the rules stored by the event with the given name

```
Examples:

:pr fn ; prints the rules from the definition of fn (including any
       ; :type-prescription rule and :definition rule)

:pr assoc-append ; if assoc-append is a rewrite rule, prints that rule
```

Also see Section 7.12 [PR!], page 212, which is similar but works at the command level instead of at the event level.

Pr takes one argument, a logical name, and prints the rules associated with it. In each case it prints the rune, the current enabled/disabled status, and other appropriate fields from the rule. It may be helpful to read the documentation for various kinds of rules in order to understand the information printed by this command. For example, the information printed for a linear rule might be:

```
Rune:     (:LINEAR ABC)
Status:   Enabled
Hyps:     ((CONSP X))
Concl:    (< (ACL2-COUNT (CAR X)) (ACL2-COUNT X))
Max-term: (ACL2-COUNT (CAR X))
Fixer:    NIL
```

The `hyps` and `concl` fields for this rule are fairly self-explanatory, but it is useful to see Section 14.10 [LINEAR], page 643 to learn about maximal terms (which, as one might guess, are stored under "Max-term"). In order to learn about the "Fixer" one has to know to see Section 14.11 [LINEAR-ALIAS], page 646, which is quite unreasonable. The state of this documentation will surely improve.

Currently, this function does not print congruence rules or equivalence rules.

The expert user might also wish to use `find-rules-of-rune^`. See Section 8.34 [FIND-RULES-OF-RUNE], page 276.

7.12 PR!

PR! print rules stored by the command with a given command descriptor

```
Examples:

:pr! fn ; prints the rules from the definition of fn (including any
        ; :type-prescription rule and :definition rule), as well as all other
        ; rules created by the command that created by fn (which could be
        ; many rules if, for example, fn was defined by an include-book
        ; command).

:pr! :max ; prints all the rules stored by the most recent command
```

Also see Section 7.11 [PR], page 211, which is similar but works at the event level instead of at the command level.

`Pr^` takes one argument, a command descriptor, and prints the rules created by the corresponding event. In each case it prints the rune, the current enabled/disabled status, and other appropriate fields from the rule. See Section 7.11 [PR], page 211 for further details.

7.13 PUFF

PUFF replace a compound command^ by its immediate subevents

```
Example Forms:
ACL2 !>:puff :max
ACL2 !>:puff :x
ACL2 !>:puff 15
ACL2 !>:puff "book"

General Form:
:puff cd
```

where `cd` is a command^ descriptor (see Section 8.11 [COMMAND-DESCRIPTOR], page 245) for a "puffable" command^ (see below). `Puff` replaces the command^ at `cd` by the immediate subevents of the command^, executed as command^s. `Puff` then prints, using `pcs^`, the `puff`ed region.

A command^ is "puffable" if it is an `encapsulate^` command^, an `include-book^` command^, or any command^ other than those consisting of a single primitive event. For example, since `defun^` is a primitive event, a `defun^` command^ is not puffable. But a macro form that expands into several `defun^` events^ is puffable. The only primitive event command^s that are puffable are `encapsulate^` and `include-book^` command^s. A puffable command^ contains (interesting) subevents, namely, the events^ in the body of the `encapsulate^`, in the file of the book included, or in the command^ block.

The puff command^ "lifts" the immediate subevents of the indicated command^ so that they become command^s themselves. The command^ `puff*^` recursively puffs the newly introduced command^s. See Section 7.14 [PUFF*], page 215, which also gives an example illustrating both `puff` and `puff*^`. `Puff` undoes the command^ at `cd` and replaces it by its immediate subevents. Thus, in general the length of the history^ grows when a puff command^ is executed. If `puff` causes an error (see below), the logical world^ remains unchanged from its initial configuration.

The intended use of `puff` is to allow the user access to the events^ "hidden" inside compound command^s. For example, while trying to prove some theorem, `p`, about a constrained function, `fn`, one might find that the `encapsulate^`, `cd`, that introduced `fn` failed to include an important constraint^, `q`. Without `puff`, the only way to proceed is to undo back through `cd`, create a suitable `encapsulate^` that proves and exports `q` as well as the old constraint^s, re-execute the new `encapsulate^`, re-execute the events^ since `cd`, and then try `p` again. Unfortunately, it may be hard to prove `q` and additional events^ may have to be inserted into the `encapsulate^` to prove it. It may also be hard to formulate the "right" `q`, i.e., one that is provable in the `encapsulate^` and provides the appropriate facts for use in the proof of `p`.

Using `puff`, the user can erase the `encapsulate^` at `cd`, replacing it by the events^ in its body. Now the formerly constrained function, `fn`, is defined as its witness. The user can experiment with formulations and proofs of `q` suitable for `p`. Of course, to get into the ultimately desired state^ — where `fn` is constrained rather than defined and `q` is exported by an `encapsulate^` at `cd` — the user must ultimately undo back to `cd` and carry out the more tedious program described above. But by using `puff` it is easier to experiment.

Similar applications of `puff` allow the user of a book to expose the innards of the book as though they had all be typed as command^s. The user might then "partially undo" the book, keeping only some of the events^ in it.

`Puff` operates as follows. First, it determines the list of immediate subevents of the command^ indicated by `cd`. It causes an error if there is only one subevent and that subevent is identical to the command^ — i.e., if the command^ at `cd` is a primitive. Next, `puff` undoes back through the indicated command^. This not only erases the command^ at `cd` but all the command^s executed after it. Finally, `puff` re-executes the subevents of (the now erased) `cd` followed by all the command^s that were executed afterwards.

Observe that the command^s executed after `cd` will generally have higher command^ numbers than they did before the puff. For example, suppose 100 command^s have been executed and that `:puff 80` is then executed. Suppose command^ 80 contains 5 immediate subevents (i.e., is an encapsulation of five events^). Then, after puffing, command^ 80 is the first event of the puffed command^, command^ 81 is the second, and so on; 104 command^s appear to have been executed.

When puffing an `encapsulate^` or `include-book^`, the `local^` command^s are executed. Note that this will replace constrained functions by their witnesses.

Finally, it is impossible to `puff` in the presence of `include-book^` command^s involving certified files that have been altered since they were included. To be specific, suppose `"arith"` is a certified book that has been included in a session. Suppose that after `"arith"` was included, the source file is modified. (This might happen if the user of `"arith"` is not its author and the author happens to be working on a new version of `"arith"` during the same time period.) Now suppose the user tries to `puff` the command^ that included `"arith"`. The attempt to obtain the subevents in `"arith"` will discover that the check sum of `"arith"` has changed and an error will be caused. No change is made in the logical world^. A similar error is caused if, in this same situation, the user tries to puff **any command that occurred before the inclusion** of `"arith"`! That is, `puff` may cause an error and leave the world^ unchanged even if the command^ puffed is not one involving the modified book. This happens because in order to reconstruct the world^ after the puffed command^, `puff` must obtain the events^ in the book and if the book's source file has changed there is no assurance that the reconstructed world^ is the one the user intends.

Warning: We do not detect changes to uncertified books^ that have been included and are then puffed or re-included! The act of including an uncertified book leaves no trace of the check sum of the book. Furthermore, the act prints a warning message disclaiming soundness. In light of this, `:puff` quietly "re-"executes the current contents of the book.

7.14 PUFF*

PUFF* replace a compound command^ by its subevents

```
Example Forms:
ACL2 !>:puff* :max
ACL2 !>:puff* :x
ACL2 !>:puff* 15
ACL2 !>:puff* "book"

General Form:
:puff* cd
```

where `cd` is a command^ descriptor (see Section 8.11 [COMMAND-DESCRIPTOR], page 245) for a "puffable" command^. See Section 7.13 [PUFF], page 212 for the definition of "puffable" and for a description of the basic act of "puffing" a command^. `Puff*` is just the recursive application of puff^. `Puff*` prints the region puff^ed, using `pcs^`.

To puff^ a command^ is to replace it by its immediate subevents, each of which is executed as a command^. To `puff*` a command^ is to replace the command^ by each of its immediate subevents and then to `puff*` each of the puffable command^s among the newly introduced ones.

For example, suppose `"ab"` is a book containing the following

```
(in-package "ACL2")
(include-book "a")
(include-book "b")
```

Suppose that book `"a"` only contained `defun^`s for the functions `a1` and `a2` and that `"b"` only contained `defun^`s for `b1` and `b2`.

Now consider an ACL2 state^ in which only two command^s have been executed, the first being `(include-book "ab")` and the second being `(include-book "c")`. Thus, the relevant part of the display produced by `:pbt`^ 1 would be:

```
1 (INCLUDE-BOOK "ab")
2 (INCLUDE-BOOK "c")
```

Call this state^ the "starting state^" in this example, because we will refer to it several times.

Suppose `:puff 1` is executed in the starting state^. Then the first command^ is replaced by its immediate subevents and `:pbt 1` would show:

```
1 (INCLUDE-BOOK "a")
2 (INCLUDE-BOOK "b")
3 (INCLUDE-BOOK "c")
```

Contrast this with the execution of `:puff* 1` in the starting state^. `Puff*` would first puff^ `(include-book "ab")` to get the state^ shown above. But then it would recursively `puff*` the puffable command^s introduced by the first puff^. This continues recursively as long as any puff^ introduced a puffable command^. The end result of `:puff* 1` in the starting state^ is

```
1 (DEFUN A1 ...)
2 (DEFUN A2 ...)
3 (DEFUN B1 ...)
4 (DEFUN B2 ...)
5 (INCLUDE-BOOK "c")
```

Observe that when `puff*` is done, the originally indicated command^, `(include-book "ab")`, has been replaced by the corresponding sequence of primitive events^. Observe also that puffable command^s elsewhere in the history^, for example, command^ 2 in the starting state^, are not affected (except that their command^ numbers grow as a result of the splicing in of earlier command^s).

7.15 U

U undo last command^, without a query

```
Example:
:u
```

The keyword command^ :u is the same as :ubt^ :max^, but with related queries suppressed appropriately. :Oops^ will undo the last :u. See Section 7.16 [UBT], page 217 and see Section 7.17 [UBT!], page 217.

7.16 UBT

UBT undo the command^s back through a command^ descriptor

```
Examples:
:ubt :max      ; undo back through the most recent command
               ; (which just means undo the most recent command)
:ubt :x        ; same as :ubt :max
:u             ; same as :ubt :max with no questions asked
:ubt fn        ; undo back through the introduction of fn
               ; (including all the other events in fn's block)
:ubt 5         ; undo back through the fifth command executed
:ubt (:max -4) ; undo back through the most recent five commands
:ubt (:x -4)   ; undo back through the most recent five commands
```

See Section 8.11 [COMMAND-DESCRIPTOR], page 245.

ubt takes one argument, a command^ descriptor, and undoes the command^s from :max^ (aka :x) through the one described. See Section 8.11 [COMMAND-DESCRIPTOR], page 245. Pbt^ will print the command^s that ubt will undo. :Oops^ will undo the undo. See Section 7.1 [OOPS], page 203.

It is important to remember that a command^ may create several events^. That is, the command^ that introduces fn1 may also introduce fn2. Undoing the command^ that created either of these will undo them both. The events^ created by a command^ constitute the command^'s "block" and we can only undo entire blocks. Use pcb^ to print the command^ block of a command^ if you wish to see what will be lost by undoing the command^.

Ubt will not undo into "prehistory". :Ubt 1 will undo all of your command^s. But :ubt -5 will cause an error, warning you that :ubt cannot undo system initialization.

7.17 UBT!

UBT! undo command^s, without a query or an error

```
Example:
:ubt! :x-4
```

The keyword command^ `:ubt!` is the same as `:ubt^`, but with related queries suppressed appropriately, and with a guarantee that it is "error-free." More precisely, the error triple returned by `:ubt!` will always have a first component of `nil`. `:Oops^` will undo the last `:ubt!`. See Section 7.16 [UBT], page 217 and see Section 7.15 [U], page 216.

8 MISCELLANEOUS

MISCELLANEOUS a miscellany of documented functions and concepts
(often cited in more accessible documentation^)

* ACCUMULATED-PERSISTENCE:: to get statistics on which rune^s are being tried
* ACKNOWLEDGEMENTS:: some contributors to the well-being of ACL2
* ACL2-COUNT:: a commonly used measure for justifying recursion
* ACL2-CUSTOMIZATION:: customizing ACL2 for a particular user
* APROPOS:: searching `:doc^` and `:more-doc^` text
* ARRAYS:: an introduction to ACL2 arrays.
* BIBLIOGRAPHY:: reports about ACL2
* BREAKS:: Common Lisp breaks
* CHECK-SUM:: assigning "often unique" integers to files and objects
* COMMAND:: forms you type at the top-level, but...
* COMMAND-DESCRIPTOR:: an object describing a particular command^ typed by the user
* CONSTRAINT:: restrictions on certain functions introduced in `encapsulate^` events^
* COPYRIGHT:: ACL2 copyright, license, sponsorship
* COROLLARY:: the corollary formula of a rune^
* CURRENT-PACKAGE:: the package used for reading and printing
* DEFAULT-DEFUN-MODE:: the default defun-mode^ of `defun^`'d functions
* DEFAULT-PRINT-PROMPT:: the default prompt^ printed by `ld^`
* DEFUN-MODE:: determines whether a function definition is a logical act
* DEFUN-MODE-CAVEAT:: functions with defun-mode^ of `:program^` considered unsound
* DEFUNS:: an alternative to `mutual-recursion^`
* DISABLE-FORCING:: to disallow force^d case splits
* DISABLEDP:: determine whether a given name or rune is disabled
* E0-ORD-<:: the well-founded less-than relation on ordinals up to `epsilon-0`
* E0-ORDINALP:: a recognizer for the ordinals up to epsilon-0
* EMBEDDED-EVENT-FORM:: forms that may be embedded in other events^
* ENABLE-FORCING:: to allow force^d case splits
* ENTER-BOOT-STRAP-MODE:: The first millisecond of the Big Bang
* ESCAPE-TO-COMMON-LISP:: escaping to Common Lisp
* EVISCERATE-HIDE-TERMS:: to print `(hide ...)` as `<hidden>`
* EXECUTABLE-COUNTERPART:: a rule for computing the value of a function
* EXIT-BOOT-STRAP-MODE:: the end of pre-history
* FAILED-FORCING:: how to deal with a proof failure^ in a forcing round
* FAILURE:: how to deal with a proof failure
* FIND-RULES-OF-RUNE:: find the rules named rune
* FORCE:: identity function used to force a case split
* FORCING-ROUND:: a section of a proof dealing with force^d assumptions
* GENERALIZED-BOOLEANS:: potential soundness issues related to ACL2 predicates
* GOAL-SPEC:: to indicate where a hint is to be used
* GUARD:: restricting the domain of a function
* HIDE:: hide a term from the rewriter
* HINTS:: advice to the theorem proving process
* I-AM-HERE:: a convenient marker for use with `rebuild^`

* WORMHOLE:: `ld^` without `state^` — a short-cut to a parallel universe
* XARGS:: giving hints^ to `defun^`

Perhaps as the system matures this section will become more structured.

8.1 ACCUMULATED-PERSISTENCE

ACCUMULATED-PERSISTENCE to get statistics on which rune^s are being tried

```
Useful Forms:
(accumulated-persistence t)             ; activate statistics gathering

(show-accumulated-persistence :frames)  ; display statistics ordered by
(show-accumulated-persistence :tries)   ; frames built, times tried,
(show-accumulated-persistence :ratio)   ; or their ratio

(accumulated-persistence nil)           ; deactivate
```

Generally speaking, the more ACL2 knows, the slower it runs. That is because the search space grows with the number of alternative rules. Often, the system tries to apply rules that you have forgotten were even there, if you knew about them in the first place! “Accumulated-persistence” is a statistic (originally developed for Nqthm) that helps you identify the rules that are causing ACL2's search space to explode.

Accumulated persistence tracking can be turned on or off. It is generally off. When on, the system runs about 4 times slower than otherwise! But some useful numbers are collected. When it is turned on, by

```
ACL2 !>(accumulated-persistence t)
```

an accumulation site is initialized and henceforth data about which rules are being tried is accumulated into that site. That accumulated data can be displayed with `show-accumulated-persistence`, as described in detail below. When accumulated persistence is turned off, with `(accumulated-persistence nil)`, the accumulation site is wiped out and the data in it is lost.

The “accumulated persistence” of a rune^ is the number of rune^s the system has attempted to apply (since accumulated persistence was last activated) while the given rune^ was being tried.

Consider a `:rewrite^` rule named `rune^`. For simplicity, let us imagine that `rune^` is tried only once in the period during which accumulated persistence is being monitor^ed. Recall that to apply a rewrite rule we must match the left-hand side of the conclusion to some term we are trying to rewrite, establish the hypotheses of `rune^` by rewriting, and, if successful, then rewrite the right-hand side of the conclusion. We say `rune^` is "being tried" from the time we have matched its left-hand side to the time we have either abandoned the attempt or rewritten its right-hand side. During that period of time other rules might be tried, e.g., to establish the hypotheses. The rules tried while `rune^` is being tried are "billed" to `rune^` in the sense that they are being considered here only because of the demands of `rune^`. Thus, if no other rules are tried during that period, the accumulated persistence of `rune^` is `1` — we "bill" `rune^` once for its own application attempt. If, on the other hand, we tried `10` rules on behalf of that application of `rune^`, then `rune^`'s accumulated persistence would be `11`.

One way to envision accumulated persistence is to imagine that every time a rune^ is tried it is pushed onto a stack. The rules tried on behalf of a given application of a rune^ are thus pushed and popped on the stack above that rune^. A lot of work might be done on its behalf — the stack above the rune^ grows and shrinks repeatedly as the search continues for a way to use the rune^. All the while, the rune^ itself "persists" in the stack, until we finish with the attempt to apply it, at which time we pop it off. The accumulated persistence of a rune^ is thus the number of stack frames built while the given rune^ was on the stack.

Note that accumulated persistence is not concerned with whether the attempt to apply a rune^ is successful. Each of the rules tried on its behalf might have failed and the attempt to apply the rune^ might have also failed. The ACL2 proof script would make no mention of the rune^ or the rules tried on its behalf because they did not contribute to the proof. But time was spent pursuing the possible application of the rune^ and accumulated persistence is a measure of that time.

A high accumulated persistence might come about in two extreme ways. One is that the rule causes a great deal of work every time it is tried. The other is that the rule is "cheap" but is tried very often. We therefore keep track of the number of times each rule is tried as well as its persistence. The ratio between the two is the average amount of work done on behalf of the rule each time it is tried.

When the accumulated persistence totals are displayed by the function `show-accumulated-persistence` we sort them so that the most expensive rune^s are shown first. We can sort according to one of three keys:

```
:frames - the number of frames built on behalf of the rune
:tries  - the number of times the rune was tried
:ratio  - frames built per try
```

The key simply determines the order in which the information is presented. If no argument is supplied to `show-accumulated-persistence`, `:frames` is used.

Note that a rune^ with high accumulated persistence may not actually be the "culprit." For example, suppose `rune1` is reported to have a `:ratio` of `101`, meaning that on the average a hundred and one frames were built each time `rune1` was tried. Suppose `rune2` has a `:ratio` of `100`. It could be that the attempt to apply `rune1` resulted in the attempted application of `rune2` and no other rune^. Thus, in some sense, `rune1` is "cheap" and `rune2` is the "culprit" even though it costs less than `rune1`.

Users are encouraged to think about other meters we could install in ACL2 to help diagnose performance problems.

8.2 ACKNOWLEDGEMENTS

ACKNOWLEDGEMENTS some contributors to the well-being of ACL2

The development of ACL2 was made possible by funding from the U. S. Department of Defense, including ARPA and ONR.

ACL2 was started in August, 1989 by Boyer and Moore working together. They co-authored the first versions of axioms.lisp and basis.lisp, with Boyer taking the lead in the formalization of "state^" and the most primitive io^ functions. Boyer also had a significant hand in the development of the early versions of the files interface-raw.lisp and translate.lisp. For several years, Moore alone was responsible for developing the ACL2 system code, though he consulted often with both Boyer and Kaufmann. In August, 1993, Kaufmann became jointly responsible with Moore for developing the system. Boyer has continued to provide valuable consulting on an informal basis.

Bishop Brock was the heaviest early user of ACL2, and provided many suggestions for improvements. In particular, the `:cases` and `:restrict` hints^ were his idea; he developed an early version of congruence-based reasoning for Nqthm; and he helped in the development of some early books^ about arithmetic.

John Cowles also helped in the development of some early books^ about arithmetic, and also provided valuable feedback and bug reports.

Other early users of ACL2 at Computational Logic, Inc. helped influence its development. In particular, Warren Hunt helped with the port to Macintosh Common Lisp, and Art Flatau and Mike Smith provided useful general feedback.

Mike Smith helped develop the Emacs portion of the implementation of proof trees.

Bill Schelter made some enhancements to akcl (now gcl) that helped to enhance ACL2 performance in that Common Lisp implementation, and more generally, responded helpfully to our bug reports.

Kent Pitman helped in our interaction with the ANSI Common Lisp standardization committee, X3J13.

Regarding the documentation:

Bill Young wrote significant portions of the ACL2-TUTORIAL^ section of the ACL2 documentation, an important task for which we are grateful. He, Bishop Brock, Rich Cohen, and Noah Friedman read over considerable amounts of the documentation, and made many useful comments. Others, particularly Bill Bevier and John Cowles, have also made useful comments on the documentation^.

Art Flatau helped develop the ACL2 markup^ language and translators from that language to Texinfo and HTML.

Laura Lawless provided many hours of help in marking up appropriate parts of the documentation^ in typewriter font.

Noah Friedman developed an Emacs tool that helped us insert "invisible links" into the documentation^, which improve the usability of that documentation under HTML readers such as Mosaic.

Richard Stallman contributed a texinfo patch, to be found in the file `doc/texinfo.tex`.

8.3 ACL2-COUNT

ACL2-COUNT a commonly used measure for justifying recursion

`(Acl2-count x)` returns a nonnegative integer that indicates the "size" of its argument `x`.

All characters^ and symbols have `acl2-count 0`. The `acl2-count` of a string is the number of characters^ in it, i.e., its length. The `acl2-count` of a `cons^` is one greater than the sum of the `acl2-counts` of the `car^` and `cdr^`. The `acl2-count` of an integer is its absolute value. The `acl2-count` of a rational is the sum of the `acl2-counts` of the numerator and denominator. The `acl2-count` of a complex rational is one greater than the sum of the `acl2-counts` of the real and imaginary parts.

8.4 ACL2-CUSTOMIZATION

ACL2-CUSTOMIZATION customizing ACL2 for a particular user

The file `"acl2-customization.lisp"` is automatically loaded, via `ld^`, the first time `lp^` is called in an ACL2 session, provided such a file exists on the current directory. Except for the fact that this `ld^` command is not typed explicitly by you, it is a standard `ld^` command, with one exception: any settings of `ld^` specials are remembered once this call of `ld^` has completed. For example, suppose that you start your customization file with `(set-ld-skip-proofsp t state)`, so that proofs are skipped as it is loaded with `ld^`. Then the `ld^` special `ld-skip-proofsp^` will remain `t` after the `ld^` has completed, causing proofs to be skipped in your ACL2 session, unless your customization file sets this variable back to `nil`, say with `(set-ld-skip-proofsp nil state)`.

The customization file `"acl2-customization.lisp"` actually resides on the connected book directory; see Section 3.4 [CBD], page 97. Except, if that file does not exist, then ACL2 looks for `"acl2-customization.lisp"` on your home directory. If ACL2 does not find that file either, then no customization occurs and `lp^` enters the standard ACL2 read-eval-print loop.

If the customization file exists, it is loaded with `ld^` using the usual default values for the `ld^` specials (see Section 9.7 [LD], page 380). Thus, if an error is encountered, no subsequent forms in the file will be evaluated.

To create a customization file it is recommended that you first give it a name other than `"acl2-customization.lisp"` so that ACL2 does not try to include it prematurely when you next enter `lp^`. Then, while in the uncustomized `lp^`, explicitly invoke `ld^` on your evolving (but renamed) customization file until all forms are successfully evaluated. The same procedure is recommended if for some reason ACL2 cannot successfully evaluate all forms in your customization file: rename your customization file so that ACL2 does not try to `ld^` it automatically and then debug the new file by explicit calls to `ld^`.

When you have created a file that can be loaded with `ld^` without error and that you wish to be your customization file, name it `"acl2-customization.lisp"` and put it on the current directory or in your home directory. The first time after starting up ACL2 that you invoke `(lp)`, ACL2 will automatically load the `"acl2-customization.lisp"` file from the cbd (see Section 3.4 [CBD], page 97) if there is one, and otherwise will load it from your home directory.

Note that if you certify a book after the (automatic) loading of an `acl2-customization` file, the forms in that file **will be part of the portcullis** of the books^ you certify! That is, the forms in your customization file at certification time will be loaded whenever anybody uses the books^ you are certifying. Since customization files generally contain idiosyncratic command^s, you may not want yours to be part of the books^ you create for others. Thus, if you have a customization file then you may want to invoke `:ubt 1` before certifying any books^.

The conventions concerning ACL2 customization are liable to change as we get more experience with the interaction between customization, certification of books^ for others, and routine undoing. For example, at the moment it is regarded as a **feature** of customization that it can be undone but it might be regarded as a bug if you accidentally undo your customization.

8.5 APROPOS

APROPOS searching `:doc^` and `:more-doc^` text

NOTE: The `:docs^` command only makes sense at the terminal.

```
Example:
:Docs "compile" will find all documented topics mentioning the
string "compile"
```

When the `:docs^` command is given a `stringp^` argument it searches the text produced by `:doc^` and `:more-doc^` and lists all the documented topics whose text contains the given string. For purposes of this string matching we ignore distinctions of case and the amount and kind (but not presence) of white space. We also treat hyphen as whitespace.

8.6 ARRAYS

ARRAYS an introduction to ACL2 arrays.

Below we begin a detailed presentation of ACL2 arrays. Related topics:

* AREF1:: access the elements of a 1-dimensional array
* AREF2:: access the elements of a 2-dimensional array
* ARRAY1P:: recognize a 1-dimensional array
* ARRAY2P:: recognize a 2-dimensional array
* ASET1:: set the elements of a 1-dimensional array
* ASET2:: set the elements of a 2-dimensional array
* COMPRESS1:: remove irrelevant pairs from a 1-dimensional array
* COMPRESS2:: remove irrelevant pairs from a 2-dimensional array
* DEFAULT:: return the `:default` from the header^ of a 1- or 2-dimensional array
* DIMENSIONS:: return the `:dimensions` from the header^ of a 1- or 2-dimensional array
* HEADER:: return the header of a 1- or 2-dimensional array
* MAXIMUM-LENGTH:: return the `:maximum-length` from the header^ of an array
* SLOW-ARRAY-WARNING:: a warning issued when arrays^ are used inefficiently

ACL2 provides relatively efficient 1- and 2-dimensional arrays. Arrays are awkward to provide efficiently in an applicative language because the programmer rightly expects to be able to "modify" an array object with the effect of changing the behavior of the element accessing function on that object. This, of course, does not make any sense in an applicative setting. The element accessing function is, after all, a function, and its behavior on a given object is immutable. To "modify" an array object in an applicative setting we must actually produce a new array object. Arranging for this to be done efficiently is a challenge to the implementors of the language. In addition, the programmer accustomed to the von Neumann view of arrays must learn how to use immutable applicative arrays efficiently.

In this note we explain 1-dimensional arrays. In particular, we explain briefly how to create, access, and "modify" them, how they are implemented, and how to program with them. 2-dimensional arrays are dealt with by analogy.

The Logical Description of ACL2 Arrays

An ACL2 1-dimensional array is an object that associates arbitrary objects with certain integers, called "indices." Every array has a dimension, `dim`, which is a positive integer. The indices of an array are the consecutive integers from `0` through `dim-1`. To obtain the object associated with the index `i` in an array `a`, one uses `(aref1 name a i)`. `Name` is a symbol that is irrelevant to the semantics of `aref1`^ but affects the speed with which it computes. We will talk more about array "names" later. To produce a new array object that is like `a` but which associates `val` with index `i`, one uses `(aset1 name a i val)`.

An ACL2 1-dimensional array is actually an alist. There is no special ACL2 function for creating arrays; they are generally built with the standard list processing functions `list^` and `cons^`. However, there is a special ACL2 function, called `compress1^`, for speeding up access to the elements of such an alist. We discuss `compress1^` later.

One element of the alist must be the "header" of the array. The header^ of a 1-dimensional array with dimension `dim` is of the form:

```
(:HEADER :DIMENSIONS (dim)
         :MAXIMUM-LENGTH max
         :DEFAULT obj
         :NAME name).
```

`Obj` may be any object and is called the "default value" of the array. `Max^` must be an integer greater than `dim`. `Name` must be a symbol. The `:default^` and `:name` entries are optional; if `:default^` is omitted, the default value is `nil`. The function `header^`, when given a name and a 1- or 2-dimensional array, returns the header^ of the array. The functions `dimensions^`, `maximum-length^`, and `default^` are similar and return the corresponding fields of the header^ of the array. The role of the `:dimensions^` field is obvious: it specifies the legal indices into the array. The roles played by the `:maximum-length^` and `:default^` fields are described below.

Aside from the header^, the other elements of the alist must each be of the form `(i . val)`, where `i` is an integer and `0 <= i < dim`, and `val` is an arbitrary object.

`(Aref1 name a i)` is guard^ed so that `name` must be a symbol, `a` must be an array and `i` must be an index into `a`. The value of `(aref1 name a i)` is either `(cdr (assoc i a))` or else is the default value of `a`, depending on whether there is a pair in `a` whose `car^` is `i`. Note that `name` is irrelevant to the value of an `aref1^` expression. You might `:pe aref1` to see how simple the definition is.

`(Aset1 name a i val)` is guard^ed analogously to the `aref1^` expression. The value of the `aset1^` expression is essentially `(cons (cons i val) a)`. Again, `name` is irrelevant. Note `(aset1 name a i val)` is an array, `a'`, with the property that `(aref1 name a' i)` is `val` and, except for index `i`, all other indices into `a'` produce the same value as in `a`. Note also that if `a` is viewed as an alist (which it is) the pair "binding" `i` to its old value is in `a'` but "covered up" by the new pair. Thus, the length of an array grows by one when `aset1^` is done.

Because `aset1^` covers old values with new ones, an array produced by a sequence of `aset1^` calls may have many irrelevant pairs in it. The function `compress1^` removes these irrelevant pairs. Thus, `(compress1 name a)` returns an array that is equivalent (vis-a-vis `aref1^`) to `a` but which may be shorter. For technical reasons, the alist returned by `compress1^` may also list the pairs in a different order than listed in `a`.

To prevent arrays from growing excessively long due to repeated `aset1^` operations, `aset1^` actually calls `compress1^` on the new alist whenever the length of the new alist exceeds the `:maximum-length^` entry, `max^`, in the header^ of the array. See the definition of `aset1^` (for example by using `:pe^`). This is primarily just a mechanism for freeing up `cons^` space consumed while doing `aset1^` operations.

This completes the logical description of 1-dimensional arrays. 2-dimensional arrays are analogous. The `:dimensions^` entry of the header^ of a 2-dimensional array should be `(dim1 dim2)`. A pair of indices, `i` and `j`, is legal iff `0 <= i < dim1` and `0 <= j < dim2`. The `:maximum-length^` must be greater than `dim1*dim2`. `Aref2^`, `aset2^`, and `compress2^` are like their counterparts but take an additional `index` argument. Finally, the pairs in a 2-dimensional array are of the form `((i . j) . val)`.

The Implementation of ACL2 Arrays

Very informally speaking, the function `compress1^` "creates" an ACL2 array that provides fast access, while the function `aref1^` "maintains" fast access. We now describe this informal idea more carefully.

`Aref1^` is essentially `assoc^`. If `aref1^` were implemented naively the time taken to access an array element would be linear in the dimension of the array and the number of "assignments" to it (the number of `aset1^` calls done to create the array from the initial alist). This is intolerable; arrays are "supposed" to provide constant-time access and change.

The apparently irrelevant names associated with ACL2 arrays allow us to provide constant-time access and change when arrays are used in "conventional" ways. The implementation of arrays makes it clear what we mean by "conventional."

Recall that array names are symbols. Behind the scenes, ACL2 associates two objects with each ACL2 array name. The first object is called the "semantic value" of the name and is an alist. The second object is called the "raw lisp array" and is a Common Lisp array.

When `(compress1 name alist)` builds a new alist, `a'`, it sets the semantic value of `name` to that new alist. Furthermore, it creates a Common Lisp array and writes into it all of the index/value pairs of `a'`, initializing unassigned indices with the default value. This array becomes the raw lisp array of `name`. `Compress1^` then returns `a'`, the semantic value, as its result, as required by the definition of `compress1^`.

When `(aref1 name a i)` is invoked, `aref1^` first determines whether the semantic value of `name` is `a` (i.e., is `eq^` to the alist `a`). If so, `aref1^` can determine the `i`th element of `a` by invoking Common Lisp's `aref` function on the raw lisp array associated with name. Note that no linear search of the alist `a` is required; the operation is done in constant time and involves retrieval of two global variables, an `eq^` test and `jump`, and a raw lisp array access. In fact, an ACL2 array access of this sort is about 5 times slower than a C array access. On the other hand, if `name` has no semantic value or if it is different from `a`, then `aref1^` determines the answer by linear search of `a` as suggested by the `assoc-like` definition of `aref1^`. Thus, `aref1^` always returns the axiomatically specified result. It returns in constant time if the array being accessed is the current semantic value of the name used. The ramifications of this are discussed after we deal with `aset1^`.

When `(aset1 name a i val)` is invoked, `aset1^` does two `cons^`es to create the new array. Call that array `a'`. It will be returned as the answer. (In this discussion we ignore the case in which `aset1^` does a `compress1^`.) However, before returning, `aset1^` determines if `name`'s semantic value is `a`. If so, it makes the new semantic value of `name` be `a'` and it smashes the raw lisp array of `name` with `val` at index `i`, before returning `a'` as the result. Thus, after doing an `aset1^` and obtaining a new semantic value `a'`, all `aref1^`s on that new array will be fast. Any `aref1^`s on the old semantic value, `a`, will be slow.

To understand the performance implications of this design, consider the chronological sequence in which ACL2 (Common Lisp) evaluates expressions: basically inner-most first, left-to-right, call-by-value. An array use, such as `(aref1 name a i)`, is "fast" (constant-time) if the alist supplied, `a`, is the value returned by the most recently executed `compress1^` or `aset1^` on the name supplied. In the functional expression of "conventional" array processing, all uses of an array are fast.

The `:name` field of the header^ of an array is completely irrelevant. Our convention is to store in that field the symbol we mean to use as the name of the raw lisp array. But no ACL2 function inspects `:name` and its primary value is that it allows the user, by inspecting the semantic value of the array — the alist — to recall the name of the raw array that probably holds that value. We say "probably" since there is no enforcement that the alist was compressed under the name in the header^ or that all `aset`s used that name. Such enforcement would be inefficient.

Some Programming Examples

In the following examples we will use ACL2 "global variables" to hold several arrays. See Section 9.1 [`@`], page 373, and see Section 9.3 [ASSIGN], page 378.

Let the `state^` global variable `a` be the 1-dimensional compressed array of dimension `5` constructed below.

```
ACL2 !>(assign a (compress1 'demo
                            '((:header :dimensions (5)
                                       :maximum-length 15
                                       :default uninitialized
                                       :name demo)
                              (0 . zero))))
```

Then `(aref1 'demo (@ a) 0)` is `zero` and `(aref1 'demo (@ a) 1)` is `uninitialized`.

Now execute

```
ACL2 !>(assign b (aset1 'demo (@ a) 1 'one))
```

Then `(aref1 'demo (@ b) 0)` is `zero` and `(aref1 'demo (@ b) 1)` is `one`.

All of the `aref1^`s done so far have been "fast."

Note that we now have two array objects, one in the global variable `a` and one in the global variable `b`. `B` was obtained by assigning to `a`. That assignment does not affect the alist `a` because this is an applicative language. Thus, `(aref1 'demo (@ a) 1)` must **still** be `uninitialized`. And if you execute that expression in ACL2 you will see that indeed it is. However, a rather ugly comment is printed, namely that this array access is "slow." The reason it is slow is that the raw lisp array associated with the name `demo` is the array we are calling `b`. To access the elements of `a`, `aref1^` must now do a linear search. Any reference to `a` as an array is now "unconventional;" in a conventional language like Ada or Common Lisp it would simply be impossible to refer to the value of the array before the assignment that produced our `b`.

Now let us define a function that counts how many times a given object, `x`, occurs in an array. For simplicity, we will pass in the name and highest index of the array:

```
ACL2 !>(defun cnt (name a i x)
         (declare (xargs :guard
                         (and (array1p name a)
                              (integerp i)
                              (>= i -1)
                              (< i (car (dimensions name a))))
                         :mode :logic
                         :measure (nfix (+ 1 i))))
```

```
              (cond ((zp (1+ i)) 0) ; return 0 if i is at most -1
                    ((equal x (aref1 name a i))
                     (1+ (cnt name a (1- i) x)))
                    (t (cnt name a (1- i) x))))
```

To determine how many times `zero` appears in `(@ b)` we can execute:

```
ACL2 !>(cnt 'demo (@ b) 4 'zero)
```

The answer is `1`. How many times does `uninitialized` appear in `(@ b)`?

```
ACL2 !>(cnt 'demo (@ b) 4 'uninitialized)
```

The answer is `3`, because positions `2`, `3` and `4` of the array contain that default value.

Now imagine that we want to assign `'two` to index `2` and then count how many times the 2nd element of the array occurs in the array. This specification is actually ambiguous. In assigning to `b` we produce a new array, which we might call `c`. Do we mean to count the occurrences in `c` of the 2nd element of `b` or the 2nd element of `c`? That is, do we count the occurrences of `uninitialized` or the occurrences of `two`? If we mean the former the correct answer is `2` (positions `3` and `4` are `uninitialized` in `c`); if we mean the latter, the correct answer is `1` (there is only one occurrence of `two` in `c`).

Below are ACL2 renderings of the two meanings, which we call `[former]` and `[latter]`. (Warning: Our description of these examples, and of an example `[fast former]` that follows, assumes that only one of these three examples is actually executed; for example, they are not executed in sequence. See "A Word of Warning" below for more about this issue.)

```
(cnt 'demo (aset1 'demo (@ b) 2 'two) 4 (aref1 'demo (@ b) 2))  ; [former]

(let ((c (aset1 'demo (@ b) 2 'two)))                          ; [latter]
  (cnt 'demo c 4 (aref1 'demo c 2)))
```

Note that in `[former]` we create `c` in the second argument of the call to `cnt` (although we do not give it a name) and then refer to `b` in the fourth argument. This is unconventional because the second reference to `b` in `[former]` is no longer the semantic value of `demo`. While ACL2 computes the correct answer, namely `2`, the execution of the `aref1^` expression in `[former]` is done slowly.

A conventional rendering with the same meaning is

```
(let ((x (aref1 'demo (@ b) 2)))                          ; [fast former]
  (cnt 'demo (aset1 'demo (@ b) 2 'two) 4 x))
```

which fetches the 2nd element of b before creating c by assignment. It is important to understand that [former] and [fast former] mean exactly the same thing: both count the number of occurrences of uninitialized in c. Both are legal ACL2 and both compute the same answer, 2. Indeed, we can symbolically transform [fast former] into [former] merely by substituting the binding of x for x in the body of the let^. But [fast former] can be evaluated faster than [former] because all of the references to demo use the then-current semantic value of demo, which is b in the first line and c throughout the execution of the cnt in the second line. [Fast former] is the preferred form, both because of its execution speed and its clarity. If you were writing in a conventional language you would have to write something like [fast former] because there is no way to refer to the 2nd element of the old value of b after smashing b unless it had been saved first.

We turn now to [latter]. It is both clear and efficient. It creates c by assignment to b and then it fetches the 2nd element of c, two, and proceeds to count the number of occurrences in c. The answer is 1. [Latter] is a good example of typical ACL2 array manipulation: after the assignment to b that creates c, c is used throughout.

It takes a while to get used to this because most of us have grown accustomed to the peculiar semantics of arrays in conventional languages. For example, in raw lisp we might have written something like the following, treating b as a "global variable":

```
(cnt 'demo (aset 'demo b 2 'two) 4 (aref 'demo b 2))
```

which sort of resembles [former] but actually has the semantics of [latter] because the b from which aref fetches the 2nd element is not the same b used in the aset! The array b is destroyed by the aset and b henceforth refers to the array produced by the aset, as written more clearly in [latter].

A Word of Warning: Users must exercise care when experimenting with [former], [latter] and [fast former]. Suppose you have just created b with the assignment shown above,

```
ACL2 !>(assign b (aset1 'demo (@ a) 1 'one))
```

If you then evaluate [former] in ACL2 it will complain that the aref1^ is slow and compute the answer, as discussed. Then suppose you evaluate [latter] in ACL2. From our discussion you might expect it to execute fast — i.e., issue no complaint. But in fact you will find that it complains repeatedly. The problem is that the evaluation of [former] changed the semantic value of `demo` so that it is no longer `b`. To try the experiment correctly you must make `b` be the semantic value of `demo` again before the next example is evaluated. One way to do that is to execute

```
ACL2 !>(assign b (compress1 'demo (@ b)))
```

before each expression. Because of issues like this it is often hard to experiment with ACL2 arrays at the top-level. We find it easier to write functions that use arrays correctly and efficiently than to so use them interactively.

This last assignment also illustrates a very common use of `compress1^`. While it was introduced as a means of removing irrelevant pairs from an array built up by repeated assignments, it is actually most useful as a way of insuring fast access to the elements of an array.

Many array processing tasks can be divided into two parts. During the first part the array is built. During the second part the array is used extensively but not modified. If your programming^ task can be so divided, it might be appropriate to construct the array entirely with list processing, thereby saving the cost of maintaining the semantic value of the name while few references are being made. Once the alist has stabilized, it might be worthwhile to treat it as an array by calling `compress1^`, thereby gaining constant time access to it.

ACL2's theorem prover uses this technique in connection with its implementation of the notion of whether a rune^ is disable^d or not. Associated with every rune^ is a unique integer `index`, called its "nume." When each rule is stored, the corresponding nume is stored as a component of the rule. Theories^ are lists of rune^s and membership in the "current theory" indicates that the corresponding rule is enable^d. But these lists are very long and membership is a linear-time operation. So just before a proof begins we map the list of rune^s in the current theory into an alist that pairs the corresponding numes with `t`. Then we compress this alist into an array. Thus, given a rule we can obtain its nume (because it is a component) and then determine in constant time whether it is enable^d. The array is never modified during the proof, i.e., `aset1^` is never used in this example. From the logical perspective this code looks quite odd: we have replaced a linear-time membership test with an apparently linear-time `assoc^` after going to the trouble of mapping from a list of rune^s to an alist of numes. But because the alist of numes is an array, the "apparently linear-time `assoc^`" is more apparent than real; the operation is constant-time.

8.6.1 AREF1

AREF1   access the elements of a 1-dimensional array

```
Example Form:
(aref1 'delta1 a (+ i k))

General Form:
(aref1 name alist index)
```

where `name` is a symbol, `alist` is a 1-dimensional array and `index` is a legal index into `alist`. This function returns the value associated with `index` in `alist`, or else the default value of the array. See Section 8.6 [ARRAYS], page 226 for details.

This function executes in virtually constant time if `alist` is in fact the "semantic value" associated with `name` (see Section 8.6 [ARRAYS], page 226). When it is not, `aref1` must do a linear search through `alist`. In that case the correct answer is returned but a **slow array** comment is printed to the comment window. See Section 8.6.13 [SLOW-ARRAY-WARNING], page 241.

8.6.2 AREF2

AREF2 access the elements of a 2-dimensional array

```
Example Form:
(aref2 'delta1 a i j)

General Form:
(aref2 name alist i j)
```

where `name` is a symbol, `alist` is a 2-dimensional array and `i` and `j` are legal indices into `alist`. This function returns the value associated with `(i . j)` in `alist`, or else the default value of the array. See Section 8.6 [ARRAYS], page 226 for details.

This function executes in virtually constant time if `alist` is in fact the "semantic value" associated with `name` (see Section 8.6 [ARRAYS], page 226). When it is not, `aref2` must do a linear search through `alist`. In that case the correct answer is returned but a **slow array** comment is printed to the comment window. See Section 8.6.13 [SLOW-ARRAY-WARNING], page 241.

8.6.3 ARRAY1P

ARRAY1P recognize a 1-dimensional array

```
Example Form:
```

```
(array1p 'delta1 a)

General Form:
(array1p name alist)
```

where `name` and `alist` are arbitrary objects. This function returns `t` if `alist` is a 1-dimensional ACL2 array. Otherwise it returns `nil`. The function operates in constant time if `alist` is the semantic value of `name`. See Section 8.6 [ARRAYS], page 226.

8.6.4 ARRAY2P

ARRAY2P recognize a 2-dimensional array

```
Example Form:
(array2p 'delta1 a)

General Form:
(array2p name alist)
```

where `name` and `alist` are arbitrary objects. This function returns `t` if `alist` is a 2-dimensional ACL2 array. Otherwise it returns `nil`. The function operates in constant time if `alist` is the semantic value of `name`. See Section 8.6 [ARRAYS], page 226.

8.6.5 ASET1

ASET1 set the elements of a 1-dimensional array

```
Example Form:
(aset1 'delta1 a (+ i k) 27)

General Form:
(aset1 name alist index val)
```

where `name` is a symbol, `alist` is a 1-dimensional array named `name`, `index` is a legal index into `alist`, and `val` is an arbitrary object. See Section 8.6 [ARRAYS], page 226 for details. Roughly speaking this function "modifies" `alist` so that the value associated with `index` is `val`. More precisely, it returns a new array, `alist'`, of the same name and dimension as `alist` that, under `aref1^`, is everywhere equal to `alist` except at `index` where the result is `val`. That is, `(aref1` `name alist' i)` is `(aref1 name alist i)` for all legal indices `i` except `index`, where `(aref1 name alist' i)` is `val`.

In order to "modify" `alist`, `aset1` `cons^`es a new pair onto the front. If the length of the resulting alist exceeds the `:maximum-length^` entry in the array header^, `aset1` compresses the array as with `compress1^`.

It is generally expected that the "semantic value" of `name` will be `alist` (see Section 8.6 [ARRAYS], page 226). This function operates in virtually constant time whether this condition is true or not (unless the `compress1^` operation is required). But the value returned by this function cannot be used efficiently by subsequent `aset1` operations unless `alist` is the semantic value of `name` when `aset1` is executed. Thus, if the condition is not true, `aset1` prints a **slow array** warning to the comment window. See Section 8.6.13 [SLOW-ARRAY-WARNING], page 241.

8.6.6 ASET2

ASET2 set the elements of a 2-dimensional array

```
Example Form:
(aset2 'delta1 a i j 27)

General Form:
(aset2 name alist i j val)
```

where `name` is a symbol, `alist` is a 2-dimensional array named `name`, `i` and `j` are legal indices into `alist`, and `val` is an arbitrary object. See Section 8.6 [ARRAYS], page 226 for details. Roughly speaking this function "modifies" `alist` so that the value associated with `(i . j)` is `val`. More precisely, it returns a new array, `alist'`, of the same name and dimension as `alist` that, under `aref2^`, is everywhere equal to `alist` except at `(i . j)` where the result is `val`. That is, `(aref2 name alist' x y)` is `(aref2 name alist x y)` for all legal indices `x y` except `i` and `j` where `(aref2 name alist' i j)` is `val`.

In order to "modify" `alist`, `aset2` `cons^`es a new pair onto the front. If the length of the resulting `alist` exceeds the `:maximum-length^` entry in the array header^, `aset2` compresses the array as with `compress2^`.

It is generally expected that the "semantic value" of `name` will be `alist` (see Section 8.6 [ARRAYS], page 226). This function operates in virtually constant time whether this condition is true or not (unless the `compress2^` operation is required). But the value returned by this function cannot be used efficiently by subsequent aset2 operations unless alist is the semantic value of name when aset2 is executed. Thus, if the condition is not true, aset2 prints a **slow array** warning to the comment window. See Section 8.6.13 [SLOW-ARRAY-WARNING], page 241.

8.6.7 COMPRESS1

COMPRESS1 remove irrelevant pairs from a 1-dimensional array

```
Example Form:
(compress1 'delta1 a)

General Form:
(compress1 name alist)
```

where name is a symbol and alist is a 1-dimensional array named name. See Section 8.6 [ARRAYS], page 226 for details. Logically speaking, this function removes irrelevant pairs from alist, possibly shortening it. The function returns a new array, alist', of the same name and dimension as alist, that, under aref1^, is everywhere equal to alist. That is, (aref1 name alist' i) is (aref1 name alist i), for all legal indices i. Alist' may be shorter than alist and the non-irrelevant pairs may occur in a different order than in alist.

Practically speaking, this function plays an important role in the efficient implementation of aref1^. In addition to creating the new array, alist', compress1 makes that array the "semantic value" of name and allocates a raw lisp array to name. For each legal index, i, that raw lisp array contains (aref1 name alist' i) in slot i. Thus, subsequent aref1^ operations can be executed in virtually constant time provided they are given name and the alist' returned by the most recently executed compress1 or aset1^ on name. See Section 8.6 [ARRAYS], page 226.

8.6.8 COMPRESS2

COMPRESS2 remove irrelevant pairs from a 2-dimensional array

```
Example Form:
(compress2 'delta1 a)

General Form:
(compress2 name alist)
```

where name is a symbol and alist is a 2-dimensional array named name. See Section 8.6 [ARRAYS], page 226 for details. Logically speaking, this function removes irrelevant pairs from alist, possibly shortening it. The function returns a new array, alist', of the same name and dimension as alist, that, under aref2^, is everywhere equal to alist. That is, (aref2 name alist' i j) is (aref2 name alist i j), for all legal indices i and j. Alist' may be shorter than alist and the non-irrelevant pairs may occur in a different order in alist' than in alist.

Practically speaking, this function plays an important role in the efficient implementation of aref2^. In addition to creating the new array, alist', compress2 makes that array the "semantic value" of name and allocates a raw lisp array to name. For all legal indices, i and j, that raw lisp array contains (aref2 name alist' i j) in slot i,j. Thus, subsequent aref2^ operations can be executed in virtually constant time provided they are given name and the alist' returned by the most recently executed compress2 or aset2^ on name. See Section 8.6 [ARRAYS], page 226.

8.6.9 DEFAULT

DEFAULT return the :default from the header^ of a 1- or 2-dimensional array

```
Example Form:
(default 'delta1 a)

General Form:
(default name alist)
```

where name is an arbitrary object and alist is a 1- or 2-dimensional array. This function returns the contents of the :default field of the header^ of alist. When aref1^ or aref2^ is used to obtain a value for an index (or index pair) not bound in alist, the default value is returned instead. Thus, the array alist may be thought of as having been initialized with the default value. default operates in virtually constant time if alist is the semantic value of name. See Section 8.6 [ARRAYS], page 226.

8.6.10 DIMENSIONS

DIMENSIONS return the :dimensions from the header^ of a 1- or 2-dimensional array

```
Example Form:
(dimensions 'delta1 a)

General Form:
```

```
(dimensions name alist)
```

where `name` is arbitrary and `alist` is a 1- or 2-dimensional array. This function returns the dimensions list of the array `alist`. That list will either be of the form `(dim1)` or `(dim1 dim2)`, depending on whether `alist` is a 1- or 2-dimensional array. `Dim1` and `dim2` will be integers and each exceed by 1 the maximum legal corresponding index. Thus, if `dimensions` returns, say, `'(100)` for an array `a` named `'delta1`, then `(aref1 'delta1 a 99)` is legal but `(aref1 'delta1 a 100)` violates the guard^s on `aref1^`. `Dimensions` operates in virtually constant time if `alist` is the semantic value of `name`. See Section 8.6 [ARRAYS], page 226.

8.6.11 HEADER

HEADER return the header of a 1- or 2-dimensional array

```
Example Form:
(header 'delta1 a)

General Form:
(header name alist)
```

where `name` is arbitrary and `alist` is a 1- or 2-dimensional array. This function returns the header of the array `alist`. The function operates in virtually constant time if `alist` is the semantic value of `name`. See Section 8.6 [ARRAYS], page 226.

8.6.12 MAXIMUM-LENGTH

MAXIMUM-LENGTH return the `:maximum-length` from the header^ of an array

```
Example Form:
(maximum-length 'delta1 a)

General Form:
(maximum-length name alist)
```

where `name` is an arbitrary object and `alist` is a 1- or 2-dimensional array. This function returns the contents of the `:maximum-length` field of the header^ of `alist`. Whenever an `aset1^` or `aset2^` would cause the length of the alist to exceed its maximum length, a `compress1^` or `compress2^` is done automatically to remove irrelevant pairs from the array. `Maximum-length` operates in virtually constant time if `alist` is the semantic value of `name`. See Section 8.6 [ARRAYS], page 226.

8.6.13 SLOW-ARRAY-WARNING

SLOW-ARRAY-WARNING a warning issued when arrays^ are used inefficiently

If you use ACL2 arrays^ you may sometimes see a **slow array** warning. We here explain what that warning means and some likely "mistakes" it may signify.

The discussion in the documentation for arrays^ defines what we mean by the semantic value of a name. As noted there, behind the scenes ACL2 maintains the invariant that with some names there is associated a pair consisting of an ACL2 array `alist`, called the semantic value of the name, and an equivalent raw lisp array. Access to ACL2 array elements, as in `(aref1 name alist i)`, is executed in constant time when the array alist is the semantic value of the name, because we can just use the corresponding raw lisp array to obtain the answer. `Aset1`^ and `compress1`^ modify the raw lisp array appropriately to maintain the invariant.

If `aref1`^ is called on a name and alist, and the alist is not the then-current semantic value of the name, the correct result is computed but it requires linear time because the alist must be searched. When this happens, `aref1`^ prints a **slow array** warning message to the comment window. `Aset1`^ behaves similarly because the array it returns will cause the **slow array** warning every time it is used.

From the purely logical perspective there is nothing "wrong" about such use of arrays^ and it may be spurious to print a warning message. But because arrays^ are generally used to achieve efficiency, the **slow array** warning often means the user's intentions are not being realized. Sometimes merely performance expectations are not met; but the message may mean that the functional behavior of the program is different than intended.

Here are some "mistakes" that might cause this behavior. In the following we suppose the message was printed by `aset1`^ about an array named `name`. Suppose the alist supplied `aset1`^ is `alist`.

(1) `Compress1`^ was never called on `name` and `alist`. That is, perhaps you created an alist that is an `array1p`^ and then proceeded to access it with `aref1`^ but never gave ACL2 the chance to create a raw lisp array for it. After creating an alist that is intended for use as an array, you must do `(compress1 name alist)` and pass the resulting `alist'` as the array.

(2) `Name` is misspelled. Perhaps the array was compressed under the name `'delta-1` but accessed under `'delta1`?

(3) An `aset1^` was done to modify `alist`, producing a new array, `alist'`, but you subsequently used `alist` as an array. Inspect all `(aset1 name ...)` occurrences and make sure that the alist modified is never used subsequently (either in that function or any other). It is good practice to adopt the following syntactic style. Suppose the alist you are manipulating is the value of the local variable `alist`. Suppose at some point in a function definition you wish to modify `alist` with `aset1^`. Then write

```
(let ((alist (aset1 name alist i val))) ...)
```

and make sure that the subsequent function body is entirely within the scope of the `let^`. Any uses of `alist` subsequently will refer to the new alist and it is impossible to refer to the old alist. Note that if you write

```
(foo (let ((alist (aset1 name alist i val))) ...)  ; arg 1
     (bar alist))                                  ; arg 2
```

you have broken the rules, because in `arg 1` you have modified `alist` but in `arg 2` you refer to the old value. An appropriate rewriting is to lift the `let^` out:

```
(let ((alist (aset1 name alist alist i val)))
  (foo ...                                         ; arg 1
       (bar alist)))                               ; arg 2
```

Of course, this may not mean the same thing.

(4) A function which takes `alist` as an argument and modifies it with `aset1^` fails to return the modified version. This is really the same as (3) above, but focuses on function interfaces. If a function takes an array `alist` as an argument and the function uses `aset1^` (or a subfunction uses `aset1^`, etc.), then the function probably "ought" to return the result produced by `aset1^`. The reasoning is as follows. If the array is passed into the function, then the caller is holding the array. After the function modifies it, the caller's version of the array is obsolete. If the caller is going to make further use of the array, it must obtain the latest version, i.e., that produced by the function.

8.7 BIBLIOGRAPHY

BIBLIOGRAPHY reports about ACL2

Below is a list of notes and reports pertaining to ACL2.

R. Boyer, M. Kaufmann, and J Moore, "A short note on some advantages of Acl2." CLI Internal Note 215, Jan., 1991. See the files `reports/note-215-acl2-advantages.*` under the ACL2 source directory.

"High-level Correctness of ACL2: A Story." See the file `reports/story.txt` under the ACL2 source directory.

M. Kaufmann and J Moore, "Design Goals for ACL2." CLI Technical Report 101, August, 1994. See the files `reports/acl2-design.*` under the ACL2 source directory.

8.8 BREAKS

BREAKS Common Lisp breaks

```
Example:
Broken at PROVE.  Type :H for Help.
>>:Q

ACL2 !>
```

If a break occurs, e.g. because of a bug in ACL2 or a user interrupt, the break will run a Common Lisp read-eval-print loop, not an ACL2 read-eval-print loop. This may not be obvious if the prompt^s in the two loops are similar. Because you are typing to a Common Lisp evaluator, you must be careful. It is possible to damage your ACL2 state in irreparable ways by executing non-ACL2 Common Lisp. It is even possible to disrupt and render inaccurate the interrupted evaluation of a simple ACL2 expression.

Quitting from the break (as with `:q^` in AKCL) will return to the innermost ACL2 read-eval-print loop. Before the loop is continued, any pending cleanup forms from `acl2-unwind-protect`s are evaluated (unless `acl2::*acl2-panic-exit-flg*` is non-`nil`, in which case no cleanup is done).

If at any time you type the token `#.` to either a raw lisp break or to the ACL2 read-eval-print loop, an abort is executed. Control is passed to the outermost ACL2 read-eval-print loop `(lp)`. Again, unwind-protection cleanup forms are executed first.

8.9 CHECK-SUM

CHECK-SUM assigning "often unique" integers to files and objects

A "check sum" is an integer in some fixed range computed from the printed representation of an object, e.g., the sum, modulo `2**32`, of the ascii codes of all the characters^ in the printed representation.

Ideally, you would like the check sum of an object to be uniquely associated with that object, like a fingerprint. It could then be used as a convenient way to recognize the object in the future: you could remember the check sum (which is relatively small) and when an object is presented to you and alleged to be the special one you could compute its check sum and see if indeed it was. Alas, there are many more objects than check sums (after all, each check sum is an object, and then there's `t`). So you try to design a check sum algorithm that maps similar looking objects far apart, in the hopes that corruptions and counterfeits — which appear to be similar to the object — have different check sums. Nevertheless, the best you can do is a many-to-one map. If an object with a different check sum is presented, you can be positive it is not the special object. But if an object with the same check sum is presented, you have no grounds for positive identification.

The basic check sum algorithm in ACL2 is called `check-sum-obj`, which computes the check sum of an ACL2 object. Roughly speaking, we scan the print representation of the object and, for each character encountered, we multiply the ascii code of the character times its position in the stream (modulo a certain prime) and then add (modulo a certain prime) that into the running sum. This is inaccurate in many senses (for example, we don't always use the ascii code and we see numbers as though they were printed in base 127) but indicates the basic idea.

ACL2 uses check sums to increase security in the books^ mechanism; see Section 3.5 [CERTIFICATE], page 99.

8.10 COMMAND

COMMAND forms you type at the top-level, but...

...the word "command" usually refers to a top-level form whose evaluation produces a new logical world^.

```
Typical commands are:
(defun foo (x) (cons x x))
(defthm consp-foo (consp (foo x)))
(defrec pair (hd . tl) nil)
```

The first two forms are examples of commands that are in fact primitive events^. See Chapter 6 [EVENTS], page 153. `defrec`, on the other hand, is a macro that expands into a `progn^` of several primitive events^. In general, a world^ extending command generates one or more events^.

Both events^ and commands leave landmarks on the world^ that enable us to determine how the given world^ was created from the previous one. Most of your interactions will occur at the command level, i.e., you type commands, you print previous commands, and you undo back through commands. Commands are denoted by command descriptors. See Section 8.11 [COMMAND-DESCRIPTOR], page 245.

8.11 COMMAND-DESCRIPTOR

COMMAND-DESCRIPTOR   an object describing a particular command^ typed by the user

```
Examples:

:max      ; the command most recently typed by the user
:x        ; synonymous with :max
(:x -1)   ; the command before the most recent one
(:x -2)   ; the command before that
:x-2      ; synonymous with (:x -2)
5         ; the fifth command typed by the user
1         ; the first command typed by the user
0         ; the last command of the system initialization
-1        ; the next-to-last initialization command
:min      ; the first command of the initialization
fn        ; the command that introduced the logical name fn
(:search (defmacro foo-bar))
          ; the first command encountered in a search from :max to
          ; 0 that either contains defmacro and foo-bar in the
          ; command form or contains defmacro and foo-bar in some
          ; event within its block.
```

The recorded history^ of your interactions with the top-level ACL2 command^ loop is marked by the command^s you typed that changed the logical world^. Each such command^ generated one or more events^, since the only way for you to change the logical world^ is to execute an event function. See Section 8.10 [COMMAND], page 244 and see Chapter 6 [EVENTS], page 153. We divide history^ into "command^ blocks," grouping together each world^ changing command^ and its events^. A "command^ descriptor" is an object that can be used to describe a particular command^ in the history^ of the ongoing session.

Each command^ is assigned a unique integer called its "command^ number" which indicates the command^'s position in the chronological ordering of all of the command^s ever executed in this session (including those executed to initialize the system). We assign the number 1 to the first command^ you type to ACL2. We assign 2 to the second and so on. The non-positive integers are assigned to "prehistoric" command^s, i.e., the command^s used to initialize the ACL2 system: 0 is the last command^ of the initialization, -1 is the one before that, etc.

The legal command^ descriptors are described below. We use `n` to denote any integer, `sym` to denote any logical name (see Section 8.61 [LOGICAL-NAME], page 322), and `cd` to denote, recursively, any command^ descriptor.

```
 command                    command
descriptor                 described

:max   -- the most recently executed command (i.e., the one with
          the largest command number)
:x     -- synonymous with :max
:x-k   -- synonymous with (:x -k), if k is an integer and k>0
:min   -- the earliest command (i.e., the one with the smallest
          command number and hence the first command of the system
          initialization)
n      -- command number n  (If n is not in the
          range :min<=n<=:max, n is replaced by the nearest of :min
          and :max.)
sym    -- the command that introduced the logical name sym
(cd n) -- the command whose number is n plus the command number of
          the command described by cd
(:search pat cd1 cd2)
          In this command descriptor, pat must be either an atom or
          a true list of atoms and cd1 and cd2 must be command
          descriptors.  We search the interval from cd1 through cd2
          for the first command that matches pat.  Note that if cd1
          occurs chronologically after cd2, the search is
          ``backwards'' through history while if cd1 occurs
          chronologically before cd2, the search is ``forwards''.  A
          backwards search will find the most recent match; a
          forward search will find the chronologically earliest
          match.  A command matches pat if either the command form
          itself or one of the events in the block contains pat (or
          all of the atoms in pat if pat is a list).
(:search pat)
          the command found by (:search pat :max 0), i.e., the most
          recent command matching pat that was part of the user's
          session, not part of the system initialization.
```

8.12 CONSTRAINT

CONSTRAINT restrictions on certain functions introduced in `encapsulate^` events^

Suppose that a given theorem, `thm`, is to be functionally instantiated using a given functional substitution, `alist`, as described in `:DOC^` lemma-instance^. What is the set of proof obligations generated? It is the set of all terms, `tm`, such that (a) `tm` mentions some function symbol in the domain of `alist`, and (b) either `tm` arises from the "constraint" on a function symbol ancestral in `thm` or some `defaxiom^` or (ii) `tm` is the body of a `defaxiom^`. Here, a function symbol is "ancestral" in `thm` if either it occurs in `thm`, or it occurs in the definition of some function symbol that occurs in `thm`, and so on.

The remainder of this note explains what we mean by "constraint" in the words above.

In a certain sense, function symbols are introduced in essentially two ways. The most common way is to use `defun^` (or when there is mutual recursion, `mutual-recursion^` or `defuns^`). There is also a mechanism for introducing "witness functions"; see Section 6.3 [DEFCHOOSE], page 155. The documentation for these events^ describes the axioms they introduce, which we will call here their "definitional axioms." These definitional axioms are generally the constraints on the function symbols that these axioms introduce.

However, when a function symbol is introduced in the scope of an `encapsulate^` event, its constraints may differ from the definitional axioms introduced for it. For example, suppose that a function's definition is `local^` to the `encapsulate^`; that is, suppose the function is introduced in the signature^ of the `encapsulate^`. Then its constraints include, at the least, those non-`local^` theorems and definitions in the `encapsulate^` that mention the function symbol.

Actually, it will follow from the discussion below that if the signature^ is empty for an `encapsulate^`, then the constraint on each of its new function symbols is exactly the definitional axiom introduced for it. Intuitively, we view such `encapsulates` just as we view `include-book^` events^. But the general case, where the signature^ is not empty, is more complicated.

In the discussion that follows we describe in detail exactly which constraints are associated with which function symbols that are introduced in the scope of an `encapsulate^` event. In order to simplify the exposition we make two cuts at it. In the first cut we present an over-simplified explanation that nevertheless captures the main ideas. In the second cut we complete our explanation by explaining how we view certain events^ as being "lifted" out of the `encapsulate^`, resulting in a possibly smaller `encapsulate^`, which becomes the target of the algorithm described in the first cut.

At the end of this note we present an example showing why a more naive approach is unsound.

Finally, before we start our "first cut," we note that constrained functions always have guard^s of T. This makes sense when one considers that a constrained function's "guard^" only appears in the context of a `local^ defun^`, which is skipped. Note also that any information you want "exported" outside an `encapsulate^` event must be there as an explicit definition or theorem. For example, even if a function `foo` has output type `(mv t t)` in its signature^, the system will not know `(true-listp (foo x))` merely on account of this information. Thus, if you are using functions like `foo` (constrained `mv^` functions) in a context where you are verifying guard^s, then you should probably provide a `:type-prescription^` rule for the constrained function, for example, the `:type-prescription^` rule `(true-listp (foo x))`.

First cut at constraint-assigning algorithm. Quite simply, the formulas introduced in the scope of an `encapsulate^` are conjoined, and each function symbol introduced by the `encapsulate^` is assigned that conjunction as its constraint.

Clearly this is a rather severe algorithm. Let us consider two possible optimizations in an informal manner before presenting our second cut.

Consider the (rather artificial) event below. The function `before1` does not refer at all, even indirectly, to the locally-introduced function `sig-fn`, so it is unfortunate to saddle it with constraints about `sig-fn`.

```
(encapsulate
 ((sig-fn (x) t))

 (defun before1 (x)
   (if (consp x)
       (before1 (cdr x))
     x))

 (local (defun sig-fn (x) (cons x x)))

 (defthm sig-fn-prop
   (consp (sig-fn x)))
 )
```

We would like to imagine moving the definition of `before1` to just in front of this `encapsulate^`, as follows.

```
(defun before1 (x)
  (if (consp x)
```

```
      (before1 (cdr x))
    x))

(encapsulate
 ((sig-fn (x) t))

 (local (defun sig-fn (x) (cons x x)))

 (defthm sig-fn-prop
   (consp (sig-fn x)))
 )
```

Thus, we will only assign the constraint `(consp (sig-fn x))`, from the theorem `sig-fn-prop`, to the function `sig-fn`, not to the function `before1`.

More generally, suppose an event in an `encapsulate^` event does not mention any function symbol in the signature^ of the `encapsulate^`, nor any function symbol that mentions any such function symbol, and so on. (We might say that no function symbol from the signature^ is an "ancestor" of any function symbol occurring in the event.) Then we imagine moving the event, so that it appears in front of the `encapsulate^`. We don't actually move it, but we pretend we do when it comes time to assign constraints. Thus, such definitions only introduce definitional axioms as the constraints on the function symbols being defined, and such theorems introduce no constraints.

Once this first optimization is performed, we have in mind a set of "constrained functions." These are the functions introduced in the `encapsulate^` that would remain after moving some of them out, as indicated above. Consider the collection of all formulas introduced by the `encapsulate^`, except the definitional axioms, that mention these constrained functions. So for example, in the event below, no such formula mentions the function symbol `after1`.

```
(encapsulate
 ((sig-fn (x) t))

 (local (defun sig-fn (x) (cons x x)))

 (defthm sig-fn-prop
   (consp (sig-fn x)))

 (defun after1 (x)
   (sig-fn x))
 )
```

We can see that there is really no harm in imagining that we move the definition of `after1` out of the `encapsulate^`, to just after the `encapsulate^`.

We may summarize the observations above as follows, after which we conclude with a more elaborate example.

Second cut at constraint-assigning algorithm. Given an `encapsulate^` event, first move, to just in front of it and in the same order, all definitions and theorems for which none of the functions declared in the signature^ is ancestral. (In fact perform this process recursively, handling nested `encapsulate^`s.) Now collect up all formulas introduced in the `encapsulate^` other than the definitional axioms, and restrict the set of constrained functions to those that are ancestral in at least one such formula. Finally, assign all formulas introduced in the resulting `encapsulate^` as the common constraint on all function symbols that are introduced in the resulting `encapsulate^`. (Thus, we imagine that the definitions of functions that are omitted from this list of function symbols, together with all non-definitional formulas omitted from this list of formulas, are moved outside the `encapsulate^`, to just after it.)

Implementation note. In the implementation we do not actually move events^, but we create constraints that pretend that we did.

Here is an example illustrating our constraint-assigning algorithm. It builds on the preceding examples.

```
(encapsulate
 ((sig-fn (x) t))

 (defun before1 (x)
   (if (consp x)
       (before1 (cdr x))
     x))

 (local (defun sig-fn (x) (cons x x)))

 (defthm sig-fn-prop
   (consp (sig-fn x)))

 (defun during (x)
   (if (consp x)
       x
     (cons (car (sig-fn x))
           17)))

 (defun before2 (x)
   (before1 x))

 (defthm before2-prop
   (atom (before2 x)))
```

```
(defthm during-prop
  (implies (and (atom x)
                (before2 x))
           (equal (car (during x))
                  (car (sig-fn x)))))

(defun after1 (x)
  (sig-fn x))

(defchoose after2 (x) (u)
  (and (< u x) (during x)))
)
```

Only the functions `sig-fn` and `during` receive extra constraints. The functions `before1` and `before2` are viewed as moving in front of the `encapsulate^`, as is the theorem `before2-prop`. The functions `after1` and `after2` are viewed as being moved past the `encapsulate^`. Notice that the formula `(consp (during x))` is a conjunct of the constraint. It comes from the `:type-prescription^` rule deduced during the definition of the function `during`. The implementation reports the following.

```
(SIG-FN X) is axiomatized to return one result.

In addition, we export AFTER2, AFTER1, DURING-PROP, BEFORE2-PROP, BEFORE2,
DURING, SIG-FN-PROP and BEFORE1.

The following constraint is associated with both of the functions DURING
and SIG-FN:

(AND (EQUAL (DURING X)
            (IF (CONSP X)
                X (CONS (CAR (SIG-FN X)) 17)))
     (CONSP (DURING X))
     (CONSP (SIG-FN X))
     (IMPLIES (AND (ATOM X) (BEFORE2 X))
              (EQUAL (CAR (DURING X))
                     (CAR (SIG-FN X)))))
```

We conclude by asking (and to a certain extent, answering) the following question: Isn't there an approach to assigning constraints that avoids over-constraining more simply than our "second cut" above? Perhaps it seems that given an `encapsulate^`, we should simply assign to each locally defined function the theorems exported about that function. If we adopted that simple approach the events below would be admissible.

```
(encapsulate
 ((foo (x) t))
 (local (defun foo (x) x))
 (defun bar (x)
   (foo x))
 (defthm bar-prop
   (equal (bar x) x)
   :rule-classes nil))

(defthm foo-id
  (equal (foo x) x)
  :hints (("Goal" :use bar-prop)))

; The following event is not admissible in ACL2.

(defthm ouch!
  nil
  :rule-classes nil
  :hints
  (("Goal" :use
    ((:functional-instance foo-id
                           (foo (lambda (x) (cons x x))))))))
```

Under the simple approach we have in mind, `bar` is constrained to satisfy both its definition and `bar-prop` because `bar` mentions a function declared in the signatures of the encapsulation. In fact, `bar` is so-constrained in the ACL2 semantics of encapsulation and the first two events above (the `encapsulate` and the consequence that `foo` must be the identity function) are actually admissible. But under the simple approach to assigning constraints, `foo` is unconstrained because no theorem about it is exported. Under that approach, `ouch!` is proveable because `foo` can be instantiated in `foo-id` to a function other than the identity function.

It's tempting to think we can fix this by including definitions, not just theorems, in constraints. But consider the following slightly more elaborate example. The problem is that we need to include as a constraint on `foo` not only the definition of `bar`, which mentions `foo` explicitly, but also `abc`, which has `foo` as an ancestor.

```
(encapsulate
 ((foo (x) t))
 (local (defun foo (x) x))
 (local (defthm foo-prop
          (equal (foo x) x)))
 (defun bar (x)
   (foo x))
 (defun abc (x)
   (bar x))
```

```
 (defthm abc-prop
   (equal (abc x) x)
   :rule-classes nil))

(defthm foo-id
  (equal (foo x) x)
  :hints (("Goal" :use abc-prop)))

; The following event is not admissible in ACL2.

(defthm ouch!
  nil
  :rule-classes nil
  :hints
  (("Goal" :use
    ((:functional-instance foo-id
                           (foo (lambda (x) (cons x x)))
                           (bar (lambda (x) (cons x x))))))))
```

8.13 COPYRIGHT

COPYRIGHT ACL2 copyright, license, sponsorship

"ACL2" is an acronym for "A Computational Logic for Applicative Common Lisp." ACL2 is copyrighted by Computational Logic, Inc. All rights reserved. For copyright and license information, see the file "LICENSE". Also see Section 8.2 [ACKNOWLEDGEMENTS], page 223.

8.14 COROLLARY

COROLLARY the corollary formula of a rune^

This is a low-level system function at the present time. See Section 7.11 [PR], page 211 and see Section 7.12 [PR!], page 212 instead. Also see Chapter 14 [RULE-CLASSES], page 617 for the use of the symbol :corollary in specifying a rule class.

8.15 CURRENT-PACKAGE

CURRENT-PACKAGE the package used for reading and printing

`Current-package` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(current-package state)` and the updater is `(set-current-package val state)`, or more conventionally, `(in-package val)`. The value of `current-package` is actually the string that names the package. (Common Lisp's "package" objects do not exist in ACL2.) The current package must be known to ACL2, i.e., it must be one of the initial packages or a package defined with `defpkg^` by the user.

When printing symbols, the package prefix is displayed if it is not the `current-package` and may be optionally displayed otherwise. Thus, if `current-package` is `"ACL2"` then the symbol `'ACL2::SYMB` may be printed as `SYMB` or `ACL2::SYMB`, while `'MY-PKG::SYMB` must be printed as `MY-PKG::SYMB`. But if `current-package` is `"MY-PKG"` then the former symbol must be printed as `ACL2::SYMB` while the latter may be printed as `SYMB`.

In Common Lisp, `current-package` also affects how objects are read from character streams. Roughly speaking, read and print are inverses if the `current-package` is fixed, so reading from a stream produced by printing an object must produce an equal object.

In ACL2, the situation is more complicated because we never read objects from character streams, we only read them from object "streams" (channels). Logically speaking, the objects in such a channel are fixed regardless of the setting of `current-package`. However, our host file systems do not support the idea of Lisp object files and instead only support character files. So when you open an object input channel to a given (character file) we must somehow convert it to a list of ACL2 objects. This is done by a *deus ex machina* ("a person or thing that appears or is introduced suddenly and unexpectedly and provides a contrived solution to an apparently insoluble difficulty," Webster's Ninth New Collegiate Dictionary). Roughly speaking, the *deus ex machina* determines what sequence of calls to `read-object` will occur in the future and what the `current-package` will be during each of those calls, and then produces a channel containing the sequence of objects produced by an analogous sequence of Common Lisp reads with `*current-package*` bound appropriately for each.

A simple rule suffices to make sane file io^ possible: before you read an object from an object channel to a file created by printing to a character channel, make sure the `current-package` at read-time is the same as it was at print-time.

8.16 DEFAULT-DEFUN-MODE

DEFAULT-DEFUN-MODE the default defun-mode^ of `defun^`'d functions

When a `defun^` is processed and no `:mode xarg` is supplied, the function `default-defun-mode` is used. To find the default defun-mode^ of the current ACL2 world^, type `(default-defun-mode (w state))`. See Section 8.18 [DEFUN-MODE], page 256 for a discussion of defun-mode^s. To change the default defun-mode^ of the ACL2 world^, type one of the keywords `:program^` or `:logic^`.

The default ACL2 prompt^ displays the current default defun-mode^ by showing the character `p` for `:program^` mode, and omitting it for `:logic^` mode; see Section 8.17 [DEFAULT-PRINT-PROMPT], page 255. The default defun-mode^ may be changed using the keyword command^s `:program^` and `:logic^`, which are equivalent to the command^s `(program)` and `(logic)`. Each of these names is documented separately: see Section 6.25 [PROGRAM], page 186 and see Section 6.22 [LOGIC], page 182. The default defun-mode^ is stored in the table^ `acl2-defaults-table^` and hence may also be changed by a `table^` command^. See Section 6.35 [TABLE], page 193 and also see Section 9.2 [ACL2-DEFAULTS-TABLE], page 374. Both mode-changing command^s are events^.

While events^ that change the default defun-mode^ are permitted within an `encapsulate^` or the text of a book, their effects are local in scope to the duration of the encapsulation or inclusion. For example, if the default defun-mode^ is `:logic^` and a book is included that contains the event `(program)`, then subsequent events^ within the book are processed with the default defun-mode^ `:program^`; but when the `include-book^` event completes, the default defun-mode^ will still be `:logic^`. Command^s that change the default defun-mode^ are not permitted inside `local^` forms.

8.17 DEFAULT-PRINT-PROMPT

DEFAULT-PRINT-PROMPT the default prompt^ printed by `ld^`

```
Example prompt:
ACL2 p!s>
```

The prompt^ printed by ACL2 displays the current package, followed by a space, followed by zero or more of the three characters^ as specified below, followed by the character `>^` printed one or more times, reflecting the number of recursive calls of `ld^`. The three characters^ in the middle are as follows:

```
p     ; when (default-defun-mode (w state)) is :program
!     ; when guard checking is on
s     ; when (ld-skip-proofsp state) is t
```

See Section 8.16 [DEFAULT-DEFUN-MODE], page 254, see Section 9.12 [SET-GUARD-CHECKING], page 386, and see Section 8.57 [LD-SKIP-PROOFSP], page 317.

Also see Section 8.54 [LD-PROMPT], page 312 to see how to install your own prompt^.

Here are some examples with `ld-skip-proofsp nil`.

```
ACL2 !>    ; logic mode with guard checking on
ACL2 >     ; logic mode with guard checking off
ACL2 p!>   ; program mode with guard checking on
ACL2 p>    ; program mode with guard checking off
```

Here are some examples with `default-defun-mode^` of `:logic^`.

```
ACL2 >     ; guard checking off, ld-skip-proofsp nil
ACL2 s>    ; guard checking off, ld-skip-proofsp t
ACL2 !>    ; guard checking on, ld-skip-proofsp nil
ACL2 !s>   ; guard checking on, ld-skip-proofsp t
```

8.18 DEFUN-MODE

DEFUN-MODE determines whether a function definition is a logical act

Two "defun-mode^s" are supported, `:program^` and `:logic^`. Roughly speaking, `:program^` mode allows you to prototype a function for execution without any proof burdens, while `:logic^` mode allows you to add a new definitional axiom to the logic. The system comes up in `:logic^` mode. Execution of functions whose defun-mode^ is `:program^` may render ACL2 unsound! See Section 8.19 [DEFUN-MODE-CAVEAT], page 259.

When you define a function in the ACL2 logic, that function can be run on concrete data. But it is also possible to reason deductively about the function because each definition extends the underlying logic with a definitional axiom. To insure that the logic is sound after the addition of this axiom, certain restrictions have to be met, namely that the recursion terminates. This can be quite challenging.

Because ACL2 is a programming^ language, you often may wish simply to program in ACL2. For example, you may wish to define your system and test it, without any logical burden. Or, you may wish to define "utility" functions — functions that are executed to help manage the task of building your system but functions whose logical properties are of no immediate concern. Such functions might be used to generate test data or help interpret the results of tests. They might create files or explore the ACL2 data base. The termination arguments for such functions are an unnecessary burden provided no axioms about the functions are ever used in deductions.

Thus, ACL2 introduces the idea of the "defun-mode^" of a function. The `:mode` keyword of `defun^`'s `declare^ xarg` allows you to specify the defun-mode^ of a given definition. If no `:mode` keyword is supplied, the default defun-mode^ is used; see Section 8.16 [DEFAULT-DEFUN-MODE], page 254.

There are two defun-mode^s, each of which is written as a keyword:

`:program^` — logically undefined but executable outside deductive contexts.

`:logic^` — axiomatically defined as per the ACL2 definitional principle.

It is possible to change the defun-mode^ of a function from `:program^` to `:logic^`. We discuss this below.

We think of functions having `:program^` mode as "dangerous" functions, while functions having `:logic^` mode are "safe." The only requirement enforced on `:program^` mode functions is the syntactic one: each definition must be well-formed ACL2. Naively speaking, if a `:program^` mode function fails to terminate then no harm is done because no axiom is added (so inconsistency is avoided) and some invocations of the function may simply never return. This simplistic justification of `:program^` mode execution is faulty because it ignores the damage that might be caused by "misguarded" functions. See Section 8.19 [DEFUN-MODE-CAVEAT], page 259.

We therefore implicitly describe an imagined implementation of defun-mode^s that is safe and, we think, effective. But please see Section 8.19 [DEFUN-MODE-CAVEAT], page 259.

The default defun-mode^ is `:logic^`. This means that when you `defun^` a function the system will try to prove termination. If you wish to introduce a function of a different defun-mode^ use the `:mode xargs^` keyword. Below we show `fact` introduced as a function in `:program^` mode.

```
(defun fact (n)
  (declare (xargs :mode :program))
  (if (or (not (integerp n)) (= n 0))
      1
    (* n (fact (1- n)))))
```

No axiom is added to the logic as a result of this definition. By introducing `fact` in `:program^` mode we avoid the burden of a termination proof, while still having the option of executing the function. For example, you can type

```
ACL2 !>(fact 3)
```

and get the answer `6`. If you type `(fact -1)` you will get a hard lisp error due to "infinite recursion."

However, the ACL2 theorem prover knows no axioms about `fact`. In particular, if the term `(fact 3)` arises in a proof, the theorem prover is unable to deduce that it is `6`. From the perspective of the theorem prover it is as though `fact` were an undefined function symbol of arity `1`. Thus, modulo certain important issues (see Section 8.19 [DEFUN-MODE-CAVEAT], page 259), the introduction of this function in `:program^` mode does not imperil the soundness of the system — despite the fact that the termination argument for `fact` was omitted — because nothing of interest can be proved about `fact`. Indeed, we do not allow `fact` to be used in logical contexts such as conjectures submitted for proof.

It is possible to convert a function from `:program^` mode to `:logic^` mode at the cost of proving that it is admissible. This can be done by invoking

```
(verify-termination fact)
```

which is equivalent to submitting the `defun^` of `fact`, again, but in `:logic^` mode.

```
(defun fact (n)
  (declare (xargs :mode :logic))
  (if (or (not (integerp n)) (= n 0))
      1
    (* n (fact (1- n)))))
```

This particular event will fail because the termination argument requires that `n` be nonnegative. A repaired `defun^`, for example with `=^` replaced by `<=^`, will succeed, and an axiom about `fact` will henceforth be available.

Technically, `verify-termination^` submits a redefinition of the `:program^` mode function. This is permitted, even when `ld-redefinition-action^` is `nil`, because the new definition is identical to the old (except for its `:mode` and, possibly, other non-logical properties).

See Section 8.39 [GUARD], page 285 for a discussion of how to restrict the execution of functions. Guard^s may be "verified" for functions in `:logic^` mode; see Section 6.38 [VERIFY-GUARDS], page 199.

8.19 DEFUN-MODE-CAVEAT

DEFUN-MODE-CAVEAT functions with defun-mode^ of `:program^` considered unsound

Technically speaking, in the current implementation, the execution of functions having defun-mode^ `:program^` may damage the ACL2 system in a way that renders it unsound. See Section 8.18 [DEFUN-MODE], page 256 for a discussion of defun-mode^s. That discussion describes an imagined implementation that is slightly different from this one. This note explains that the current implementation is open to unsoundness.

For discussion of a different soundness issue that is also related to function execution, see Section 8.37 [GENERALIZED-BOOLEANS], page 281.

The execution of a function having defun-mode^ `:program^` may violate Common Lisp guard^s on the subroutines used. (This may be true even for calls of a function on arguments that satisfy its guard^, because ACL2 has not verified that its guard^ is sufficient to protect its subroutines.) When a guard^ is violated at runtime all bets are off. That is, no guarantees are made either about the answer being "right" or about the continued rationality of the ACL2 system itself.

For example, suppose you make the following `defun^`:

```
(defun crash (i)
  (declare (xargs :mode :program :guard (integerp i)))
  (car i))
```

Note that the declared guard does not in fact adequately protect the subroutines in the body of `crash`; indeed, satisfying the guard to `crash` will guarantee that the `car^` expression is in violation of its guard. Because this function is admitted in `:program^`-mode, no checks are made concerning the suitability of the guard. Furthermore, in the current ACL2 implementation, `crash` is executed directly in Common Lisp. Thus if you call `crash` on an argument satisfying its guard you will cause an erroneous computation to take place.

```
ACL2 !>(crash 7)
```

```
Error: Caught fatal error [memory may be damaged]
...
```

There is no telling how much damage is done by this errant computation. In some lisps your ACL2 job may actually crash back to the operating system. In other lisps you may be able to recover from the "hard error" and resume ACL2 in a damaged but apparently functional image.

THUS, HAVING A FUNCTION WITH DEFUN-MODE^ `:PROGRAM^` IN YOUR SYSTEM ABSOLVES US, THE ACL2 IMPLEMENTORS, FROM RESPONSIBILITY FOR THE SOUNDNESS OF OUR SYSTEM.

Furthermore

ACL2 DOES NOT YET PROVIDE ANY MEANS OF REGAINING ASSURANCES OF SOUNDNESS AFTER THE INTRODUCTION OF A FUNCTION IN `:PROGRAM^` MODE, EVEN IF IT IS ULTIMATELY CONVERTED TO `:LOGIC^` MODE (since its execution could have damaged the system in a way that makes it possible to verify its termination and guard^s unsoundly).

Finally,

THE VAST MAJORITY OF ACL2 SYSTEM CODE IS IN `:PROGRAM^` MODE AND SO ALL BETS ARE OFF FROM BEFORE YOU START!

This hopeless state of current affairs will change, we think. We think we have defined our functions "correctly" in the sense that they can be converted, without "essential" modification, to `:logic^` mode. We think it very unlikely that a mis-guarded function in `:program^` mode (whether ours or yours) will cause unsoundness without some sort of hard lisp error accompanying it. We think that ultimately we can make it possible to execute your functions (interpretively) without risk to the system, even when some have `:program^` mode. In that imagined implementation, code using functions having `:program^` mode would run more slowly, but safely. These functions could be introduced into the logic ex post facto, whereupon the code's execution would speed up because Common Lisp would be allowed to execute it directly. We therefore ask that you simply pretend that this is that imagined implementation, introduce functions in `:program^` mode, use them as convenient and perhaps ultimately introduce some of them in `:logic^` mode and prove their properties. If you use the system this way we can develop (or dismiss) this style of formal system development. BUT BE ON THE LOOKOUT FOR SCREWUPS DUE TO DAMAGE CAUSED BY THE EXECUTION OF YOUR FUNCTIONS HAVING `:PROGRAM^` MODE!

8.20 DEFUNS

DEFUNS   an alternative to `mutual-recursion^`

```
Example:
(DEFUNS
 (evenlp (x)
   (if (consp x) (oddlp (cdr x)) t))
 (oddlp (x)
   (if (consp x) (evenlp (cdr x)) nil)))

General Form:
(DEFUNS defuns-tuple1 ... defuns-tuplen)
```

is equivalent to

```
(MUTUAL-RECURSION
  (DEFUN . defuns-tuple1)
  ...
  (DEFUN . defuns-tuplen))
```

In fact, `defuns` is the more primitive of the two and `mutual-recursion^` is just a macro that expands to a call of `defun^` after stripping off the `defun^` at the `car^` of each argument to `mutual-recursion^`. We provide and use `mutual-recursion^` rather than `defuns` because by leaving the `defun^`s in place, `mutual-recursion^` forms can be processed by the Emacs `tags` program. See Section 6.24 [MUTUAL-RECURSION], page 184.

8.21 DISABLE-FORCING

DISABLE-FORCING   to disallow force^d case splits

```
General Form:
ACL2 !>:disable-forcing   ; disallow forced case splits
```

See Section 8.35 [FORCE], page 277 for a discussion of force^d case splits.

Disable-forcing is a macro that disable^s the executable counterpart of the function symbol `force^`; see Section 8.35 [FORCE], page 277. When you want to disable^ forcing in hints^, use a form such as:

```
:in-theory (disable (:executable-counterpart force))
```

8.22 DISABLEDP

DISABLEDP determine whether a given name or rune is disabled

```
Examples:

:disabledp foo   ; returns a list of all disabled runes whose base
                 ; symbol is foo (see Section 15.10 [RUNE], page 675)
(disabledp 'foo) ; same as above (i.e., :disabledp foo)
:disabledp (:rewrite bar . 1) ; returns t if the indicated rune is
                              ; disabled, else nil
(disabledp (:rewrite bar . 1)); same as immediately above
```

Also see Section 7.11 [PR], page 211, which gives much more information about the rules associated with a given event.

`Disabledp` takes one argument, an event name or a rune. In the former case it returns the list of disabled runes associated with that name (in the sense that the rune's "base symbol" is that name; see Section 15.10 [RUNE], page 675). In the latter case it returns `t` if the given rune is disabled, and `nil` otherwise.

8.23 E0-ORD-<

E0-ORD-< the well-founded less-than relation on ordinals up to `epsilon-0`

If `x` and `y` are both `e0-ordinalp`^s (see Section 8.24 [E0-ORDINALP], page 263) then `(e0-ord-< x y)` is true iff `x` is strictly less than `y`. `e0-ord-<` is well-founded on the `e0-ordinalp`^s. When `x` and `y` are both nonnegative integers, `e0-ord-<` is just the familiar `<`^ relation.

`e0-ord-<` plays a key role in the formal underpinnings of the ACL2 logic. In order for a recursive definition to be admissible it must be proved to "terminate." By terminate we mean that the arguments to the function "get smaller" as the function recurses and this sense of size comparison must be such that there is no "infinitely descending" sequence of ever smaller arguments. That is, the relation used to compare successive arguments must be well-founded on the domain being measured.

The most basic way ACL2 provides to prove termination requires the user to supply (perhaps implicitly) a mapping of the argument tuples into the ordinals with some "measure" expression in such a way that the measures of the successive argument tuples produced by recursion decrease according to the relation `e0-ord-<`. The validity of this method rests on the well-foundedness of `e0-ord-<` on the `e0-ordinalp^`s.

Without loss of generality, suppose the definition in question introduces the function `f`, with one formal parameter `x` (which might be a list of objects). Then we require that there exist a measure expression, `(m x)`, that always produces an `e0-ordinalp^`. Furthermore, consider any recursive call, `(f (d x))`, in the body of the definition. Let `hyps` be the conjunction terms (each of which is either the test of an `if^` in the body or else the negation of such a test) describing the path through the body to the recursive call in question. Then it must be a theorem that

```
(IMPLIES hyps (E0-ORD-< (m (d x)) (m x))).
```

When we say `e0-ord-<` is "well-founded" on the `e0-ordinalp^`s we mean that there is no infinite sequence of `e0-ordinalp^`s such that each is smaller than its predecessor in the sequence. Thus, the theorems that must be proved about `f` when it is introduced establish that it cannot recur forever because each time a recursive call is taken `(m x)` gets smaller. From this, and the syntactic restrictions on definitions, it can be shown (as on page 44 in "A Computational Logic", Boyer and Moore, Academic Press, 1979) that there exists a function satisfying the definition; intuitively, the value assigned to any given `x` by the alleged function is that computed by a sufficiently large machine. Hence, the logic is consistent if the axiom defining `f` is added.

See Section 8.24 [E0-ORDINALP], page 263 for a discussion of the ordinals and how to compare two ordinals.

The definitional principle permits the use of relations other than `e0-ord-<` but they must first be proved to be well-founded on some domain. See Section 14.17 [WELL-FOUNDED-RELATION], page 662. Roughly put, alternative relations are shown well-founded by providing an order-preserving mapping from their domain into the ordinals. See Section 6.16 [DEFUN], page 169 for details on how to specify which well-founded relation is to be used.

8.24 E0-ORDINALP

E0-ORDINALP a recognizer for the ordinals up to epsilon-0

Using the nonnegative integers and lists we can represent the ordinals up to `epsilon-0`. The ACL2 notion of `ordinal` is the same as that found in `nqthm-1992` and both are very similar to the development given in "New Version of the Consistency Proof for Elementary Number Theory" in The Collected Papers of Gerhard Gentzen, ed. M.E. Szabo, North-Holland Publishing Company, Amsterdam, 1969, pp 132-213.

The following essay is intended to provide intuition about ordinals. The truth, of course, lies simply in the ACL2 definitions of `e0-ordinalp` and `e0-ord-<`.

Very intuitively, think of each non-zero natural number as by being denoted by a series of the appropriate number of strokes, i.e.,

```
0             0
1             |
2             ||
3             |||
4             ||||
...           ...
```

Then "`omega`," here written as `w`, is the ordinal that might be written as

```
w             |||||...,
```

i.e., an infinite number of strokes. Addition here is just concatenation. Observe that adding one to the front of `w` in the picture above produces `w` again, which gives rise to a standard definition of `w`: `w` is the least ordinal such that adding another stroke at the beginning does not change the ordinal.

We denote by `w+w` or `w*2` the "`doubly infinite`" sequence that we might write as follows.

```
w*2           |||||... |||||...
```

One way to think of `w*2` is that it is obtained by replacing each stroke in `2 (||)` by `w`. Thus, one can imagine `w*3`, `w*4`, etc., which leads ultimately to the idea of "`w*w`," the ordinal obtained by replacing each stroke in `w` by `w`. This is also written as "`omega` squared" or `w^2`, or:

```
 2
w             |||||... |||||... |||||... |||||... |||||... ...
```

We can analogously construct `w^3` by replacing each stroke in `w` by `w^2` (which, it turns out, is the same as replacing each stroke in `w^2` by `w`). That is, we can construct `w^3` as `w` copies of `w^2`,

```
 3                 2        2        2        2
w                 w  ...  w  ...  w  ...  w ... ...
```

Then we can construct w^4 as w copies of w^3, w^5 as w copies of w^4, etc., ultimately suggesting w^w. We can then stack omegas, i.e., (w^w)^w etc. Consider the "limit" of all of those stacks, which we might display as follows.

```
         .
        .
       .
      w
     w
    w
   w
  w
```

That is epsilon-0.

Below we begin listing some ordinals up to epsilon-0; the reader can fill in the gaps at his or her leisure. We show in the left column the conventional notation, using w as "omega," and in the right column the ACL2 object representing the corresponding ordinal.

```
ordinal            ACL2 representation

0                  0
1                  1
2                  2
3                  3
...                ...
w                  '(1 . 0)
w+1                '(1 . 1)
w+2                '(1 . 2)
...                ...
w*2                '(1 1 . 0)
(w*2)+1            '(1 1 . 1)
...                ...
w*3                '(1 1 1 . 0)
(w*3)+1            '(1 1 1 . 1)
...                ...

 2
w                  '(2 . 0)
...                ...

 2
```

```
w +w*4+3           '(2 1 1 1 1 . 3)
...                 ...

 3
w                  '(3 . 0)
...                 ...


 w
w                  '((1 . 0) . 0)
...                 ...

 w  99
w +w  +4w+3        '((1 . 0) 99 1 1 1 1 . 3)
...                 ...

  2
 w
w                  '((2 . 0) . 0)

...                 ...

  w
 w
w                  '(((1 . 0) . 0) . 0)
...                 ...
```

Observe that the sequence of `e0-ordinalp`s starts with the nonnegative integers. This is convenient because it means that if a term, such as a measure expression for justifying a recursive function (see Section 8.23 [E0-ORD-<], page 262) must produce an `e0-ordinalp` it suffices for it to produce a nonnegative integer.

The ordinals listed above are listed in ascending order. This is the ordering tested by `e0-ord-<^`.

The “`epsilon-0` ordinals” of ACL2 are recognized by the recursively defined function `e0-ordinalp`. The base case of the recursion tells us that positive integers are `epsilon-0` ordinals. Otherwise, an `epsilon-0` ordinal is a `cons^` pair `(o1 . o2)`, where `o1` is a non-0 `epsilon-0` ordinal, `o2` is an `epsilon-0` ordinal, and if `o2` is not an integer then its `car^` (which, by the foregoing, must be an `epsilon-0` ordinal) is no greater than `o1`. Thus, if you think of a (non-integer) `epsilon-0` ordinal as a list, each element is an non-0 `epsilon-0` ordinal, the ordinals are listed in weakly descending order, and the final `cdr^` of the list is an integer.

The function `e0-ord-<^` compares two `epsilon-0` ordinals, `x` and `y`. If both are integers, `e0-ord-<^` is just `x<y`. If one is an integer and the other is a `cons^`, the integer is the smaller. Otherwise,

the ordinals in their car^s are compared recursively and determines which is smaller unless the car^s are equal, in which case the ordinals in their cdr^s are compared.

Fundamental to ACL2 is the fact that e0-ord-<^ is well-founded on epsilon-0 ordinals. That is, there is no "infinitely descending chain" of such ordinals. See Section 8.71 [PROOF-OF-WELL-FOUNDEDNESS], page 332.

8.25 EMBEDDED-EVENT-FORM

EMBEDDED-EVENT-FORM forms that may be embedded in other events^

```
Examples:
(defun hd (x) (if (consp x) (car x) 0))
(local (defthm lemma23 ...))
(progn (defun fn1 ...)
       (local (defun fn2 ...))
       ...)

General Form:
An embedded event form is a term, x, such that

  x is a call of an event function other than DEFPKG (see the
  documentation for `events' for a listing of the event functions);

  x is of the form (LOCAL x1) where x1 is an embedded event form;

  x is of the form (PROGN x1 ... xn), where each xi is an embedded
  event form;

  x is of the form (VALUE &), where & is any term;

  x macroexpands to one of the forms above.
```

An exception: an embedded event form may not set the acl2-defaults-table^ when in the context of local^. Thus for example, the form

```
(local (table acl2-defaults-table :defun-mode :program))
```

is not an embedded event form, nor is the form (local (program)), since the latter sets the acl2-defaults-table^ implicitly. An example at the end of the discussion below illustrates why there is this restriction.

When an embedded event is executed while `ld-skip-proofsp^` is `'include-book^`, those parts of it inside `local^` forms are ignored. Thus,

```
(progn (defun f1 () 1)
       (local (defun f2 () 2))
       (defun f3 () 3))
```

will define `f1`, `f2`, and `f3` when `ld-skip-proofsp^` is `nil` but will define only `f1` and `f3` when `ld-skip-proofsp^` is `'include-book^`.

Discussion:

`Encapsulate^` and `include-book^` place restrictions on the kinds of forms that may be processed. These restrictions insure that the non-local events^ (which will ultimately be processed with `ld-skip-proofs t`) are indeed admissible provided that the sequence of local^ and non-local events^ is admissible when `ld-skip-proofs` is `nil`.

`Local^` permits the hiding of an event or group of events^ in the sense that local^ events^ are processed when we are trying to establish the admissibility of a sequence of embedded events^ but are ignored when we are constructing the world^ produced by assuming that sequence. Thus, for example, a particularly ugly and inefficient `:rewrite^` rule might be made local^ to an encapsulate^ that "exports" a desirable theorem whose proof requires the ugly lemma.

To see why we can't allow just anything in as an embedded event, consider allowing the form

```
(if (ld-skip-proofsp state)
    (defun foo () 2)
    (defun foo () 1))
```

followed by

```
(defthm foo-is-1 (equal (foo) 1)).
```

When we process the events^ with `ld-skip-proofsp^`, `nil` the second `defun^` is executed and the `defthm^` succeeds. But when we process the events^ with `ld-skip-proofsp^ 'include-book^`, the second `defun^` is executed, so that `foo` no longer has the same definition it did when we proved `foo-is-1`. Thus, an invalid formula is assumed when we process the `defthm^` while skipping proofs. Thus, the first form above is not a legal embedded event form.

`Defpkg^` is not allowed because it affects how things are read after it is executed. But all the forms embedded in an event are read before any are executed. That is,

```
(encapsulate nil
             (defpkg "MY-PKG" nil)
             (defun foo () 'my-pkg::bar))
```

makes no sense since `my-pkg::bar` must have been read before the `defpkg^` for `"MY-PKG"` was executed.

Finally, let us elaborate on the restriction mentioned earlier related to the `acl2-defaults-table^`. Consider the following form.

```
(encapsulate
 ()
 (local (program))
 (defun foo (x)
   (if (equal 0 x)
       0
     (1+ (foo (- x))))))
```

See Section 8.60 [LOCAL-INCOMPATIBILITY], page 321 for a discussion of how `encapsulate^` processes event forms. Briefly, on the first pass through the events^ the definition of `foo` will be accepted in `defun^` mode `:program^`, and hence accepted. But on the second pass the form `(local (program))` is skipped because it is marked as local^, and hence `foo` is accepted in `defun^` mode `:logic^`. Yet, no proof has been performed in order to admit `foo`, and in fact, it is not hard to prove a contradiction from this definition!

8.26 ENABLE-FORCING

ENABLE-FORCING to allow force^d case splits

```
General Form:
ACL2 !>:enable-forcing    ; allowed forced case splits
```

See Section 8.35 [FORCE], page 277 for a discussion of force^d case splits.

Enable-forcing is a macro that enable^s the executable counterpart of the function symbol `force^`; see Section 8.35 [FORCE], page 277. When you want to enable^ forcing in hints^, use a form such as:

```
:in-theory (enable (:executable-counterpart force))
```

8.27 ENTER-BOOT-STRAP-MODE

ENTER-BOOT-STRAP-MODE The first millisecond of the Big Bang

ACL2 functions, e.g., `if^`, that show `enter-boot-strap-mode` as their defining command^ are in fact primitives. It is impossible for the system to display defining axioms about these symbols.

`Enter-boot-strap-mode` is a Common Lisp function but not an ACL2 function. It magically creates from `nil` an ACL2 property list world^ that lets us start the boot-strapping process. That is, once `enter-boot-strap-mode` has created its world^, it is possible to process the `defconst^`s, `defun^`s, and `defaxiom^`s, necessary to bring up the rest of the system. Before that world^ is created, the attempt by ACL2 even to translate a `defun^` form, say, would produce an error because `defun^` is undefined.

Several ACL2 functions show `enter-boot-strap-mode` as their defining command^. Among them are `if^`, `cons^`, `car^`, and `cdr^`. These functions are characterized by axioms rather than definitional equations — axioms that in most cases are built into our code and hence do not have any explicit representation among the rules and formulas in the system.

8.28 ESCAPE-TO-COMMON-LISP

ESCAPE-TO-COMMON-LISP escaping to Common Lisp

```
Example:
ACL2 !>:Q
```

There is no Common Lisp escape feature in the `lp^`. This is part of the price of purity. To execute a form in Common Lisp as opposed to ACL2, exit `lp^` with `:q^`, submit the desired forms to the Common Lisp read-eval-print loop, and reenter ACL2 with `(lp)`.

8.29 EVISCERATE-HIDE-TERMS

EVISCERATE-HIDE-TERMS   to print `(hide ...)` as `<hidden>`

```
Example:
(assign eviscerate-hide-terms t)
(assign eviscerate-hide-terms nil)
```

`Eviscerate-hide-terms` is a `state^` global variable whose value is either `t` or `nil`. The variable affects how terms are displayed. If `t`, terms of the form `(hide ...)` are printed as `<hidden>`. Otherwise, they are printed normally.

8.30 EXECUTABLE-COUNTERPART

EXECUTABLE-COUNTERPART   a rule for computing the value of a function

```
Examples:
(:executable-counterpart length)
```

which may be abbreviated in theories^ as

```
(length)
```

Every `defun^` introduces at least two rules used by the theorem prover. Suppose `fn` is the name of a `defun^`'d function. Then `(:definition fn)` is the rune (see Section 15.10 [RUNE], page 675) naming the rule that allows the simplifier to replace calls of `fn` by its instantiated body. `(:executable-counterpart fn)` is the rune^ for the rule for how to evaluate the function on known constants.

When typing theories^ it is convenient to know that `(fn)` is a runic designator that denotes `(:executable-counterpart fn)`. See Chapter 15 [THEORIES], page 667.

If `(:executable-counterpart fn)` is enable^d, then when applications of `fn` to known constants are seen by the simplifier they are computed out by executing the Common Lisp code for `fn` (with the appropriate handling of guard^s). Suppose `fact` is defined as the factorial function. If the executable counterpart rune^ of `fact`, `(:executable-counterpart fact)`, is enable^d when the simplifier encounters `(fact 12)`, then that term will be "immediately" expanded to `479001600`.

Such one-step expansions are sometimes counterproductive because they prevent the anticipated application of certain lemmas about the subroutines of the expanded function. Such computed expansions can be prevented by disabling the executable counterpart rune^ of the relevant function. For example, if `(:executable-counterpart fact)` is disable^d, `(fact 12)` will not be expanded by computation. In this situation, `(fact 12)` may be rewritten to `(* 12 (fact 11))`, using the rule named `(:definition fact)`, provided the system's heuristics permit the introduction of the term `(fact 11)`. Note that lemmas about multiplication may then be applicable (while such lemmas would be inapplicable to `479001600`). In many proofs it is desirable to disable^ the executable counterpart rune^s of certain functions to prevent their expansion by computation. See Section 15.4 [EXECUTABLE-COUNTERPART-THEORY], page 672.

Finally: What do we do about functions that are "constrained" rather than defined, such as the following? (See Section 6.18 [ENCAPSULATE], page 176.)

```
(encapsulate ((foo (x) t))
             (local (defun foo (x) x)))
```

Does `foo` have an executable counterpart? Yes: since the vast majority of functions have sensible executable counterparts, it was decided that **all** functions, even such "constrained" ones, have executable counterparts. We essentially "trap" when such calls are inappropriate. Thus, consider for example:

```
(defun bar (x)
  (if (rationalp x)
      (+ x 1)
    (foo x)))
```

If the term `(bar '3)` is encountered by the ACL2 rewriter during a proof, and if the `:executable-counterpart` of `bar` is enable^d, then it will be invoked to reduce this term to `'4`. However, if the term `(bar 'a)` is encountered during a proof, then since `'a` is not a `rationalp`^ and since the `:executable-counterpart` of `foo` is only a "trap," then this call of the `:executable-counterpart` of `bar` will result in a "trap." In that case, the rewriter will return the term `(hide (bar 'a))` so that it never has to go through this process again. See Section 8.40 [HIDE], page 296.

8.31 EXIT-BOOT-STRAP-MODE

EXIT-BOOT-STRAP-MODE the end of pre-history

`Exit-boot-strap-mode` is the last step in creating the ACL2 world^ in which the user lives. It has command^ number `0`. Command^s before it are part of the system initialization and extend all the way back to `:min^`. Command^s after it are those of the user.

`Exit-boot-strap-mode` is a Common Lisp function but not an ACL2 function. It is called when every `defconst^`, `defun^`, etc., in our source code has been processed under ACL2 and the world^ is all but complete. `exit-boot-strap-mode` has only one job: to signal the completion of the boot-strapping.

8.32 FAILED-FORCING

FAILED-FORCING how to deal with a proof failure^ in a forcing round

See Section 8.36 [FORCING-ROUND], page 279 for a background discussion of the notion of forcing rounds. When a proof fails during a forcing round it means that the "gist" of the proof succeeded but some "technical detail" failed. The first question you must ask yourself is whether the force^d goals are indeed theorems. We discuss the possibilities below.

If you believe the force^d goals are theorems, you should follow the usual methodology for "fixing" failed ACL2 proofs, e.g., the identification of key lemmas and their timely and proper use as rules. See Section 8.33 [FAILURE], page 276 and see Chapter 12 [PROOF-TREE], page 579.

The rules designed for the goals of forcing rounds are often just what is needed to prove the force^d hypothesis at the time it is force^d. Thus, you may find that when the system has been "taught" how to prove the goals of the forcing round no forcing round is needed. This is intended as a feature to help structure the discovery of the necessary rules.

If a hint must be provided to prove a goal in a forcing round, the appropriate "goal specifier" (the string used to identify the goal to which the hint is to be applied) is just the text printed on the line above the formula, e.g., `"[1]Subgoal *1/3''"`. See Section 8.38 [GOAL-SPEC], page 283.

If you solve a forcing problem by giving explicit hints^ for the goals of forcing rounds, you might consider whether you could avoid forcing the assumption in the first place by giving those hints^ in the appropriate places of the main proof. This is one reason that we print out the origins of each force^d assumption. An argument against this style, however, is that an assumption might be force^d in hundreds of places in the main goal and proved only once in the forcing round, so that by delaying the proof you actually save time.

We now turn to the possibility that some goal in the forcing round is not a theorem.

There are two possibilities to consider. The first is that the original theorem has insufficient hypotheses to insure that all the force^d hypotheses are in fact always true. The “fix” in this case is to amend the original conjecture so that it has adequate hypotheses.

A more difficult situation can arise and that is when the conjecture has sufficient hypotheses but they are not present in the forcing round goal. This can be caused by what we call “premature” forcing.

Because ACL2 rewrites from the inside out, it is possible that it will force^ hypotheses while the context is insufficient to establish them. Consider trying to prove `(p x (foo x))`. We first rewrite the formula in an empty context, i.e., assuming nothing. Thus, we rewrite `(foo x)` in an empty context. If rewriting `(foo x)` force^s anything, that force^d assumption will have to be proved in an empty context. This will likely be impossible.

On the other hand, suppose we did not attack `(foo x)` until after we had expanded `p`. We might find that the value of its second argument, `(foo x)`, is relevant only in some cases and in those cases we might be able to establish the hypotheses force^d by `(foo x)`. Our premature forcing is thus seen to be a consequence of our “over eager” rewriting.

Here, just for concreteness, is an example you can try. In this example, `(foo x)` rewrites to `x` but has a force^d hypothesis of `(rationalp x)`. `P` does a case split on that very hypothesis and uses its second argument only when `x` is known to be rational. Thus, the hypothesis for the `(foo x)` rewrite is satisfied. On the false branch of its case split, `p` simplies to `(p1 x)` which can be proved under the assumption that `x` is not rational.

```
(defun p1 (x) (not (rationalp x)))
(defun p (x y)(if (rationalp x) (equal x y) (p1 x)))
(defun foo (x) x)
(defthm foo-rewrite (implies (force (rationalp x)) (equal (foo x) x)))
(in-theory (disable foo))
```

The attempt then to do `(thm (p x (foo x)))` force^s the unprovable goal `(rationalp x)`.

Since all “formulas” are presented to the theorem prover as single terms with no hypotheses (e.g., since `implies^` is a function), this problem would occur routinely were it not for the fact that the theorem prover expands certain “simple” definitions immediately without doing anything that can cause a hypothesis to be force^d. See Section 8.79 [SIMPLE], page 340. This does not solve the problem, since it is possible to hide the propositional structure arbitrarily deeply. For example, one could define `p`, above, recursively so that the test that `x` is rational and the subsequent first "real" use of `y` occurred arbitrarily deeply.

Therefore, the problem remains: what do you do if an impossible goal is force^d and yet you know that the original conjecture was adequately protected by hypotheses?

One alternative is to disable forcing entirely. See Section 8.21 [DISABLE-FORCING], page 261. Another is to disable^ the rule that caused the force^.

A third alternative is to prove that the negation of the main goal implies the force^d hypothesis. For example,

```
(defthm not-p-implies-rationalp
  (implies (not (p x (foo x))) (rationalp x))
  :rule-classes nil)
```

Observe that we make no rules from this formula. Instead, we merely `:use` it in the subgoal where we must establish `(rationalp x)`.

```
(thm (p x (foo x))
     :hints (("Goal" :use not-p-implies-rationalp)))
```

When we said, above, that `(p x (foo x))` is first rewritten in an empty context we were misrepresenting the situation slightly. When we rewrite a literal we know what literal we are rewriting and we implicitly assume it false. This assumption is "dangerous" in that it can lead us to simplify our goal to `nil` and give up — we have even seen people make the mistake of assuming the negation of what they wished to prove and then via a very complicated series of transformations convince themselves that the formula is false. Because of this "tail biting" we make very weak use of the negation of our goal. But the use we make of it is sufficient to establish the force^d hypothesis above.

A fourth alternative is to weaken your desired theorem so as to make explicit the required hypotheses, e.g., to prove

```
(defthm rationalp-implies-main
  (implies (rationalp x) (p x (foo x)))
  :rule-classes nil)
```

This of course is unsatisfying because it is not what you originally intended. But all is not lost. You can now prove your main theorem from this one, letting the `implies^` here provide the necessary case split.

```
(thm (p x (foo x))
     :hints (("Goal" :use rationalp-implies-main)))
```

8.33 FAILURE

FAILURE how to deal with a proof failure

When ACL2 gives up it does not mean that the submitted conjecture is invalid, even if the last formula ACL2 printed in its proof attempt is manifestly false. Since ACL2 sometimes generalize^s the goal being proved, it is possible it adopted an invalid subgoal as a legitimate (but doomed) strategy for proving a valid goal. Nevertheless, conjectures submitted to ACL2 are often invalid and the proof attempt often leads the careful reader to the realization that a hypothesis has been omitted or that some special case has been forgotten. It is good practice to ask yourself, when you see a proof attempt fail, whether the conjecture submitted is actually a theorem.

If you think the conjecture is a theorem, then you must figure out from ACL2's output what you know that ACL2 doesn't about the functions in the conjecture and how to impart that knowledge to ACL2 in the form of rules. Books could be written about this, but they haven't been yet. However, see Chapter 12 [PROOF-TREE], page 579 for a utility that may be very helpful in locating parts of the failed proof that are of particular interest. See also the discussion of how to read Nqthm proofs and how to use Nqthm rules in "A Computational Logic Handbook" by Boyer and Moore (Academic Press, 1988).

If the failure occurred during a forcing round, see Section 8.32 [FAILED-FORCING], page 273.

8.34 FIND-RULES-OF-RUNE

FIND-RULES-OF-RUNE find the rules named rune

```
General Form:
(find-rules-of-rune rune wrld)
```

This function finds all the rules in `wrld` with `:rune^` rune. It returns a list of rules in their internal form (generally as described by the corresponding `defrec`). Decyphering these rules is difficult since one cannot always look at a rule object and decide what kind of record it is without exploiting many system invariants (e.g., that `:rewrite^` runes only name rewrite-rules).

At the moment this function returns `nil` if the rune in question is a `:refinement^` rune, because there is no object representing `:refinement^` rules. (`:refinement^` rules cause changes in the `'coarsenings` properties.) In addition, if the rune is an `:equivalence^` rune, then congruence rules with that rune will be returned — because `:equivalence^` lemmas generate some congruence rules — but the fact that a certain function is now known to be an equivalence relation is not represented by any rule object and so no such rule is returned. (The fact that the function is an equivalence relation is encoded entirely in its presence as a `'coarsening` of `equal^`.)

8.35 FORCE

FORCE identity function used to force a case split

When a hypothesis of a conditional rule has the form `(force hyp)` it is logically equivalent to `hyp` but has a pragmatic effect. In particular, when the rule is considered, the needed instance of the hypothesis, `hyp'`, is assumed and a special case is generated, requiring the system to prove that `hyp'` is true in the current context. The proofs of all such "forced assumptions" are delayed until the successful completion of the main goal. See Section 8.36 [FORCING-ROUND], page 279.

Forcing should only be used on hypotheses that are always expected to be true, such as the guard^s of functions. All the power of the theorem prover is brought to bear on a forced hypothesis and no backtracking is possible. If the `:executable-counterpart^` of the function `force` is disable^d, then no hypothesis is forced. See Section 8.26 [ENABLE-FORCING], page 269 and see Section 8.21 [DISABLE-FORCING], page 261.

It sometimes happens that a conditional rule is not applied because some hypothesis, `hyp`, could not be relieved, even though the required instance of `hyp`, `hyp'`, can be shown true in the context. This happens when insufficient resources are brought to bear on `hyp'` at the time we try to relieve it. A sometimes desirable alternative behavior is for the system to assume `hyp'`, apply the rule, and to generate explicitly a special case to show that `hyp'` is true in the context. This is called "forcing" `hyp`. It can be arranged by restating the rule so that the offending hypothesis, `hyp`, is embedded in a call of `force`, as in `(force hyp)`. By using the `:corollary^` field of the `rule-classes^` entry, a hypothesis can be forced without changing the statement of the theorem from which the rule is derived.

Technically, `force` is just a function of one argument that returns that argument. It is generally enable^d and hence evaporates during simplification. But its presence among the hypotheses of a conditional rule causes case splitting to occur if the hypothesis cannot be conventionally relieved.

Since a forced hypothesis must be provable whenever the rule is otherwise applicable, forcing should be used only on hypotheses that are expected always to be true. A common situation is when the hypothesis is in fact a guard^ (or part of a guard^) of some function involved in the pattern that triggers the rule. Intuitively, if that pattern term occurs in the current conjecture, then its guard^s had better be true, since otherwise nothing is known about the term.

A particularly common situation in which some hypotheses should be forced is in "most general" type-prescription^ lemmas. If a single lemma describes the "expected" type of a function, for all "expected" arguments, then it is probably a good idea to force the hypotheses of the lemma. Thus, every time a term involving the function arises, the term will be given the expected type and its arguments will be required to be of the expected type. In applying this advice it might be wise to avoid forcing those hypotheses that are in fact just type predicates on the arguments, since the routine that applies type-prescription^ lemmas has fairly thorough knowledge of the types of all terms.

`Force` can have the additional benefit of causing the ACL2 typing mechanism to interact with the ACL2 rewriter to establish the hypotheses of type-prescription^ rules. To understand this remark, think of the ACL2 type reasoning system as a rather primitive rule-based theorem prover for questions about Common Lisp types, e.g., "does this expression produce a `consp^`?" "does this expression produce some kind of ACL2 number, e.g., an `integerp^`, a `rationalp^`, or a `complex-rationalp^`?" etc. It is driven by type-prescription^ rules. To relieve the hypotheses of such rules, the type system recursively invokes itself. This can be done for any hypothesis, whether it is "type-like" or not, since any proposition, `p`, can be phrased as the type-like question "does `p` produce an object of type `nil`?" However, as you might expect, the type system is not very good at establishing hypotheses that are not type-like, unless they happen to be assumed explicitly in the context in which the question is posed, e.g., "If `p` produces a `consp^` then does `p` produce `nil`?" If type reasoning alone is insufficient to prove some instance of a hypothesis, then the instance will not be proved by the type system and a type-prescription^ rule with that hypothesis will be inapplicable in that case. But by embedding such hypotheses in `force` expressions you can effectively cause the type system to "punt" them to the rest of the theorem prover. Of course, as already noted, this should only be done on hypotheses that are "always true." In particular, if rewriting is required to establish some hypothesis of a type-prescription^ rule, then the rule will be found inapplicable because the hypothesis will not be established by type reasoning alone.

The ACL2 rewriter uses the type reasoning system as a subsystem. It is therefore possible that the type system will force a hypothesis that the rewriter could establish. Before a forced hypothesis is reported out of the rewriter, we try to establish it by rewriting.

This makes the following surprising behavior possible: A type-prescription^ rule fails to apply because some true hypothesis is not being relieved. The user changes the rule so as to **force** the hypothesis. The system then applies the rule but reports no forcing. How can this happen? The type system "punted" the forced hypothesis to the rewriter, which established it.

Finally, we should mention that the rewriter is never willing to force when there is an `if^` term present in the goal being simplified. Since `and^` and `or^` terms are merely abbreviations for `if^` terms, they also prevent forcing.

8.36 FORCING-ROUND

FORCING-ROUND a section of a proof dealing with force^d assumptions

If ACL2 "force^s" some hypothesis of some rule to be true, it is obliged later to prove the hypothesis. See Section 8.35 [FORCE], page 277. ACL2 delays the consideration of force^d hypotheses until the main goal has been proved. It then undertakes a new round of proofs in which the main goal is essentially the conjunction of all hypotheses force^d in the preceding proof. Call this round of proofs the "Forcing Round." Additional hypotheses may be force^d by the proofs in the Forcing Round. The attempt to prove these hypotheses is delayed until the Forcing Round has been successfully completed. Then a new Forcing Round is undertaken to prove the recently force^d hypotheses and this continues until no hypotheses are force^d. Thus, there is a succession of Forcing Rounds.

The Forcing Rounds are enumerated starting from 1. The Goals and Subgoals of a Forcing Round are printed with the round's number displayed in square brackets. Thus, `"[1]Subgoal 1.3"` means that the goal in question is Subgoal 1.3 of the 1st forcing round. To supply a hint for use in the proof of that subgoal, you should use the goal specifier `"[1]Subgoal 1.3"`. See Section 8.38 [GOAL-SPEC], page 283.

When a round is successfully completed — and for these purposes you may think of the proof of the main goal as being the 0th forcing round — the system collects all of the assumptions force^d by the just-completed round. Here, an assumption should be thought of as an implication, `(implies context hyp)`, where context describes the context in which hyp was assumed true. Before undertaking the proofs of these assumptions, we try to "clean them up" in an effort to reduce the amount of work required. This is often possible because the force^d assumptions are generated by the same rule being applied repeatedly in a given context.

For example, suppose the main goal is about some term `(pred (xtrans i) i)` and that some rule rewriting `pred` contains a force^d hypothesis that the first argument is a `good-inputp`. Suppose that during the proof of Subgoal 14 of the main goal, `(good-inputp (xtrans i))` is force^d in a context in which `i` is an `integerp^` and `x` is a `consp^`. (Note that `x` is irrelevant.) Suppose finally that during the proof of Subgoal 28, `(good-inputp (xtrans i))` is force^d "again," but this time in a context in which `i` is a `rationalp^` and `x` is a `symbolp^`. Since the force^d hypothesis does not mention `x`, we deem the contextual information about `x` to be irrelevant and discard it from both contexts. We are then left with two force^d assumptions: `(implies (integerp i) (good-inputp (xtrans i)))` from Subgoal 14, and `(implies (rationalp i) (good-inputp (xtrans i)))` from Subgoal 28. Note that if we can prove the assumption required by Subgoal 28 we can easily get that for Subgoal 14, since the context of Subgoal 28 is the more general. Thus, in the next forcing round we will attempt to prove just

```
(implies (rationalp i) (good-inputp (xtrans i)))
```

and "blame" both Subgoal 14 and Subgoal 28 of the previous round for causing us to prove this.

By delaying and collecting the `forced` assumptions until the completion of the "main goal" we gain two advantages. First, the user gets confirmation that the "gist" of the proof is complete and that all that remains are "technical details." Second, by delaying the proofs of the force^d assumptions ACL2 can undertake the proof of each assumption only once, no matter how many times it was force^d in the main goal.

In order to indicate which proof steps of the previous round were responsible for which force^d assumptions, we print a sentence explaining the origins of each newly force^d goal. For example,

```
[1]Subgoal 1, below, will focus on
(GOOD-INPUTP (XTRANS I)),
which was forced in
 Subgoal 14, above,
  by applying (:REWRITE PRED-CRUNCHER) to
  (PRED (XTRANS I) I),
 and
 Subgoal 28, above,
  by applying (:REWRITE PRED-CRUNCHER) to
  (PRED (XTRANS I) I).
```

In this entry, "[1]Subgoal 1" is the name of a goal which will be proved in the next forcing round. On the next line we display the force^d hypothesis, call it `x`, which is `(good-inputp (xtrans i))` in this example. This term will be the conclusion of the new subgoal. Since the new subgoal will be printed in its entirety when its proof is undertaken, we do not here exhibit the context in which `x` was force^d. The sentence then lists (possibly a succession of) a goal name from the just-completed round and some step in the proof of that goal that force^d `x`. In the example above we see that Subgoals 14 and 28 of the just-completed proof force^d `(good-inputp (xtrans i))` by applying `(:rewrite pred-cruncher)` to the term `(pred (xtrans i) i)`.

If one were to inspect the theorem prover's description of the proof steps applied to Subgoals 14 and 28 one would find the word "force^d" (or sometimes "forcibly") occurring in the commentary. Whenever you see that word in the output, you know you will get a subsequent forcing round to deal with the hypotheses force^d. Similarly, if at the beginning of a forcing round a rune^ is blamed for causing a force^ in some subgoal, inspection of the commentary for that subgoal will reveal the word "force^d" after the rule name blamed.

Most force^d hypotheses come from within the prover's simplifier. When the simplifier encounters a hypothesis of the form `(force hyp)` it first attempts to establish it by rewriting `hyp` to, say, `hyp'`. If the truth or falsity of `hyp'` is known, forcing is not required. Otherwise, the simplifier actually force^s `hyp'`. That is, the `x` mentioned above is `hyp'`, not `hyp`, when the force^d subgoal was generated by the simplifier.

Once the system has printed out the origins of the newly force^d goals, it proceeds to the next forcing round, where those goals are individually displayed and attacked.

At the beginning of a forcing round, the enable^d structure defaults to the global enable^d structure. For example, suppose some rune^, `rune`, is globally enable^d. Suppose in some event you disable^ the rune^ at `"Goal"` and successfully prove the goal but force^ `"[1]Goal"`. Then during the proof of `"[1]Goal"`, rune^ is enable^d "again." The right way to think about this is that the rune^ is "still" enable^d. That is, it is enable^d globally and each forcing round resumes with the global enable^d structure.

8.37 GENERALIZED-BOOLEANS

GENERALIZED-BOOLEANS potential soundness issues related to ACL2 predicates

The discussion below outlines a potential source of unsoundness in ACL2. Although to our knowledge there is no existing Lisp implementation in which this problem can arise, nevertheless we feel compelled to explain this situation.

Consider for example the question: What is the value of `(equal 3 3)`? According to the ACL2 axioms, it's `t`. And as far as we know, based on considerable testing, the value is `t` in every Common Lisp implementation. However, according the Common Lisp draft proposed ANSI standard, any non-`nil` value could result. Thus for example, `(equal 3 3)` is allowed by the standard to be `17`.

The Common Lisp standard specifies (or soon will) that a number of Common Lisp functions that one might expect to return Boolean values may, in fact, return arbitrary values. Examples of such functions are `equal^`, `rationalp^`, and `<^`; a much more complete list is given below. Indeed, we dare to say that every Common Lisp function that one may believe returns only Booleans is, nevertheless, not specified by the standard to have that property, with the exceptions of the functions `not` and `null`. The standard refers to such arbitrary values as "generalized Booleans," but in fact this terminology makes no restriction on those values. Rather, it merely specifies that they are to be viewed, in an informal sense, as being either `nil` or not `nil`.

This situation is problematic for ACL2, which axiomatizes these functions to return Booleans. The problem is that because (for efficiency and simplicity) ACL2 makes very direct use of compiled Common Lisp functions to support the execution of its functions, there is in principle a potential for unsoundness due to these "generalized Booleans." For example, ACL2's `equal^` function is defined to be Common Lisp's `equal`. Hence if in Common Lisp the form `(equal 3 3)` were to evaluate to 17, then in ACL2 we could prove (using the `:executable-counterpart^` of `equal^`) `(equal (equal 3 3) 17)`. However, ACL2 can also prove `(equal (equal x x) t)`, and these two terms together are contradictory, since they imply (instantiating `x` in the second term by `3`) that `(equal 3 3)` is both equal to `17` and to `t`.

To make matters worse, the standard allows `(equal 3 3)` to evaluate to *different* non-`nil` values every time. That is: `equal` need not even be a function!

Fortunately, no existing Lisp implementation takes advantage of the flexibility to have (most of) its predicates return generalized Booleans, as far as we know. We may implement appropriate safeguards in future releases of ACL2, in analogy to (indeed, probably extending) the existing safeguards by way of guards (see Section 8.39 [GUARD], page 285). For now, we'll sleep a bit better knowing that we have been up-front about this potential problem.

The following list of functions contains all those that are defined in Common Lisp to return generalized Booleans but are assumed by ACL2 to return Booleans. In addition, the functions `acl2-numberp^` and `complex-rationalp^` are directly defined in terms of respective Common Lisp functions `numberp` and `complexp`.

```
/=
<
=
alpha-char-p
atom
char-equal
char<
char<=
char>
char>=
characterp
consp
digit-char-p
endp
eq
eql
equal
evenp
integerp
keywordp
listp
logbitp
logtest
lower-case-p
minusp
oddp
plusp
rationalp
standard-char-p
string-equal
string<
string<=
string>
string>=
stringp
subsetp
symbolp
upper-case-p
zerop
```

8.38 GOAL-SPEC

GOAL-SPEC to indicate where a hint is to be used

```
Examples:
"Goal"
"goal"
"Subgoal *1/3''"
"subgoal *1/3''"
"[2]Subgoal *1/3''"
```

When hints^ are given to the theorem prover, a goal-spec is provided to specify the goal to which the hints^ are to be applied. The hints^ provided are carried along innocuously until the named goal arises. When it arises, the hints^ are "activated" for that goal and its descendents.

A legal goal specification may be extracted from the theorem prover's output. Certain lines clearly label formulas, as in

```
Subgoal *1/3.2'
(IMPLIES ... ...)
```

and these lines all give rise to goal specifications. In general, these lines all start either with "Goal" or "Subgoal" or else with those words preceded by a number in square brackets, as in

```
[1]Subgoal *1/3.2'
(IMPLIES ... ...).
```

A goal specification may be obtained by deleting any surrounding whitespace from such a line and embedding the text in string quotation marks. Thus

```
"[1]Subgoal *1/3.2'"
```

is the goal specifier for the goal above.

As noted, a hint is applied to a goal when the hint's goal specification matches the name ACL2 assigns to the goal. The matching algorithm is case-insensitive. Thus, alternative goal specifications for the goal above are `"[1]subgoal *1/3.2'"` and `"[1]SUBGOAL *1/3.2'"`. The matching algorithm does not tolerate non-case discrepancies. Thus, `"[1]Subgoal*1/3.2'"` and `" [1]Subgoal *1/3.2'"` are unacceptable.

Sometimes a formula is given two names, e.g.,

```
Subgoal *1/14.2'
(IMPLIES ...
         ...)
Name the formula above *1.1.
```

It is the first name (the one that starts with "Goal" or "Subgoal") and not the second which constitutes a legal goal-spec. Roughly speaking, when the system prints the line containing the goal specification, it activates any hints^ that are attached to that goal-spec. Consider the example above. Suppose `Subgoal *1/14.2'` could be proved by using a certain lemma instance. Then the appropriate entry in the hints^ would be:

```
("Subgoal *1/14.2'" :use ...)
```

This might surprise you because the system appears to do nothing to `*1/14.2'` besides push it for a subsequent induction. But actually between the time the system printed the goal-spec line and the time it decides to push the goal, you can think of the system as trying everything it has. So a `use` hint activated when `Subgoal *1/14.2'` arises is just what you want.

But what if you want to give an `:induct` hint? By the time induction is tried, the formula has been given the name `*1.1`. Well, this is one you just have to remember:

```
("Subgoal *1/14.2'" :induct ...).
```

When the above hint is activated the `:induct` directive short-circuits the rest of the processing and sends immediately the formula into the pool of goals to prove by induction. The induct hint is attached to the formula in the pool and when the time comes to turn our attention to that goal, the induct advice is followed.

8.39 GUARD

GUARD   restricting the domain of a function

The ACL2 system provides a mechanism for restricting a function definition to a particular domain. Although such restrictions, which are called "guards," are actually ignored by the ACL2 *logic*, they can be useful as a specification device or as a means of causing the execution of definitions directly in Common Lisp. To make such a restriction, use the `:guard xarg` to `defun^`. See Section 8.91 [XARGS], page 369 for a discussion of how to use `:guard`. The general issue of guards and guard verification is discussed in the topics listed below.

Related topics other than immediate subtopics:

To begin further discussion of guards, see Section 8.39.1 [GUARD-INTRODUCTION], page 286. That section directs the reader to further sections for more details. To see just a summary of some command^s related to guards, see Section 8.39.3 [GUARD-QUICK-REFERENCE], page 288.

8.39.1 GUARD-INTRODUCTION

GUARD-INTRODUCTION   introduction to guard^s in ACL2

Most users can probably profit by avoiding dealing with guards most of the time. If they seem to get in the way, they can be "turned off" using the command `:set-guard-checking^ nil`; for more about this, see Section 9.12 [SET-GUARD-CHECKING], page 386. For more about guards in general, see Section 8.39 [GUARD], page 285.

The guard on a function symbol is a formula about the formals of the function. To see the guard on a function, use the keyword command `:args^`. See Section 5.2 [ARGS], page 132. To specify the guard on a function at `defun-time`, use the `:guard^ xarg`. See Section 8.91 [XARGS], page 369.

Guards can be seen as having either of two roles: (a) they are a specification device allowing you to characterize the kinds of inputs a function "should" have, or (b) they are an efficiency device allowing logically defined functions to be executed directly in Common Lisp. Briefly: If the guards of a function definition are "verified" (see Section 6.38 [VERIFY-GUARDS], page 199), then the evaluation of a call of that function on arguments satisfying its guard will have the following property:

> All subsequent function calls during that evaluation will be on arguments satisfying the guard of the called function.

The consequence of this fact for (a) is that your specification function is well-formed, in the sense that the values returned by this function on appropriate arguments only depend on the restrictions of the called functions to their intended domains. The consequence of this fact for (b) is that in the ACL2 system, when a function whose guards have been verified is called on arguments that satisfy its guard, then the raw lisp function defined by this functions's `defun^` event is used to evaluate the call. Note however that even when the user-supplied `defun^` is not used, ACL2 uses a corresponding "executable counterpart" that generally performs, we expect, nearly as well as the raw lisp function. See Section 9.5 [COMP], page 379 to see how compilation^ can speed up both kinds of execution.

Let us turn next to the issue of the relationship between guards and evaluation. See Section 8.39.4 [GUARDS-AND-EVALUATION], page 290.

8.39.2 GUARD-MISCELLANY

GUARD-MISCELLANY miscellaneous remarks about guards

The discussion of guards concludes here with a few miscellaneous remarks. (Presumably you found this documentation by following a link; see Section 8.39.5 [GUARDS-FOR-SPECIFICATION], page 295.) For further information related to guards other than what you find under "guard^," see any of the following documentation topics: guard-example^, `set-verify-guards-eagerness^`, `set-guard-checking^`, and `verify-guards^`.

`Defun^` can be made to try to verify the guards on a function. This is controlled by the "defun-mode^" of the `defun^`; see Section 8.18 [DEFUN-MODE], page 256. The defun-mode^ is either as specified with the `:mode xarg` of the `defun^` or else defaults to the default defun-mode^. See Section 8.16 [DEFAULT-DEFUN-MODE], page 254. If the defun-mode^ of the `defun^` is `:logic^` and either a `:guard^` is specified or `:verify-guards^ t` is specified in the `xargs^`, then we attempt to verify the guards of the function. Otherwise we do not.

It is sometimes impossible for the system to verify the guards of a recursive function at definition time. For example, the guard conjectures might require the invention and proof of some inductively derived property of the function (as often happens when the value of a recursive call is fed to a guarded subroutine). So sometimes it is necessary to define the function using `:verify-guards nil` then to state and prove key theorems about the function, and only then have the system attempt guard verification. Post-`defun^` guard verification is achieved via the event `verify-guards^`. See Section 6.38 [VERIFY-GUARDS], page 199.

It should be emphasized that guard verification affects only two things: how fast ACL2 can evaluate the function and whether the function is executed correctly by raw Common Lisp, without guard violations. Since ACL2 does not use the raw Common Lisp definition of a function to evaluate its calls unless that function's guards have been verified, the latter effect is felt only if you run functions in raw Common Lisp rather than via ACL2's command loop.

Guard verification does not otherwise affect the theorem prover or the semantics of a definition. If you are not planning on running your function on "big" inputs and you don't care if your function runs correctly in raw Common Lisp (e.g., you have formalized some abstract mathematical property and just happened to use ACL2 as your language), there is no need to suffer through guard verification. Often users start by not doing guard verification and address that problem later. Sometimes you are driven to it, even in mathematical projects, because you find that you want to run your functions particularly fast or in raw Common Lisp.

If `certify-book^` is used to compile a file, and the file contains functions with unverified guard conjectures, then you will be warned that the compiled file cannot be loaded into raw Common Lisp with the expectation that the functions will run correctly. This is just the same point we have been making: ACL2 and Common Lisp agree only on the restricted domains specified by our guards. When guards are violated, Common Lisp can do anything. When you call a compiled function on arguments violating its guards, the chances are only increased that Common Lisp will go berserk, because compiled functions generally check fewer things at runtime and tend to be more fragile than interpreted ones.

8.39.3 GUARD-QUICK-REFERENCE

GUARD-QUICK-REFERENCE brief summary of guard checking and guard verification

For a careful introduction to guards, see Section 8.39 [GUARD], page 285.

I. GUARD CHECKING DURING EXECUTION

Effect

Guards on definitions are checked at execution time (except for guards on subsidiary calls of recursive or mutually recursive functions).

When does it happen

By default, guards are checked for all forms submitted at the top level.

To disable

```
:set-guard-checking nil
```

To (re-)enable

```
:set-guard-checking t
```

II. GUARD VERIFICATION

Effect

A proof is attempted of the obligations arising from the guards of subsidiary functions in a `defun^`, `defthm^`, or `defaxiom^` event.

When does it happen

Only names of defined functions, `defthm^`s, and `defaxiom^`s are subject to guard verification. Guard verification may occur when functions are defined (using `defun^`), but it requires an explicit call of `verify-guards^` in order to verify guards for `defthm^`s and `defaxiom^`s. Constrained functions (see Section 6.18 [ENCAPSULATE], page 176) may not have their guards verified.

```
(verify-guards foo ...)
```

causes guard verification for the `defun^`, `defthm^`, or `defaxiom^` named by `foo`, if it has not already been successfully done. The default defun-mode^ (see Section 8.16 [DEFAULT-DEFUN-MODE], page 254) must be `:logic^`, or else this event is ignored.

```
(defun foo ...)
```

causes guard verification of `foo` if and only if the following conditions are both met. (However, see Section 6.33 [SET-VERIFY-GUARDS-EAGERNESS], page 192 for how to change this behavior.)

1. Foo is processed in `:logic^` mode (either by setting mode `:logic^` globally, or by including `:mode :logic` in the `xargs^` declaration).

2. The `xargs^` declaration (see Section 8.91 [XARGS], page 369) either specifies `:guard^` or specifies `:verify-guards^ t` (or both).

```
(verify-termination foo ...)
```

causes guard verification of `foo` if `foo` is a function currently defined in `:program^` mode and the appropriate `xargs^` are supplied, as discussed for the case of `defun^` above. The default defun-mode^ (see Section 8.16 [DEFAULT-DEFUN-MODE], page 254) must be `:logic^`, or else this event is ignored.

8.39.4 GUARDS-AND-EVALUATION

GUARDS-AND-EVALUATION the relationship between guards and evaluation

The guard has no effect on the logical axiom added by the definition of a function. It does, however, have consequences for how calls of that function are evaluated in ACL2. We begin by explaining those consequences, when ACL2 is in its default "mode," i.e., as originally brought up. In subsequent discussion we'll consider other ways that guards can interact with evaluation. For more about guards in general, see Section 8.39 [GUARD], page 285. Also see Section 8.37 [GENERALIZED-BOOLEANS], page 281 for discussion about a subtle issue in the evaluation of certain Common Lisp functions.

Guards and evaluation I: the default

Consider the following very simple definition.

```
(defun foo (x) (cons 1 (cdr x)))
```

First consider how raw Common Lisp behaves when evaluating calls of this function. To evaluate `(foo x)` for some expression `x`, first `x` is evaluated to some value `v`, and then `(cons 1 (cdr x))` is evaluated with `x` bound to `v`. For example, if `v` is `(cons 'a 3)`, then Common Lisp computes `(cons 1 3)`. But if (for example) `v` is a number, e.g., `7`, then there is no way to predict what Common Lisp might do. Some implementations would cause "sensible" errors, others might return nonsense, still others might crash the host machine. The results tend toward the catastrophic if the call of `foo` in question is in compiled code.

Now by default, ACL2 evaluates calls of `foo` exactly as Common Lisp does, except that it uses guards to check the "legality" of each function call. So for example, since `(cdr x)` has a guard of `(or (consp x) (equal x nil))`, the call `(foo 7)` would cause a "guard violation," as illustrated below.

```
ACL2 !>(foo 7)

ACL2 Error in TOP-LEVEL:  The guard for the function symbol CDR, which
is (OR (CONSP X) (EQUAL X NIL)), is violated by the arguments in the
call (CDR 7).

ACL2 !>
```

Thus, the relation between evaluation in ACL2 and evaluation in Common Lisp is that the two produce the very same results, provided there is no guard violation.

Guards and evaluation II: `:set-guard-checking^`.

The ACL2 logic is a logic of total functions. That is, every application of a function defined has a completely specified result. See the documentation^ for each individual primitive for the specification of what it returns when its guard is violated; for example, see Section 10.71 [CDR], page 422.

The presence of guards thus introduces a choice in the sense of evaluation. When you type a form for evaluation do you mean for guards to be checked or not? Put another way, do you mean for the form to be evaluated in Common Lisp (if possible) or in the ACL2 logic? Note: If Common Lisp delivers an answer, it will be the same as in the logic, but it might be erroneous to execute the form in Common Lisp. For example, the ACL2 logic definition of `cdr^` implies that the `cdr^` of an atom^ is `nil`; see Section 10.71 [CDR], page 422. So: should `(cdr 7)` cause a guard violation error or return `nil`?

The top-level ACL2 loop has a variable which controls which sense of execution is provided. By default, "guard checking" is on, by which we mean that guards are checked at runtime, in the sense already described. To turn it off, do `:set-guard-checking nil`. To turn "guard checking" back on, execute the top-level form `:set-guard-checking t`. The status of this variable is reflected in the prompt^; guard-checking is "on" when the prompt^ contains an exclamation mark (also see Section 8.17 [DEFAULT-PRINT-PROMPT], page 255). For example,

```
ACL2 !>
```

means guard checking is on and

```
ACL2 >
```

means guard checking is off. The exclamation mark can be thought of as "barring" certain computations. The absence of the mark suggests the absence of error messages or unbarred access to the logical axioms. Thus, for example

```
ACL2 !>(car 'abc)
```

will signal an error, while

```
ACL2 >(car 'abc)
```

will return `nil`. To return to our previous example: with guard checking off, `(foo 7)` evaluates to `(cons 1 nil)`.

Guards and evaluation III: guard verification

Consider the defininition of `foo` given above, but modified so that a reasonable guard of `(consp x)` is specified, as shown below.

```
(defun foo (x)
  (declare (xargs :guard (consp x)))
  (cons 1 (cdr x)))
```

We say "reasonable guard" above because if `x` is such that `(consp x)` holds, then the call of `cdr^` in the evaluation of `(foo x)` will not cause a guard violation. Thus, it "should" be legal to evaluate `(foo x)`, for any such `x`, simply by evaluating this form in raw Common Lisp.

The `verify-guards^` event has been provided for this purpose. Details may be found elsewhere; see Section 6.38 [VERIFY-GUARDS], page 199. Briefly, for any defined function `fn`, the event `(verify-guards fn)` attempts to check the condition discussed above, that whenever `fn` is called on arguments that satisfy its guard, the evaluation of this call will proceed without any guard violation. If this check is successful, then future calls of this sort will be evaluated in raw Common Lisp.

Returning to our example above, the `(verify-guards foo)` will succeed because the guard `(consp x)` of `foo` implies the guard generated from the call `(cdr x)` in the body of the definition, namely, `(or (consp x) (equal x nil))` (see Section 10.71 [CDR], page 422). Then the evaluation of `(foo (cons 'a 3))` will take place in raw Common Lisp, because `(cons 'a 3)` satisfies the guard of `foo`.

This ability to dive into raw Common Lisp hinges on the proof that the guards you attach to your own functions are sufficient to insure that the guards encountered in the body are satisfied. This is called "guard verification." Once a function has had its guards verified, then ACL2 can evaluate the function somewhat faster (but see "Guards and evaluation V: efficiency issues" below). Perhaps more importantly, ACL2 can also guarantee that the function will be evaluated correctly by any implementation of Common Lisp (provided the guard of the function is satisfied on the input). That is, if you have verified the guards of a system of functions and you have determined that they work as you wish in your host ACL2 (perhaps by proving it, perhaps by testing), then they will work identically in any Common Lisp.

There is a subtlety to our treatment of evaluation of calls of functions whose guards have been verified. If the function's guard is not satisfied by such a call, then no further attempt is made to evaluate any call of that function in raw lisp during the course of evaluation of that call. This is obvious if guard checking is on, because an error is signalled the first time its guard is violated; but in fact it is also true when guard checking is off. See Section 1.5.6.2 [GUARD-EXAMPLE], page 59 for an example.

Guards and evaluation IV: functions having :program mode

Strictly speaking, functions in `:program^` mode (see Section 8.18 [DEFUN-MODE], page 256) do not have definitions in the ACL2 logic. So what does it mean to evaluate calls of such functions in ACL2? Our decision is to treat `:program^` functions exactly as we treat `:logic^` functions whose guards have been verified, except that when there is a guard violation we signal an error regardless of whether guard checking is currently on or off. Note that when the guard of a function in `:logic^` mode is violated, there is still a value that the ACL2 logic proves is equal to the given call. But the same cannot be said for a function in `:program^` mode, so we really have no choice but to signal an error when there is a guard violation for such a function.

To summarize: as with `:logic^` functions, when a guard has been satisfied on a call of a function with `:program^` mode, no subsidiary guard checking will be done.

Consider the same example `foo` as above, but where `foo` is submitted in `:program^` mode. Then even with guard checking off, top-level calls of `foo` will signal a guard violation when the arguments do not satisfy the guard of `foo`, e.g., `(foo 7)`.

Notice that by treating functions in `:program^` mode like functions whose guards have been verified, we are using raw lisp to compute their values when their guards are met. We do not check guards any further once raw lisp is invoked. This can lead to hard lisp errors if the guards are not appropriate, as illustrated below.

```
ACL2 >:program
ACL2 p>(defun foo (x)
         (declare (xargs :guard t))
         (cons 1 (cdr x)))

Summary
Form:  ( DEFUN FOO ...)
Rules: NIL
Warnings:  None
Time:  0.02 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other: 0.02)
 FOO
ACL2 p>(foo 3)

Error: 3 is not of type LIST.
Fast links are on: do (use-fast-links nil) for debugging
Error signalled by CDR.
Broken at COND.  Type :H for Help.
ACL2>>
```

See Section 8.19 [DEFUN-MODE-CAVEAT], page 259.

Guards and evaluation V: efficiency issues

We have seen that by verifying the guards for a `:logic^` function, we arrange that raw lisp is used for evaluation of calls of such functions when the arguments satisfy its guard.

This has several apparent advantages over the checking of guards as we go. First, the savings is magnified as your system of functions gets deeper: the guard is checked upon the top-level entry to your system and then raw Common Lisp does all the computing. Second, if the raw Common Lisp is compiled, enormous speed-ups are possible. Third, if your Common Lisp or its compiler does such optimizations as `tail-recursion` removal, raw Common Lisp may be able to compute your functions on input much "bigger" than ACL2 can.

The first of these advantages is quite important if you have complicated guards. However, the other two advantages are probably not very important, as we now explain.

When a function is defined in `:logic^` mode, its `defun^` is executed in raw Common Lisp. (We might call this the "primary" raw lisp definition of the function.) However, a corresponding "logic definition" is also executed. The "logic definition" is a `defun^` in raw lisp that checks guards at runtime and escapes to the primary raw lisp definition if the guard holds of the arguments and the function has already had its guards verified. Otherwise the logic definition executes the body of the function by calling the logic definitions of each subroutine. Now it is true that compilation^ generally speeds up execution enormously. However, the `:comp^` command (see Section 9.5 [COMP], page 379) compiles both of the raw lisp definitions associated with a `:logic^` function. Also, we have attempted to arrange that for every tail recursion removal done on the actual `defun^`, a corresonding tail recursion removal is done on the "logic definition."

We believe that in most cases, the logic definition executes almost as fast as the primary raw lisp definition, at least if the evaluation of the guards is fast. So, the main advantage of guard verification is probably that it lets you know that the function may be executed safely in raw lisp, returning the value predicted by the ACL2 logic, whenever its arguments satisfy its guard. We envision the development of systems of applicative lisp functions that have been developed and reasoned about using ACL2 but which are intended for evaluation in raw Common Lisp (perhaps with only a small "core" of ACL2 loaded), so this advantage of guard verification is important.

Nevertheless, guard verification might be important for optimal efficiency when the functions make use of type declarations. For example, at this writing, the GCL implementation of Common Lisp can often take great advantage of `declare^` forms that assign small integer types to formal parameters.

To continue the discussion of guards, see Section 8.39.5 [GUARDS-FOR-SPECIFICATION], page 295 to read about the use of guards as a specification device.

8.39.5 GUARDS-FOR-SPECIFICATION

GUARDS-FOR-SPECIFICATION guards as a specification device

A use of guard verification that has nothing to do with efficiency is as a way to gain confidence in specifications. This use has the feel of "types" in many traditional programming^ languages, though guards allow much greater expressiveness than most systems of types (and unfortunately, as a result they are not syntactically checkable).

For more discussion of guards in general, see Section 8.39 [GUARD], page 285.

Suppose you have written a collection of function definitions that are intended to specify the behavior of some system. Perhaps certain functions are only intended to be called on certain sorts of inputs, so you attach guards to those functions in order to "enforce" that requirement. And then, you verify the guards for all those functions.

Then what have you gained, other than somewhat increased efficiency of execution (as explained above), which quite possibly isn't your main concern? You have gained the confidence that when evaluating any call of a (specification) function whose arguments satisfy that function's guard, all subsequent function calls during the course of evaluation will have this same property, that the arguments satisfy the guard of the calling function. In logical terms, we can say that the equality of the original call with the returned value is provable from weakened versions of the definitions, where each definitional axiom is replaced by an implication whose antecedent is the requirement that the arguments satisfy the guard and whose consequent is the original axiom. For example,

```
(defun foo (x)
  (declare (xargs :guard (consp x)))
  (cons 1 (cdr x)))
```

originally generates the axiom

```
(equal (foo x)
       (cons 1 (cdr x)))
```

but in fact, when evaluation involves no guard violation then the following weaker axiom suffices in the justification of the evaluation.

```
(implies (consp x)
         (equal (foo x)
                (cons 1 (cdr x))))
```

If you are following links to read this documentation as a hypertext style document, then please see Section 8.39.2 [GUARD-MISCELLANY], page 287. This concludes our discussion of guards with miscellaneous remarks, and also contains pointers to related topics.

8.40 HIDE

HIDE hide a term from the rewriter

Hide is actually the identity^ function: (hide x) = x for all x. However, terms of the form (hide x) are ignored by the ACL2 rewriter, except when explicit :expand hints^ are given for such terms (see Section 8.41 [HINTS], page 297). They are also ignored by the induction heuristics.

See Section 8.29 [EVISCERATE-HIDE-TERMS], page 271 for how to affect the printing of such terms.

8.41 HINTS

HINTS advice to the theorem proving process

```
Examples:
The following :hints value is nonsensical.  Nevertheless, it
illustrates all of the available hint keywords:

:hints (("Goal"
         :do-not-induct t
         :do-not '(generalize fertilize)
         :expand ((assoc x a) (member y z))
         :restrict ((<-trans ((x x) (y (foo x)))))
         :hands-off (length binary-append)
         :in-theory (set-difference-theories
                      (current-theory :here)
                      '(assoc))
         :induct (and (nth n a) (nth n b))
         :use ((:instance assoc-of-append
                          (x a) (y b) (z c))
               (:functional-instance
                 (:instance p-f (x a) (y b))
                 (p consp)
                 (f assoc)))
         :bdd (:vars (c a0 b0 a1 b1) :prove nil :bdd-constructors (cons))
         :cases ((true-listp a) (consp a))
         :by ((:instance rev-rev (x (cdr z))))))
```

A very common hint is the :use hint, which in general takes as its value a list of “lemma instances” (see Section 8.59 [LEMMA-INSTANCE], page 319) but which allows a single lemma name as a special case:

```
:hints (("[1]Subgoal *1/1.2'" :use lemma23))
```

Background: `Hints` are allowed in all events^ that use the theorem prover. During `defun^` events^ there are two different uses of the theorem prover: one to prove termination and another to verify the guard^s. To pass a hint to the theorem prover during termination proofs, use the `:hints` keyword in the `defun^`'s `xargs^` declaration. To pass a hint to the theorem prover during the guard^ verification of `defun^`, use the `:guard-hints` keyword in the `defun^`'s `xargs^` declaration. The `verify-guards^` event and the `defthm^` event also use the theorem prover. To pass hints to them, use the `:hints` keyword argument to the event.

```
General Form of :hints:
  ((goal-spec :key1 val1 ... :keyn valn)
   ...
   (goal-spec :key1 val1 ... :keyn valn))
```

where `goal-spec^` is as described in the documentation for goal-spec^ and the keys and their respective values are shown below with their interpretations.

`:DO-NOT-INDUCT`

`Value` is `t`, `name` or `nil`, indicating whether induction^ is permitted under the specified goal. If `value` is `t`, then the attempt to apply induction^ to the indicated goal or any subgoal under the indicated goal will immediately cause the theorem prover to report failure^. Thus, the indicated goal must be proved entirely by simplification, destructor elimination, and the other "waterfall" processes. Induction^ to prove the indicated goal (or any subgoal) is not permitted. See however the `:induct` hint below. If `value` is a symbol other than `t` or `nil`, the theorem prover will give a "bye" to any subgoal that would otherwise be attacked with induction. This will cause the theorem prover to fail eventually but will collect the necessary subgoals. If `value` is `nil`, this hint means induction^ is permitted. Since that is the default, there is no reason to use the value `nil`.

`:DO-NOT`

`Value` is a term having at most the single free variable `world^`, which when evaluated (with `world^` bound to the current ACL2 logical world^) produces a list of symbols that is a subset of the list

```
(preprocess ;propositional logic, simple rules
 simplify   ;as above plus rewriting, linear arithmetic
 eliminate-destructors
 fertilize  ;use of equalities
 generalize
 eliminate-irrelevance).
```

The hint indicates that the "processes" named should not be used at or below the goal in question. Thus, to prevent generalization and fertilization, say, include the hint

```
:do-not '(generalize fertilize)
```

If `value` is a single symbol, as in

```
:do-not generalize,
```

it is taken to be `'(value)`.

:EXPAND

`Value` is a true list of terms, each of which is of one of the forms `(let ((v1 t1)...) b)` or `(fn t1 ... tn)`, where `fn` is a defined function symbol with formals `v1, ..., vn,` and `body b`. Such a term is said to be "expandable:" it can be replaced by the result of substituting the `ti`'s for the `vi`'s in `b`. The terms listed in the `:expand` hint are expanded when they are encountered by the simplifier while working on the specified goal or any of its subgoals. We permit `value` to be a single such term instead of a singleton list. **Note**: Also allowed are "terms" of the form `(:free (var1 var2 ... varn) pattern)` where the indicated variables are distinct and `pattern` is a term. Such "terms" indicate that we consider the indicated variables to be instantiatable, in the following sense: whenever the simplifier encounters a term that can be obtained from `pattern` by instantiating the variables `(var1 var2 ... varn)`, then it expands that term.

:HANDS-OFF

`Value` is a true list of function symbols or lambda expressions, indicating that under the specified goal applications of these functions are not to be rewritten. `value` may also be a single function symbol or lambda expression instead of a list.

:IN-THEORY^

`Value` is a "theory expression," i.e., a term having at most the single free variable `world^` which when evaluated (with `world^` bound to the current ACL2 logical world (see Section 8.89 [WORLD], page 362)) will produce a theory to use as the current theory for the goal specified. See Chapter 15 [THEORIES], page 667.

:INDUCT

`Value` is either `t` or a term containing at least one recursively defined function symbol; if `t`, this hint indicates that the system should proceed to apply its induction heuristic to the specified goal produced (without trying simplification, etc.); if `value` is a term other than `t`, then not only should the system apply induction immediately, but it should analyze `value` rather than the goal to generate its induction^ scheme. Merging and the other induction^ heuristics are applied. Thus, if `value` contains several mergeable induction^s, the "best" will be created and chosen. E.g., the `:induct` hint

```
(and (nth i a) (nth j a))
```

suggests simultaneous induction^ on `i`, `j`, and `a`.

If both an `:induct` and a `:do-not-induct` hint are supplied for a given goal then the indicated induction^ is applied to the goal and the `:do-not-induct` hint is inherited by all subgoals generated.

:USE

`Value` is a lemma-instance^ or a true list of lemma-instance^s, indicating that the propositions denoted by the instances be added as hypotheses to the specified goal. See Section 8.59 [LEMMA-INSTANCE], page 319. Note that `:use` makes the given instances available as ordinary hypotheses of the formula to be proved. The `:instance` form of a lemma-instance^ permits you to instantiate the free variables of previously proved theorems any way you wish; but it is up to you to provide the appropriate instantiations because once the instances are added as hypotheses their variables are no longer instantiable. These new hypotheses participate fully in all subsequent rewriting, etc. If the goal in question is in fact an instance of a previously proved theorem, you may wish to use `:by` below.

:BDD^

This hint indicates that ordered binary decision diagrams (BDDs) with rewriting are to be used to prove or simplify the goal. See Chapter 2 [BDD], page 69 for an introduction to the ACL2 BDD algorithm.

`Value` is a list of even length, such that every other element, starting with the first, is one of the keywords `:vars`, `:bdd-constructors`, `:prove`, or `literal`. Each keyword that is supplied should be followed by a value of the appropriate form, as shown below; for others, a default is used. Although `:vars` must always be supplied, we expect that most users will be content with the defaults used for the other values.

`:vars` — A list of ACL2 variables, which are to be treated as Boolean variables. The prover must be able to check, using trivial reasoning (see Section 8.87 [TYPE-SET], page 358), that each of these variables is Boolean in the context of the current goal. Note that the prover will use very simple heuristics to order any variables that do not occur in `:vars` (so that they are "greater than" the variables that do occur in `:vars`), and these heuristics are often far from optimal. In addition, any variables not listed may fail to be assumed Boolean by the prover, which is likely to seriously impede the effectiveness of ACL2's BDD algorithm. Thus, users are encouraged *not* to rely on the default order, but to supply a list of variables instead. Finally, it is allowed to use a value of `t` for `vars`. This means the same as a `nil` value, except that the BDD algorithm is directed to fail unless it can guarantee that all variables in the input term are known to be Boolean (in a sense discussed elsewhere; see Section 2.1 [BDD-ALGORITHM], page 69).

`:literal` — An indication of which part of the current goal should receive BDD processing. Possible values are:

```
:all     treat entire goal as a single literal (the default)
:conc    process the conclusion
n        process the hypothesis with index n (1, 2, ...)
```

`:bdd-constructors` — When supplied, this value should be a list of function symbols in the current ACL2 world^; it is `(cons)` by default, unless `:bdd-constructors` has a value in the `acl2-defaults-table^` by default, in which case that value is the default. We expect that most users will be content with the default. See Section 2.1 [BDD-ALGORITHM], page 69 for information about how this value is used.

`:prove` — When supplied, this value should be `t` or `nil`; it is `t` by default. When the goal is not proved and this value is `t`, the entire proof will abort. Use the value `nil` if you are happy to the proof to go on with the simplified term.

:CASES

`Value` is a non-empty list of terms. For each term in the list, a new goal is created from the current goal by assuming that term; and also, in essence, one additional new goal is created by assuming all the terms in the list false. We say "in essence" because if the disjunction of the terms supplied is a tautology, then that final goal will be a tautology and hence will in fact never actually be created.

:BY

`Value` is a lemma-instance^, `nil`, or a new event name. If the value is a lemma-instance^ (see Section 8.59 [LEMMA-INSTANCE], page 319), then it indicates that the goal (when viewed as a clause) is subsumed by the proposition denoted by the instance (when viewed as a clause). To view a formula as a clause, union together the negations of the hypotheses and add the conclusion. For example,

```
(IMPLIES (AND (h1 t1) (h2 t2)) (c t1))
```

may be viewed as the clause

```
{~(h1 t1) ~(h2 t2) (c t1)}.
```

Clause `c1` is "subsumed" by clause `c2` iff some instance of `c2` is a subset of `c1`. For example, the clause above is subsumed by `{~(h1 x) (c x)}`, which when viewed as a formula is `(implies (h1 x) (c x))`.

If the value is `nil` or a new name, the prover does not even attempt to prove the goal to which this hint is attached. Instead the goal is given a "bye", i.e., it is skipped and the proof attempt continues as though the goal had been proved. If the prover terminates without error then it reports that the proof would have succeeded had the indicated goals been proved and it prints an appropriate defthm^ form to define each of the `:by` names. The "name" `nil` means "make up a name."

The system does not attempt to check the uniqueness of the `:by` names (supplied or made up), since by the time those goals are proved the namespace will be cluttered still further. Therefore, the final list of "appropriate" `defthm^` forms may be impossible to admit without some renaming by the user. If you must invent new names, remember to substitute the new ones for the old ones in the `:by` hints themselves.

:RESTRICT

(Warning: This is a sophisticated hint, intended for advanced users. This hint was suggested by Bishop Brock.)

`Value` is an association list. Its members are of the form `(x subst1 subst2 ...)`, where: `x` is either (1) a rune^ whose `car^` is `:rewrite^` or (2) an event name corresponding to one or more such rune^s; and `(subst1 subst2 ...)` is a non-empty list of substitutions, i.e., of association lists pairing variables with terms. First consider the case that `x` is a `:rewrite^` rune^. First recall that without this hint, the rewrite rule named `x` is used by matching its left-hand side (call it `lhs`) against the term currently being considered by the rewriter, that is, by attempting to find a substitution `s` such that the instantiation of `lhs` using `s` is equal to that term. If however the `:restrict` hint contains `(x subst1 subst2 ...)`, then this behavior will be modified by restricting `s` so that it must extend `subst1`; and if there is no such `s`, then `s` is restricted so that it must extend `subst2`; and so on, until the list of substitutions is exhausted. If no such `s` is found, then the rewrite rule named `x` is not applied to that term. Finally, if `x` is an event name corresponding to one or more `:rewrite^` rune^s (that is, `x` is the "base symbol" of such rune^s; see Section 15.10 [RUNE], page 675), say rune^s `r1`, ... `rn`, then the meaning is the same except that `(x subst1 subst2 ...)` is replaced by `(ri subst1 subst2 ...)` for each `i`. Once this replacement is complete, the hint may not contain two members whose `car^` is the same rune^.

Note that the substitutions in `:restrict` hints refer to the variables actually appearing in the goals, not to the variables appearing in the `:rewrite^` rule being restricted.

Here is an example, supplied by Bishop Brock. Suppose that the database includes the following rewrite rule, which is probably kept disable^d. (We ignore the question of how to prove this rule.)

```
cancel-<-*$free:
(implies (and (rationalp x)
              (rationalp y)
              (rationalp z))
         (equal (< y z)
                (if (< x 0)
                    (> (* x y) (* x z))
                  (if (> x 0)
                      (< (* x y) (* x z))
                    (hide (< y z))))))
```

Then ACL2 can prove the following theorem (unless other rules get in the way), essentially by multiplying both sides by `x`.

```
(thm
```

```
  (implies (and (rationalp x)
                (< 1 x))
           (< (/ x) 1))
  :hints
  (("Goal"
    :in-theory (enable cancel-<-*$free)
    :restrict ((cancel-<-*$free ((x x) (y (/ x)) (z 1)))))))
```

The `:restrict` hint above says that the variables `x`, `y`, and `z` in the rewrite rule `cancel-<-*$free` above should be instantiated respectively by `x`, `(/ x)`, and `1`. Thus `(< y z)` becomes `(< (/ x) 1)`, and this inequality is replaced by the corresponding instance of the right-hand-side of `cancel-<-*$free`. Since the current conjecture assumes `(< 1 x)`, that instance of the right-hand side simplifies to

```
(< (* x (/ x)) (* x 1))
```

which in turn simplifies to `(< 1 x)`, a hypothesis in the present theorem.

8.42 I-AM-HERE

I-AM-HERE a convenient marker for use with `rebuild^`

```
Example Input File for Rebuild:
(defun fn1 (x y) ...)
(defthm lemma1 ...)
(defthm lemma2 ...)
(i-am-here)
The following lemma won't go through.  I started
typing the hint but realized I need to prove a
lemma first.  See the failed proof attempt in foo.bar.
I'm going to quit for the night now and resume tomorrow
from home.

(defthm lemma3 ...
  :hints (("Goal" :use (:instance ???
...
```

By putting an `(i-am-here)` form at the “frontier” of an evolving file of command^s, you can use `rebuild^` to load the file up to the `(i-am-here)`. `I-am-here` simply returns an `ld^` error triple and any form that “causes an error” will do the same job. Note that the text of the file after the `(i-am-here)` need not be machine readable.

8.43 IMMEDIATE-FORCE-MODEP

IMMEDIATE-FORCE-MODEP when executable counterpart is enable^d,
force^d hypotheses are attacked immediately

This function symbol is defined simply to provide a rune^ which can be enable^d and disable^d. Enabling

```
(:executable-counterpart immediate-force-modep)
```

causes ACL2 to attack force^d hypotheses immediately instead of delaying them to the next forcing round.

```
Example Hints
:in-theory (disable (:executable-counterpart immediate-force-modep))
           ; delay forced hyps to forcing round
:in-theory (enable (:executable-counterpart immediate-force-modep))
           ; split on forced hyps immediately
```

See Section 8.35 [FORCE], page 277 for background information. When a `force^`d hypothesis cannot be established a record is made of that fact and the proof continues. When the proof succeeds a "forcing round" is undertaken in which the system attempts to prove each of the force^d hypotheses explicitly. However, if the rune^ `(:executable-counterpart immediate-force-modep)` is enable^d at the time the hypothesis is force^d, then ACL2 does not delay the attempt to prove that hypothesis but undertakes the attempt more or less immediately.

8.44 IN-PACKAGE

IN-PACKAGE select current package

```
Example:
(in-package "MY-PKG")

General Form:
(in-package str)
```

where `str` is a string that names an existing ACL2 package, i.e., one of the initial packages such as `"KEYWORD"` or `"ACL2"` or a package introduced with `defpkg^`. For a complete list of the known packages created with `defpkg^`, evaluate

```
(strip-cars (known-package-alist state)).
```

See Section 6.11 [DEFPKG], page 164. `In-package` forms can only be typed at the top-level of the ACL2 loop and as the first form in a file being loaded or compiled.

8.45 INVISIBLE-FNS-ALIST

INVISIBLE-FNS-ALIST functions that are invisible to the loop-stopper^ algorithm

```
Examples:
ACL2 !>(invisible-fns-alist (w state))
((binary-+ unary--)
 (binary-* unary-/)
 (unary-- unary--)
 (unary-/ unary-/))
```

Among other things, the setting above has the effect of making `unary--`^ "invisible" for the purposes of applying permutative `:rewrite`^ rules to `binary-+`^ trees. See Section 6.30 [SET-INVISIBLE-FNS-ALIST], page 189.

The notion of "invisible functions" has to do with the control mechanism on permutative `:rewrite`^ rules. See Section 8.62 [LOOP-STOPPER], page 324 for a detailed discussion of the control mechanism. See Section 6.30 [SET-INVISIBLE-FNS-ALIST], page 189 for a discussion of how to set the invisible functions alist.

8.46 KEYWORD-COMMANDS

KEYWORD-COMMANDS how keyword commands are processed

```
Examples:
user type-in                  form evaluated
:pc 5                         (ACL2::PC '5)
:pcs app rev                  (ACL2::PCS 'app 'rev)
:length (1 2 3)               (ACL2::LENGTH '(1 2 3))
```

When a keyword, `:key`, is read as a command, ACL2 determines whether the symbol with the same name in the `"ACL2"` package, `acl2::key`, is a function or simple macro of n arguments. If so, ACL2 reads `n` more objects, `obj1`, ..., `objn`, and then acts as though it had read the following form (for a given `key`):

```
(ACL2::key 'obj1 ... 'objn)
```

Thus, by using the keyword command hack you avoid typing the parentheses, the `"ACL2"` package name, and the quotation marks.

Note the generality of this hack. Almost any function or macro in the `"ACL2"` package can be so invoked, not just "commands." Indeed, there is no such thing as a distinguished class of commands. The one caveat is that the keyword hack can be used to invoke a macro only if that macro has a simple argument list — one containing no lambda keywords (such as `&rest`), since they complicate or render impossible the task of deciding how many objects to read. Users may take advantage of the keyword command hack by defining functions and macros in the `"ACL2"` package.

8.47 LD-ERROR-ACTION

LD-ERROR-ACTION determines `ld^`'s response to an error

`Ld-error-action` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-error-action state)` and the updater is `(set-ld-error-action val state)`. `Ld-error-action` must be `:continue`, `:return`, or `:error`. The initial value of `ld-error-action` is `:continue`, which means that the top-level ACL2 command^ loop will not exit when an error is caused by `user-typein`. But the default value for `ld-error-action` on calls of `ld^` is `:return`.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-error-action` is one of them. If, while `ld-error-triples^` is `t`, a form evaluates to three results, the first of which is non-`nil` and the third of which is `state^`, an error is said to have occurred. If an error occurs, `ld^`'s action depends on `ld-error-action`. If it is `:continue`, `ld^` just continues processing the forms in `standard-oi^`. If it is `:return`, `ld^` stops and returns as though it had emptied the channel. If it is `:error`, `ld^` stops and returns, signalling an error to its caller.

8.48 LD-ERROR-TRIPLES

LD-ERROR-TRIPLES determines whether a form caused an error during `ld^`

`Ld-error-triples` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-error-triples state)` and the updater is `(set-ld-error-triples val state)`. `Ld-error-triples` must be either `t` or `nil`. The initial value of `ld-error-triples` is `t`.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-error-triples` is one of them. If this variable has the value `t` then when a form evaluates to 3 values, the first of which is non-`nil` and the third of which is `state^`, an error is deemed to have occurred. When an error occurs in evaluating a form, `ld^` rolls back the ACL2 world^ to the configuration it had at the conclusion of the last error-free form. Then `ld^` takes the action determined by `ld-error-action^`.

8.49 LD-EVISC-TUPLE

LD-EVISC-TUPLE determines whether `ld^` suppresses details when printing

`Ld-evisc-tuple` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-evisc-tuple state)` and the updater is `(set-ld-evisc-tuple val state)`. `Ld-evisc-tuple` must be either `nil` or a list of the form

```
(alist nil nil print-level print-length hiding-cars)
```

where `alist` is an alist that pairs objects to strings, `print-level` and `print-length` are either `nil` or non-negative integers, and `hiding-cars` is a list of symbols. The initial value of `ld-evisc-tuple` is `nil`.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-evisc-tuple` is one of them. `Ld^` may print the forms it is evaluating and/or the results of evaluation. If the value of `ld-evisc-tuple` is a list as shown above, then `ld^` "eviscerates" the objects it prints before printing them. To "eviscerate" an object we replace certain substructures within it by strings which are printed in their stead. `Print-level` and `print-length`, above, are used as described in CLTL (pp 372) to replace those substructures deeper than `print-level` by "`#`" and those longer than `print-length` by "`...`". `Alist` is used to replace any substructure occuring as a key in `alist` by the corresponding string. Finally, any `consp^ x` that starts with one of the symbols in `hiding-cars` is printed as `<hidden>`.

8.50 LD-KEYWORD-ALIASES

LD-KEYWORD-ALIASES allows the abbreviation of some keyword commands

```
Example:
(set-ld-keyword-aliases '((:q 0 q-fn)
                          (:e 0 exit-acl2-macro))
                        state)
```

`Ld-keyword-aliases` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-keyword-aliases state)` and the updater is `(set-ld-keyword-aliases val state)`. `Ld-keyword-aliases` must be an alist, each element of which is of the form `(:keyword n fn)`, where `:keyword` is a keyword, `n` is a nonnegative integer, and `fn` is a function symbol of arity `n`, a macro symbol, or a `lambda` expression of arity `n`. When `keyword` is typed as an `ld^` command, `n` more forms are read, `x1, ..., xn`, and the form `(fn 'x1 ... 'xn)` is then evaluated. The initial value of `ld-keyword-aliases` is `nil`.

In the example above, `:q^` has been redefined to have the effect of executing `(q-fn)`, so for example if you define

```
(defmacro q-fn ()
  '(er soft 'q "You un-bound :q and now we have a soft error."))
```

then `:q^` will cause an error, and if you define

```
(defmacro exit-acl2-macro () '(exit-ld state))
```

then `:e` will cause the effect (it so happens) that `:q^` normally has. If you prefer `:e` to `:q^` for exiting the ACL2 loop, you might even want to put such definitions of `q-fn` and `exit-acl2-macro` together with the `set-ld-keyword-aliases` form above in your `"acl2-customization.lisp"` file; see Section 8.4 [ACL2-CUSTOMIZATION], page 225.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-keyword-aliases` is one of them. `Ld-keyword-aliases` affects how keyword commands are parsed. Generally speaking, `ld^`'s command interpreter reads "`:fn x1 ... xn`" as "`(fn 'x1 ... 'xn)`" when `:fn` is a keyword and `fn` is the name of an `n`-ary function. But this parse is overridden, as described above, for the keywords bound in `ld-keyword-aliases`.

8.51 LD-POST-EVAL-PRINT

LD-POST-EVAL-PRINT determines whether and how `ld^` prints the result of evaluation

`Ld-post-eval-print` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-post-eval-print state)` and the updater is `(set-ld-post-eval-print val state)`. `Ld-post-eval-print` must be either `t`, `nil`, or `:command-conventions`. The initial value of `ld-post-eval-print` is `:command-conventions`.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-post-eval-print` is one of them. If this global variable is `t`, `ld^` prints the result. In the case of a form that produces multiple values, `ld^` prints the list containing them all (which, logically speaking, is what the form returned). If `ld-post-eval-print` is `nil`, `ld^` does not print the values. This is most useful when `ld^` is used to load a previously processed file.

Finally, if `ld-post-eval-print` is `:command-conventions` then `ld^` prints the result but treats "error triples" specially. An "error triple" is a result, `(mv erp val state)`, that consists of three values, the third of which is `state^`. Many ACL2 functions use such triples to signal errors. The convention is that if `erp` (the first value) is `nil`, then the function is returning `val` (the second value) as its conventional single result and possibly side-effecting state^ (as with some output). If `erp` is `t`, then an error has been caused, `val` is irrelevant and the error message has been printed in the returned state^. Example ACL2 functions that follow this convention include `defun^` and `in-package^`. If such "error producing" functions are evaluated while `ld-post-eval-print` is set to `t`, then you would see them producing lists of length 3. This is disconcerting to users accustomed to Common Lisp (where these functions produce single results but sometimes cause errors or side-effect state^).

When `ld-post-eval-print` is `:command-conventions` and a form produces an error triple `(mv erp val state)` as its value, `ld^` prints nothing if `erp` is non-`nil` and otherwise `ld^` prints just `val`. Because it is a misrepresentation to suggest that just one result was returned, `ld^` prints the value of the global variable `'triple-print-prefix` before printing `val`. `'triple-print-prefix` is initially `" "`, which means that when non-erroneous error triples are being abbreviated to `val`, `val` appears one space off the left margin instead of on the margin.

In addition, when `ld-post-eval-print` is `:command-conventions` and the value component of an error triple is the keyword `:invisible` then `ld^` prints nothing. This is the way certain commands (e.g., `:pc^`) appear to return no value.

By printing nothing when an error has been signalled, `ld^` makes it appear that the error (whose message has already appeared in state^) has "thrown" the computation back to load without returning a value. By printing just `val` otherwise, we suppress the fact that state^ has possibly been changed.

8.52 LD-PRE-EVAL-FILTER

LD-PRE-EVAL-FILTER determines which forms `ld^` evaluates

`Ld-pre-eval-filter` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-pre-eval-filter state)` and the updater is `(set-ld-pre-eval-filter val state)`. `Ld-pre-eval-filter` must be either `:all`, `:query`, or a new name that could be defined (e.g., by `defun^` or `defconst^`). The initial value of `ld-pre-eval-filter` is `:all`.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-pre-eval-filter` is one of them. If the filter is `:all`, then every form read is evaluated. If the filter is `:query`, then after a form is read it is printed to `standard-co^` and the user is asked if the form is to be evaluated or skipped. If the filter is a new name, then all forms are evaluated until that name is introduced, at which time `ld^` terminates normally.

The `:all` filter is, of course, the normal one. `:Query` is useful if you want to replay selected the command^s in some file. The new name filter is used if you wish to replay all the command^s in a file up through the introduction of the given one.

8.53 LD-PRE-EVAL-PRINT

LD-PRE-EVAL-PRINT determines whether `ld^` prints the forms to be `eval`'d

`Ld-pre-eval-print` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-pre-eval-print state)` and the updater is `(set-ld-pre-eval-print val state)`. `Ld-pre-eval-print` must be either `t`, `nil`, or `:never`. The initial value of `ld-pre-eval-print` is `nil`.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-pre-eval-print` is one of them. If this global variable is `t`, then before evaluating the form just read from `standard-oi^`, `ld^` prints the form to `standard-co^`. If the variable is `nil`, no such printing occurs. The `t` option is useful if you are reading from a file of command^s and wish to assemble a complete script of the session in `standard-co^`.

The value `:never` of `ld-pre-eval-print` is rarely used. During the evaluation of `encapsulate^` and of `certify-book^` forms, subsidiary events are normally printed, even if `ld-pre-eval-print` is `nil`. Thus for example, when the user submits an `encapsulate^` form, all subsidiary events are generally printed even in the default situation where `ld-pre-eval-print` is `nil`. But occasionally one may want to suppress such printing. In that case, `ld-pre-eval-print` should be set to `:never`.

8.54 LD-PROMPT

LD-PROMPT determines the prompt^ printed by `ld^`

`Ld-prompt` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-prompt state)` and the updater is `(set-ld-prompt val state)`. `Ld-prompt` must be either `nil`, `t`, or a function symbol that, when given an open output character channel and state^, prints the desired prompt^ to the channel and returns two values: the number of characters^ printed and the state^. The initial value of `ld-prompt` is `t`.

The general-purpose ACL2 read-eval-print loop, `ld^`, reads forms from `standard-oi^`, evaluates them and prints the result to `standard-co^`. However, there are various flags that control `ld^`'s behavior and `ld-prompt` is one of them. `Ld-prompt` determines whether `ld^` prints a prompt^ before reading the next form from `standard-oi^`. If `ld-prompt` is `nil`, `ld^` prints no prompt^. If `ld-prompt` is `t`, the default prompt^ printer is used, which displays information that includes the current package, default defun-mode^, guard^ checking status (on or off), and `ld-skip-proofsp^`; see Section 8.17 [DEFAULT-PRINT-PROMPT], page 255.

If `ld-prompt` is neither `nil` nor `t`, then it should be a function name, `fn`, such that `(fn channel state)` will print the desired prompt^ to `channel` in `state^` and return `(mv col state)`, where `col` is the number of characters^ output (on the last line output). You may define your own prompt^ printing function.

If you supply an inappropriate prompt^ function, i.e., one that causes an error or does not return the correct number and type of results, the following odd prompt^ will be printed instead:

```
Bad Prompt
See :DOC ld-prompt>
```

which will lead you to this message. You should either call ld^ appropriately next time or assign an appropriate value to ld-prompt.

8.55 LD-QUERY-CONTROL-ALIST

LD-QUERY-CONTROL-ALIST how to default answers to queries

Ld-query-control-alist is an ld^ special (see Section 9.7 [LD], page 380). The accessor is (ld-query-control-alist state) and the updater is (set-ld-query-control-alist val state). Roughly speaking, ld-query-control-alist is either nil (meaning all queries should be interactive), t (meaning all should default to the first accepted response), or an alist that pairs query ids to keyword responses. The alist may end in either t or nil, indicating the default value for all ids not listed explicitly. Formally, the ld-query-control-alist must satisfy ld-query-control-alistp. The initial ld-query-control-alist is nil, which means all queries are handled interactively.

When an ACL2 query is raised, a unique identifying symbol is printed in parentheses after the word "Query". This symbol, called the "query id," can be used in conjunction with ld-query-control-alist to prevent the query from being handled interactively. By "handled interactively" we mean that the query is printed to *standard-co*^ and a response is read from *standard-oi*^. The alist can be used to obtain a "default value" for each query id. If this value is nil, then the query is handled interactively. In all other cases, the system handles the query without interaction (although text may be printed to standard-co^ to make it appear that an interaction has occurred; see below). If the default value is t, the system acts as though the user responded to the query by typing the first response listed among the acceptable responses. If the default value is neither nil nor t, then it must be a keyword and one of the acceptable responses. In that case, the system acts as though the user responded with the given keyword.

Next, we discuss how the ld-query-control-alist assigns a default value to each query id. It assigns each id the first value paired with the id in the alist, or, if no such pair appears in the alist, it assigns the final cdr^ of the alist as the value. Thus, nil assigns all ids nil. T assigns all ids t. '((:filter . nil) (:sysdef . :n) . t) assigns nil to the :filter query, :n to the :sysdef query, and t to all others.

It remains only to discuss how the system prints text when the default value is not nil, i.e., when the query is handled without interaction. In fact, it is allowed to pair a query id with a singleton list containing a keyword, rather than a keyword, and this indicates that no printing is to be done. Thus for the example above, :sysdef queries would be handled noninteractively, with printing done to simulate the interaction. But if we change the example so that :sysdef is paired with

`(:n)`, i.e., if `ld-query-control-alist` is `'((:filter . nil) (:sysdef :n) . t)`, then no such printing would take place for `sysdef` queries. Instead, the default value of `:n` would be assigned "quietly".

8.56 LD-REDEFINITION-ACTION

LD-REDEFINITION-ACTION to allow redefinition without undoing

`Ld-redefinition-action` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-redefinition-action state)` and the updater is `(set-ld-redefinition-action val state)`. **If `ld-redefinition-action` is `non-nil` then ACL2 is liable to be made unsafe or unsound by ill-considered definitions**. The keyword command `:redef^` will set `ld-redefinition-action` to a convenient setting allowing unsound redefinition. See below.

When `ld-redefinition-action` is `nil`, redefinition is prohibited. In that case, an error message is printed upon any attempt to introduce a name that is already in use. There is one exception to this rule. It is permitted to redefine a function symbol in `:program^` mode to be a function symbol in `:logic^` mode provided the formals and body remain the same. This is the standard way a function "comes into" logical existence.

Throughout the rest of this discussion we exclude from our meaning of "redefinition" the case in which a function in `:program^` mode is identically redefined in `:logic^` mode. At one time, ACL2 freely permitted the signature^-preserving redefinition of `:program^` mode functions but it no longer does. See Section 10.200 [REDEFINING-PROGRAMS], page 484.

When `ld-redefinition-action` is non-`nil`, you are allowed to redefine a name that is already in use. **The system may be rendered unsound** by such an act. It is important to understand how dangerous redefinition is. Suppose `fn` is a function symbol that is called from within some other function, say `g`. Suppose `fn` is redefined so that its arity changes. Then the definition of `g` is rendered syntactically ill-formed by the redefinition. This can be devastating since the entire ACL2 system assumes that terms in its data base are well-formed. For example, if ACL2 executes `g` by running the corresponding function in raw Common Lisp the redefinition of `fn` may cause raw lisp to break in irreparable ways. As Lisp programmers we live with this all the time by following the simple rule: after changing the syntax of a function don't run any function that calls it via its old syntax. This rule also works in the context of the evaluation of ACL2 functions, but it is harder to follow in the context of ACL2 deductions, since it is hard to know whether the data base contains a path leading the theorem prover from facts about one function to facts about another. Finally, of course, even if the data base is still syntactically well-formed there is no assurance that all the rules stored in it are valid. For example, theorems proved about `g` survive the redefinition of `fn` but may have crucially depended on the properties of the old `fn`. In summary, we repeat the warning: **all bets are off if you set `ld-redefinition-action` TO non-`nil`.**

If at any point in a session you wish to see the list of all names that have been redefined, see Section 8.75 [REDEFINED-NAMES], page 337.

That said, we'll give you enough rope to hang yourself. When `ld-redefinition-action` is non-`nil`, it must be a pair, `(a . b)`. The value of `a` determines how the system interacts with you when a redefinition is submitted. The value of `b` allows you to specify how the property list of the redefined name is to be "renewed" before the redefinition.

There are several dimensions to the space of possibilities controlled by part a: Do you want to be queried each time you redefine a name, so you can confirm your intention? (We sometimes make typing mistakes or simply forget we have used that name already.) Do you want to see a warning stating that the name has been redefined? Do you want ACL2 system functions given special protection from possible redefinition? Below are the choices for part a:

> `:query` — every attempt to redefine a name will produce a query. The query will allow you to abort the redefinition or proceed. It will will also allow you to specify the part `b` for this redefinition. `:Query` is the recommended setting for users who wish to dabble in redefinition.

> `:warn` — any user-defined function may be redefined but a post-redefinition warning is printed. The attempt to redefine a system name produces a query. If you are prototyping and testing a big system in ACL2 this is probably the desired setting for part `a`.

> `:doit` — any user-defined function may be redefined silently (without query or warning) but when an attempt is made to redefine a system function, a query is made. This setting is recommended when you start making massive changes to your prototyped system (and tire of even the warning messages issued by `:warn`).

In support of our own ACL2 systems programming^ there are two other settings. We suggest ordinary users not use them.

> `:warn!` — every attempt to redefine a name produces a warning but no query. Since ACL2 system functions can be redefined this way, this setting should be used by the only-slightly-less-than supremely confident ACL2 system hacker.

`:doit!` — this setting allows any name to be redefined silently (without query or warnings). ACL2 system functions are fair game. This setting is reserved for the supremely confident ACL2 system hacker. (Actually, this setting is used when we are loading massively modified versions of the ACL2 source files.)

Part `b` of `ld-redefinition-action` tells the system how to "renew" the property list of the name being redefined. There are two choices:

`:erase` — erase all properties stored under the name, or

`:overwrite` — preserve existing properties and let the redefining overwrite them.

It should be stressed that neither of these `b` settings is guaranteed to result in an entirely satisfactory state of affairs after the redefinition. Roughly speaking, `:erase` returns the property list of the name to the state it was in when the name was first introduced. Lemmas, type information, etc., stored under that name are lost. Is that what you wanted? Sometimes it is, as when the old definition is "completely wrong." But other times the old definition was "almost right" in the sense that some of the work done with it is still (intended to be) valid. In that case, `:overwrite` might be the correct `b` setting. For example if `fn` was a function and is being re-`defun^`'d with the same signature^, then the properties stored by the new `defun^` should overwrite those stored by the old `defun^` but the properties stored by `defthm^`s will be preserved.

The `b` setting is only used as the default action when no query is made. If you choose a setting for part a that produces a query then you will have the opportunity, for each redefinition, to specify whether the property list is to be erased or overwritten.

The keyword command `:redef^` sets `ld-redefinition-action` to the pair `(:query . :overwrite)`. Since the resulting query will give you the chance to specify `:erase` instead of `:overwrite`, this setting is quite convenient. But when you are engaged in heavy-duty prototyping, you may wish to use a setting such as `:warn` or even `:doit`. For that you will have to invoke a form such as:

```
(set-ld-redefinition-action '(:doit . :overwrite) state) .
```

`Encapsulate^` causes somewhat odd interaction with the user if redefinition occurs within the encapsulation because the encapsulate^d event list is processed several times. For example, if the redefinition action causes a query and a non-local definition is actually a redefinition, then the query will be posed twice, once during each pass. C'est la vie.

Finally, it should be stressed again that redefinition is dangerous because not all of the rules about a name are stored on the property list of the name. Thus, redefinition can render ill-formed terms stored elsewhere in the data base or can preserve now-invalid rules.

8.57 LD-SKIP-PROOFSP

LD-SKIP-PROOFSP how carefully ACL2 processes your command^s

```
Examples:
ACL2 !>(set-ld-skip-proofsp t state)
 T
ACL2 !s>(set-ld-skip-proofsp nil state)
 NIL
ACL2 !>(set-ld-skip-proofsp 'include-book state)
 INCLUDE-BOOK
ACL2 !s>
```

A global variable in the ACL2 `state^`, called `'ld-skip-proofsp`, determines the thoroughness with which ACL2 processes your command^s. This variable may take on one of three values: `t`, `nil` or `'include-book^`. When `ld-skip-proofsp` is non-`nil`, the system assumes that which ought to be proved and is thus unsound. The form `(set-ld-skip-proofsp flg state)` is the general-purpose way of setting `ld-skip-proofsp`. This global variable is an "`ld^` special," which is to say, you may call `ld^` in such a way as to "bind" this variable for the dynamic extent of the `ld^`.

When `ld-skip-proofsp` is non-`nil`, the default prompt^ displays the character `s`. Thus, the prompt^

```
ACL2 !s>
```

means that the default defun-mode^ is `:logic^` (otherwise the character `p`, for `:program^`, would also be printed; see Section 8.17 [DEFAULT-PRINT-PROMPT], page 255) but "proofs are being skipped."

Observe that there are two legal non-`nil` values, `t` and `'include-book^`. When `ld-skip-proofsp` is `t`, ACL2 skips all proof obligations but otherwise performs all other required analysis of input events^. When `ld-skip-proofsp` is `'include-book^`, ACL2 skips not only proof obligations but all analysis except that required to compute the effect of successfully executed events^. To explain the distinction, let us consider one particular event, say a `defun^`. Very roughly speaking, a `defun^` event normally involves a check of the syntactic well-formedness of the submitted definition, the generation and proof of the termination conditions, and the computation and storage of various rules such as a `:definition^` rule and some `:type-prescription^` rules. By "normally" above we mean when `ld-skip-proofsp` is `nil`. How does a `defun^` behave when `ld-skip-proofsp` is non-`nil`?

If `ld-skip-proofsp` is `t`, then `defun^` performs the syntactic well-formedness checks and computes and stores the various rules, but it does not actually carry out the termination proofs. If `ld-skip-proofsp` is `'include-book^`, `defun^` does not do the syntactic well-formedness check nor does it carry out the termination proof. Instead, it merely computes and stores the rules under the assumption that the checks and proofs would all succeed. Observe that a setting of `'include-book^` is "stronger" than a setting of `t` in the sense that `'include-book^` causes `defun^` to assume even more about the admissibility of the event than `t` does.

As one might infer from the choice of name, the `include-book^` event sets `ld-skip-proofsp` to `'include-book^` when processing the events^ in a book being loaded. Thus, `include-book^` does the miminal work necessary to carry out the effects of every event in the book. The syntactic checks and proof obligations were, presumably, successfully carried out when the book was certified.

A non-`nil` value for `ld-skip-proofsp` also affects the system's output messages. Event summaries (the paragraphs that begin "Summary" and display the event forms, rules used, etc.) are not printed when `ld-skip-proofsp` is non-`nil`. Warnings and observations are printed when `ld-skip-proofsp` is `t` but are not printed when it is `'include-book^`.

Intuitively, `ld-skip-proofsp t` means skip just the proofs and otherwise do all the work normally required for an event; while `ld-skip-proofsp 'include-book^` is "stronger" and means do as little as possible to process events^. In accordance with this intuition, `local^` events^ are processed when `ld-skip-proofsp` is `t` but are skipped when `ld-skip-proofsp` is `'include-book^`.

The ACL2 system itself uses only two settings, `nil` and `'include-book^`, the latter being used only when executing the events^ inside of a book being included. The `ld-skip-proofsp` setting of `t` is provided as a convenience to the user. For example, suppose one has a file of events^. By loading it with `ld^` with `ld-skip-proofsp` set to `t`, the events^ can all be checked for syntactic correctness and assumed without proof. This is a convenient way to recover a state lost by a system crash or to experiment with a modification of an events^ file.

The foregoing discussion is actually based on a lie. `ld-skip-proofsp` is allowed two other values, `'initialize-acl2` and `'include-book-with-locals`. The first causes behavior similar to `t` but skips `local^` events^ and avoids some error checks that would otherwise prevent ACL2 from properly booting. The second is identical to `'include-book^` but also executes `local^` events^.

These additional values are not intended for use by the user, but no barriers to their use have been erected.

We close by reminding the user that ACL2 is potentially unsound if `ld-skip-proofsp` is ever set by the user. We provide access to it simply to allow experimentation and rapid reconstruction of lost or modified logical world^s.

8.58 LD-VERBOSE

LD-VERBOSE determines whether `ld^` prints "ACL2 Loading ..."

`Ld-verbose` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(ld-verbose state)` and the updater is `(set-ld-verbose val state)`. `Ld-verbose` must be `t`, `nil` or a string or `consp^` suitable for `fmt^` printing via the `~@` command. The initial value of `ld-verbose` is a `fmt^` message that prints the ACL2 version number, `ld^` level and connected book directory.

Before processing the forms in `standard-oi^`, `ld^` may print a header. The printing of this header is controlled by `ld-verbose`. If `ld-verbose` is `nil`, no header is printed. If it is `t`, `ld^` prints the message

```
ACL2 loading <file>
```

where `<file>` is the input channel supplied to `ld^`. A similar message is printed when `ld^` completes. If `ld-verbose` is neither `t` nor `nil` then it is presumably a header and is printed with the `~@ fmt^` directive before `ld^` begins to read and process forms. In this case the `~@ fmt^` directive is interpreted in an environment in which `#\v` is the ACL2 version string, `#\l` is the level of the current recursion in `ld^` and/or `wormhole^`, and `#\c` is the connected book directory `(cbd)`.

8.59 LEMMA-INSTANCE

LEMMA-INSTANCE an object denoting an instance of a theorem

Lemma instances are the objects one provides via `:use` and `:by` hints^ (see Section 8.41 [HINTS], page 297) to bring to the theorem prover's attention some previously proved or easily provable fact. A typical use of the `:use` hint is given below. The value specified is a list of five lemma instances.

```
:use (reverse-reverse
      (:type-prescription app)
      (:instance assoc-of-app
                 (x a) (y b) (z c))
      (:functional-instance p-f
                            (p consp) (f flatten))
      (:instance (:theorem (equal x x))
                 (x (flatten a))))
```

Observe that an event name can be a lemma instance. The `:use` hint allows a single lemma instance to be provided in lieu of a list, as in:

```
:use reverse-reverse
```

or

```
:use (:instance assoc-of-app (x a) (y b) (z c))
```

A lemma instance denotes a formula which is either known to be a theorem or which must be proved to be a theorem before it can be used. To use a lemma instance in a particular subgoal, the theorem prover adds the formula as a hypothesis to the subgoal before the normal theorem proving heuristics are applied.

A lemma instance, or `lmi`, is of one of the following five forms:

(1) `name`, where `name` names a previously proved theorem, axiom, or definition and denotes the formula (theorem) of that name.

(2) `rune`, where `rune` is a rune^ (see Section 15.10 [RUNE], page 675) denoting the `:corollary^` justifying the rule named by the rune^.

(3) `(:theorem term)`, where `term` is any term alleged to be a theorem. Such a lemma instance denotes the formula `term`. But before using such a lemma instance the system will undertake to prove `term`.

(4) `(:instance lmi (v1 t1) ... (vn tn))`, where `lmi` is recursively a lemma instance, the `vi`'s are distinct variables and the `ti`'s are terms. Such a lemma instance denotes the formula obtained by instantiating the formula denoted by `lmi`, replacing each `vi` by `ti`.

(5) `(:functional-instance lmi (f1 g1) ... (fn gn))`, where `lmi` is recursively a lemma instance and each `fi` is an "instantiable" function symbol of arity `ni` and `gi` is a function symbol or a pseudo-lambda expression of arity `ni`. An instantiable function symbol is any defined or constrained function symbol except the primitives `not^`, `member^`, `implies^`, and `e0-ord-<^`, and a few others, as listed by the constant `*non-instantiable-primitives*`. These are built-in in such a way that we cannot recover the constraint^s on them. A pseudo-lambda expression is an expression of the form `(lambda (v1 ... vn) body)` where the `vi` are distinct variable symbols and `body` is any term. No *a priori* relation is imposed between the `vi` and the variables of `body`, i.e., `body` may ignore some `vi`'s and may contain "free" variables. However, we do not permit `v` to occur freely in `body` if the functional substitution is to be applied to any formula (`lmi` or the constraint^s to be satisfied) that contains `v` as a variable. This is our draconian restriction to avoid capture. If you happen to violate this restriction by choosing a `v` that does occur, say in one of the relevant constraint^s, an informative error message will be printed. That message will list for you the illegal choices for `v` in the context in which the functional substitution is offered. A `:functional-substitution` lemma instance denotes the formula obtained by functionally instantiating the formula denoted by `lmi`, replacing `fi` by `gi`. However, before such a lemma instance can be used, the system will undertake to prove that the `gi`'s satisfy the constraints (see Section 8.12 [CONSTRAINT], page 247) on the `fi`'s. Any such constraint^ that was generated and proved by ACL2 on behalf of a previously-proved event will be considered proved.

8.60 LOCAL-INCOMPATIBILITY

LOCAL-INCOMPATIBILITY when non-local events^ won't replay in isolation

Sometimes a "local^ incompatibility" is reported while attempting to embed some events^, as in an `encapsulate^` or `include-book^`. This is generally due to the use of a locally defined name in a non-local event or the failure to make a witnessing definition local^.

Local^ incompatibilities may be detected while trying to execute the strictly non-local events^ of an embedding. For example, `encapsulate^` operates by first executing all the events^ (local^ and non-local) with `ld-skip-proofsp^ nil`, to confirm that they are all admissible. Then it attempts merely to assume the non-local ones to create the desired theory, by executing the events^ with `ld-skip-proofsp^` set to `'include-book^`. Similarly, `include-book^` assumes the non-local ones, with the understanding that a previously successful `certify-book^` has performed the admissiblity check.

How can a sequence of events^ admitted with `ld-skip-proofsp^ nil` fail when `ld-skip-proofsp^` is `'include-book^`? The key observation is that in the latter case only the non-local events^ are processed. The local^ ones are skipped and so the non-local ones must not depend upon them.

Two typical mistakes are suggested by the detection of a local^ incompatibility: (1) a locally defined function or macro was used in a non-`local^` event (and, in the case of `encapsulate^`, was not included among the signature^s) and (2) the witnessing definition of a function that was included among the signature^s of an `encapsulate^` was not made `local^`.

An example of mistake (1) would be to include among your encapsulate^d events^ both `(local (defun fn ...))` and `(defthm lemma (implies (fn ...) ...))`. Observe that `fn` is defined locally but a formula involving `fn` is defined non-locally. In this case, either the `defthm^` should be made local^ or the `defun^` should be made non-local.

An example of mistake (2) would be to include `(fn (x) t)` among your signature^s and then to write `(defun fn (x) ...)` in your events^, instead of `(local (defun fn ...))`.

One subtle aspect of `encapsulate^` is that if you constrain any member of a mutually recursive clique you must define the entire clique locally and then you must constrain those members of it you want axiomatized non-locally.

Errors due to local^ incompatibility should never occur in the assumption of a fully certified book. Certification insures against it. Therefore, if `include-book^` reports an incompatibility, we assert that earlier in the processing of the `include-book^` a warning was printed advising you that some book was uncertified. If this is not the case — if `include-book^` reports an incompatibility and there has been no prior warning about lack of certification — please report it to us.

When a local^ incompatibility is detected, we roll-back to the world^ in which we started the `encapsulate^` or `include-book^`. That is, we discard the intermediate world^ created by trying to process the events^ skipping proofs. This is clean, but we realize it is very frustrating because the entire sequence of events^ must be processed from scratch. Assuming that the embedded events^ were, once upon a time, processed as top-level command^s (after all, at some point you managed to create this sequence of command^s so that the local^ and non-local ones together could survive a pass in which proofs were done), it stands to reason that we could define a predicate that would determine then, before you attempted to embed them, if local^ incompatibilities exist. We hope to do that, eventually.

8.61 LOGICAL-NAME

LOGICAL-NAME a name created by a logical event

```
Examples:
assoc
caddr
+
"ACL2-USER"
"arith"
"project/task-1/arith.lisp"
:here
```

A logical name is either a name introduced by some event, such as `defun^`, `defthm^`, or `include-book^`, or else is the keyword `:here`, which refers to the most recent such event. See Chapter 6 [EVENTS], page 153. Every logical name is either a symbol or a string. For the syntactic rules on names, see Section 8.65 [NAME], page 329. The symbols name functions, macros, constants, axioms, theorems, labels, and theories^. The strings name packages or books^. We permit the keyword symbol `:here` to be used as a logical name denoting the most recently completed event.

The logical name introduced by an include-book^ is the full book name string for the book (see Section 3.7 [FULL-BOOK-NAME], page 103). Thus, under the appropriate setting for the current book directory (see Section 3.4 [CBD], page 97) the event `(include-book "arith")` may introduce the logical name

```
"/usr/home/smith/project/task-1/arith.lisp" .
```

Under a different `cbd^` setting, it may introduce a different logical name, perhaps

```
"/local/src/acl2/library/arith.lisp" .
```

It is possible that identical `include-book^` events forms in a session introduce two different logical names because of the current book directory.

A logical name that is a string is either a package name or a book name. If it is not a package name, we support various conventions to interpret it as a book name. If it does not end with the string `".lisp"` we extend it appropriately. Then, we search for any book name that has the given logical name as a terminal substring. Suppose `(include-book "arith")` is the only include-book^ so far and that `"/usr/home/smith/project/task-1/arith.lisp"` is the source file it processed. Then `"arith"`, `"arith.lisp"` and `"task-1/arith.lisp"` are all logical names identifying that `include-book^` event (unless they are package names). Now suppose a second `(include-book "arith")` is executed and processes `"/local/src/acl2/library/arith.lisp"`. Then `"arith"` is no longer a logical name, because it is ambiguous. However, `"task-1/arith"` is a logical name for the first `include-book^` and `"library/arith"` is a logical name for the second. Indeed, the first can be named by `"1/arith"` and the second by `"y/arith"`.

Logical names are used primarily in the theory manipulation functions, e.g., `universal-theory^` and `current-theory^` with which you may obtain some standard theories^ as of some point in the historical past. The reference points are the introductions of logical names, i.e., the past is determined by the events^ it contains. One might ask, "Why not discuss the past with the much more flexible language of command^ descriptors?" (See Section 8.11 [COMMAND-DESCRIPTOR], page 245.) The reason is that inside of such events^ as `encapsulate^` or macro command^s that expand to `progn^`s of events^, command^ descriptors provide too coarse a grain.

When logical names are used as referents in theory expressions used in books^, one must consider the possibility that the defining event within the book in question becomes redundant by the definition of the name prior to the assumption of the book. See Section 8.76 [REDUNDANT-EVENTS], page 337.

8.62 LOOP-STOPPER

LOOP-STOPPER limit application of permutative rewrite rules

See Chapter 14 [RULE-CLASSES], page 617 for a discussion of the syntax of the `:loop-stopper` field of `:rewrite^` rule-classes. Here we describe how that field is used, and also how that field is created when the user does not explicitly supply it.

For example, the built-in `:rewrite^` rule `commutativity-of-+`,

```
(implies (and (acl2-numberp x)
              (acl2-numberp y))
         (equal (+ x y) (+ y x))),
```

creates a rewrite rule with a loop-stopper of `((x y binary-+))`. This means, very roughly, that the term corresponding to `y` must be "smaller" than the term corresponding to `x` in order for this rule to apply. However, the presence of `binary-+^` in the list means that certain functions that are "invisible" with respect to `binary-+^` (by default, `unary--^` is the only such function) are more or less ignored when making this "smaller" test. We are much more precise below.

Our explanation of loop-stopping is in four parts. First we discuss ACL2's notion of "term order." Next, we bring in the notion of "invisibility", and use it together with term order to define orderings on terms that are used in the loop-stopping algorithm. Third, we describe that algorithm. These topics all assume that we have in hand the `:loop-stopper` field of a given rewrite rule; the fourth and final topic describes how that field is calculated when it is not supplied by the user.

ACL2 must sometimes decide which of two terms is syntactically simpler. It uses a total ordering on terms, called the "term order." Under this ordering constants such as `'(a b c)` are simpler than terms containing variables such as `x` and `(+ 1 x)`. Terms containing variables are ordered according to how many occurrences of variables there are. Thus `x` and `(+ 1 x)` are both simpler than `(cons x x)` and `(+ x y)`. If variable counts do not decide the order, then the number of function applications are tried. Thus `(cons x x)` is simpler than `(+ x (+ 1 y))` because the latter has one more function application. Finally, if the number of function applications do not decide the order, a lexicographic ordering on Lisp objects is used. See Section 8.86 [TERM-ORDER], page 355 for details.

When the loop-stopping algorithm is controlling the use of permutative `:rewrite^` rules it allows `term1` to be moved leftward over `term2` only if `term1` is smaller, in a suitable sense. Note: The sense used in loop-stopping is **not** the above explained term order but a more complicated ordering described below. The use of a total ordering stops rules like commutativity from looping indefinitely because it allows `(+ b a)` to be permuted to `(+ a b)` but not vice versa, assuming `a` is smaller than `b` in the ordering. Given a set of permutative rules that allows arbitrary permutations of the tips of a tree of function calls, this will normalize the tree so that the smallest argument is leftmost and the arguments ascend in the order toward the right. Thus, for example, if the same argument appears twice in the tree, as `x` does in the `binary-+^` tree denoted by the term `(+ a x b x)`, then when the allowed permutations are done, all occurrences of the duplicated argument in the tree will be adjacent, e.g., the tree above will be normalized to `(+ a b x x)`.

Suppose the loop-stopping algorithm used term order, as noted above, and consider the `binary-+^` tree denoted by `(+ x y (- x))`. The arguments here are in ascending term order already. Thus, no permutative rules are applied. But because we are inside a `+-expression` it is very convenient if `x` and `(- x)` could be given virtually the same position in the ordering so that `y` is not allowed to separate them. This would allow such rules as `(+ i (- i) j) = j` to be applied. In support of this, the ordering used in the control of permutative rules allows certain unary functions, e.g., the unary minus function above, to be "invisible" with respect to certain "surrounding" functions, e.g., `+^` function above.

Briefly, a unary function symbol `fn1` is invisible with respect to a function symbol `fn2` if `fn2` belongs to the value of `fn1` in `invisible-fns-alist^`; also see Section 6.30 [SET-INVISIBLE-FNS-ALIST], page 189, which explains its format and how it can be set by the user. Roughly speaking, "invisible" function symbols are ignored for the purposes of the term-order test.

Consider the example above, `(+ x y (- x))`. The translated version of this term is `(binary-+ x (binary-+ y (unary-- x)))`. The initial `invisible-fns-alist^` makes `unary--^` invisible with repect to `binary-+^`. The commutativity rule for `binary-+^` will attempt to swap `y` and `(unary-- x)` and the loop-stopping algorithm is called to approve or disapprove. If term order is used, the swap will be disapproved. But term order is not used. While the loop-stopping algorithm is permuting arguments inside a `binary-+^` expression, `unary--^` is invisible. Thus, insted of comparing `y` with `(unary-- x)`, the loop-stopping algorithm compares `y` with `x`, approving the swap because `x` comes before `y`.

Here is a more precise specification of the total order used for loop-stopping with respect to a list, `fns`, of functions that are to be considered invisible. Let `x` and `y` be distinct terms; we specify when "`x` is smaller than `y` with respect to `fns`." If `x` is the application of a unary function symbol that belongs to `fns`, replace `x` by its argument. Repeat this process until the result is not the application of such a function; let us call the result `x-guts`. Similarly obtain `y-guts` from `y`. Now if `x-guts` is the same term as `y-guts`, then `x` is smaller than `y` in this order iff `x` is smaller than `y` in the standard term order. On the other hand, if `x-guts` is different than `y-guts`, then `x` is smaller than `y` in this order iff `x-guts` is smaller than `y-guts` in the standard term order.

Now we may describe the loop-stopping algorithm. Consider a rewrite rule with conclusion `(equiv lhs rhs)` that applies to a term `x` in a given context; see Section 14.14 [REWRITE], page 657. Suppose that this rewrite rule has a loop-stopper field (technically, the `:heuristic-info` field) of `((x1 y1 . fns-1) ... (xn yn . fns-n))`. (Note that this field can be observed by using the command `:pr^` with the name of the rule; see Section 7.11 [PR], page 211.) We describe when rewriting is permitted. The simplest case is when the loop-stopper list is `nil` (i.e., `n` is `0`); in that case, rewriting is permitted. Otherwise, for each `i` from 1 to `n` let `xi'` be the actual term corresponding to the variable `xi` when `lhs` is matched against the term to be rewritten, and similarly correspond `yi'` with `y`. If `xi'` and `yi'` are the same term for all `i`, then rewriting is not permitted. Otherwise, let `k` be the least `i` such that `xi'` and `yi'` are distinct. Let `fns` be the list of all functions that are invisible with respect to every function in `fns-k`, if `fns-k` is non-empty; otherwise, let `fns` be `nil`. Then rewriting is permitted if and only if `yi'` is smaller than `xi'` with respect to `fns`, in the sense defined in the preceding paragraph.

It remains only to describe how the loop-stopper field is calculated for a rewrite rule when this field is not supplied by the user. (On the other hand, to see how the user may specify the `:loop-stopper`, see Chapter 14 [RULE-CLASSES], page 617.) Suppose the conclusion of the rule is of the form `(equiv lhs rhs)`. First of all, if `rhs` is not an instance of the left hand side by a substitution whose range is a list of distinct variables, then the loop-stopper field is `nil`. Otherwise, consider all pairs `(u . v)` from this substitution with the property that the first occurrence of `v` appears in front of the first occurrence of `u` in the print representation of `rhs`. For each such `u` and `v`, form a list `fns` of all functions `fn` in `lhs` with the property that `u` or `v` (or both) appears as a top-level argument of a subterm of `lhs` with function symbol `fn`. Then the loop-stopper for this rewrite rule is a list of all lists `(u v . fns)`.

8.63 LP

LP the Common Lisp entry to ACL2

To enter the ACL2 command^ loop from Common Lisp, call the Common Lisp program `lp` (which stands for "loop," as in "read-eval-print loop" or "command^ loop.") The ACL2 command^ loop is actually coded in ACL2 as the function `ld^` (which stands for "load"). The command^ loop is just what you get by loading from the standard object input channel, `*standard-oi*^`. Calling `ld^` directly from Common Lisp is possible but fragile because hard lisp errors or aborts throw you out of `ld^` and back to the top-level of Common Lisp. `Lp` calls `ld^` in such a way as to prevent this and is thus the standard way to get into the ACL2 command^ loop. Also see Section 8.4 [ACL2-CUSTOMIZATION], page 225 for information on the loading of an initialization file.

All of the visible functionality of `lp` is in fact provided by `ld^`, which is written in ACL2 itself. Therefore, you should see Section 9.7 [LD], page 380 for detailed documentation^ of the ACL2 command^ loop. We sketch it below, for novice users.

Every expression typed to the ACL2 top-level must be an ACL2 expression.

Any ACL2 expression containing no variables may be evaluated. Because of the applicative nature of ACL2, we make a special exception for the variable `state^`. If this variable occurs in the form, it is taken to mean the "current" ACL2 state^ object. If the form returns a new state^ object as one of its values, then that is considered the new "current" state^ for the evaluation of the subsequent form. See Section 8.81 [STATE], page 344.

If the form read is a single keyword, e.g., `:pe^` or `:ubt^`, then special procedures are followed. See Section 8.46 [KEYWORD-COMMANDS], page 306.

The command^ loop keeps track of the command^s you have typed and allows you to review them, display them, and roll the logical state^ back to that created by any command^. See Chapter 7 [HISTORY], page 203.

ACL2 makes the convention that if a top-level form returns three values, the last of which is an ACL2 state^, then the first is treated as a flag that means "an error occurred," the second is the value to be printed if no error occurred, and the third is (of course) the new state^. When "an error occurs" no value is printed. Thus, if you execute a top-level form that happens to return three such values, only the second will be printed (and that will only happen if the first is `nil`!). See Section 9.7 [LD], page 380 for details.

8.64 MACRO-ARGS

MACRO-ARGS the formals list of a macro definition

```
Examples:
(x y z)
(x y z &optional max (base '10 basep))
(x y &rest rst)
(x y &key max base)
(&whole sexpr x y)
```

The "lambda-list" of a macro definition may include simple formal parameter names as well as appropriate uses of the following `lambda`-list keywords from CLTL (pp. 60 and 145):

```
&optional,
&rest,
&key,
&whole,
&body, and
&allow-other-keys.
```

ACL2 does not support `&aux` and `&environment`. In addition, we make the following restrictions:

(1) initialization forms in `&optional` and `&key` specifiers must be quoted values;

(2) `&allow-other-keys` may only be used once, as the last specifier; and

(3) destructuring is not allowed.

You are encouraged to experiment with the macro facility. One way to do so is to define a macro that does nothing but return the quotation of its arguments, e.g.,

```
(defmacro demo (x y &optional opt &key key1 key2)
  (list 'quote (list x y opt key1 key2)))
```

You may then execute the macro on some sample forms, e.g.,

```
  term                          value
(demo 1 2)                 (1 2 NIL NIL NIL)
(demo 1 2 3)               (1 2 3 NIL NIL)
(demo 1 2 :key1 3)         error:  non-even key/value arglist
                           (because :key1 is used as opt)
(demo 1 2 3 :key2 5)       (1 2 3 NIL 5)
```

Also see Section 9.16 [TRANS], page 390.

8.65 NAME

NAME syntactic rules on logical names

```
Examples                    Counter-Examples

PRIMEP                 91          (not a symbolp)
F-AC-23                :CHK-LIST  (in KEYWORD package)
1AX                    *PACKAGE*  (in the Lisp Package)
|Prime Number|         1E3         (not a symbolp)
```

Many different ACL2 entities have names, e.g., functions, macros, variables, constants, packages, theorems, theories^, etc. Package names are strings. All other names are symbols and may not be in the "KEYWORD" package. Moreover, names of functions, macros, constrained functions, and constants, other than those that are predefined, must not be in the "main Lisp package" (which is ("LISP" or "COMMON-LISP", according to whether we are following CLTL1 or CLTL2). An analogous restriction on variables is given below.

T, nil, and all names above except those that begin with ampersand (&) are names of variables or constants. T, nil, and those names beginning and ending with star (*) are constants. All other such names are variables.

Note that names that start with ampersand, such as &rest, may be lambda list keywords and are reserved for such use whether or not they are in the CLTL constant lambda-list-keywords. (See pg 82 of CLTL2.) That is, these may not be used as variables. Unlike constants, variables may be in the main Lisp package as long as they are in the original list of imports from that package to ACL2, the list *common-lisp-symbols-from-main-lisp-package*, and do not belong to a list of symbols that are potentially "global." This latter list is the value of *common-lisp-specials-and-constants*.

Our restrictions pertaining to the main Lisp package take into account that some symbols, e.g., `lambda-list-keywords` and `boole-c2`, are "special." Permitting them to be bound could have harmful effects. In addition, the Common Lisp language does not allow certain manipulations with many symbols of that package. So, we stay away from them, except for allowing certain variables as indicated above.

8.66 OBDD

OBDD   ordered binary decision diagrams with rewriting

See Chapter 2 [BDD], page 69 for information on this topic.

8.67 OTF-FLG

OTF-FLG   pushing all the initial subgoals

The value of this flag is normally `nil`. If you want to prevent the theorem prover from abandoning its initial work upon pushing the second subgoal, set `:otf-flg` to `t`.

Suppose you submit a conjecture to the theorem prover and the system splits it up into many subgoals. Any subgoal not proved by other methods is eventually set aside for an attempted induction proof. But upon setting aside the second such subgoal, the system chickens out and decides that rather than prove n>1 subgoals inductively, it will abandon its initial work and attempt induction on the originally submitted conjecture. The `:otf-flg` (Onward Thru the Fog) allows you to override this chickening out. When `:otf-flg` is `t`, the system will push all the initial subgoals and proceed to try to prove each, independently, by induction.

Even when you don't expect induction to be used or to succeed, setting the `:otf-flg` is a good way to force the system to generate and display all the initial subgoals.

8.68 PACKAGE-REINCARNATION-IMPORT-RESTRICTIONS

PACKAGE-REINCARNATION-IMPORT-RESTRICTIONS   re-defining undone `defpkg^s`

Suppose `(defpkg "pkg" imports)` is the most recently executed successful definition of `"pkg"` in this ACL2 session and that it has since been undone, as by `:ubt^`. Any future attempt in this session to define `"pkg"` as a package must specify an identical imports list.

The restriction stems from the need to implement the reinstallation of saved logical world^s as in error recovery and the `:oops^` command^. Suppose that the new `defpkg^` attempts to import some symbol, `a::sym`, not imported by the previous definition of `"pkg"`. Because it was not imported in the original package, the symbol `pkg::sym`, different from `a::sym`, may well have been created and may well be used in some saved world^s. Those saved world^s are Common Lisp objects being held for you "behind the scenes." In order to import `a::sym` into `"pkg"` now we would have to unintern `pkg::sym`, rendering those saved world^s ill-formed. It is because of saved world^s that we do not actually clear out a package when it is undone.

At one point we thought it was sound to allow the new `defpkg^` to import a subset of the old. But that is incorrect. Suppose the old definition of `"pkg"` imported `a::sym` but the new one does not. Suppose we allowed that and implemented it simply by setting the imports of `"pkg"` to the new subset. Then consider the conjecture `(eq a::sym pkg::sym)`. This ought not be a theorem because we did not import `a::sym` into `"pkg"`. But in fact in AKCL it is a theorem because `pkg::sym` is read as `a::sym` because of the old imports.

8.69 PRINT-DOC-START-COLUMN

PRINT-DOC-START-COLUMN printing the one-liner

```
Examples:
(assign print-doc-start-column nil)
(assign print-doc-start-column 17)
```

This state^ global variable controls the column in which the "one-liner" of a formatted documentation^ string is printed. Generally, when `:doc^` is used to print a documentation^ string, the name of the documented concept is printed and then `:doc^` tabs over to `print-doc-start-column` and prints the one-liner. If the name extends past the desired column, `:doc^` outputs a carriage return and then tabs over to the column. If `print-doc-start-column` is `nil`, `:doc^` just starts the one-liner two spaces from the end of the name, on the same line. The initial value of `print-doc-start-column` is 15.

8.70 PROMPT

PROMPT the prompt printed by `ld^`

The prompt printed by ACL2 conveys information about various "modes." See Section 8.17 [DEFAULT-PRINT-PROMPT], page 255 and see Section 8.54 [LD-PROMPT], page 312 for details.

8.71 PROOF-OF-WELL-FOUNDEDNESS

PROOF-OF-WELL-FOUNDEDNESS a proof that `e0-ord-<^` is well-founded on `e0-ordinalp^`s

The soundness of ACL2 rests in part on the well-foundedness of `e0-ord-<^` on `e0-ordinalp^`s. This can be taken as obvious if one is willing to grant that those concepts are simply encodings of the standard mathematical notions of the ordinals below `epsilon-0` and its natural ordering relation. But it is possible to prove that `e0-ord-<^` is well-founded on `e0-ordinalp^`s without having to assert any connection to the ordinals and that is what we do here.

We first observe three facts about `e0-ord-<^` on ordinals that have been proved by ACL2 using only structural induction on lists.

```
(defthm transitivity
  (implies (and (e0-ordinalp x)
                (e0-ordinalp y)
                (e0-ordinalp z)
                (e0-ord-< x y)
                (e0-ord-< y z))
           (e0-ord-< x z))
  :rule-classes nil)

(defthm non-circularity
  (implies (and (e0-ordinalp x)
                (e0-ordinalp y)
                (e0-ord-< x y))
           (not (e0-ord-< y x)))
  :rule-classes nil)

(defthm trichotomy
  (implies (and (e0-ordinalp x)
                (e0-ordinalp y))
           (or (equal x y)
               (e0-ord-< x y)
               (e0-ord-< y x)))
  :rule-classes nil)
```

These three properties establish that `e0-ord-<^` orders the `e0-ordinalp^`s. To put such a statement in the most standard mathematical nomenclature, we can define the function:

```
(defun e0-ord-<= (x y)
  (or (equal x y) (e0-ord-< x y)))
```

and then establish that `e0-ord-<=` is a relation that is a simple, complete (i.e., total) order on ordinals by the following three lemmas, which have been proved:

```
(defthm antisymmetry
  (implies (and (e0-ordinalp x)
                (e0-ordinalp y)
                (e0-ord-<= x y)
                (e0-ord-<= y x))
           (equal x y))
  :rule-classes nil
  :hints (("Goal" :use non-circularity)))

(defthm e0-ord-<=-transitivity
          (implies (and (e0-ordinalp x)
                        (e0-ordinalp y)
                        (e0-ordinalp z)
                        (e0-ord-<= x y)
                        (e0-ord-<= y z))
                   (e0-ord-<= x z))
  :rule-classes nil
  :hints (("Goal" :use transitivity)))

(defthm trichotomy-of-e0-ord-<
  (implies (and (e0-ordinalp x)
                (e0-ordinalp y))
           (or (e0-ord-<= x y)
               (e0-ord-<= y x)))
  :rule-classes nil
  :hints (("Goal" :use trichotomy)))
```

Crucially important to the proof of the well-foundedness of `e0-ord-<^` on `e0-ordinalp^`s is the concept of ordinal-depth, abbreviated `od`:

```
(defun od (l) (if (atom l) 0 (1+ (od (car l)))))
```

If the `od` of an `e0-ordinalp^ x` is smaller than that of an `e0-ordinalp^ y`, then `x` is `e0-ord-<^ y`:

```
(defthm od-implies-ordlessp
```

```
(implies (and (e0-ordinalp x) (e0-ordinalp y)
              (< (od x) (od y)))
         (e0-ord-< x y)))
```

Remark. A consequence of this lemma is the fact that if `s = s(1), s(2), ...` is an infinite, `e0-ord-<^` descending sequence, then `od(s(1)), od(s(2)), ...` is a "weakly" descending sequence of non-negative integers: `od(s(i))` is greater than or equal to `od(s(i+1))`.

Lemma Main. For each non-negative integer `n`, `e0-ord-<^` well-orders the set of `e0-ordinalp^s` with `od` less than or equal to `n` .

```
Base Case.  n = 0.  The e0-ordinalps with 0 od are the non-negative
integers.  On the non-negative integers, e0-ord-< is the same as <.

Induction Step.  n > 0.  We assume that e0-ord-< well-orders the
e0-ordinalps with od less than n.

  If e0-ord-< does not well-order the e0-ordinalps with od less
  than or equal to n, we may let O be the e0-ord-<-least
  e0-ordinalp which is the car of the first member of an infinite,
  e0-ord-< descending sequence of e0-ordinalps of od less than or
  equal to n.  The od of O is n-1.

   Let k be the least integer > 0 such that for some infinite,
   e0-ord-< descending sequence s of e0-ordinalps with od less than
   or equal to n, the first element of s begins with k occurrences
   of O but not k+1 occurrences of O.

     Having fixed O and k, let s = s(1), s(2), ... be an infinite,
     e0-ord-< descending sequence of e0-ordinalps with od less than
     or equal to n such that O occurs exactly k times at the
     beginning of s(1).

        O occurs exactly k times at the beginning of each s(i).
        For suppose that s(j) is the first member of s with exactly
        m occurrences of O at the beginning, m /= k.  If m = 0,
        then the first member of s(j) must be e0-ord-< O,
        contradicting the minimality of O.  If 0 < m < k, then the
        fact that the sequence beginning at s(j) is infinitely
        descending contradicts the minimality of k.  If m > k, then
        s(j) is greater than its predecessor, which has only k
        occurrences of O at the beginning; but this contradicts the
        fact that s is descending.

        Let t = t(1), t(2), ... be the sequence of e0-ordinalps
        that is obtained by letting t(i) be the result of removing
```

```
O from the front of s(i) exactly k times.  t is infinitely
descending.  Furthermore, t(1) begins with an e0-ordinalp
O' that is e0-ord-< O, and hence has od at most N-1 by the
lemma od-implies-ordlessp.  But this contradicts the
minimality of O. Q.E.D.
```

Theorem. `e0-ord-<^` well-orders the `e0-ordinalp^`s. Proof. Every infinite, `e0-ord-<` descending sequence of `e0-ordinalp^`s has the property that each member has `od` less than or equal to the `od`, `n`, of the first member of the sequence. This contradicts Lemma Main. Q.E.D.

8.72 PSEUDO-TERMP

PSEUDO-TERMP a predicate for recognizing term-like s-expressions

```
Example Forms:
(pseudo-termp '(car (cons x 'nil)))      ; has value t
(pseudo-termp '(car x y z))              ; also has value t!
(pseudo-termp '(delta (h x)))            ; has value t
(pseudo-termp '(delta (h x) . 7))        ; has value nil (not a true-listp)
(pseudo-termp '((lambda (x) (car x)) b)) ; has value t
(pseudo-termp '(if x y 123))             ; has value nil (123 is not quoted)
(pseudo-termp '(if x y '123))            ; has value t
```

If `x` is the quotation of a term, then `(pseudo-termp x)` is `t`. However, if `x` is not the quotation of a term it is not necessarily the case that `(pseudo-termp x)` is `nil`.

See Section 8.85 [TERM], page 352 for a discussion of the various meanings of the word "term" in ACL2. In its most strict sense, a term is either a legal variable symbol, a quoted constant, or the application of an `n`-ary function symbol or closed `lambda`-expression to `n` terms. By "legal variable symbol" we exclude constant symbols, such as `t`, `nil`, and `*ts-rational*`. By "quoted constants" we include `'t` (aka `(quote t)`), `'nil`, `'31`, etc., and exclude constant names such as `t`, `nil` and `*ts-rational*`, unquoted constants such as `31` or `1/2`, and ill-formed `quote` expressions such as `(quote 3 4)`. By "closed lambda expression" we exclude expressions, such as `(lambda (x) (cons x y))`, containing free variables in their bodies. Terms typed by the user are translated into strict terms for internal use in ACL2.

The predicate `termp` checks this strict sense of "term" with respect to a given ACL2 logical world; See Section 8.89 [WORLD], page 362. Many ACL2 functions, such as the rewriter, require certain of their arguments to satisfy `termp`. However, as of this writing, `termp` is in `:program^` mode and thus cannot be used effectively in conjectures to be proved. Furthermore, if regarded simply from the perspective of an effective guard^ for a term-processing function, `termp` checks many irrelevant things. (Does it really matter that the variable symbols encountered never start and end with an asterisk?) For these reasons, we have introduced the notion of a "pseudo-term" and embodied it in the predicate `pseudo-termp`, which is easier to check, does not require the logical world^ as input, has `:logic`^ mode, and is often perfectly suitable as a guard^ on term-processing functions.

A `pseudo-termp` is either a symbol, a true list of length 2 beginning with the word `quote`, the application of an `n`-ary pseudo-`lambda` expression to a true list of `n` pseudo-terms, or the application of a symbol to a true list of `n` `pseudo-termp`s. By an "`n`-ary pseudo-`lambda` expression" we mean an expression of the form `(lambda (v1 ... vn) pterm)`, where the `vi` are symbols (but not necessarily distinct legal variable symbols) and `pterm` is a `pseudo-termp`.

Metafunctions may use `pseudo-termp` as a guard^.

8.73 REDEF

REDEF a common way to set ld-redefinition-action^

```
Example and General Form:
ACL2 !>:redef
```

This command sets `ld-redefinition-action`^ to `'(:query . :overwrite)`.

As explained elsewhere (see Section 8.56 [LD-REDEFINITION-ACTION], page 314), this allows redefinition of functions and other events^ without undoing. A query will be made every time a redefinition is command^ed; the user must explicitly acknowledge that the redefinition is intentional. It is possible to set `ld-redefinition-action`^ so that the redefinition of non-system functions occurs quietly. See Section 8.56 [LD-REDEFINITION-ACTION], page 314.

8.74 REDEF!

REDEF! system hacker's redefinition command^

```
Example and General Form:
ACL2 !>:redef!
ACL2 p!>
```

This command^ sets `ld-redefinition-action`^ to `'(:warn! . :overwrite)` and sets the default defun-mode^ to `:program`^.

This is the ACL2 system hacker's redefinition command^. Note that even system functions can be redefined with a mere warning. Be careful!

8.75 REDEFINED-NAMES

REDEFINED-NAMES to collect the names that have been redefined

```
Example and General Forms:
(redefined-names state)
```

This function collects names that have been redefined in the current ACL2 state^. `:Program`^ mode functions that were reclassified to `:logic`^ functions are not collected, since such reclassification cannot imperil soundness because it is allowed only when the new and old definitions are identical.

Thus, if `(redefined-names state)` returns `nil` then no unsafe definitions have been made, regardless of `ld-redefinition-action`^. See Section 8.56 [LD-REDEFINITION-ACTION], page 314.

8.76 REDUNDANT-EVENTS

REDUNDANT-EVENTS allowing a name to be introduced "twice"

Sometimes an event will announce that it is "redundant". When this happens, no change to the logical world^ has occurred. This happens when the logical name being defined is already defined and has exactly the same definition, from the logical point of view. This feature permits two independent books^, each of which defines some name, to be included sequentially provided they use exactly the same definition.

When are two logical-name^ definitions considered exactly the same? It depends upon the kind of name being defined.

A `deflabel`^ event is never redundant. This means that if you have a `deflabel`^ in a book and that book has been included (without error), then references to that label denote the point in history^ at which the book introduced the label. See the note about shifting logical names, below.

A `defun^` or `mutual-recursion^` (or `defuns^`) event is redundant if for each function to be introduced, there has already been introduced a function with the same name, the same formals, and syntactically identical `guard^`, type declarations, and `body` (before macroexpansion).

A `verify-guards^` event is redundant if the function has already had its guard^s verified.

A `defaxiom^` or `defthm^` event is redundant if there is already an axiom or theorem of the given name and both the formula (after macroexpansion) and the rule-classes^ are syntactically identical. Note that a `defaxiom^` can make a subsequent `defthm^` redundant, and a `defthm^` can make a subsequent `defaxiom^` redundant as well.

A `defconst^` is redundant if the name has been defined to have the same value.

A `defmacro^` is redundant if there is already a macro defined with the same name and syntactically identical arguments, guard^, and body.

A `defpkg^` is redundant if a package of the same name with exactly the same imports has been defined.

A `deftheory^` is never redundant. The "natural" notion of equivalent `deftheory^`s is that the names and values of the two theory expressions are the same. But since most theory expressions are sensitive to the context in which they occur, it seems unlikely to us that two `deftheory^`s coming from two sequentially included books^ will ever have the same values. So we prohibit redundant theory definitions. If you try to define the same theory name twice, you will get a "name in use" error.

An `in-theory^` event is never redundant because it doesn't define any name.

`Table^` and `defdoc^` events^ are never redundant because they don't define any name.

An `encapsulate^` event is redundant if and only if a syntactically identical `encapsulate^` has already been executed under the same `default-defun-mode^`.

An `include-book^` is redundant if the book has already been included.

Note About Shifting Logical Names:

Suppose a book defines a function `fn` and later uses `fn` as a logical name in a theory expression. Consider the value of that theory expression in two different sessions. In session A, the book is included in a world^ in which `fn` is not already defined, i.e., in a world^ in which the book's definition of `fn` is not redundant. In session B, the book is included in a world^ in which `fn` is already identically defined. In session B, the book's definition of `fn` is redundant. When `fn` is used as a logical name in a theory expression, it denotes the point in history^ at which `fn` was introduced. Observe that those points are different in the two sessions. Hence, it is likely that theory expressions involving `fn` will have different values in session A than in session B.

This may adversely affect the user of your book. For example, suppose your book creates a theory via `deftheory^` that is advertised just to contain the names generated by the book. But suppose you compute the theory as the very last event in the book using:

```
(set-difference-theories (universal-theory :here)
                         (universal-theory fn))
```

where `fn` is the very first event in the book and happens to be a `defun^` event. This expression returns the advertised set if `fn` is not already defined when the book is included. But if `fn` were previously (identically) defined, the theory is larger than advertised.

The moral of this is simple: when building books^ that other people will use, it is best to describe your theories^ in terms of logical names that will not shift around when the books^ are included. The best such names are those created by `deflabel^`.

8.77 SAVING-AND-RESTORING

SAVING-AND-RESTORING saving and restoring your logical state

```
Examples:
ACL2 !>:Q
>(make-lib "file")
...
>(note-lib "file")
>(LP)
ACL2 !>
```

To save the current ACL2 logical world^ to a file, exit ACL2 with `:q^` and invoke `(make-lib "file")` in Common Lisp. This creates a file `"file.lib"` and a file `"file.lisp"`. The latter will be compiled. It generally takes half an hour to save an ACL2 logical world^ and creates a 20Mb file. All things considered it is probably better to just save your core image.

To restore such a saved ACL2 world^, invoke `(note-lib "file")` from Common Lisp, and then enter ACL2 with `(lp)`. We do not save the `io^` system, the stacks, or the global table, hence bindings of your globals will not be restored.

This save/restore mechanism is a temporary expedient. We know of faster mechanisms, mechanisms that consume less disk space, and mechanisms that provide more functionality. We don't know of good compromises between these various desirable features.

8.78 SIGNATURE

SIGNATURE how to specify the arity of a constrained function

```
Examples:
(hd (x) t)
(printer (x state) (mv t t state))
(printer (x state) (mv er-flg val state))

General Form:
(fn formals result)
```

where `fn` is the constrained function symbol, `formals` is a suitable list of formal parameters for it, and `result` is either a symbol denoting that the function returns one result (which is a state^ object or not depending on whether the symbol is `state^`) or else result is an `mv^` expression, `(mv s1 ... sn)`, where `n>1`, each `si` is a symbol, and at most one of the `si` is the symbol `state^`. The latter form of result indicates that the function returns `n` results and indicates which of them (if any) is a state^ object. The non-`state^` `si` are just place holders and may all be identical, e.g., `t`, though we often use symbols that suggest the type of the corresponding value. It is illegal for `state^` to be used in result if `state^` does not appear in `formals`.

8.79 SIMPLE

SIMPLE `:definition^` and `:rewrite^` rules used in preprocessing

```
Example of simple rewrite rule:
(equal (car (cons x y)) x)
```

```
Examples of simple definition:
(defun file-clock-p (x) (integerp x))
(defun naturalp (x)
  (and (integerp x) (>= x 0)))
```

The theorem prover output sometimes refers to "simple" definitions and rewrite rules. These rules can be used by the preprocessor, which is one of the theorem prover's "processes" understood by the `:do-not` hint; see Section 8.41 [HINTS], page 297.

The preprocessor expands certain definitions and uses certain rewrite rules that it considers to be "fast". There are two ways to qualify as fast. One is to be an "abbreviation", where a rewrite rule with no hypotheses or loop stopper is an "abbreviation" if the right side contains no more variable occurrences than the left side, and the right side does not call the functions `if^`, `not^` or `implies^`. Definitions and rewrite rules can both be abbreviations; the criterion for definitions is similar, except that the definition must not be recursive. The other way to qualify applies only to a non-recursive definition, and applies when its body is a disjunction or conjunction, according to a perhaps subtle criterion that is intended to avoid case splits.

8.80 SPECIOUS-SIMPLIFICATION

SPECIOUS-SIMPLIFICATION nonproductive proof steps

Occasionally the ACL2 theorem prover reports that the current goal simplifies to itself or to a set including itself. Such simplifications are said to be "specious" and are ignored in the sense that the theorem prover acts as though no simplification were possible and tries the next available proof technique. Specious simplifications are almost always caused by forcing.

The simplification of a formula proceeds primarily by the local application of `:rewrite^`, `:type-prescription^`, and other rules to its various subterms. If no rewrite rules apply, the formula cannot be simplified and is passed to the next ACL2 proof technique, which is generally the elimination of destructors. The experienced ACL2 user pays special attention to such "maximally simplified" formulas; the presence of unexpected terms in them indicates the need for additional rules or the presence of some conflict that prevents existing rules from working harmoniously together.

However, consider the following interesting possibility: local rewrite rules apply but, when applied, reproduce the goal as one of its own subgoals. How can rewrite rules apply and reproduce the goal? Of course, one way is for one rule application to undo the effect of another, as when commutativity is applied twice in succession to the same term. Another kind of example is when rules conflict and undermine each other. For example, under suitable hypotheses, `(length x)` might be rewritten to `(+ 1 (length (cdr x)))` by the `:definition^` of `length^` and then a `:rewrite^` rule might be used to "fold" that back to `(length x)`. Generally speaking the presence of such "looping" rewrite rules causes ACL2's simplifier either to stop gracefully because of heuristics such as that described in the documentation for `loop-stopper^` or to cause a stack overflow because of indefinite recursion.

A more insidious kind of loop can be imagined: two rewrites in different parts of the formula undo each other's effects "at a distance," that is, without ever being applied to one another's output. For example, perhaps the first hypothesis of the formula is simplified to the second, but then the second is simplified to the first, so that the end result is a formula propositionally equivalent to the original one but with the two hypotheses commuted. This is thought to be impossible unless forcing occurs, but if forcing is exploited (see Section 8.35 [FORCE], page 277) it can be made to happen relatively easily.

Here is a simple example. Declare `foo` to be a function of one argument returning one result:

```
(defstub foo (x) t)
```

Add the following `:type-prescription^` rule about `foo`:

```
(defaxiom forcer
 (implies (force (not (true-listp x)))
          (equal (foo x) t))
 :rule-classes :type-prescription)
```

Note that we could define a `foo` with this property; `defstub^` and `defaxiom^` are only used here to get to the gist of the problem immediately. Consider the proof attempt for the following formula.

```
(thm (implies (and (consp x)              ; hyp 1
                   (true-listp (cdr x))   ; hyp 2
                   (true-listp x))        ; hyp 3
              (foo x)))                   ; concl
```

When we simplify this goal, `hyp 1` cannot be simplified. `Hyp 2` simplifies to `t`, because `x` is known to be a non-`nil` true list so its `cdr^` is a true list by type reasoning; because true hypotheses are dropped, `hyp 2` simply disappears. `Hyp 3` simplifies to `(true-listp (cdr x))` by opening up the `:definition^` of `true-listp^`. Note that `hyp 3` has simplified to the old `hyp 2`. So at this point, the "current (intermediate) goal" is

```
(implies (and (consp x)                    ; rewritten hyp 1
              (true-listp (cdr x)))        ; rewritten hyp 3
         (foo x))                          ; unrewritten concl
```

and we are working on `(foo x)`. But the `:type-prescription^` rule above tells us that `(foo x)` is `t` if the hypothesis of the rule is true. Thus, in the case that the hypothesis of the rule is true, we are done. It remains to prove the current intermediate goal under the assumption that the hypothesis of the rule is false. This is done by adding the negation of the `:type-prescription^` rule's hypothesis to the current intermediate goal. This is what `force^` does in this situation. The negation of the hypothesis is `(true-listp x)`. Adding it to the current goal produces the subgoal

```
(implies (and (consp x)                    ; rewritten hyp 1
              (true-listp (cdr x))         ; rewritten hyp 3
              (true-listp x))              ; FORCEd hyp
         (foo x)).                         ; unrewritten concl
```

Observe that this is just our original goal. Despite all the rewriting, no progress was made! In more common cases, the original goal may simplify to a set of subgoals, one of which includes the original goal.

If ACL2 were to adopt the new set of subgoals, it would loop indefinitely. Therefore, it checks whether the input goal is a member of the output subgoals. If so, it announces that the simplification is "specious" and pretends that no simplification occurred.

"Maximally simplified" formulas that produce specious simplifications are maximally simplified in a very technical sense: were ACL2 to apply every applicable rule to them, no progress would be made. Since ACL2 can only apply every applicable rule, it cannot make further progress with the formula. But the informed user can perhaps identify some rule that should not be applied and make it inapplicable by disabling it, allowing the simplifier to apply all the others and thus make progress.

When specious simplifications are a problem it might be helpful to disable^ all forcing and resubmit the formula to observe whether forcing is involved in the loop or not. See Section 8.35 [FORCE], page 277. The commands

```
ACL2 !>:disable-forcing
and
ACL2 !>:enable-forcing
```

disable^ and enable^ forcing. If the loop is broken when forcing is disable^d, then it is very likely some force^d hypothesis of some rule is "undoing" a prior simplification. The most common cause of this is when we force^ a hypothesis that is actually false but whose falsity is somehow temporarily hidden (more below). To find the offending rule, compare the specious simplification with its non-specious counterpart and look for rules that were speciously applied that are not applied in the non-specious case. Most likely you will find at least one such rule and it will have a `force^d` hypothesis. By disabling that rule, at least for the subgoal in question, you may allow the simplifier to make progress on the subgoal.

To illustrate what we mean by the claim that specious simplifications often arise because the system force^s a false hypothesis, reconsider the example above. At the time we used the `:type-prescription^` rule, the known assumptions were `(consp x)` and `(true-listp (cdr x))`. Observe that this tells us that `x` is a true list. But the hypothesis force^d to be true was `(not (true-listp x))`. Why was the falsity of this hypothesis missed? The most immediate reason is that the encoding of the two assumptions above does not produce a context ("type-alist") in which `x` is recorded to be a true-list. When we look up `(not (true-listp x))` in that context, we are not told that it is false. More broadly, the problem stems from the fact that when we force^ a hypothesis we do not bring to bear on it all of the resources of the theorem prover. Thus it could be — as here — that the hypothesis could be proved false in the current context but is not obviously so. No matter how sophisticated we made the forcing mechanism, the unavoidable incompleteness of the theorem prover would still permit the occasional specious simplification. While that does not excuse us from trying to avoid specious simplifications when we can — and we may well strengthen the type mechanism to deal with the problem illustrated here — specious simplifications will probably remain a problem deserving of the user's attention.

8.81 STATE

STATE the von Neumannesque ACL2 state object

The ACL2 state object is used extensively in programming the ACL2 system, and has been used in other ACL2 programs as well. However, most users, especially those interested in specification and verification (as opposed to programming *per se*), need not be aware of the role of the state object in ACL2, and will not write functions that use it explicitly. We say more about this point at the end of this documentation topic.

The *global table* is perhaps the most visible portion of the state object. Using the interface functions `@` and `assign`, a user will bind global variables to the results of function evaluations (much as an

Nqthm user exploits the Nqthm utility `r-loop`). See Section 9.1 [@], page 373 and see Section 9.3 [ASSIGN], page 378.

ACL2 supports several facilities of a truly von Neumannesque state machine character, including file io^ and global variables. Logically speaking, the state is a true list of the 14 components as described below. There is a "current" state object at the top-level of the ACL2 command^ loop. This object is understood to be the value of what would otherwise be the free variable `state` appearing in top-level input. When any command^ returns a state object as one of its values, that object becomes the new current state. But ACL2 provides von Neumann style speed for state operations by maintaining only one physical (as opposed to logical) state object. Operations on the state are in fact destructive. This implementation does not violate the applicative semantics because we enforce certain draconian syntactic rules regarding the use of state objects. For example, one cannot "hold on" to an old state, access the components of a state arbitrarily, or "modify" a state object without passing it on to subsequent state-sensitive functions.

Every routine that uses the state facilities (e.g. does io^, or calls a routine that does io^), must be passed a "state object." And a routine must return a state object if the routine modifies the state in any way. Rigid syntactic rules governing the use of state objects are enforced by the function `translate`, through which all ACL2 user input first passes. State objects can only be "held" in the formal parameter `state`, never in any other formal parameter and never in any structure (excepting a multiple-values return list field which is always a state object). State objects can only be accessed with the primitives we specifically permit. Thus, for example, one cannot ask, in code to be executed, for the length of `state` or the `car^` of `state`. In the statement and proof of theorems, there are no syntactic rules prohibiting arbitrary treatment of state objects.

Logically speaking, a state object is a true list whose members are as follows:

> `Open-input-channels`, an alist with keys that are symbols in package `"ACL2-INPUT-CHANNEL"`. The value (`cdr^`) of each pair has the form `((:header type file-name open-time) . elements)`, where `type` is one of `:character`, `:byte`, or `:object` and `elements` is a list of things of the corresponding `type`, i.e. characters, integers of type `(mod 255)`, or lisp objects in our theory. `File-name` is a string. `Open-time` is an integer. See Section 10.135 [IO], page 456.

> `Open-output-channels`, an alist with keys that are symbols in package `"ACL2-OUTPUT-CHANNEL"`. The value of a pair has the form `((:header type file-name open-time) . current-contents)`. See Section 10.135 [IO], page 456.

> `Global-table`, an alist associating symbols (to be used as "global variables") with values. See Section 9.1 [@], page 373, and see Section 9.3 [ASSIGN], page 378.

T-stack, a list of arbitrary objects accessed and changed by the functions aref-t-stack and aset-t-stack.

32-bit-integer-stack, a list of arbitrary 32-bit-integers accessed and changed by the functions aref-32-bit-integer-stack and aset-32-bit-integer-stack.

Big-clock-entry, an integer, that is used logically to bound the amount of effort spent to evaluate a quoted form.

Idates, a list of dates and times, used to implement the function print-current-idate, which prints the date and time.

Run-times, a list of integers, used to implement the functions that let ACL2 report how much time was used, but inaccessible to the user.

File-clock, an integer that is increased on every file opening and closing and used to maintain the consistency of the io^ primitives.

Readable-files, an alist whose keys have the form (string type time), where string^ is a file name and time is an integer. The value associated with such a key is a list of characters, bytes, or objects, according to type. The time field is used in the following way: when it comes time to open a file for input, we will only look for a file of the specified name and type whose time field is that of file-clock. This permits us to have a "probe-file" aspect to open-file: one can ask for a file, find it does not exist, but come back later and find that it does now exist.

Written-files, an alist whose keys have the form (string type time1 time2), where string^ is a file name, type is one of :character, :byte or :object, and time1 and time2 are integers. Time1 and time2 correspond to the file-clock time at which the channel for the file was opened and closed. This field is write-only; the only operation that affects this field is close-output-channel, which cons^es a new entry on the front.

Read-files, a list of the form (string type time1 time2), where string^ is a file name and time1 and time2 were the times at which the file was opened for reading and closed. This field is write only.

Writeable-files, an alist whose keys have the form (string type time). To open a file for output, we require that the name, type, and time be on this list.

List-all-package-names-lst, a list of true-listps. Roughly speaking, the car^ of this list is the list of all package names known to this Common Lisp right now and the cdr^ of this list is the value of this state variable after you look at its car^. The function, list-all-package-names, which takes the state as an argument, returns the car^ and cdr^s the list (returning a new state too). This essentially gives ACL2 access to what is provided by CLTL's list-all-packages. Defpkg^ uses this feature to insure that the about-to-be-created package is new in this lisp. Thus, for example, in akcl it is impossible to create the package "COMPILER" with defpkg^ because it is on the list, while in Lucid that package name is not initially on the list.

We recommend avoiding the use of the state object when writing ACL2 code intended to be used as a formal model of some system, for several reasons. First, the state object is complicated and contains many components that are oriented toward implementation and are likely to be irrelevant to the model in question. Second, there is currently not much support for reasoning about ACL2 functions that manipulate the state object. Third, the documentation about state is not as complete as one might wish for serious programming that uses state.

If a user is building a model that includes a system state, it is better to represent that state explicitly in the model rather than use the ACL2 state object. ACL2 functions that manipulate association lists (for example, see Section 10.28 [ASSOC], page 409) can be used in place of @ and assign to access and update the state component of the model. As of this writing, the "books" directory of the ACL2 distribution contains a number of theorems already proved about such functions.

A consequence of this recommendation is that ACL2 constructs like pprogn and er-progn that depend on the state object will not appear in user-built models.

8.82 SUBVERSIVE-INDUCTIONS

SUBVERSIVE-INDUCTIONS why we restrict encapsulate^d recursive functions

It is illegal for one of the functions introduced in the signature^ of an encapsulate^ event to be involved in the induction^ scheme suggested by a non-local^ly defined recursive function in that encapsulate^. Such recursive functions can give rise to so-called "subversive" induction^ schemes in the sense that when used outside the encapsulate^ the scheme is unsound. Normally, the induction^ suggested by a function is justified by the termination proof for the recursion. But in the case of recursive functions defined within encapsulations, the termination proof is constructed in a context in which the functions are to be viewed as "constrained." The termination proof might therefore take advantage of properties of the witnesses that are not exported from the encapsulate^. Hence, crucial restrictions on the constrained functions may be lost. We give examples and advice below.

The following (illegal) event illustrates the problem posed by subversive induction^s.

```
(encapsulate ((test (x) t) (d (x) t) (p (x) t))

 (local (defun test (x) (consp x)))

 (local (defun d (x)
          (declare (xargs :mode :logic))
          (cdr x)))

 (defun foo (x)
   (declare (xargs :mode :logic))
   (if (test x) (foo (d x)) t))

 (local (defun p (x) (declare (ignore x)) t))

 (defthm base-case (implies (not (test x)) (p x)))

 (defthm induction-step (implies (and (test x) (p (d x))) (p x))))
```

Consider just the first three events^ above: the local^ definitions of `test` and `d` and the non-local recursive function `foo`. `Test` and `d` will be constrained to have certain properties, but within the `encapsulate^` they are locally witnessed by `consp^` and `cdr^`, respectively. Observe that `foo` recurses by stepping from `x` to `(d x)` when `(test x)` is true. This recursion terminates because of properties of the witnesses for `test` and `d`. These properties need not be exported from the encapsulation. But when `foo` is exported from the encapsulation, it suggests that it is permissible to recur/induct by stepping "down" to `(d x)` from `x` when `(test x)` is true. Since we do not constrain `(d x)` to be smaller than `x` when `(test x)` the induction^ suggestion by `foo` can be used to prove non-theorems. Because `foo` uses `test` and `d` in its recursion we say the induction^ suggested by `foo` is "subversive" outside the encapsulation and do not permit such a `foo` to be defined non-locally. That is, the attempt to submit the encapsulate^ event above will cause an error and complain about `foo`.

We give some advice below on how to carry out such encapsulations as might have been intended by that above. But first we show how an inconsistency can be deduced when the above `encapsulate^` is permitted.

Note that the `encapsulate^` introduced one other function symbol, `p`. Furthermore, the only constraint^s on `test`, `d`, and `p` are the last two theorems above. These theorems are the base case and induction^ step for a proof of `(p x)` based on the induction^ suggested by `foo`. We arrange for these theorems to be provable simply by defining `p` locally to be `t`.

Once outside the encapsulate^ we can prove `(p x)` by the spurious induction^ suggested by `foo`.

```
(defthm p-true (p x) :hints (("Goal" :induct (foo x))))
```

The proof is immediate given the two theorems exported from the `encapsulate^`.

But then we can functionally instantiate `p` in the theorem above to be `nil`! We must, of course, satisfy the constraint^s on `p`, namely the two theorems about `test`, `d`, and `p` exported from the `encapsulate^`. We can satisfy the constraint^s by choosing `test` to be always `t`.

```
(defthm subversion! nil
  :hints (("Goal" :use (:functional-instance p-true
                          (test (lambda (x) t))
                          (p (lambda (x) nil)))))
  :rule-classes nil)
```

This concludes our illustration of what can go wrong were we to permit recursive functions in encapsulations to use the constrained functions in their recursions. We now move on to our advice about how to achieve the (non-nefarious) ends intended by the disallowed encapsulations.

One often desires to define recursive functions such as `foo`, i.e., functions that use constrained functions in their recursion. However, one should not try to introduce such functions within the same encapsulation as the constrained functions. Instead, introduce the constrained functions, including among their constraint^s the measure theorems sufficient to justify the intended recursive uses, and define the recursive functions outside the encapsulation environment. Thus, the following pair of events^ achieves, perhaps, the intended effect of the original encapsulation:

```
(encapsulate ((test (x) t) (d (x) t))
  (local (defun test (x) (consp x)))
  (local (defun d (x) (declare (xargs :mode :logic)) (cdr x)))
  (defthm d-decreases
    (implies (test x) (< (acl2-count (d x)) (acl2-count x)))))

(defun foo (x) (if (test x) (foo (d x)) t))
```

Another alternative, depending on one's original intentions, is to include the definition of `foo` in the encapsulation but to make it local^. This allows `foo` to be used in other local^ events^ of that encapsulation but does not export it and its subversive induction^ scheme.

Finally, it may be that `foo` is needed outside the encapsulation environment but the user does not intend for `foo` to suggest any induction^ schemes. If this is the case, one should include `foo` in the signature^ of the encapsulate^ (and so make its definition local^) and prove a non-local `:definition^` rule which states the recurrence equation for `foo` without requiring a termination argument. The `:clique` and `:controller-alist` fields for the `:definition^` rule should be those for the recursive definition of `foo`. See Section 14.4 [DEFINITION], page 628.

Note that the functions introduced in the signature^ may not even occur ancestrally in the induction^ scheme suggested by a non-`local^`ly defined recursive function in the `encapsulate^`. That is, they may not occur in definitions of, or constraint^s on, functions that occur in such induction^ schemes; and **those** functions may not occur in such induction^ schemes; and so on.

8.83 SYNTAX

SYNTAX the syntax of ACL2 is that of Common Lisp

For the details of ACL2 syntax, see CLTL. For examples of ACL2 syntax, use `:pe^` to print some of the ACL2 system code. For example:

```
:pe assoc-equal
:pe dumb-occur
:pe fn-var-count
:pe add-linear-variable-to-alist
```

There is no comprehensive description of the ACL2 syntax yet, except that found in CLTL. Also see Section 8.85 [TERM], page 352.

8.84 SYNTAXP

SYNTAXP to attach a heuristic filter on a `:rewrite^` rule

```
Example:
Consider the :REWRITE rule created from

(IMPLIES (SYNTAXP (NOT (AND (CONSP X) (EQ (CAR X) 'NORM))))
         (EQUAL (LXD X) (LXD (NORM X)))).
```

The `syntaxp` hypothesis in this rule will allow the rule to be applied to `(lxd (trn a b))` but will not allow it to be applied to `(lxd (norm a))`.

```
General Form:
(SYNTAXP test)
```

may be used as the nth hypothesis in a `:rewrite^` rule whose `:corollary^` is `(implies (and hyp1 ... hypn ... hypk) (equiv lhs rhs))` provided `test` is a term, `test` contains at least one variable, and every variable occuring freely in `test` occurs freely in `lhs` or in some `hypi`, `i<n`. Formally, `syntaxp` is a function of one argument; `syntaxp` always returns `t` and so may be added as a vacuous hypothesis. However, the test "inside" the `syntaxp` form is actually treated as a meta-level proposition about the proposed instantiation of the rule's variables and that proposition must evaluate to true (non-`nil`) to "establish" the `syntaxp` hypothesis.

We illustrate this by slightly elaborating the example given above. Consider a `:rewrite^` rule whose `:corollary^` is:

```
(IMPLIES (AND (RATIONALP X)
              (SYNTAXP (NOT (AND (CONSP X) (EQ (CAR X) 'NORM)))))
         (EQUAL (LXD X) (LXD (NORM X))))
```

How is this rule applied to `(lxd (trn a b))`? First, we form a substitution that instantiates the left-hand side of the conclusion of the rule so that it is identical to the target term. In the present case, the substitution replaces `x` with `(trn a b)`. Then we backchain to establish the hypotheses, in order. Ordinarily this means that we instantiate each hypothesis with our substitution and then attempt to rewrite the resulting instance to true. Of course, most users are aware of some exceptions to this general rule. For example, if a hypothesis contains a "free-variable" — one not bound by the current substitution — we attempt to extend the substitution by searching for an instance of the hypothesis among known truths. `Force^`d hypotheses are another exception to the general rule of how hypotheses are relieved. Hypotheses marked with `syntaxp`, as in `(syntaxp test)`, are also exceptions. Instead of instantiating the hypothesis and trying to establish it, we evaluate `test` in an environment in which its variable symbols are bound to the quotations of the terms to which those variables are bound in the instantiating substitution. In the case in point, we evaluate the test

```
(NOT (AND (CONSP X) (EQ (CAR X) 'NORM)))
```

in an environment where `x` is bound to `'(trn a b)`, i.e., the list of length three whose `car^` is the symbol `'trn`. Thus, the test returns `t` because `x` is a `consp^` and its `car^` is not the symbol `'norm`. When the `syntaxp` test evaluates to `t`, we consider the `syntaxp` hypothesis to have been established; this is sound because `(syntaxp test)` is `t` regardless of `test`. If the test evaluates to `nil` (or fails to evaluate because of guard^ violations) we act as though we cannot establish the hypothesis and abandon the attempt to apply the rule; it is always sound to give up.

Note that the test of a `syntaxp` hypothesis does not deal with the meaning or semantics or values of the terms but with their syntactic forms. In the example above, the `syntaxp` hypothesis allows the rule to be applied to every target of the form `(lxd u)`, provided `(rationalp u)` can be established and `u` is not of the form `(norm v)`. Observe that without this syntactic restriction the rule above could loop producing a sequence of increasingly complex targets `(lxd a)`, `(lxd (norm a))`, `(lxd (norm (norm a)))`, etc. An intuitive reading of the rule might be "`norm` the argument of `lxd` (when it is `rationalp^`) unless it has already been `norm`ed."

Another common syntactic restriction is

```
(SYNTAXP (AND (CONSP X) (EQ (CAR X) 'QUOTE))).
```

A rule with such a hypothesis can be applied only if `x` is bound to a specific constant. Thus, if `x` is `23` (which is actually represented internally as `(quote 23)`), the test evaluates to `t`; but if `x` is `(+ 11 12)` it evaluates to `nil` (because `(car x)` is the symbol `'+^`). It is often desirable to delay the application of a rule until certain subterms are in some kind of normal form so that the application doesn't produce excessive case splits.

8.85 TERM

TERM the three senses of well-formed ACL2 expressions or formulas

```
Examples of Terms:
(cond ((caar x) (cons t x)) (t 0))   ; an untranslated term

(if (car (car x)) (cons 't x) '0)    ; a translated term

(car (cons x y) 'nil v)              ; a pseudo-term
```

In traditional first-order predicate calculus a "term" is a syntactic entity denoting some object in the universe of individuals. Often, for example, the syntactic characterization of a term is that it is either a variable symbol or the application of a function symbol to the appropriate number of argument terms. Traditionally, "atomic formulas" are built from terms with predicate symbols such as "equal" and "member;" "formulas" are then built from atomic formulas with propositional "operators" like "not," "and," and "implies." Theorems are formulas. Theorems are "valid" in the sense that the value of a theorem is true, in any model of the axioms and under all possible assignments of individuals to variables.

However, in ACL2, terms are used in place of both atomic formulas and formulas. ACL2 does not have predicate symbols or propositional operators as distinguished syntactic entities. The ACL2 universe of individuals includes a "true" object (denoted by `t`) and a "false" object (denoted by `nil`), predicates and propositional operators are functions that return these objects. Theorems in ACL2 are terms and the "validity" of a term means that, under no assignment to the variables does the term evaluate to `nil`.

We use the word "term" in ACL2 in three distinct senses. We will speak of "translated" terms, "untranslated" terms, and "pseudo-" terms.

Translated Terms: The Strict Sense and Internal Form

In its most strict sense, a "term" is either a legal variable symbol, a quoted constant, or the application of an n-ary function symbol or closed `lambda` expression to a true list of n terms.

The legal variable symbols are symbols other than `t` or `nil` which are not in the keyword package, do not start with ampersand, do not start and end with asterisks, and if in the main Lisp package, do not violate an appropriate restriction (see Section 8.65 [NAME], page 329).

Quoted constants are expressions of the form `(quote x)`, where `x` is any ACL2 object. Such expressions may also be written `'x`.

Closed `lambda` expressions are expressions of the form `(lambda (v1 ... vn) body)` where the `vi` are distinct legal variable symbols, `body` is a term, and the only free variables in `body` are among the `vi`.

The function `termp`, which takes two arguments, an alleged term `x` and a logical world `w` (see Section 8.89 [WORLD], page 362), recognizes terms of a given extension of the logic. `Termp` is defined in `:program^` mode. Its definition may be inspected with `:pe^ termp` for a complete specification of what we mean by "term" in the most strict sense. Most ACL2 term-processing functions deal with terms in this strict sense and use `termp` as a guard^. That is, the "internal form" of a term satisfies `termp`, the strict sense of the word "term."

Untranslated Terms: What the User Types

While terms in the strict sense are easy to explore (because their structure is so regular and simple) they can be cumbersome to type. Thus, ACL2 supports a more sugary syntax that includes uses of macros and constant symbols. Very roughly speaking, macros are functions that produce terms as their results. Constants are symbols that are associated with quoted objects. Terms in this sugary syntax are "translated" to terms in the strict sense; the sugary syntax is more often called "untranslated." Roughly speaking, translation just implements macroexpansion, the replacement of constant symbols by their quoted values, and the checking of all the rules governing the strict sense of "term."

More precisely, macro symbols are as described in the documentation for `defmacro^`. A macro, `mac`, can be thought of as a function, `mac-fn`, from ACL2 objects to an ACL2 object to be treated as an untranslated term. For example, `caar^` is defined as a macro symbol; the associated macro function maps the object `x` into the object `(car (car x))`. A macro form is a "call" of a macro symbol, i.e., a list whose `car^` is the macro symbol and whose `cdr^` is an arbitrary true list of objects, used as a term. Macroexpansion is the process of replacing in an untranslated term every occurrence of a macro form by the result of applying the macro function to the appropriate arguments. The "appropriate" arguments are determined by the exact form of the definition of the macro; macros support positional, keyword, optional and other kinds of arguments. See Section 6.10 [DEFMACRO], page 163.

In addition to macroexpansion and constant symbol dereferencing, translation implements the mapping of `let^` and `let*^` forms into applications of `lambda` expressions and closes `lambda` expressions containing free variables. Thus, the translation of

```
(let ((x (1+ i))) (cons x k))
```

can be seen as a two-step process that first produces

```
((lambda (x) (cons x k)) (1+ i))
```

and then

```
((lambda (x k) (cons x k)) (1+ i) k) .
```

Observe that the body of the `let^` and of the first `lambda` expression contains a free `k` which is finally bound and passed into the second `lambda` expression.

When we say, of an event-level function such as `defun^` or `defthm^`, that some argument "must be a term" we mean an untranslated term. The event functions translate their term-like arguments.

To better understand the mapping between untranslated terms and translated terms it is convenient to use the keyword command `:trans^` to see examples of translations. See Section 9.16 [TRANS], page 390 and also see Section 9.17 [TRANS1], page 390.

Pseudo-Terms: A Common Guard for Metafunctions

Because `termp` is defined in `:program^` mode, it cannot be used effectively in conjectures to be proved. Furthermore, from the perspective of merely guarding a term processing function, `termp` often checks more than is required. Finally, because `termp` requires the logical world^ as one of its arguments it is impossible to use `termp` as a guard^ in places where the logical world^ is not itself one of the arguments.

For these reasons we support the idea of "pseudo-terms." A pseudo-term is either a symbol (but not necessarily one having the syntax of a legal variable symbol), a true list beginning with `quote` (but not necessarily well-formed), or the "application of" a symbol or pseudo `lambda` expression to a true list of pseudo-terms. A pseudo `lambda` expression is an expression of the form `(lambda (v1 ... vn) body)` where the `vi` are all symbols and `body` is a pseudo-term.

Pseudo-terms are recognized by the unary function `pseudo-termp^`. If `(termp x w)` is true, then `(pseudo-termp x)` is true. However, if `x` fails to be a (strict) term it may nevertheless still be a pseudo-term. For example, `(car a b)` is not a term, because `car^` is applied to the wrong number of arguments, but it is a pseudo-term.

The structures recognized by `pseudo-termp^` can be recursively explored with the same simplicity that terms can be. In particular, if `x` is not a `variablep` or an `fquotep`, then `(ffn-symb x)` is the function (`symbol` or `lambda` expression) and `(fargs x)` is the list of argument pseudo-terms. A metafunction may use `pseudo-termp^` as the guard^.

8.86 TERM-ORDER

TERM-ORDER the ordering relation on terms used by ACL2

ACL2 must occasionally choose which of two terms is syntactically smaller. The need for such a choice arises, for example, when using equality hypotheses in conjectures (the smaller term is substituted for the larger elsewhere in the formula), in stopping loops in permutative rewrite rules (see Section 8.62 [LOOP-STOPPER], page 324), and in choosing the order in which to try to cancel the addends in linear arithmetic inequalities. When this notion of syntactic size is needed, ACL2 uses "term order." Popularly speaking, term order is just a lexicographic ordering on terms. But the situation is actually more complicated.

We define term order only with respect to terms in translated form. See Section 9.16 [TRANS], page 390.

`Term1` comes before `term2` in the term order iff

(a) the number of variable occurrences in `term1` is less than that in `term2`, or

(b) the numbers of variable occurrences in the two terms are equal but the number of function applications in `term1` is less than that in `term2`, or

(c) the numbers of variable occurrences in the two terms are equal, the numbers of functions applications in the two terms are equal, and `term1` comes before `term2` in a certain lexicographic ordering based their structure as Lisp objects.

The function `term-order`, when applied to the translations of two ACL2 terms, returns `t` iff the first is "less than or equal" to the second in the term order.

By "number of variable occurrences" we do not mean "number of distinct variables" but "number of times a variable symbol is mentioned." `(cons x x)` has two variable occurrences, not one. Thus, perhaps counterintuitively, a large term that contains only one variable occurrence, e.g., `(standard-char-p (car (reverse x)))` comes before `(cons x x)` in the term order.

Since constants contain no variable occurrences and non-constant expressions must contain at least one variable occurrence, constants come before non-constants in the term order, no matter how large the constants. For example, the list constant

```
'(monday tuesday wednesday thursday friday)
```

comes before `x` in the term order. Because term order is involved in the control of permutative rewrite rules and used to shift smaller terms to the left, a set of permutative rules designed to allow the permutation of any two tips in a tree representing the nested application of some function will always move the constants into the left-most tips. Thus,

```
(+ x 3 (car (reverse klst)) (dx i j)) ,
```

which in translated form is

```
(binary-+ x
          (binary-+ '3
                    (binary-+ (dx i j)
                              (car (reverse klst))))),
```

will be permuted under the built-in commutativity rules to

```
(binary-+ '3
```

```
(binary-+ x
          (binary-+ (car (reverse klst))
                    (dx i j))))
```

or

```
(+ 3 x (car (reverse klst)) (dx i j)).
```

Clearly, two constants are ordered using cases (b) and (c) of term order, since they each contain 0 variable occurrences. This raises the question "How many function applications are in a constant?" Because we regard the number of function applications as a more fundamental measure of the size of a constant than lexicographic considerations, we decided that for the purposes of term order, constants would be seen as being built by primitive constructor functions. These constructor functions are not actually defined in ACL2 but merely imagined for the purposes of term order. We here use suggestive names for these imagined functions, ignoring entirely the prior use of these names within ACL2.

The constant function `z` constructs `0`. Positive integers are constructed from `(z)` by the successor function, `s`. Thus 2 is `(s (s (z)))` and contains three function applications. `100` contains one hundred and one applications. Negative integers are constructed from their positive counterparts by `-^`. Thus, `-2` is `(- (s (s (z))))` and has four applications. Ratios are constructed by the dyadic function `/^`. Thus, `-1/2` is

```
(/ (- (s (z))) (s (s (z))))
```

and contains seven applications. Complex rationals are similarly constructed from rationals. All character objects are considered primitive and are constructed by constant functions of the same name. Thus `#\a` and `#\b` both contain one application. Strings are built from the empty string, `(o)` by the "string-cons" function written `cs`. Thus `"AB"` is `(cs (#\a) (cs (#\b) (o)))` and contains five applications. Symbols are obtained from strings by "packing" the `symbol-name^` with the unary function `p`. Thus `'ab` is

```
(p (cs (#\a) (cs (#\b) (o))))
```

and has six applications. Note that packages are here ignored and thus `'acl2::ab` and `'my-package::ab` each contain just six applications. Finally, cons^es are built with `cons^`, as usual. So `'(1 . 2)` is `(cons '1 '2)` and contains six applications, since `'1` contains two and `'2` contains three. This, for better or worse, answers the question "How many function applications are in a constant?"

Two terms with the same numbers of variable occurrences and function applications are ordered by lexicographic means, based on their structures. In the lexicographic ordering, two atom^s are ordered "alphabetically" as described below, an atom^ and a cons^ are ordered so that the atom^ comes first, and two cons^es are ordered so that the one with the recursively smaller `car^` comes first, with the `cdr^`s being compared only if the `car^`s are equal. Thus, if two terms `(member ...)` and `(reverse ...)` contain the same numbers of variable occurrences and function applications, then the `member^` term is first in the term order because `member^` comes before `reverse^` in the term order (which is here reduced to alphabetic ordering).

It remains only to define what we mean by the alphabetic ordering on Lisp atom^s. Within a single type, numbers are compared arithmetically, characters^ are compared via their (char) codes, and strings and symbols are compared with the conventional alphabetic ordering on sequences of characters^. Across types, numbers come before characters^, characters^ before strings, and strings before symbols.

8.87 TYPE-SET

TYPE-SET how type information is encoded in ACL2

To help you experiment with type-sets we briefly note the following utility functions.

`(type-set-quote x)` will return the type-set of the object `x`. For example, `(type-set-quote "test")` is 2048 and `(type-set-quote '(a b c))` is 512.

`(type-set 'term nil nil nil nil (ens state) (w state) nil)` will return the type-set of `term`. For example,

```
(type-set '(integerp x) nil nil nil nil (ens state) (w state) nil)
```

will return (mv 192 nil). 192, otherwise known as `*ts-boolean*`, is the type-set containing `t` and `nil`. The second result may be ignored in these experiments. `Term` must be in the `translated`, internal form shown by `:trans^`. See Section 9.16 [TRANS], page 390 and see Section 8.85 [TERM], page 352.

`(type-set-implied-by-term 'x nil 'term (ens state)(w state) nil)` will return the type-set deduced for the variable symbol `x` assuming the `translated` term, `term`, true. The second result may be ignored in these experiments. For example,

```
(type-set-implied-by-term 'v nil '(integerp v)
                          (ens state) (w state) nil)
```

returns `11`.

`(convert-type-set-to-term 'x ts (ens state) (w state) nil)` will return a term whose truth is equivalent to the assertion that the term `x` has type-set `ts`. The second result may be ignored in these experiments. For example

```
(convert-type-set-to-term 'v 523 (ens state) (w state) nil)
```

returns a term expressing the claim that `v` is either an integer or a non-`nil` true-list. `523` is the `logical-or` of `11` (which denotes the integers) with `512` (which denotes the non-`nil` true-lists).

The "actual primitive types" of ACL2 are listed in `*actual-primitive-types*`. These primitive types include such types as `*ts-zero*`, `*ts-positive-integer*`, `*ts-nil*` and `*ts-proper-consp*`. Each actual primitive type denotes a set — sometimes finite and sometimes not — of ACL2 objects and these sets are pairwise disjoint. For example, `*ts-zero*` denotes the set containing 0 while `*ts-positive-integer*` denotes the set containing all of the positive integers.

The actual primitive types were chosen by us to make theorem proving convenient. Thus, for example, the actual primitive type `*ts-nil*` contains just `nil` so that we can encode the hypothesis "`x` is `nil`" by saying "`x` has type `*ts-nil*`" and the hypothesis "`x` is non-`nil`" by saying "`x` has type complement of `*ts-nil*`." We similarly devote a primitive type to `t`, `*ts-t*`, and to a third type, `*ts-non-t-non-nil-symbol*`, to contain all the other ACL2 symbols.

Let `*ts-other*` denote the set of all Common Lisp objects other than those in the actual primitive types. Thus, `*ts-other*` includes such things as floating point numbers and CLTL array objects. The actual primitive types together with `*ts-other*` constitute what we call `*universe*`. Note that `*universe*` is a finite set containing one more object than there are actual primitive types; that is, here we are using `*universe*` to mean the finite set of primitive types, not the infinite set of all objects in all of those primitive types. `*Universe*` is a partitioning of the set of all Common Lisp objects: every object belongs to exactly one of the sets in `*universe*`.

Abstractly, a "type-set" is a subset of `*universe*`. To say that a term, `x`, "has type-set `ts`" means that under all possible assignments to the variables in `x`, the value of `x` is a member of some member of `ts`. Thus, `(cons x y)` has type-set `{*ts-proper-cons* *ts-improper-cons*}`. A term can have more than one type-set. For example, `(cons x y)` also has the type-set `{*ts-proper-cons* *ts-improper-cons* *ts-nil*}`. Extraneous types can be added to a type-set without invalidating the claim that a term "has" that type-set. Generally we are interested in the smallest type-set a term has, but because the entire theorem-proving problem for ACL2 can be encoded as a type-set question, namely, "Does `p` have type-set complement of `*ts-nil*`?," finding the smallest type-set for a term is an undecidable problem. When we speak informally of "the" type-set we generally mean "the type-set found by our heuristics" or "the type-set assumed in the current context."

Note that if a type-set, `ts`, does not contain `*ts-other*` as an element then it is just a subset of the actual primitive types. If it does contain `*ts-other*` it can be obtained by subtracting from `*universe*` the complement of `ts`. Thus, every type-set can be written as a (possibly complemented) subset of the actual primitive types.

By assigning a unique bit position to each actual primitive type we can encode every subset, `s`, of the actual primitive types by the nonnegative integer whose ith bit is on precisely if `s` contains the ith actual primitive type. The type-sets written as the complement of `s` are encoded as the `twos-complement` of the encoding of `s`. Those type-sets are thus negative integers. The bit positions assigned to the actual primitive types are enumerated from `0` in the same order as the types are listed in `*actual-primitive-types*`. At the concrete level, a type-set is an integer between `*min-type-set*` and `*max-type-set*`, inclusive.

For example, `*ts-nil*` has bit position `6`. The type-set containing just `*ts-nil*` is thus represented by `64`. If a term has type-set `64` then the term is always equal to `nil`. The type-set containing everything but `*ts-nil*` is the twos-complement of `64`, which is `-65`. If a term has type-set `-65`, it is never equal to `nil`. By "always" and "never" we mean under all, or under no, assignments to the variables, respectively.

Here is a more complicated example. Let `s` be the type-set containing all of the symbols and the natural numbers. The relevant actual primitive types, their bit positions and their encodings are:

```
actual primitive type       bit    value

*ts-zero*                    0        1
*ts-positive-integer*        1        2
*ts-nil*                     6       64
*ts-t*                       7      128
*ts-non-t-non-nil-symbol*    8      256
```

Thus, the type-set `s` is represented by `(+ 1 2 64 128 256)` = `451`. The complement of `s`, i.e., the set of all objects other than the natural numbers and the symbols, is `-452`.

8.88 WHY-BRR

WHY-BRR   an explanation of why ACL2 has an explicit `brr^` mode

Why isn't `brr^` mode automatically disabled when there are no monitored runes? The reason is that the list of monitored runes is kept in a wormhole state.

See Section 8.90 [WORMHOLE], page 364 for more information on wormholes in general. But the fundamental property of the wormhole function is that it is a logical `no-op`, a constant function that does not take state as an argument. When entering a wormhole, arbitrary information can be passed in (including the external state). That information is used to construct a near copy of the external state and that "wormhole state" is the one with respect to which interactions occur during breaks. But no information is carried by ACL2 out of a wormhole — if that were allowed wormholes would not be logical no-ops. The only information carried out of a wormhole is in the user's head.

`Break-rewrite^` interacts with the user in a wormhole state because the signature of the ACL2 rewrite function does not permit it to modify `state^`. Hence, only wormhole interaction is possible. (This has the additional desirable property that the correctness of the rewriter does not depend on what the user does during interactive breaks within it; indeed, it is logically impossible for the user to affect the course of `rewrite^`.)

Now consider the list of monitored runes. Is that kept in the external state as a normal state global or is it kept in the wormhole state? If it is in the external state then it can be inspected within the wormhole but not changed. This is unacceptable; it is common to change the monitor^ed rules as the proof attempt progresses, installing monitors when certain rules are about to be used in certain contexts. Thus, the list of monitored runes must be kept as a wormhole variable. Hence, its value cannot be determined outside the wormhole, where the proof attempt is ongoing.

This raises another question: If the list of monitored runes is unknown to the rewriter operating on the external state, how does the rewriter know when to break? The answer is simple: it breaks every time, for every rune, if `brr^` mode is enabled. The wormhole is entered (silently), computations are done within the wormhole state to determine if the user wants to see the break, and if so, interactions begin. For unmonitored runes and runes with false break conditions, the silent wormhole entry is followed by a silent wormhole exit and the user perceives no break.

Thus, the penalty for running with `brr^` mode enabled when there are no monitored runes is high: a wormhole is entered on every application of every rune and the user is simply unware of it. The user who has finally unmonitored all runes is therefore strongly advised to carry this information out of the wormhole and to do `:brr^ nil` in the external state when the next opportunity arises.

8.89 WORLD

WORLD ACL2 property lists and the ACL2 logical data base

A "world" is a list of triples, each of the form `(sym prop . val)`, implementing the ACL2 notion of property lists. ACL2 permits the simultaneous existence of many property list worlds. "The world" is often used as a shorthand for "the ACL2 logical world" which is the particular property list world used within the ACL2 system to maintain the data base of rules.

Common Lisp provides the notion of "property lists" by which one can attach "properties" and their corresponding "values" to symbols. For example, one can arrange for the `'color` property of the symbol `'box-14` to be `'purple` and the `'color` property of the symbol `'triangle-7` to be `'yellow`. Access to property lists is given via the Common Lisp function `get`. Thus, `(get 'box-14 'color)` might return `'purple`. Property lists can be changed via the special form `setf`. Thus, `(setf (get 'box-14 'color) 'blue)` changes the Common Lisp property list configuration so that `(get 'box-14 'color)` returns `'blue`. It should be obvious that ACL2 cannot provide this facility, because Common Lisp's `get` "function" is not a function of its argument, but instead a function of some implicit state object representing the property list settings for all symbols.

ACL2 provides the functions `getprop` and `putprop` which allow one to mimic the Common Lisp property list facility. However, ACL2's `getprop` takes as one of its arguments a list that is a direct encoding of what was above called the "state object representing the property list settings for all symbols." Because ACL2 already has a notion of "state^" that is quite distinct from that used here, we call this property list object a "world." A world is just a true list of triples. Each triple is of the form `(sym prop . val)`. This world can be thought of as a slightly elaborated form of association list and `getprop` is a slightly elaborated form of `assoc^` that takes two keys. When `getprop` is called on a symbol, `s`, property `p`, and world, `w`, it scans `w` for the first triple whose `sym` is `s` and `prop` is `p` and returns the corresponding `val`. `Getprop` has two additional arguments, one of which that controls what it returns if no such `sym` and `prop` exist in `w`, and other other of which allows an extremely efficient implementation. To set some property's value for some symbol, ACL2 provides `putprop`. `(putprop sym prop val w)` merely returns a new world, `w'`, in which `(sym prop . val)` has been `cons^`ed onto the front of `w`, thus "overwriting" the `prop` value of `sym` in `w` to `val` and leaving all other properties in `w` unchanged.

One aspect of ACL2's property list arrangment is that it is possible to have many different property list worlds. For example, `'box-14` can have `'color 'purple` in one world and can have `'color 'yes` in another, and these two worlds can exist simultaneously because `getprop` is explicitly provided the world from which the property value is to be extracted.

The efficiency alluded to above stems from the fact that Common Lisp provides property lists. Using Common Lisp's provisions behind the scenes, ACL2 can "install" the properties of a given world into the Common Lisp property list state so as to make retrieval via `getprop` very fast in the special case that the world provided to `getprop` has been installed. To permit multiple installed worlds, each of which is permitted to be changed via `putprop`, ACL2 requires that worlds be named and these names are used to distinquish installed versions of the various worlds. At the moment we do not further document `getprop` and `putprop`.

However, the ACL2 system uses a property list world, named `'current-acl2-world`, in which to store the succession of user command^s and their effects on the logic. This world is often referred to in our documentation^ as "the world" though it should be stressed that the user is permitted to have worlds and ACL2's is in no way distinguished except that the user is not permitted to modify it except via event command^s. The ACL2 world is part of the ACL2 state^ and may be obtained via `(w state)`.

Warning: The ACL2 world is very large. Its length as of this writing (Version 1.5) is over `31,000` and it grows with each release. Furthermore, some of the values stored in it are pointers to old versions of itself. Printing `(w state)` is something you should avoid because you likely will not have the patience to await its completion. For these practical reasons, the only thing you should do with `(w state)` is provide it to `getprop`, as in the form

```
(getprop sym prop default 'current-acl2-world (w state))
```

to inspect properties within it, or to pass it to ACL2 primitives, such as theory functions, where it is expected.

Some ACL2 command^ forms, such as theory expressions (see Chapter 15 [THEORIES], page 667) and the values to be stored in tables (see Section 6.35 [TABLE], page 193), are permitted to use the variable symbol `world` freely with the understanding that when these forms are evaluated that variable is bound to `(w state)`. Theoretically, this gives those forms complete knowledge of the current logical configuration of ACL2. However, at the moment, few world scanning functions have been documented for the ACL2 user. Instead, supposedly convenient macro forms have been created and documented. For example, `(current-theory :here)`, which is the theory expression which returns the currently enable^d theory, actually macroexpands to `(current-theory-fn`

:here world). When evaluated with world bound to (w state), current-theory-fn scans the current ACL2 world and computes the set of rune^s currently enable^d in it.

8.90 WORMHOLE

WORMHOLE ld^ without state^ — a short-cut to a parallel universe

```
Example Form:
(wormhole t 'interactive-break nil '(value 'hi!))
                             ; Enters a recursive read-eval-print loop
                             ; on a copy of the ``current state^'' and
                             ; returns nil!

General Form:
(wormhole pseudo-flg name input form
  :current-package    ...  ; known package name
  :ld-skip-proofsp    ...  ; t, nil or 'include-book
  :ld-redefinition-action  ; nil or '(:a . :b)
  :ld-prompt          ...  ; nil, t, or some prompt printer fn
  :ld-keyword-aliases ...  ; an alist pairing keywords to parse info
  :ld-pre-eval-filter ...  ; :all, :query, or some new name
  :ld-pre-eval-print  ...  ; nil, t, or :never
  :ld-post-eval-print ...  ; nil, t, or :command-conventions
  :ld-evisc-tuple     ...  ; nil or '(alist nil nil level length)
  :ld-error-triples   ...  ; nil or t
  :ld-error-action    ...  ; :continue, :return, or :error
  :ld-query-control-alist  ; alist supplying default responses
  :ld-verbose         ...) ; nil or t
```

The keyword arguments above are exactly those of ld^ (see Section 9.7 [LD], page 380) except that three of ld^'s keyword arguments are missing: the three that specify the channels standard-oi^, standard-co^ and proofs-co^. Essentially wormhole is just a call of ld^ on the current state^ with the given keyword arguments. Wormhole always returns nil. The **amazing** thing about wormhole is that it calls ld^ and interacts with the user even though state^ is not available as an argument!

Wormhole does this by manufacturing a "wormhole state^," a copy of the "current state^" (whatever that is) modified so as to contain some of the wormhole arguments. Ld^ is called on that wormhole state^ with the three ld^ channels directed to ACL2's "comment window." At the moment, the comment window is overlaid on the terminal and you cannot tell when output is going to *standard-co*^ and when it is going to the comment window. But we imagine that eventually a different window will pop up on your screen. In any case, the interaction provided by this call of `ld^` does not modify the `state^` "from which" wormhole was called, it modifies the copied `state^`. When `ld^` exits (e.g., in response to `:q^` being typed in the comment window) the wormhole state^ evaporates and `wormhole` returns `nil`. Logically and actually (from the perspective of the ongoing computation) nothing has happened except that a "no-op" function was called and returned `nil`.

The name `wormhole` is meant to suggest the idea that the function provides easy access to `state^` in situations where it is apparently impossible to get `state^`. Thus, for example, if you define the `factorial` function, say, except that you sprinkled into its body appropriate calls of `wormhole`, then the execution of `(factorial 6)` would cause interactive breaks in the comment window. During those breaks you would apparently be able to inspect the "current state^" even though `factorial` does not take `state^` as an argument. The whole notion of there being a "current state^" during the evaluation of `(factorial 6)` is logically ill-defined. And yet, we know from practical experience with the sequential computing machines upon which ACL2 is implemented that there is a "current state^" (to which the `factorial` function is entirely insensitive) and that is the state^ to which `wormhole` "tunnels." A call of `wormhole` from within `factorial` can pass `factorial`-specific information that is embedded in the wormhole state^ and made available for inspection by the user in an interactive setting. But no information ever flows out of a wormhole state^: `wormhole` always returns `nil`.

There are four arguments to `wormhole` that need further explanation: `pseudo-flg`, `name`, `input`, and `form`. Roughly speaking, the value of `pseudo-flg` should be `t` or `nil` and indicates whether we are actually to enter a wormhole or just return `nil` immediately. The actual handling of `pseudo-flg` is more sophisticated and is explained in detail at the end of this documentation^.

`Name` and `input` are used as follows. Recall that `wormhole` copies the "current state^" and then modifies it slightly to obtain the `state^` upon which `ld^` is called. We now describe the modifications. First, the `state^` global variable `'wormhole-name` is set to `name`, which may be any non-`nil` ACL2 object but is usually a symbol. Then, `'wormhole-input` is set to `input`, which may be any ACL2 object. Finally, and inexplicably, `'wormhole-output` is set to the value of `'wormhole-output` the last time a wormhole named `name` was exited (or `nil` if this is the first time a wormhole named `name` was entered). `this` last aspect of wormholes, namely the preservation of `'wormhole-output`, allows all the wormholes of a given name to communicate with each other.

We can now explain how `form` is used. The modified `state^` described above is the `state^` on which `ld^` is called. However, `standard-oi^` — the input channel from which `ld^` reads command^s — is set so that the first command^ that `ld^` reads and evaluates is `form`. If `form` returns an error triple with value `:q^`, i.e., `form` returns via `(value :q)`, then no further command^s are read, `ld^` exits, and the wormhole exits and returns `nil`. But if `form` returns any other value (or is not an error triple), then subsequent command^s are read from the comment window.

As usual, the `ld^`-specials affect whether a herald is printed upon entry, whether `form` is printed before evaluation, whether a prompt^ is printed, how errors are handled, etc. The `ld^`-specials can be specified with the corresponding arguments to `wormhole`. It is standard practice to call `wormhole` so that the entry to `ld^` and the evaluation of `form` are totally silent. Then, tests in `form` can inspect the `state^` and decide whether user interaction is desired. If so, `form` can appropriately set `ld-prompt^`, `ld-error-action^`, etc., print a herald, and then return `(value :invisible)`. Recall (see Section 9.7 [LD], page 380) that `(value :invisible)` causes `ld^` not to print a value for the just executed form. The result of this arrangement is that whether interaction occurs can be based on tests that are performed on the wormhole state^ after `(@ wormhole-input)` and the last `(@ wormhole-output)` are available for inspection. This is important because outside the wormhole you can access `wormhole-input` (you are passing it into the wormhole) but you may not be able to access the current `state^` (because you might be in `factorial`) and you definitely cannot access the `wormhole-output` of the last wormhole because it is not part of the ACL2 `state^`. Thus, if the condition under which you wish to interact depends upon the `state^` or that part of it preserved from the last wormhole interaction, that condition can only be tested from within the wormhole, via `form`.

It is via this mechanism that `break-rewrite^` (see Chapter 4 [BREAK-REWRITE], page 113) is implemented. To be more precise, the list of monitor^ed rune^s is maintained as part of the preserved `wormhole-output` of the `break-rewrite^` wormhole. Because it is not part of the normal `state^`, it may be changed by the user during proofs. That is what allows you to install new monitor^s while debugging proofs. But that means that the list of monitor^ed rune^s cannot be inspected from outside the wormhole. Therefore, to decide whether a break is to occur when a given rule is applied, the rewriter must enter the `break-rewrite^` wormhole, supplying a form that causes interaction if the given rule's break condition is satisfied. The user perceives this as though the wormhole was conditionally entered — a perception that is happily at odds with the informed user's knowledge that the list of monitor^ed rune^s is not part of the `state^`. In fact, the wormhole was unconditionally entered and the condition was checked from within the wormhole, that being the only state^ in which the condition is known.

Another illustrative example is available in the implemention of the `monitor^` command. How can we add a new rune^ to the list of monitor^ed rune^s while in the normal ACL2 `state^` (i.e., while not in a wormhole)? The answer is: by getting into a wormhole. In particular, when you type `(monitor rune expr)` at the top-level of ACL2, `monitor^` enters the `break-rewrite^` wormhole with a cleverly designed first `form`. That form adds rune^ and `expr` to the list of monitor^ed rune^s — said list only being available in `break-rewrite^` wormhole state^s. Then the first form returns `(value :q)`, which causes us to exit the wormhole. By using `ld^`-specials that completely suppress all output during the process, it does not appear to the user that a wormhole was entered. The moral here is rather subtle: the first form supplied to `wormhole` may be the entire computation you want to perform in the wormhole; it need not just be a predicate that decides if interaction is to occur. Using wormholes of different names you can maintain a variety of "hidden" data structures that are always accessible (whether passed in or not). This appears to violate completely the applicative semantics of ACL2, but it does not: because these data structures are only accessible via `wormhole`s, it is impossible for them to affect any ACL2 computation (except in the comment window).

As one might imagine, there is some overhead associated with entering a wormhole because of the need to copy the current `state^`. This brings us back to `pseudo-flg`. Ostensibly, `wormhole` is a function and hence all of its argument expressions are evaluated outside the function (and hence, outside the wormhole it creates) and then their values are passed into the function where an appropriate wormhole is created. In fact, `wormhole` is a macro that permits the `pseudo-flg` expression to peer dimly into the wormhole that will be created before it is created. In particular, `pseudo-flg` allows the user to access the `wormhole-output` that will be used to create the wormhole state^.

This is done by allowing the user to mention the (apparently unbound) variable `wormhole-output` in the first argument to `wormhole`. Logically, `wormhole` is a macro that wraps

```
(let ((wormhole-output nil)) ...)
```

around the expression supplied as its first argument. So logically, `wormhole-output` is always `nil` when the expression is evaluated. However, actually, `wormhole-output` is bound to the value of `(@ wormhole-output)` on the last exit from a wormhole of the given name (or `nil` if this is the first entrance). Thus, the `pseudo-flg` expression, while having to handle the possibility that `wormhole-output` is `nil`, will sometimes see non-`nil` values. The next question is, of course, "But how can you get away with saying that logically `wormhole-output` is always `nil` but actually it is not? That doesn't appear to be sound." But it is sound because whether `pseudo-flg` evaluates to `nil` or non-`nil` doesn't matter, since in either case `wormhole` returns `nil`. To make that point slightly more formal, imagine that `wormhole` did not take `pseudo-flg` as an argument. Then it could be implemented by writing

```
(if pseudo-flg (wormhole name input form ...) nil).
```

Now since wormhole always returns `nil`, this expression is equivalent to `(if pseudo-flg nil nil)` and we see that the value of `pseudo-flg` is irrelevant. So we could in fact allow the user to access arbitrary information to decide which branch of this if to take. We allow access to `wormhole-output` because it is often all that is needed. We don't allow access to `state^` (unless `state^` is available at the level of the wormhole call) for technical reasons having to do with the difficulty of overcoming `translate`'s prohibition of the sudden appearance of the variable `state^`.

We conclude with an example of the use of `pseudo-flg`. This example is a simplification of our implementation of `break-rewrite^`. To enter `break-rewrite^` at the beginning of the attempted application of a rule, `rule`, we use

```
(wormhole
 (and (f-get-global 'brr-mode state)
      (member-equal (access rewrite-rule rule :rune)
                    (cdr (assoc-eq 'monitored-runes wormhole-output))))
 'break-rewrite
 ...)
```

The function in which this call of `wormhole` occurs has `state^` as a formal. The `pseudo-flg` expression can therefore refer to `state^` to determine whether `'brr-mode` is set. But the `pseudo-flg` expression above mentions the variable `wormhole-output`; this variable is not bound in the context of the call of `wormhole`; if `wormhole` were a simple function symbol, this expression would be illegal because it mentions a free variable.

However, it is useful to think of `wormhole` as a simple function that evaluates all of its arguments but to also imagine that somehow `wormhole-output` is magically bound around the first argument so that `wormhole-output` is the output of the last `break-rewrite^` wormhole. If we so imagine, then the `pseudo-flg` expression above evaluates either to `nil` or non-`nil` and we will enter the wormhole named `break-rewrite^` in the latter case.

Now what does the `pseudo-flg` expression above actually test? We know the format of our own `wormhole-output` because we are responsible for maintaining it. In particular, we know that the list of monitor^ed rune^s can be obtained via

```
(cdr (assoc-eq 'monitored-runes wormhole-output)).
```

Using that knowledge we can design a `pseudo-flg` expression which tests whether (a) we are in `brr-mode` and (b) the rune^ of the current rule is a member of the monitor^ed rune^s. Question (a) is answered by looking into the current `state^`. Question (b) is answered by looking into that part of the about-to-be-created wormhole state^ that will differ from the current `state^`. To reiterate the reason we can make `wormhole-output` available here even though it is not in the current `state^`: logically speaking the value of `wormhole-output` is irrelevant because it is only used to choose between two identical alternatives. This example also makes it clear that `pseudo-flg` provides no additional functionality. The test made in the `pseudo-flg` expression could be moved into the first form evaluated by the wormhole — changing the free variable `wormhole-output` to `(@ wormhole-output)` and arranging for the first form to return `(value :q)` when the `pseudo-flg` expression returns `nil`. The only reason we provide the `pseudo-flg` feature is because it allows the test to be carried out without the overhead of entering the wormhole.

8.91 XARGS

XARGS giving hints^ to `defun^`

Common Lisp's `defun^` function does not easily allow one to pass extra arguments such as "hints^". ACL2 therefore supports a peculiar new declaration (see Section 10.94 [DECLARE], page 432) designed explicitly for passing additional arguments to `defun^` via a keyword-like syntax.

The following declaration is nonsensical but does illustrate all of the `xargs^` keywords:

```
(declare (xargs :guard (symbolp x)
                :measure (- i j)
                :well-founded-relation my-wfr
                :hints (("Goal" :in-theory (theory batch1)))
                :guard-hints (("Goal" :in-theory (theory batch1)))
                :mode :logic
                :verify-guards t
                :otf-flg t))

General Form:
(xargs :key1 val1 ... :keyn valn)
```

where the keywords and their respective values are as shown below. Note that once "inside" the xargs form, the "extra arguments" to `defun^` are passed exactly as though they were keyword arguments.

`:GUARD^`

`Value` is a term involving only the formals of the function being defined. The actual guard^ used for the definition is the conjunction of all the guard^s and types (see Section 10.94 [DECLARE], page 432) declare^d.

`:GUARD-HINTS`

`Value`: hints (see Section 8.41 [HINTS], page 297), to be used during the guard^ verification proofs as opposed to the termination proofs of the `defun^`.

`:MEASURE`

`Value` is a term involving only the formals of the function being defined. This term is indicates what is getting smaller in the recursion. The well-founded relation with which successive measures are compared is `e0-ord-<^`.

`:WELL-FOUNDED-RELATION^`

`Value` is a function symbol that is known to be a well-founded relation in the sense that a rule of class `:well-founded-relation^` has been proved about it. See Section 14.17 [WELL-FOUNDED-RELATION], page 662.

`:HINTS^`

Value: hints (see Section 8.41 [HINTS], page 297), to be used during the termination proofs as opposed to the guard^ verification proofs of the `defun^`.

`:MODE`

`Value` is `:program^` or `:logic^`, indicating the `defun^` mode of the function introduced. See Section 8.18 [DEFUN-MODE], page 256. If unspecified, the `defun^` mode defaults to the default `defun^` mode of the current world^. To convert a function from `:program^` mode to `:logic^` mode, see Section 6.39 [VERIFY-TERMINATION], page 201.

`:VERIFY-GUARDS^`

`Value` is `t` or `nil`, indicating whether or not guard^s are to be verified upon completion of the termination proof. This flag should only be `t` if the `:mode` is unspecified but the default `defun^` mode is `:logic^`, or else the `:mode` is `:logic^`.

`:OTF-FLG^`

Value is a flag indicating "onward through the fog" (see Section 8.67 [OTF-FLG], page 330).

9 OTHER

OTHER other commonly used top-level functions

* atsign:: (`@`) get the value of a global variable in `state^`
* ACL2-DEFAULTS-TABLE:: a table^ specifying certain defaults, e.g., the default defun-mode^
* ASSIGN:: assign to a global variable in `state^`
* CERTIFY-BOOK!:: a variant of `certify-book^`
* COMP:: compile some ACL2 functions
* GOOD-BYE:: quit entirely out of Lisp
* LD:: the ACL2 read-eval-print loop, file loader, and command^ processor
* PROPS:: print the ACL2 properties on a symbol
* Q:: quit ACL2 (type `:q`) — reenter with `(lp)`
* REBUILD:: a convenient way to reconstruct your old state^
* RESET-LD-SPECIALS:: restores initial settings of the `ld^` specials
* SET-GUARD-CHECKING:: control checking guard^s during execution of top-level forms
* SET-INHIBIT-OUTPUT-LST:: control output
* SKIP-PROOFS:: skip proofs for an event — a quick way to introduce unsoundness
* THM:: prove a theorem
* TRANS:: print the macroexpansion of a form
* TRANS1:: print the one-step macroexpansion of a form

Related topics other than immediate subtopics:
* CERTIFY-BOOK:: how to produce a certificate^ for a book

This section contains an assortment of functions that fit into none of the other categories and yet are suffiently useful as to merit "`advertisement`" in the `:help^` command.

9.1 `@`

`@` get the value of a global variable in `state^`

```
Examples:
(+ (@ y) 1)
(assign a (aset1 'ascii-map-array (@ a) 66 'Upper-case-B))

General Form:
(@ symbol)
```

where `symbol` is any symbol to which you have `assign^`ed a global value. This macro expands into `(f-get-global 'symbol state)`, which retrieves the stored value of the symbol.

The macro `assign^` makes it convenient to set the value of a symbol. The `:ubt^` operation has no effect on the `global-table` of `state^`. Thus, you may use these globals to hang onto useful data structures even though you may undo back past where you computed and saved them.

9.2 ACL2-DEFAULTS-TABLE

ACL2-DEFAULTS-TABLE a table^ specifying certain defaults, e.g., the default defun-mode^

```
Example Forms:
(table acl2-defaults-table :defun-mode) ; current default defun-mode
(table acl2-defaults-table :defun-mode :program)
           ; set default defun-mode to :program
```

See Section 6.35 [TABLE], page 193 for a discussion of tables in general. The legal keys for this table^ are shown below. They may be accessed and changed via the general mechanisms provided by table^s. However, there are often more convenient ways to access and/or change the defaults. (See also the note below.)

`:defun-mode`

the default defun-mode^, which must be `:program^` or `:logic^`. See Section 8.18 [DEFUN-MODE], page 256 for a general discussion of defun-mode^s. The `:defun-mode^` key may be conveniently set by keyword commands naming the new defun-mode^, `:program^` and `:logic^`. See Section 6.25 [PROGRAM], page 186 and see Section 6.22 [LOGIC], page 182.

`:verify-guards-eagerness`

an integer between 0 and 2 indicating how eager the system is to verify the guard^s of a defun^ event. See Section 6.33 [SET-VERIFY-GUARDS-EAGERNESS], page 192.

`:compile-fns`

When this key's value is `t`, functions are compiled when they are `defun^`'d; otherwise, the value is `nil`. To set the flag, see Section 6.27 [SET-COMPILE-FNS], page 187.

`:measure-function`

the default measure function used by `defun^` when no `:measure` is supplied in `xargs^`. The default measure function must be a function symbol of one argument. Let `mfn` be the default measure function and suppose no `:measure` is supplied with some recursive function definition. Then `defun^` finds the first formal, `var`, that is tested along every branch and changed in each recursive call. The system then "guesses" that `(mfn var)` is the `:measure` for that `defun^`.

```
:well-founded-relation
```

the default well-founded relation used by `defun^` when no `:well-founded-relation^` is supplied in `xargs^`. The default well-founded relation must be a function symbol, `rel`, of two arguments about which a `:well-founded-relation^` rule has been proved. See Section 14.17 [WELL-FOUNDED-RELATION], page 662.

```
:invisible-fns-alist
```

an alist used in the control of permutative rewriting. Each entry in this alist must be of the form `(fn ufn1 ufn2 ... ufnk)`, where `fn` and each `ufni` is a function symbol and the `ufni` are all unary functions. Such an entry makes each of the `ufni` "invisible" when ordering the arguments of `fn` via commutative rules. See Section 6.30 [SET-INVISIBLE-FNS-ALIST], page 189.

```
:irrelevant-formals-ok
```

When this key's value is `t`, the check for irrelevant formals is bypassed; otherwise, the value is the keyword `nil` (the default). See Section 10.136 [IRRELEVANT-FORMALS], page 458 and see Section 6.31 [SET-IRRELEVANT-FORMALS-OK], page 190.

```
:ignore-ok
```

When this key's value is `t`, the check for ignored variables is bypassed; otherwise, the value is the keyword `nil` (the default). See Section 6.28 [SET-IGNORE-OK], page 188.

```
:inhibit-warnings
```

ACL2 prints warnings that may, from time to time, seem excessive to experienced users. Each warning is "labeled" with a string identifying the type of warning. Consider for example

```
ACL2 Warning [Use] in ( THM ...):  It is unusual to :USE ....
```

Here, the label is "Use". The value of the key `:inhibit-warnings` is a list of such labels, where case is ignored. Any warning whose label is a member of this list (where again, case is ignored) is suppressed. See Section 6.29 [SET-INHIBIT-WARNINGS], page 189 and also see Section 9.13 [SET-INHIBIT-OUTPUT-LST], page 387.

```
:bdd-constructors
```

This key's value is a list of function symbols used to define the notion of "BDD normal form." See Section 2.1 [BDD-ALGORITHM], page 69 and see Section 8.41 [HINTS], page 297.

Note: Unlike all other table^s, `acl2-defaults-table` can affect the soundness of the system. The table^ mechanism therefore enforces on it a restriction not imposed on other table^s: when `table^` is used to update the `acl2-defaults-table`, the key and value must be variable-free forms. Thus, while

```
(table acl2-defaults-table :defun-mode :program),

(table acl2-defaults-table :defun-mode ':program), and

(table acl2-defaults-table :defun-mode (compute-mode *my-data*))
```

are all examples of legal events^ (assuming `compute-mode` is a function of one non-`state^` argument that produces a defun-mode^ as its single value),

```
(table acl2-defaults-table :defun-mode (compute-mode (w state)))
```

is not legal because the value form is `state^`-sensitive.

Consider for example the following three events^ which one might make into the text of a book.

```
(in-package "ACL2")

(table acl2-defaults-table
  :defun-mode
  (if (ld-skip-proofsp state) :logic :program))

(defun crash-and-burn (x) (car x))
```

The second event is illegal because its value form is `state^`-sensitive. If it were not illegal, then it would set the `:defun-mode^` to `:program^` when the book was being certified but would set the defun-mode^ to `:logic^` when the book was being loaded by `include-book^`. That is because during certification, `ld-skip-proofsp^` is `nil` (proof obligations are generated and proved), but during book inclusion `ld-skip-proofsp^` is non-`nil` (those obligations are assumed to have been satisfied.) Thus, the above book, when loaded, would create a function in `:logic^` mode that does not actually meet the conditions for such status.

For similar reasons, `table^` events^ affecting `acl2-defaults-table` are illegal within the scope of `local^` forms. That is, the text

```
(in-package "ACL2")

(local (table acl2-defaults-table :defun-mode :program))

(defun crash-and-burn (x) (car x))
```

is illegal because `acl2-defaults-table` is changed locally. If this text were acceptable as a book, then when the book was certified, `crash-and-burn` would be processed in `:program^` mode, but when the certified book was included later, `crash-and-burn` would have `:logic^` mode because the `local^` event would be skipped.

The text

```
(in-package "ACL2")

(program) ;which is (table acl2-defaults-table :defun-mode :program)

(defun crash-and-burn (x) (car x))
```

is acceptable and defines `crash-and-burn` in `:program^` mode, both during certification and subsequent inclusion.

We conclude with an important observation about the relation between `acl2-defaults-table` and `include-book^`, `certify-book^`, and `encapsulate^`. Including or certifying a book never has an effect on the `acl2-defaults-table`, nor does executing an `encapsulate^` event; we always restore the value of this table^ as a final act. (Also see Section 6.20 [INCLUDE-BOOK], page 180, see Section 6.18 [ENCAPSULATE], page 176, and see Section 3.6 [CERTIFY-BOOK], page 101.) That is, no matter how a book fiddles with the `acl2-defaults-table`, its value immediately after including that book is the same as immediately before including that book. If you want to set the `acl2-defaults-table` in a way that persists, you need to do so using command^s that are not inside books^. It may be useful to set your favorite defaults in your `acl2-customization^` file; see Section 8.4 [ACL2-CUSTOMIZATION], page 225.

9.3 ASSIGN

ASSIGN assign to a global variable in `state^`

```
Examples:
(assign x (expt 2 10))
(assign a (aset1 'ascii-map-array (@ a) 66 'Upper-case-B))

General Form:
(assign symbol term)
```

where `symbol` is any symbol (with certain enforced exclusions to avoid overwriting ACL2 system "globals") and `term` is any ACL2 term that could be evaluated at the top-level. `Assign` evaluates the term, stores the result as the value of the given symbol in the `global-table` of `state^`, and returns the result. (Note: the actual implementation of the storage of this value is much more efficient than this discussion of the logic might suggest.) `Assign` is a macro that effectively expands to the more complicated but understandable:

```
(pprogn (f-put-global 'symbol term state)
        (mv nil (f-get-global 'symbol term state) state)).
```

The macro `@^` gives convenient access to the value of such globals. The `:ubt^` operation has no effect on the `global-table` of `state^`. Thus, you may use these globals to hang onto useful data structures even though you may undo back past where you computed and saved them.

9.4 CERTIFY-BOOK!

CERTIFY-BOOK! a variant of `certify-book^`

```
Examples:
(certify-book! "my-arith" 3)     ;Certify in a world with 3
                                   ; commands, starting in a world
                                   ; with at least 3 commands.
(certify-book! "my-arith")       ;Certify in the initial world.
(certify-book! "my-arith" 0 nil) ;As above, but do not compile.

General Form:
(certify-book! book-name k compile-flg)
```

where book-name is a book name (see Section 3.3 [BOOK-NAME], page 95), k is a nonnegative integer used to indicate the "certification world^," and compile-flg indicates whether you wish to compile the (functions in the) book.

This command^ is identical to certify-book^, except that the second argument k may not be t in certify-book! and if k exceeds the current command^ number, then an appropriate ubt!^ will be executed first. See Section 3.6 [CERTIFY-BOOK], page 101 and see Section 7.17 [UBT!], page 217.

9.5 COMP

COMP compile some ACL2 functions

```
Examples:
:comp t          ; compile all uncompiled ACL2 functions
:comp foo        ; compile the defined function foo
:comp (foo bar)  ; compile the defined functions foo and bar

General Form:
:comp specifier
where specifier is one of the following:

  t                     compile all defined ACL2 functions that are currently
                          uncompiled
  (name-1 ... name-k)   a non-empty list of names of functions defined by
                          DEFUN in ACL2
  name                  same as (name)
```

Note: Unless a single function is specified (either as a symbol or as a one-element list containing its name), ACL2 will write out files "TMP.lisp" and "TMP1.lisp" (the latter is for the executable counterparts; see Section 8.30 [EXECUTABLE-COUNTERPART], page 271) that are subsequently compiled. Also see Section 6.27 [SET-COMPILE-FNS], page 187 for a way to get each function to be compiled as you go along.

9.6 GOOD-BYE

GOOD-BYE quit entirely out of Lisp

```
Example:
```

```
ACL2 !>:good-bye
```

Note: Your entire session will disappear forever when you type `:good-bye`.

The command `:good-bye` quits not only out of the ACL2 command^ loop, but in fact quits entirely out of the underlying Lisp. Thus, there is no going back! You will **not** be able to re-enter the command^ loop after typing `:good-bye`! All your work will be lost!!!

This command may not work in some underlying Common Lisp implementations. But we don't expect there to be any harm in trying. It **does** work in AKCL and GCL, at least as of this writing.

If you merely want to exit the ACL2 command^ loop, use `:q^` instead (see Section 9.9 [Q], page 384).

9.7 LD

LD   the ACL2 read-eval-print loop, file loader, and command^ processor

```
Example:
(LD standard-oi               ; open obj in channel, stringp file name
                              ; to open and close, or list of forms

                              ; Optional keyword arguments:
    :standard-co        ...   ; open char out or file to open and close
    :proofs-co          ...   ; open char out or file to open and close
    :current-package    ...   ; known package name
    :ld-skip-proofsp    ...   ; nil, 'include-book, t, or
                              ;   see Section 8.57 [LD-SKIP-PROOFSP], page 317
    :ld-redefinition-action   ; nil or '(:a . :b)
    :ld-prompt          ...   ; nil, t, or some prompt printer fn
    :ld-keyword-aliases ...   ; an alist pairing keywords to parse info
    :ld-pre-eval-filter ...   ; :all, :query, or some new name
    :ld-pre-eval-print  ...   ; nil, t, or :never
    :ld-post-eval-print ...   ; nil, t, or :command-conventions
    :ld-evisc-tuple     ...   ; nil or '(alist nil nil level length)
    :ld-error-triples   ...   ; nil or t
    :ld-error-action    ...   ; :return (default), :continue or :error
    :ld-query-control-alist   ; alist supplying default responses
    :ld-verbose         ...)  ; nil or t
```

`Ld` is the top-level ACL2 read-eval-print loop. (When you call `lp^`, a little initialization is done in raw Common Lisp and then `ld` is called.) `ld` is also a general-purpose ACL2 file loader and a command^ interpreter. `Ld` is actually a macro that expands to a function call involving `state^`. `Ld` returns an "error/value/state" triple as explained below.

The arguments to `ld` all happen to be global variables in `state^`. For example, `'current-package^` and `'ld-verbose^` are global variables, which may be accessed via `(@ current-package)` and `(@ ld-verbose)`. When `ld` is called, it "binds" these variables. By "binds" we actually mean the variables are globally set but restored to their old values on exit. Because `ld` provides the illusion of state^ global variables being bound, they are called "`ld` specials" (after the Lisp convention of calling a variable "special" if it is referenced freely after having been bound).

Note that all arguments but the first are passed via keyword. Any variable not explicitly given a value in a call retains its pre-call value, with the exception of `:ld-error-action^`, which defaults to `:return` if not explicitly specified.

Just as an example to drive the point home: If `current-package^` is `"ACL2"` and you typed

```
(ld *standard-oi* :current-package "MY-PKG")
```

you would find yourself in (an inner) read-eval-print loop in which the current-package^ was `"MY-PKG"`. You could operate there as long as you wished, changing the current package at will. But when you typed `:q^` you would return to the outer read-eval-print loop where the current package would still be `"ACL2"`.

Roughly speaking, `ld` repeatedly reads a form from `standard-oi^`, evaluates it, and prints its result to `standard-co^`. It does this until the form evaluates to an error triple whose value component is `:q^` or until the input channel or list is emptied. However, `ld` has many bells and whistles controlled by the `ld` specials. Each such special is documented individually. For example, see the documentation for `standard-oi^`, `current-package^`, `ld-pre-eval-print^`, etc.

A more precise description of `ld` is as follows. In the description below we use the `ld` specials as variables, e.g., we say "a form is read from `standard-oi^`." By this usage we refer to the current value of the named state^ global variable, e.g., we mean "a form is read from the current value of `'standard-oi^`." This technicality has an important implication: If while interacting with `ld` you change the value of one of the `ld` specials, e.g., `'standard-oi^`, you will change the behavior of `ld`, e.g., subsequent input will be taken from the new value.

Three `ld` specials are treated as channels: `standard-oi^` is treated as an object input channel and is the source of forms evaluated by `ld`; `standard-co^` and `proofs-co^` are treated as character output channels and various flavors of output are printed to them. However, the supplied values of these specials need not actually be channels; several special cases are recognized. If the supplied value of one of these is in fact an open channel of the appropriate type, that channel is used and is not closed by `ld`. If the supplied value of one of these specials is a string, the string is treated as a file name and a channel of the appropriate type is opened to/from that file. (Note that `ld` does **not** support structured pathname^s, as described in the documentation for `pathname^`.) Any channel opened by `ld` during the binding of the `ld` specials is automatically closed by `ld` upon termination. If `standard-co^` and `proofs-co^` are equal strings, only one channel to that file is opened and is used for both.

Several other alternatives are allowed for `standard-oi^`. If `standard-oi^` is a true list then it is taken as the list of forms to be processed. If `standard-oi^` is a list ending in an open channel, then `ld` processes the forms in the list and then reads and processes the forms from the channel. Analogously, if `standard-oi^` is a list ending a string, an object channel from the named file is opened and `ld` processes the forms in the list followed by the forms in the file. That channel is closed upon termination of `ld`.

The remaining `ld` specials are handled more simply and generally have to be bound to one of a finite number of tokens described in the `:doc^` entries for each `ld` special. Should any `ld` special be supplied an inappropriate value, an error message is printed.

Next, if `ld-verbose^` is `t`, `ld` prints the message "ACL2 loading name" where `name` names the file or channel from which forms are being read. At the conclusion of `ld`, it will print "Finished loading name" if `ld-verbose^` is `t`.

Finally, `ld` repeatedly executes the ACL2 read-eval-print step, which may be described as follows. A prompt^ is printed to `standard-co^` if `ld-prompt^` is non-`nil`. The format of the prompt^ is determined by `ld-prompt^`. If it is `t`, the default ACL2 prompt^ is used. If it is any other non-`nil` value then it is treated as an ACL2 function that will print the desired prompt^. See Section 8.54 [LD-PROMPT], page 312. In the exceptional case where `ld`'s input is coming from the terminal `(*standard-oi*)` but its output is going to a different sink (i.e., `standard-co^` is not `*standard-co*^`), we also print the prompt^ to the terminal.

`Ld` then reads a form from `standard-oi^`. If the object read is a keyword, `ld` constructs a "keyword command form" by possibly reading several more objects. See Section 8.46 [KEYWORD-COMMANDS], page 306. This construction process is sensitive to the value of `ld-keyword-aliases^`. See Section 8.50 [LD-KEYWORD-ALIASES], page 309. Otherwise, the object read is treated as the command form.

Ld next decides whether to evaluate or skip this form, depending on ld-pre-eval-filter^. Initially, the filter must be either :all, :query, or a new name. If it is :all, it means all forms are evaluated. If it is :query, it means each form that is read is displayed and the user is queried. Otherwise, the filter is a name and each form that is read is evaluated as long as the name remains new, but if the name is ever introduced then no more forms are read and ld terminates. See Section 8.52 [LD-PRE-EVAL-FILTER], page 311.

If the form is to be evaluated, first prints the form to standard-co^, if ld-pre-eval-print^ is t. With this feature, ld can process an input file or form list and construct a script of the session that appears as though each form was typed in. See Section 8.53 [LD-PRE-EVAL-PRINT], page 311.

Ld then evaluates the form, with state^ bound to the current state^. The result is some list of multiple values. If a state^ is among the values, then ld uses that state^ as the subsequent current state^.

Depending on ld-error-triples^, ld may interpret the result as an "error." See Section 8.48 [LD-ERROR-TRIPLES], page 307. We first discuss ld's behavior if no error signal is detected (either because none was sent or because ld is ignoring them as per ld-error-triples^).

In the case of a non-erroneous result, ld does two things: First, if the logical world^ in the now current state^ is different than the world^ before execution of the form, ld adds to the world^ a "command^ landmark" containing the form evaluated. See Section 8.11 [COMMAND-DESCRIPTOR], page 245. Second, ld prints the result to standard-co^, according to ld-post-eval-print^. If ld-post-eval-print^ is nil, no result is printed. If it is t, all of the results are printed as a list of multiple values. Otherwise, it is :command-conventions and only the non-erroneous "value" component of the result is printed. See Section 8.51 [LD-POST-EVAL-PRINT], page 310.

Whenever ld prints anything (whether the input form, a query, or some results) it "eviscerates" it if ld-evisc-tuple^ is non-nil. Essentially, evisceration is a generalization of Common Lisp's use of *print-level* and *print-length* to hide large substructures. See Section 8.49 [LD-EVISC-TUPLE], page 308.

We now return to the case of a form whose evaluation signals an error. In this case, ld first restores the ACL2 logical world^ to what it was just before the erroneous form was evaluated. Thus, a form that partially changes the world^ (i.e., begins to store properties) and then signals an error, has no effect on the world^. You may see this happen on command^s that execute several events^ (e.g., an encapsulate^ or a progn^ of several defuns^): even though the output makes it appear that the initial events^ were executed, if an error is signalled by a later event the entire block of events^ is discarded.

After rolling back, `ld` takes an action determined by `ld-error-action^`. If the action is `:continue`, `ld` merely iterates the read-eval-print step. Note that nothing suggestive of the value of the "erroneous" form is printed. If the action is `:return`, `ld` terminates normally. If the action is `:error`, `ld` terminates signalling an error to its caller. If its caller is in fact another instance of `ld` and that instance is watching out for error signals, the entire world^ created by the erroneous inner `ld` will be discarded by the outer `ld`.

`Ld` returns an error triple, `(mv erp val state)`. `Erp` is `t` or `nil` indicating whether an error is being signalled. If no error is signalled, `val` is the "reason" `ld` terminated and is one of `:exit` (meaning `:q^` was read), `:eof` (meaning the input source was exhausted), `:error` (meaning an error occurred but has been supressed) or `:filter` (meaning the `ld-pre-eval-filter^` terminated `ld`).

9.8 PROPS

PROPS print the ACL2 properties on a symbol

```
Example:
:props assoc-eq
```

`Props` takes one argument, a symbol, and prints all of the properties that are on that symbol in the ACL2 world^.

9.9 Q

Q quit ACL2 (type `:q`) — reenter with `(lp)`

```
Example:
ACL2 !>:Q
```

The keyword command `:q` typed at the top-level of the ACL2 loop will terminate the loop and return control to the Common Lisp top-level (or, more precisely, to whatever program invoked `lp^`). To reenter the ACL2 loop, execute `(acl2::lp)` in Common Lisp. You will be in the same state as you were when you exited with `:q`, unless during your stay in Common Lisp you messed the data structures representating the ACL2 state^.

Unlike all other keyword commands, typing `:q` is not equivalent to invoking the function `q`. There is no function `q`.

9.10 REBUILD

REBUILD a convenient way to reconstruct your old state^

```
Examples:
ACL2 !>(rebuild "project.lisp")
ACL2 !>(rebuild "project.lisp" t)
ACL2 !>(rebuild "project.lisp" :all)
ACL2 !>(rebuild "project.lisp" :query)
ACL2 !>(rebuild "project.lisp" 'lemma-22)
```

`Rebuild` allows you to assume all the command^s in a given file or list, supplied in the first argument. Because `rebuild` processes an arbitrary sequence of command^s with `ld-skip-proofsp^ t`, it is unsound! However, if each of these command^s is in fact admissible, then processing them with `rebuild` will construct the same logical state^ that you would be in if you typed each command^ and waited through the proofs again. Thus, `rebuild` is a way to reconstruct a state^ previously obtained by proving your way through the command^s.

The second, optional argument to `rebuild` is a "filter" (see Section 8.52 [LD-PRE-EVAL-FILTER], page 311) that lets you specify which command^s to process. You may specify `t`, `:all`, `:query`, or a new logical name. `t` and `:all` both mean that you expect the entire file or list to be processed. `:query` means that you will be asked about each command^ in turn. A new name means that all command^s will be processed as long as the name is new, i.e., `rebuild` will stop processing command^s immediately after executing a command^ that introduces name. `Rebuild` will also stop if any command^ causes an error. You may therefore wish to plant an erroneous form in the file, e.g., `(mv t nil state)`, (see Section 8.48 [LD-ERROR-TRIPLES], page 307), to cause `rebuild` to stop there. The form `(i-am-here)` is such a pre-defined form. If you do not specify a filter, `rebuild` will query you for one.

Inspection of the definition of `rebuild`, e.g., via `:pc^ rebuild-fn`, will reveal that it is just a glorified call to the function `ld^`. See Section 9.7 [LD], page 380 if you find yourself wishing that `rebuild` had additional functionality.

9.11 RESET-LD-SPECIALS

RESET-LD-SPECIALS restores initial settings of the `ld^` specials

```
Examples:
(reset-ld-specials t)
```

```
(reset-ld-specials nil)
```

Roughly speaking, the `ld^` specials are certain state^ global variables, such as `current-package^`, `ld-prompt^`, and `ld-pre-eval-filter^`, which are managed by `ld^` as though they were local variables. These variables determine the channels on which `ld^` reads and prints and control many options of `ld^`. See Section 9.7 [LD], page 380 for the details on what the `ld^` specials are.

This function, `reset-ld-specials`, takes one Boolean argument, `flg`. The function resets all of the `ld^` specials to their initial, top-level values, except for the three channel variables, `standard-oi^`, `standard-co^`, and `proofs-co^`, which are reset to their initial values only if `flg` is non-`nil`. Of course, if you are in a recursive call of `ld^`, then when you exit that call, the `ld^` specials will be restored to the values they had at the time `ld^` was called recursively. To see what the initial values are, inspect the value of the constant `*initial-ld-special-bindings*`.

9.12 SET-GUARD-CHECKING

SET-GUARD-CHECKING control checking guard^s during execution of top-level forms

The top-level ACL2 loop has a variable which controls which sense of execution is provided. To turn "guard^ checking on," by which we mean that guard^s are checked at runtime, execute the top-level form `:set-guard-checking t`. To turn it off, do `:set-guard-checking nil`. The status of this variable is reflected in the prompt^.

```
ACL2 !>
```

means guard^ checking is on and

```
ACL2 >
```

means guard^ checking is off. The exclamation mark can be thought of as "barring" certain computations. The absence of the mark suggests the absence of error messages or unbarred access to the logical axioms. Thus, for example

```
ACL2 !>(car 'abc)
```

will signal an error, while

```
ACL2 >(car 'abc)
```

will return `nil`.

Whether guard^s are checked during evaluation is independent of the `default-defun-mode^`. We note this simply because it is easy to confuse "`:program^` mode" with "evaluation in Common Lisp" and thus with "guard^ checking on;" and it is easy to confuse "`:logic^` mode" with "evaluation in the logic" and with "guard^ checking off." But the `default-defun-mode^` determines whether newly submitted definitions introduce programs or add logical axioms. That mode is independent of whether evaluation checks guard^s or not. You can operate in `:logic^` mode with runtime guard^ checking on or off. Analogously, you can operate in `:program^` mode with runtime guard^ checking on or off.

However, one caveat is appropriate: functions introduced only as programs have no logical definitions and hence when they are evaluated their Common Lisp definitions must be used and thus their guard^s (if any) checked. That is, if you are defining functions in `:program^` mode and evaluating expressions containing only such functions, guard^ checking may as well be on because guard^s are checked regardless. This same caveat applies to a few ACL2 system functions that take `state^` as an argument. This list of functions includes all the keys of the alist `*super-defun-wart-alist*` and all function whose name^s are members of the list `*initial-untouchables*`.

See Section 8.39 [GUARD], page 285 for a general discussion of guard^s.

9.13 SET-INHIBIT-OUTPUT-LST

SET-INHIBIT-OUTPUT-LST control output

```
Examples:
(set-inhibit-output-lst '(warning))
(set-inhibit-output-lst '(proof-tree prove))
:set-inhibit-output-lst (proof-tree prove)

General Form:
(set-inhibit-output-lst lst)
```

where `lst` is a form (which may mention `state^`) that evaluates to a list of names, each of which is the name of one of the following "kinds" of output produced by ACL2.

```
error          error messages
warning        warnings
observation    observations
prove          commentary produced by the theorem prover
```

```
event          non-proof commentary produced by events such as defun
               and encapsulate
summary        the summary at the successful conclusion of an event
proof-tree     proof-tree output
```

It is possible to inhibit each kind of output by putting the corresponding name into `lst`. For example, if `'warning` is included in (the value of) `lst`, then no warnings are printed. Note that proof-tree^ output is affected by `set-inhibit-output-lst`; see Chapter 12 [PROOF-TREE], page 579.

9.14 SKIP-PROOFS

SKIP-PROOFS skip proofs for an event — a quick way to introduce unsoundness

```
Example Form:
(skip-proofs
  (defun foo (x)
    (if (atom x) nil (cons (car x) (foo (reverse (cdr x)))))))

General Form:
(skip-proofs event)
```

where `event` is an event; see Chapter 6 [EVENTS], page 153. `Event` is processed as usual except that the proof obligations usually generated are merely assumed.

WARNING: Skip-proofs allows inconsistent events^ to be admitted to the logic. Use it at your own risk!

Sometimes in the development of a formal model or proof it is convenient to skip the proofs required by a given event. By embedding the event in a `skip-proofs` form, you can avoid the proof burdens generated by the event, at the risk of introducing unsoundness. Below we list three illustrative situations in which you might find `skip-proofs` useful.

1. The termination argument for a proposed function definition is complicated. You presume you could admit it, but are not sure that your definition has the desired properties. By embedding the `defun^` event in a `skip-proofs` you can "admit" the function and experiment with theorems about it before you pay the price of its admission.

2. You intend eventually to verify the guard^s for a definition but do not want to take the time now to pursue that. By embedding the `verify-guards^` event in a `skip-proofs` you can get the system to behave as though the guard^s were verified.

3. You are repeatedly recertifying a book while making many experimental changes. A certain `defthm^` in the book takes a very long time to prove and you believe the proof is not affected by the changes you are making. By embedding the `defthm^` event in a `skip-proofs` you allow the theorem to be assumed without proof during the experimental recertifications.

Unsoundness or Lisp errors may result if the presumptions underlying a use of `skip-proofs` are incorrect. Therefore, `skip-proofs` must be considered a dangerous (though useful) tool in system development.

Roughly speaking, a `defthm^` embedded in a `skip-proofs` is essentially a `defaxiom^`, except that it is not noted as an axiom for the purposes of functional instantiation (see Section 8.59 [LEMMA-INSTANCE], page 319). But a skipped `defun^` is much more subtle since not only is the definitional equation being assumed but so are formulas relating to termination and type. The situation is also difficult to characterize if the `skip-proofs` events^ are within the scope of an `encapsulate^` in which constrained functions are being introduced. In such contexts no clear logical story is maintained; in particular, constraints aren't properly tracked for definitions. A proof script involving `skip-proofs` should be regarded as work-in-progress, not as a completed proof with some unproved assumptions. A `skip-proofs` event represents a promise by the author to admit the given event without further axioms.

ACL2 allows the certification of books^ containing `skip-proofs` events^. This is contrary to the spirit of certified books^, since one is supposedly assured by a valid certificate^ that a book has been "blessed." But certification, too, takes the view of `skip-proofs` as "work-in-progress" and so allows the author of the book to promise to finish. When such books^ are certified, a warning to the author is printed, reminding him or her of the incurred obligation. When books^ containing `skip-proofs` are included into a session, a warning to the user is printed, reminding the user that the book is in fact incomplete and possibly inconsistent.

9.15 THM

THM prove a theorem

```
Example:
(thm (equal (app (app a b) c)
```

```
            (app a (app b c))))
```

Also see Section 6.15 [DEFTHM], page 168. Unlike `defthm^`, `thm` does not create an event; it merely causes the theorem prover to attempt a proof.

```
General Form:
(thm term
     :hints        hints
     :otf-flg      otf-flg
     :doc          doc-string)
```

where `term` is a term alleged to be a theorem, and `hints^`, `otf-flg^` and `doc-string^` are as described in the corresponding `:doc^` entries. The three keyword arguments above are all optional.

9.16 TRANS

TRANS print the macroexpansion of a form

```
Examples:
:trans (list a b c)
:trans (caddr x)
:trans (cond (p q) (r))
```

This function takes one argument, an alleged term, and translates it, expanding the macros in it completely. Either an error is caused or the formal meaning of the term is printed. We also print the "multiplicity" and "state-out" for the translated term. The multiplicity indicates how many values are returned (in the `mv^` sense) and is always 1 or greater. `State-out` indicates whether and where a state^ is returned. If `state-out` is `nil`, the form does not return a state^ (either as its value or among its multiple values). If `t`, the form returns one value and it is a state^. Otherwise, `state-out` is a number which indicates which of its multiple values is a state^.

For more, see Section 8.85 [TERM], page 352.

9.17 TRANS1

TRANS1 print the one-step macroexpansion of a form

```
Examples:
```

```
:trans1 (list a b c)
:trans1 (caddr x)
:trans1 (cond (p q) (r))
```

This function takes one argument, an alleged term, and expands the top-level macro in it for one step only. Either an error is caused, which happens when the form is not a call of a macro, or the result printed. Also see Section 9.16 [TRANS], page 390, which translates the given form completely.

10 PROGRAMMING

PROGRAMMING built-in ACL2 functions

The built-in ACL2 functions that one typically uses in writing programs are listed below. See their individual documentation^s. We do not bother to document the some of the more obscure functions provided by ACL2 that do not correspond to functions of Common Lisp.

* *:: multiplication macro
* *STANDARD-CI*:: an ACL2 character-based analogue of CLTL's `*standard-input*`
* *STANDARD-CO*:: the ACL2 analogue of CLTL's `*standard-output*`
* *STANDARD-OI*:: an ACL2 object-based analogue of CLTL's `*standard-input*`
* +:: addition macro
* -:: macro for subtraction and negation
* /:: macro for division and reciprocal
* /=:: test inequality of two numbers
* 1+:: increment by 1
* 1-:: decrement by 1
* <:: less-than
* <=:: less-than-or-equal test
* =:: test equality (of two numbers, symbols, or characters^)
* >:: greater-than test
* >=:: greater-than-or-equal test
* ABS:: the absolute value of a rational number
* ACL2-CHARACTERP:: recognizer for characters^
* ACL2-COMPLEX-RATIONALP:: recognizer for complex rationals
* ACL2-NUMBERP:: recognizer for numbers
* ACL2-USER:: a package the ACL2 user may prefer
* ACONS:: constructor for association lists
* ADD-TO-SET-EQ:: add a symbol to a list
* ALISTP:: recognizer for association lists
* ALPHA-CHAR-P:: recognizer for alphabetic characters
* AND:: conjunction
* APPEND:: concatenate^ two or more lists
* ASH:: arithmetic shift operation
* ASSOC:: look up key in association list, using `eql^` as test
* ASSOC-EQ:: look up key in association list, using `eq^` as test
* ASSOC-EQUAL:: look up key in association list
* ASSOC-KEYWORD:: look up key in a `keyword-value-listp^`
* ASSOC-STRING-EQUAL:: look up key, a string, in association list
* ATOM:: recognizer for atoms
* ATOM-LISTP:: recognizer for a true list of atom^s
* BINARY-*:: multiplication function
* BINARY-+:: addition function
* BINARY-APPEND:: concatenate^ two lists
* BOOLEANP:: recognizer for booleans

* BUTLAST:: all but a final segment of a list
* CAAAAR:: car^ of the caaar^
* CAAADR:: car^ of the caadr^
* CAAAR:: car^ of the caar^
* CAADAR:: car^ of the cadar^
* CAADDR:: car^ of the caddr^
* CAADR:: car^ of the cadr^
* CAAR:: car^ of the car^
* CADAAR:: car^ of the cdaar^
* CADADR:: car^ of the cdadr^
* CADAR:: car^ of the cdar^
* CADDAR:: car^ of the cddar^
* CADDDR:: car^ of the cdddr^
* CADDR:: car^ of the cddr^
* CADR:: car^ of the cdr^
* CAR:: returns the first element of a non-empty list, else nil
* CASE:: conditional based on if-then-else using eql^
* CASE-MATCH:: pattern matching or destructuring
* CDAAAR:: cdr^ of the caaar^
* CDAADR:: cdr^ of the caadr^
* CDAAR:: cdr^ of the caar^
* CDADAR:: cdr^ of the cadar^
* CDADDR:: cdr^ of the caddr^
* CDADR:: cdr^ of the cadr^
* CDAR:: cdr^ of the car^
* CDDAAR:: cdr^ of the cdaar^
* CDDADR:: cdr^ of the cdadr^
* CDDAR:: cdr^ of the cdar^
* CDDDAR:: cdr^ of the cddar^
* CDDDDR:: cdr^ of the cdddr^
* CDDDR:: cdr^ of the cddr^
* CDDR:: cdr^ of the cdr^
* CDR:: returns the second element of a cons^ pair, else nil
* CEILING:: division returning an integer by truncating toward positive infinity
* CHAR:: the nth^ element (zero-based) of a string
* CHAR-CODE:: the numeric code for a given character
* CHAR-DOWNCASE:: turn upper-case characters^ into lower-case characters^
* CHAR-EQUAL:: character equality without regard to case
* CHAR-UPCASE:: turn lower-case characters^ into upper-case characters^
* CHAR<:: less-than test for characters^
* CHAR<=:: less-than-or-equal test for characters^
* CHAR>:: greater-than test for characters^
* CHAR>=:: greater-than-or-equal test for characters^
* CHARACTER-LISTP:: recognizer for a true list of characters
* CHARACTERS:: characters in ACL2
* CODE-CHAR:: the character corresponding to a given numeric code
* COERCE:: coerce a character list to a string and a string to a list
* COMPILATION:: compiling ACL2 functions

* COMPLEX:: create an ACL2 number
* COMPLEX-RATIONALP:: recognizes complex rational numbers
* CONCATENATE:: concatenate lists or strings together
* COND:: conditional based on if-then-else
* CONJUGATE:: complex number conjugate
* CONS:: pair and list constructor
* CONSP:: recognizer for cons^ pairs
* DECLARE:: declarations
* DENOMINATOR:: divisor of a ratio in lowest terms
* DIGIT-CHAR-P:: the number, if any, corresponding to a given character
* DIGIT-TO-CHAR:: map a digit to a character
* EIGHTH:: eighth member of the list
* ENDP:: recognizer for empty lists
* EQ:: equality of symbols
* EQL:: test equality (of two numbers, symbols, or characters^)
* EQLABLE-ALISTP:: recognizer for a true list of pairs whose `car^`s are suitable for `eql^`
* EQLABLE-LISTP:: recognizer for a true list of objects each suitable for `eql^`
* EQLABLEP:: the guard^ for the function `eql^`
* EQUAL:: true equality
* ER-PROGN:: perform a sequence of state-changing “error triples”
* EVENP:: test whether an integer is even
* EXPLODE-NONNEGATIVE-INTEGER:: the list of characters^ in the decimal form of a number
* EXPT:: exponential function
* FIFTH:: fifth member of the list
* FIRST:: first member of the list
* FIX:: coerce to a number
* FIX-TRUE-LIST:: coerce to a true list
* FLOOR:: division returning an integer by truncating toward negative infinity
* FMS:: `:(str alist co-channel state evisc) => state`
* FMT:: formatted printing
* FMT1:: `:(str alist col co-channel state evisc) => (mv col state)`
* FOURTH:: fourth member of the list
* IDENTITY:: the identity function
* IF:: if-then-else function
* IFF:: logical “if and only if”
* IFIX:: coerce to an integer
* ILLEGAL:: cause a hard error
* IMAGPART:: imaginary part of a complex number
* IMPLIES:: logical implication
* IMPROPER-CONSP:: recognizer for improper (non-null-terminated) non-empty lists
* INT=:: test equality of two integers
* INTEGER-LENGTH:: number of bits in two’s complement integer representation
* INTEGER-LISTP:: recognizer for a true list of integers
* INTEGERP:: recognizer for whole numbers
* INTERN:: create a new symbol in a given package
* INTERN-IN-PACKAGE-OF-SYMBOL:: create a symbol with a given name
* INTERSECTP-EQ:: test whether two lists of symbols intersect
* INTERSECTP-EQUAL:: test whether two lists intersect

* IO:: input/output facilities in ACL2
* IRRELEVANT-FORMALS:: formals that are used but only insignificantly
* KEYWORD-VALUE-LISTP:: recognizer for true lists whose even-position elements are keywords
* KEYWORDP:: recognizer for keywords
* LAST:: the last `cons^` (not element) of a list
* LENGTH:: length of a string or proper list
* LET:: binding of lexically scoped (local) variables
* LET*:: binding of lexically scoped (local) variables
* LIST:: build a list
* LIST*:: build a list
* LISTP:: recognizer for (not necessarily proper) lists
* LOGAND:: bitwise logical 'and' of two integers
* LOGANDC1:: bitwise logical 'and' of two ints, complementing the first
* LOGANDC2:: bitwise logical 'and' of two ints, complementing the second
* LOGBITP:: the `i`th bit of an integer
* LOGCOUNT:: number of "on" bits in a two's complement number
* LOGEQV:: bitwise logical equivalence of two integers
* LOGIOR:: bitwise logical inclusive or of two integers
* LOGNAND:: bitwise logical 'nand' of two integers
* LOGNOR:: bitwise logical 'nor' of two integers
* LOGNOT:: bitwise not of a two's complement number
* LOGORC1:: bitwise logical inclusive or of two ints, complementing the first
* LOGORC2:: bitwise logical inclusive or of two ints, complementing the second
* LOGTEST:: test if two integers share a '1' bit
* LOGXOR:: bitwise logical exclusive or of two integers
* LOWER-CASE-P:: recognizer for lower case characters
* MAKE-CHARACTER-LIST:: coerce^ to a list of characters
* MAKE-LIST:: make a list of a given size
* MAX:: the larger of two rational numbers
* MEMBER:: membership predicate, using `eql^` as test
* MEMBER-EQ:: membership predicate, using `eq^` as test
* MEMBER-EQUAL:: membership predicate
* MIN:: the smaller of two rational numbers
* MINUSP:: test whether a rational number is negative
* MOD:: remainder using `floor^`
* MV:: returning multiple values
* MV-LET:: calling multi-valued ACL2 functions
* MV-NTH:: the mv-nth element (zero-based) of a list
* NFIX:: coerce to a natural number
* NINTH:: ninth member of the list
* NO-DUPLICATESP:: check for duplicates in a list
* NONNEGATIVE-INTEGER-QUOTIENT:: natural number division function
* NOT:: logical negation
* NTH:: the nth element (zero-based) of a list
* NTHCDR:: final segment of a list
* NULL:: recognizer for the empty list
* NUMERATOR:: dividend of a ratio in lowest terms
* ODDP:: test whether an integer is odd

* OR:: conjunction
* PAIRLIS:: see `pairlis$^`
* PAIRLIS$:: zipper together two lists
* PLUSP:: test whether a rational number is positive
* POSITION:: position of an item in a string or a list, using `eql^` as test
* POSITION-EQ:: position of an item in a string or a list, using `eq^` as test
* POSITION-EQUAL:: position of an item in a string or a list
* PPROGN:: evaluate a sequence of forms that return state^
* PROGN:: see the documentation for `er-progn^`
* PROOFS-CO:: the proofs character output channel
* PROPER-CONSP:: recognizer for proper (null-terminated) non-empty lists
* PUT-ASSOC-EQ:: modify an association list by associating a value with a key
* PUT-ASSOC-EQUAL:: modify an association list by associating a value with a key
* RASSOC:: look up value in association list, using `eql^` as test
* RATIONAL-LISTP:: recognizer for a true list of rational numbers
* RATIONALP:: recognizer for whole numbers
* REALPART:: real part of a complex number
* REDEFINING-PROGRAMS:: an explanation of why we restrict redefinitions
* REM:: remainder using `truncate^`
* REMOVE:: remove all occurrences, testing using `eql^`
* REMOVE-DUPLICATES:: remove duplicates from a string or (using `eql^`) a list
* REMOVE-DUPLICATES-EQUAL:: remove duplicates from a list
* REST:: rest (`cdr^`) of the list
* REVAPPEND:: concatentate the reverse^ of one list to another
* REVERSE:: reverse a list
* RFIX:: coerce to a rational number
* ROUND:: division returning an integer by rounding off
* SECOND:: second member of the list
* SET-DIFFERENCE-EQUAL:: elements of one list that are not elements of another
* SEVENTH:: seventh member of the list
* SIGNUM:: indicator for positive, negative, or zero
* SIXTH:: sixth member of the list
* STANDARD-CHAR-LISTP:: recognizer for standard characters
* STANDARD-CHAR-P:: recognizer for standard characters
* STANDARD-CO:: the character output channel to which `ld^` prints
* STANDARD-OI:: the standard object input "channel"
* STRING:: coerce^ to a string
* STRING-ALISTP:: recognizer for association lists with strings as keys
* STRING-APPEND:: concatenate^ two strings
* STRING-DOWNCASE:: in a given string, turn upper-case characters^ into lower-case
* STRING-EQUAL:: string equality without regard to case
* STRING-LISTP:: recognizer for a true list of strings
* STRING-UPCASE:: in a given string, turn lower-case characters^ into upper-case
* STRING<:: less-than test for strings
* STRING<=:: less-than-or-equal test for strings
* STRING>:: greater-than test for strings
* STRING>=:: less-than-or-equal test for strings
* STRINGP:: recognizer for strings

* STRIP-CARS:: collect up all first components of pairs in a list
* STRIP-CDRS:: collect up all second components of pairs in a list
* SUBLIS:: substitute an alist into a tree
* SUBSEQ:: subsequence of a string or list
* SUBSETP:: test if every `member^` of one list is a `member^` of the other
* SUBSETP-EQUAL:: check if all members of one list are members of the other
* SUBST:: a single substitution into a tree
* SYMBOL-<:: less-than test for symbols
* SYMBOL-ALISTP:: recognizer for association lists with symbols as keys
* SYMBOL-LISTP:: recognizer for a true list of symbols
* SYMBOL-NAME:: the name of a symbol (a string)
* SYMBOL-PACKAGE-NAME:: the name of the package of a symbol (a string)
* SYMBOLP:: recognizer for symbols
* TAKE:: initial segment of a list
* TENTH:: tenth member of the list
* THE:: run-time type check
* THIRD:: third member of the list
* TRUE-LIST-LISTP:: recognizer for true (proper) lists of true lists
* TRUE-LISTP:: recognizer for proper (null-terminated) lists
* TRUNCATE:: division returning an integer by truncating toward 0
* TYPE-SPEC:: type specifiers in declarations
* UNARY–:: arithmetic negation function
* UNARY-/:: reciprocal function
* UNION-EQ:: union of two lists of symbols
* UNION-EQUAL:: union of two lists
* UPDATE-NTH:: modify a list by putting the given value at the given position
* UPPER-CASE-P:: recognizer for upper case characters
* ZERO-TEST-IDIOMS:: how to test for 0
* ZEROP:: test an acl2-number against 0
* ZIP:: testing an "integer" against 0
* ZP:: testing a "natural" against 0

Related topics other than immediate subtopics:
* ACL2-CUSTOMIZATION:: customizing ACL2 for a particular user
* ARRAYS:: an introduction to ACL2 arrays.
* CERTIFY-BOOK:: how to produce a certificate^ for a book
* COMP:: compile some ACL2 functions
* DEFCONST:: define a constant
* DEFPKG:: define a new symbol package
* DEFUN:: define a function symbol
* INCLUDE-BOOK:: load the events^ in a file
* MUTUAL-RECURSION:: define some mutually recursive functions
* SET-COMPILE-FNS:: have each function compiled as you go along.
* SET-IGNORE-OK:: allow unused formals and locals without an `ignore` declaration
* SET-IRRELEVANT-FORMALS-OK:: allow irrelevant formals in definitions

See any documentation for Common Lisp for more details on many of these functions.

10.1 *

* multiplication macro

* is really a macro that expands to calls of the function `binary-*`^. So for example

```
(* x y 4 z)
```

represents the same term as

```
(binary-* x (binary-* y (binary-* 4 z))).
```

See Section 10.35 [BINARY-*], page 411.

* is a Common Lisp function. See any Common Lisp documentation for more information.

10.2 *STANDARD-CI*

STANDARD-CI an ACL2 character-based analogue of CLTL's `*standard-input*`

The value of the ACL2 constant `*standard-ci*` is an open character input channel that is synonymous to Common Lisp's `*standard-input*`.

ACL2 character input from `*standard-ci*` is actually obtained by reading characters^ from the stream named by Common Lisp's `*standard-input*`. That is, by changing the setting of `*standard-input*` in raw Common Lisp you can change the source from which ACL2 reads on the channel `*standard-ci*`. See Section 10.3 [*STANDARD-CO*], page 399.

10.3 *STANDARD-CO*

STANDARD-CO the ACL2 analogue of CLTL's `*standard-output*`

The value of the ACL2 constant `*standard-co*` is an open character output channel that is synonymous to Common Lisp's `*standard-output*`.

ACL2 character output to `*standard-co*` will go to the stream named by Common Lisp's `*standard-output*`. That is, by changing the setting of `*standard-output*` in raw Common Lisp you can change the actual destination of ACL2 output on the channel named by `*standard-co*`. Observe that this happens without changing the logical value of `*standard-co*` (which is some channel symbol). Changing the setting of `*standard-output*` in raw Common Lisp essentially just changes the map that relates ACL2 to the physical world of terminals, files, etc.

To see the value of this observation, consider the following. Suppose you write an ACL2 function which does character output to the constant channel `*standard-co*`. During testing you see that the output actually goes to your terminal. Can you use the function to output to a file? Yes, if you are willing to do a little work in raw Common Lisp: open a stream to the file in question, set `*standard-output*` to that stream, call your ACL2 function, and then close the stream and restore `*standard-output*` to its nominal value. Similar observations can be made about the two ACL2 input channels, `*standard-oi*^` and `*standard-ci*^`, which are analogues of `*standard-input*`.

Another reason you might have for wanting to change the actual streams associated with `*standard-oi*^` and `*standard-co*` is to drive the ACL2 top-level loop, `ld^`, on alternative input and output streams. This end can be accomplished easily within ACL2 by either calling `ld^` on the desired channels or file names or by resetting the ACL2 `state^` global variables `'standard-oi^` and `'standard-co^` which are used by `ld^`. See Section 10.218 [STANDARD-OI], page 493 and see Section 10.217 [STANDARD-CO], page 493.

10.4 *STANDARD-OI*

STANDARD-OI an ACL2 object-based analogue of CLTL's `*standard-input*`

The value of the ACL2 constant `*standard-oi*` is an open object input channel that is synonymous to Common Lisp's `*standard-input*`.

ACL2 object input from `*standard-oi*` is actually obtained by reading from the stream named by Common Lisp's `*standard-input*`. That is, by changing the setting of `*standard-input*` in raw Common Lisp you can change the source from which ACL2 reads on the channel `*standard-oi*`. See Section 10.3 [*STANDARD-CO*], page 399.

10.5 +

`+` addition macro

`+` is really a macro that expands to calls of the function `binary-+^`. So for example

```
(+ x y 4 z)
```

represents the same term as

```
(binary-+ x (binary-+ y (binary-+ 4 z))).
```

See Section 10.36 [BINARY-+], page 412.

10.6 -

- macro for subtraction and negation

See Section 10.36 [BINARY-+], page 412 for addition and see Section 10.252 [UNARY–], page 505 for negation.

Note that `-` represents subtraction as follows:

```
(- x y)
```

represents the same term as

```
(+ x (- y))
```

which is really

```
(binary-+ x (unary-- y)).
```

Also note that `-` represents arithmetic negation as follows:

```
(- x)
```

expands to

```
(unary-- x).
```

10.7 /

/ macro for division and reciprocal

See Section 10.35 [BINARY-*], page 411 for multiplication and see Section 10.253 [UNARY-/], page 506 for reciprocal.

Note that `/` represents division as follows:

```
(/ x y)
```

represents the same term as

```
(* x (/ y))
```

which is really

```
(binary-* x (unary-/ y)).
```

Also note that `/` represents reciprocal as follows:

```
(/ x)
```

expands to

```
(unary-/ x).
```

`/` is a Common Lisp macro. See any Common Lisp documentation for more information.

10.8 /=

/= test inequality of two numbers

`(/= x y)` is logically equivalent to `(not (equal x y))`.

Unlike `equal^`, `/=` has a guard^ requiring both of its arguments to be numbers. Generally, `/=` is executed more efficiently than a combination of `not^` and `equal^`.

For a discussion of the various ways to test against 0, See Section 10.258 [ZERO-TEST-IDIOMS], page 508.

`/=` is a Common Lisp function. See any Common Lisp documentation for more information.

10.9 1+

`1+` increment by 1

`(1+ x)` is the same as `(+ 1 x)`. See Section 10.5 [+], page 400.

`1+` is a Common Lisp function. See any Common Lisp documentation for more information.

10.10 1-

1- decrement by 1

`(1- x)` is the same as `(- x 1)`. See Section 10.6 [-], page 401.

`1-` is a Common Lisp function. See any Common Lisp documentation for more information.

10.11 <

`<` less-than

Basic axiom:

```
(equal (< x y)
       (if (and (rationalp x)
                (rationalp y))
           (< x y)
         (let ((x1 (if (acl2-numberp x) x 0))
               (y1 (if (acl2-numberp y) y 0)))
           (or (< (realpart x1) (realpart y1))
               (and (equal (realpart x1) (realpart y1))
                    (< (imagpart x1) (imagpart y1)))))))
```

Guard^ for `(< x y)`:

```
(and (rationalp x) (rationalp y))
```

Notice that like all arithmetic functions, `<` treats non-numeric inputs as `0`.

This function has the usual meaning on the rational numbers, but is extended to the complex rational numbers using the lexicographic order: first the real parts are compared, and if they are equal, then the imaginary parts are compared.

10.12 <=

<= less-than-or-equal test

`<=` is a macro, and `(<= x y)` expands to the same thing as `(not (< y x))`. See Section 10.11 [<], page 403.

`<=` is a Common Lisp function. See any Common Lisp documentation for more information.

10.13 =

= test equality (of two numbers, symbols, or characters^)

`(= x y)` is logically equivalent to `(equal x y)`.

Unlike `equal^`, `=` has a guard^ requiring both of its arguments to be numbers. Generally, `=` is executed more efficiently than `equal^`.

For a discussion of the various ways to test against 0, See Section 10.258 [ZERO-TEST-IDIOMS], page 508.

`=` is a Common Lisp function. See any Common Lisp documentation for more information.

10.14 >

`>` greater-than test

`>` is a macro, and `(> x y)` expands to the same thing as `(< y x)`. See Section 10.11 [<], page 403.

`>` is a Common Lisp function. See any Common Lisp documentation for more information.

10.15 >=

`>=` greater-than-or-equal test

`>=` is a macro, and `(>= x y)` expands to the same thing as `(not (< x y))`. See Section 10.11 [<], page 403.

`>=` is a Common Lisp function. See any Common Lisp documentation for more information.

10.16 ABS

ABS the absolute value of a rational number

`(Abs x)` is `-x` if `x` is negative and is `x` otherwise.

The guardˆ for `abs` requires its argument to be a rational numbers.

`Abs` is a Common Lisp function. See any Common Lisp documentation for more information.

From “Common Lisp the Language” page 205, we must not allow complex `x` as an argument to `abs` in ACL2, because if we did we would have to return a number that might not be a complex rational number (see Section 10.88 [COMPLEX-RATIONALP], page 430) and hence not an ACL2 object.

10.17 ACL2-CHARACTERP

ACL2-CHARACTERP recognizer for characters^

(acl2-characterp x) is true if and only if x is a character.

10.18 ACL2-COMPLEX-RATIONALP

ACL2-COMPLEX-RATIONALP recognizer for complex rationals

(acl2-complex-rationalp x) is true if and only if x is a complex rational number (and hence, not a rational number).

10.19 ACL2-NUMBERP

ACL2-NUMBERP recognizer for numbers

(acl2-numberp x) is true if and only if x is a number, i.e., a rational or complex rational number.

10.20 ACL2-USER

ACL2-USER a package the ACL2 user may prefer

This package imports the standard Common Lisp symbols that ACL2 supports and also a few symbols from package "ACL2" that are commonly used when interacting with ACL2. You may prefer to select this as your current package so as to avoid colliding with ACL2 system names.

This package imports the symbols listed in *common-lisp-symbols-from-main-lisp-package*, which contains hundreds of CLTL function and macro names including those supported by ACL2 such as cons^, car^, and cdr^. It also imports the symbols in *acl2-exports*, which contains a few symbols that are frequently used while interacting with the ACL2 system, such as implies^, defthm^, and rewrite^. It imports nothing else.

Thus, names such as alistp^, member-equal^, and type-set^, which are defined in the "ACL2" package are not present here. If you find yourself frequently colliding with names that are defined in "ACL2" you might consider selecting "ACL2-USER" as your current package (see Section 8.44 [IN-PACKAGE], page 305). If you select "ACL2-USER" as the current package, you may then simply type `member-equal^` to refer to `acl2-user::member-equal`, which you may define as you see fit. Of course, should you desire to refer to the `"ACL2"` version of `member-equal^`, you will have to use the `"ACL2::"` prefix, e.g., `acl2::member-equal`.

If, while using `"ACL2-USER"` as the current package, you find that there are symbols from `"ACL2"` that you wish we had imported into it (because they are frequently used in interaction), please bring those symbols to our attention. For example, should `union-theories^` and `universal-theory^` be imported? Except for stabilizing on the "frequently used" names from `"ACL2"`, we intend never to define a symbol whose `symbol-package-name^` is `"ACL2-USER"`.

10.21 ACONS

ACONS constructor for association lists

`(Acons key datum alist)` equals the result of consing the pair `(cons key datum)` to the front of the association list `alist`.

`(Acons key datum alist)` has a guard^ of `(alistp alist)`. `Acons` is a Common Lisp function. See any Common Lisp documentation for more information.

10.22 ADD-TO-SET-EQ

ADD-TO-SET-EQ add a symbol to a list

For a symbol `x` and a true list `lst`, `(add-to-set-eq x lst)` is the result of `cons^`ing `x` on to the front of `lst`, unless `x` is already a member of `lst`, in which case it equals `x`.

`(add-to-set-eq x lst)` has a guard^ that `lst` is a true list and moreover, either `x` is a symbol or `lst` is a list of symbols.

10.23 ALISTP

ALISTP recognizer for association lists

`(alistp x)` is true if and only if `x` is a list of `cons^` pairs.

`(alistp x)` has a guard^ of `t`.

10.24 ALPHA-CHAR-P

ALPHA-CHAR-P recognizer for alphabetic characters

`(Alpha-char-p x)` is true if and only if `x` is a alphabetic character, i.e., one of the characters^ `#a`, `#b`, ..., `#z`, `#A`, `#B`, ..., `#Z`.

The guard^ for `alpha-char-p` requires its argument to be a character.

`Alpha-char-p` is a Common Lisp function. See any Common Lisp documentation for more information.

10.25 AND

AND conjunction

`And` is the macro for conjunctions. `And` takes any number of arguments. `And` returns `nil` if one of the arguments is `nil`, but otherwise returns the last argument. If there are no arguments, `and` returns `t`.

`And` is a Common Lisp macro. See any Common Lisp documentation for more information.

10.26 APPEND

APPEND concatenate^ two or more lists

`Append`, which takes two or more arguments, expects all the arguments except perhaps the last to be true (null-terminated) lists. It returns the result of concatenating all the elements of all the given lists into a single list. Actually, in ACL2 `append` is a macro that expands into calls of the binary function `binary-append^`.

`Append` is a Common Lisp function. See any Common Lisp documentation for more information.

10.27 ASH

ASH arithmetic shift operation

`(ash i c)` is the result of taking the two's complement representation of the integer `i` and shifting it by `c` bits, padding with `c` `0` bits if `c` is positive, and dropping `(abs c)` bits if `c` is negative.

The guard^ for `ash` requires that its arguments are integers.

`Ash` is a Common Lisp function. See any Common Lisp documentation for more information.

10.28 ASSOC

ASSOC look up key in association list, using `eql^` as test

`(Assoc x alist)` is the first member of `alist` whose `car^` is `x`, or `nil` if no such member exists.

`(Assoc x alist)` is provably the same in the ACL2 logic as `(assoc-equal x alist)`. It has a stronger guard^ than `assoc-equal^` because it uses `eql^` to test whether `x` is equal to the `car^` of a given member of `alist`. Its guard^ requires that `alist` is an `alistp^`, and moreover, either `(eqlablep x)` or all `car^`s of members of `alist` are `eqlablep^`. See Section 10.30 [ASSOC-EQUAL], page 410 and see Section 10.29 [ASSOC-EQ], page 409.

`Assoc` is a Common Lisp function. See any Common Lisp documentation for more information. Since ACL2 functions cannot take keyword arguments (though macros can), the ACL2 functions `assoc-equal^` and `assoc-eq^` are defined to correspond to calls of the Common Lisp function `assoc` whose keyword argument `:test` is `equal^` or `eq^`, respectively.

10.29 ASSOC-EQ

ASSOC-EQ look up key in association list, using `eq^` as test

`(Assoc-eq x alist)` is the first member of `alist` whose `car^` is `x`, or `nil` if no such member exists.

`(Assoc-eq x alist)` is provably the same in the ACL2 logic as `(assoc x alist)` and `(assoc-equal x alist)`, but it has a stronger guard^ because it uses `eq^` for a more efficient test for whether `x` is equal to a given key of `alist`. Its guard^ requires that `alist` is an association list (see Section 10.23 [ALISTP], page 407), and moreover, either `x` is a symbol or all keys of `alist` are symbols, i.e., `alist` is a `symbol-alistp^`.

10.30 ASSOC-EQUAL

ASSOC-EQUAL look up key in association list

`(Assoc-equal x alist)` is the first member of `alist` whose `car^` is `x`, or `nil` if no such member exists.

`(Assoc-equal x alist)` has a guard^ of `(alistp alist)`, and returns the first member of alist whose `car^` is `x`, or `nil` if no such member exists. Thus, `assoc-equal` has the same functionality as the Common Lisp function `assoc^`, except that it uses the `equal^` function to test whether `x` is the same as each successive 'key' of `alist`. See Section 10.28 [ASSOC], page 409 and see Section 10.29 [ASSOC-EQ], page 409.

10.31 ASSOC-KEYWORD

ASSOC-KEYWORD look up key in a `keyword-value-listp^`

If `l` is a list of even length of the form `(k1 a1 k2 a2 ... kn an)`, where each `ki` is a keyword, then `(assoc-keyword key l)` is the first tail of `l` starting with `key` if key is some `ki`, and is `nil` otherwise.

The guard^ for `(assoc-keyword key l)` is `(keyword-value-listp l)`.

10.32 ASSOC-STRING-EQUAL

ASSOC-STRING-EQUAL look up key, a string, in association list

(Assoc-string-equal x alist) is similar to assoc-equal^. However, for string x and alist alist, the comparison of x with successive keys in alist is done using string-equal^ rather than equal^.

The guard^ for assoc-string-equal requires that x is a string and alist is an alist.

10.33 ATOM

ATOM recognizer for atoms

(atom x) is true if and only if x is an atom, i.e., not a cons^ pair.

Atom has a guard^ of t, and is a Common Lisp function. See any Common Lisp documentation for more information.

10.34 ATOM-LISTP

ATOM-LISTP recognizer for a true list of atom^s

The predicate atom-listp tests whether its argument is a true-listp^ of atom^s, i.e., of non-conses.

10.35 BINARY-*

BINARY-* multiplication function

Basic axiom:

```
(equal (binary-* x y)
       (if (acl2-numberp x)
           (if (acl2-numberp y)
               (binary-* x y)
             0)
         0))
```

Guard^ for (binary-* x y):

```
(and (acl2-numberp x) (acl2-numberp y))
```

Notice that like all arithmetic functions, `binary-*` treats non-numeric inputs as `0`.

Calls of the macro `*^` expand to calls of `binary-*`; see Section 10.1 [*], page 399.

10.36 BINARY-+

BINARY-+ addition function

Basic axiom:

```
(equal (binary-+ x y)
       (if (acl2-numberp x)
           (if (acl2-numberp y)
               (binary-+ x y)
             x)
         (if (acl2-numberp y)
             y
           0)))
```

Guard^ for `(binary-* x y)`:

```
(and (acl2-numberp x) (acl2-numberp y))
```

Notice that like all arithmetic functions, `binary-+` treats non-numeric inputs as `0`.

Calls of the macro `+^` expand to calls of `binary-+`; see Section 10.5 [+], page 400.

10.37 BINARY-APPEND

BINARY-APPEND concatenate^ two lists

This binary function implements `append^`, which is a macro in ACL2. See Section 10.26 [APPEND], page 408

The guard^ for `binary-append` requires the first argument to be a `true-listp^`.

10.38 BOOLEANP

BOOLEANP recognizer for booleans

`(Booleanp x)` is `t` if `x` is `t` or `nil`, and is `nil` otherwise.

See Section 8.37 [GENERALIZED-BOOLEANS], page 281 for a discussion of a potential soundness problem for ACL2 related to the question: Which Common Lisp functions are known to return Boolean values?

10.39 BUTLAST

BUTLAST all but a final segment of a list

`(Butlast l n)` is the list obtained by removing the last `n` elements from the true list `l`. The following is a theorem (though it takes some effort, including lemmas, to get ACL2 to prove it).

```
(implies (and (integerp n)
              (<= 0 n)
              (true-listp l))
         (equal (length (butlast l n))
                (if (< n (length l))
                    (- (length l) n)
                  0)))
```

For related functions, see Section 10.244 [TAKE], page 502 and see Section 10.179 [NTHCDR], page 477.

The guard^ for `(butlast l n)` requires that `n` is a nonnegative integer and `lst` is a true list.

`Butlast` is a Common Lisp function. See any Common Lisp documentation for more information. Note: In Common Lisp the second argument of `butlast` is optional, but in ACL2 it is required.

10.40 CAAAAR

CAAAAR `car^` of the `caaar^`

See any Common Lisp documentation for details.

10.41 CAAADR

CAAADR   `car^` of the `caadr^`

See any Common Lisp documentation for details.

10.42 CAAAR

CAAAR   `car^` of the `caar^`

See any Common Lisp documentation for details.

10.43 CAADAR

CAADAR   `car^` of the `cadar^`

See any Common Lisp documentation for details.

10.44 CAADDR

CAADDR   `car^` of the `caddr^`

See any Common Lisp documentation for details.

10.45 CAADR

CAADR   `car^` of the `cadr^`

See any Common Lisp documentation for details.

10.46 CAAR

CAAR `car^` of the `car^`

See any Common Lisp documentation for details.

10.47 CADAAR

CADAAR `car^` of the `cdaar^`

See any Common Lisp documentation for details.

10.48 CADADR

CADADR `car^` of the `cdadr^`

See any Common Lisp documentation for details.

10.49 CADAR

CADAR `car^` of the `cdar^`

See any Common Lisp documentation for details.

10.50 CADDAR

CADDAR `car^` of the `cddar^`

See any Common Lisp documentation for details.

10.51 CADDDR

CADDDR `car^` of the `cdddr^`

See any Common Lisp documentation for details.

10.52 CADDR

CADDR `car^` of the `cddr^`

See any Common Lisp documentation for details.

10.53 CADR

CADR `car^` of the `cdr^`

See any Common Lisp documentation for details.

10.54 CAR

CAR returns the first element of a non-empty list, else `nil`

Basic axiom:

```
(equal (car x)
       (cond
        ((consp x)
         (car x))
        (t nil)))
```

Guard^:

```
(or (consp x) (equal x nil))
```

Notice that in the ACL2 logic, `car` returns `nil` for every atom^.

10.55 CASE

CASE conditional based on if-then-else using `eql^`

```
Example Form:
(case typ
  ((:character foo)
   (open file-name :direction :output))
  (bar (open-for-bar file-name))
  (otherwise
   (my-error "Illegal.")))
```

is the same as

```
(cond ((member typ '(:character foo))
       (open file-name :direction :output))
      ((eql typ 'bar)
       (open-for-bar file-name))
      (t (my-error "Illegal.")))
```

which in turn is the same as

```
(if (member typ '(:character foo))
    (open file-name :direction :output)
    (if (eql typ 'bar)
        (open-for-bar file-name)
        (my-error "Illegal.")))

General Forms:
(case expr
  (x1 val-1)
  ...
  (xk val-k)
  (otherwise val-k+1))

(case expr
  (x1 val-1)
  ...
  (xk val-k)
  (t val-k+1))
```

where each `xi` is either `eqlablep^` or a true list of `eqlablep^` objects.

`Case` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.56 CASE-MATCH

CASE-MATCH pattern matching or destructuring

```
General Form:
(case-match x
  (pat1 dcl1 body1)
  ...
  (patk dclk bodyk))
```

where `x` is a variable symbol, the `pati` are structural patterns as described below, the `dcli` are optional `declare^` forms and the `bodyi` are terms. Return the value(s) of the `bodyi` corresponding to the first `pati` matching `x`, or `nil` if none matches.

Pattern Language:

With the few special exceptions described below, matching requires that the `cons^` structure of `x` be isomorphic to that of the pattern, down to the atom^s in the pattern. Non-symbol atom^s in the pattern match only themselves. Symbols in the pattern denote variables which match anything and which are bound by a successful match to the corresponding substructure of `x`. Variables that occur more than once must match the same (`EQUAL^`) structure in every occurrence.

```
Exceptions:
&               Matches anything and is not bound.  Multiple
                  occurrences of & in a pattern may match different
                  structures.
nil, t, *sym*   These symbols cannot be bound and match only their
                  global values.
!sym            where sym is a symbol that is already bound in the
                  context of the case-match, matches only the
                  current binding of sym.
'obj            Matches only itself.
```

Some examples are shown below.

Below we show some sample patterns and examples of things they match and do not match.

```
pattern       matches         non-matches
(x y y)       (ABC 3 3)       (ABC 3 4)    ; 3 is not 4
(fn x . rst)  (P (A I) B C)   (ABC)        ; NIL is not (x . rst)
              (J (A I))                    ; rst matches nil
```

```
('fn (g x) 3) (FN (H 4) 3)    (GN (G X) 3) ; 'fn matches only itself
(& t & !x)    ((A) T (B) (C))              ; provided x is '(C)
```

Consider the two binary trees that contain three leaves. They might be described as `(x . (y . z))` and `((x . y) . z)`, where `x`, `y`, and `z` are atomic. Suppose we wished to recognize those trees. The following `case-match` would do:

```
(case-match tree
  ((x . (y . z))
   (and (atom x) (atom y) (atom z)))
  (((x . y) . z)
   (and (atom x) (atom y) (atom z))))
```

Suppose we wished to recognize such trees where all three tips are identical. Suppose further we wish to return the tip if the tree is one of those recognized ones and to return the number `7` otherwise.

```
(case-match tree
  ((x . (x . x))
   (if (atom x) x 7))
  (((x . x) . x)
   (if (atom x) x 7))
  (& 7))
```

Note that `case-match` returns `nil` if no `pati` matches. Thus if we must return `7` in that case, we have to add as the final pattern the `&`, which always matches anything.

10.57 CDAAAR

CDAAAR `cdr`^ of the `caaar`^

See any Common Lisp documentation for details.

10.58 CDAADR

CDAADR `cdr`^ of the `caadr`^

See any Common Lisp documentation for details.

10.59 CDAAR

CDAAR `cdr`^ of the `caar`^

See any Common Lisp documentation for details.

10.60 CDADAR

CDADAR `cdr`^ of the `cadar`^

See any Common Lisp documentation for details.

10.61 CDADDR

CDADDR `cdr`^ of the `caddr`^

See any Common Lisp documentation for details.

10.62 CDADR

CDADR `cdr`^ of the `cadr`^

See any Common Lisp documentation for details.

10.63 CDAR

CDAR `cdr`^ of the `car`^

See any Common Lisp documentation for details.

10.64 CDDAAR

CDDAAR `cdr^` of the `cdaar^`

See any Common Lisp documentation for details.

10.65 CDDADR

CDDADR `cdr^` of the `cdadr^`

See any Common Lisp documentation for details.

10.66 CDDAR

CDDAR `cdr^` of the `cdar^`

See any Common Lisp documentation for details.

10.67 CDDDAR

CDDDAR `cdr^` of the `cddar^`

See any Common Lisp documentation for details.

10.68 CDDDDR

CDDDDR `cdr^` of the `cdddr^`

See any Common Lisp documentation for details.

10.69 CDDDR

CDDDR `cdr^` of the `cddr^`

See any Common Lisp documentation for details.

10.70 CDDR

CDDR `cdr^` of the `cdr^`

See any Common Lisp documentation for details.

10.71 CDR

CDR returns the second element of a `cons^` pair, else `nil`

Basic axiom:

```
(equal (cdr x)
       (cond
        ((consp x)
         (cdr x))
        (t nil)))
```

Guard^:

```
(or (consp x) (equal x nil))
```

Notice that in the ACL2 logic, `cdr` returns `nil` for every atom^.

10.72 CEILING

CEILING division returning an integer by truncating toward positive infinity

```
Example Forms:
```

```
ACL2 !>(ceiling 14 3)
5
ACL2 !>(ceiling -14 3)
-4
ACL2 !>(ceiling 14 -3)
-4
ACL2 !>(ceiling -14 -3)
5
ACL2 !>(ceiling -15 -3)
5
```

`(Ceiling i j)` is the result of taking the quotient of `i` and `j` and returning the smallest integer that is at least as great as that quotient. For example, the quotient of `-14` by `3` is `-4 2/3`, and the smallest integer at least that great is `-4`.

The guard^ for `(ceiling i j)` requires that `i` and `j` are rational numbers and `j` is non-zero.

`Ceiling` is a Common Lisp function. See any Common Lisp documentation for more information.

10.73 CHAR

CHAR the nth^ element (zero-based) of a string

`(Char s n)` is the `n`th element of `s`, zero-based. If `n` is greater than or equal to the length of `s`, then `char` returns `nil`.

`(Char s n)` has a guard^ that `n` is a non-negative integer and `s` is a `stringp^`.

`Char` is a Common Lisp function. See any Common Lisp documentation for more information.

10.74 CHAR-CODE

CHAR-CODE the numeric code for a given character

Basic axiom:

```
(equal (char-code x)
       (if (characterp x)
```

```
                (char-code x)
              0))
```

Guard^ for (char-code x):

```
(characterp x)
```

This function maps all non-characters to 0.

10.75 CHAR-DOWNCASE

CHAR-DOWNCASE turn upper-case characters^ into lower-case characters^

(Char-downcase x) is equal to #A when x is #a, #B when x is #b, ..., and #Z when x is #z, and is x for any other character.

The guard^ for char-downcase requires its argument to be a character.

Char-downcase is a Common Lisp function. See any Common Lisp documentation for more information.

10.76 CHAR-EQUAL

CHAR-EQUAL character equality without regard to case

For characters^ x and y, (char-equal x y) is true if and only if x and y are the same except perhaps for their case.

The guard^ on char-equal requires that its arguments are both characters^.

Char-equal is a Common Lisp function. See any Common Lisp documentation for more information.

10.77 CHAR-UPCASE

CHAR-UPCASE turn lower-case characters^ into upper-case characters^

`(Char-upcase x)` is equal to `#A` when `x` is `#a`, `#B` when `x` is `#b`, ..., and `#Z` when `x` is `#z`, and is `x` for any other character.

The guard^ for `char-upcase` requires its argument to be a character.

`Char-upcase` is a Common Lisp function. See any Common Lisp documentation for more information.

10.78 CHAR<

CHAR< less-than test for characters^

`(char< x y)` is true if and only if the character code of `x` is less than that of `y`. See Section 10.74 [CHAR-CODE], page 423.

The guard^ for `char<` specifies that its arguments are characters^.

`Char<` is a Common Lisp function. See any Common Lisp documentation for more information.

10.79 CHAR<=

CHAR<= less-than-or-equal test for characters^

`(char<= x y)` is true if and only if the character code of `x` is less than or equal to that of `y`. See Section 10.74 [CHAR-CODE], page 423.

The guard^ for `char<=` specifies that its arguments are characters^.

`Char<=` is a Common Lisp function. See any Common Lisp documentation for more information.

10.80 CHAR>

CHAR> greater-than test for characters^

`(char> x y)` is true if and only if the character code of `x` is greater than that of `y`. See Section 10.74 [CHAR-CODE], page 423.

The guard^ for `char>` specifies that its arguments are characters^.

`Char>` is a Common Lisp function. See any Common Lisp documentation for more information.

10.81 CHAR>=

CHAR>= greater-than-or-equal test for characters^

`(char>= x y)` is true if and only if the character code of `x` is greater than or equal to that of `y`. See Section 10.74 [CHAR-CODE], page 423.

The guard^ for `char>=` specifies that its arguments are characters^.

`Char>=` is a Common Lisp function. See any Common Lisp documentation for more information.

10.82 CHARACTER-LISTP

CHARACTER-LISTP recognizer for a true list of characters

The predicate `character-listp` tests whether its argument is a true list of characters^.

10.83 CHARACTERS

CHARACTERS characters in ACL2

ACL2 accepts 256 distinct characters, which are the characters obtained by applying the function `code-char`^ to each integer from `0` to `255`. Among these, Common Lisp designates certain ones as *standard characters*, namely those of the form `(code-char n)` where `n` is from `33` to `126`, together with `#\Newline` and `#\Space`. The actual standard characters may be viewed by evaluating the `defconst^` `*standard-chars*`.

To be more precise, Common Lisp does not specify the precise relationship between `code-char^` and the standard characters. However, we check that the underlying Common Lisp implementation uses a particular relationship that extends the usual ASCII coding of characters. We also check that Space, Tab, Newline, Page, and Rubout correspond to characters with respective `char-code^`s `32`, `9`, `10`, `12`, and `127`.

`Code-char^` has an inverse, `char-code^`. Thus, when `char-code^` is applied to an ACL2 character, `c`, it returns a number `n` between `0` and `255` inclusive such that `(code-char n)` = `c`.

The preceding paragraph implies that there is only one ACL2 character with a given character code. CLTL allows for "attributes" for characters, which could allow distinct characters with the same code, but ACL2 does not allow this.

The Character Reader

ACL2 support the '`#\`' notation for characters provided by Common Lisp, with some restrictions. First of all, for every character `c`, the notation

```
#\c
```

may be used to denote the character object `c`. That is, the user may type in this notation and ACL2 will read it as denoting the character object `c`. In this case, the character immediately following `c` must be one of the following "terminating characters": a Tab, a Newline, a Page character, a space, or one of the characters:

```
"  '  (  )  ;  `  ,
```

Other than the notation above, ACL2 accepts alternate notation for five characters.

```
#\Space
#\Tab
#\Newline
#\Page
#\Rubout
```

Again, in each of these cases the next character must be from among the set of "terminating characters" described in the single-character case. Our implementation is consistent with IS0-8859, even though we don't provide `#\` syntax for entering characters other than that described above.

Finally, we note that it is our intention that any object printed by ACL2's top-level-loop may be read back into ACL2. Please notify the implementors if you find a counterexample to this claim.

10.84 CODE-CHAR

CODE-CHAR the character corresponding to a given numeric code

Basic axiom:

```
(equal (code-char x)
       (if (and (integerp x)
                (>= x 0)
                (< x 256))
           (code-char x)
         (code-char 0)))
```

Guard^ for `(code-char x)`:

```
(and (integerp x)
     (>= x 0)
     (< x 256))
```

ACL2 supports 8-bit characters^. Inputs not between `0` and `255` are treated as `0`.

10.85 COERCE

COERCE coerce a character list to a string and a string to a list

Basic axiom:

```
(equal (coerce x y)
       (cond
        ((equal y 'list)
         (if (stringp x)
```

```
                (coerce x 'list)
              nil))
           (t
            (coerce (make-character-list x) 'string))))
```

Guard^ for (coerce x y):

```
(if (equal y 'list)
    (stringp x)
  (if (equal y 'string)
      (character-listp x)
    nil))
```

10.86 COMPILATION

COMPILATION compiling ACL2 functions

```
Example:
ACL2 !>:comp app
ACL2 !>:set-compile-fns t
```

See Section 9.5 [COMP], page 379 and see Section 6.27 [SET-COMPILE-FNS], page 187.

10.87 COMPLEX

COMPLEX create an ACL2 number

```
Examples:
(complex x 3) ; x + 3i, where i is the principal square root of -1
(complex x y) ; x + yi
(complex x 0) ; same as x, for rational numbers x
```

The function `complex` takes two rational number arguments and returns an ACL2 number. This number will be of type `(complex rational)` [as defined in the Common Lisp language], except that if the second argument is zero, then `complex` returns its first argument. The function `complex-rationalp^` is a recognizer for complex rational numbers, i.e. for ACL2 numbers that are not rational numbers.

The reader macro `#C` (which is the same as `#c`) provides a convenient way for typing in complex numbers. For explicit rational numbers `x` and `y`, `#C(x y)` is read to the same value as `(complex x y)`.

The functions `realpart^` and `imagpart^` return the real and imaginary parts (respectively) of a complex (possibly rational) number. So for example, `(realpart #C(3 4)) = 3`, `(imagpart #C(3 4)) = 4`, `(realpart 3/4) = 3/4`, and `(imagpart 3/4) = 0`.

A basic axiom that shows what `complex` returns on arguments violating its guard^ (which says that both arguments are rational numbers) is the following.

```
(equal (complex x y)
       (complex (if (rationalp x) x 0)
                (if (rationalp y) y 0)))
```

10.88 COMPLEX-RATIONALP

COMPLEX-RATIONALP recognizes complex rational numbers

```
Examples:
(complex-rationalp 3)       ; nil, as 3 is rational, not complex rational
(complex-rationalp #c(3 0)) ; nil, since #c(3 0) is the same as 3
(complex-rationalp t)       ; nil
(complex-rationalp #c(3 1)) ; t, as #c(3 1) is the complex number 3 + i
```

See Section 10.87 [COMPLEX], page 429 for more about complex rationals in ACL2.

10.89 CONCATENATE

CONCATENATE concatenate lists or strings together

```
Examples:
(concatenate 'string "ab" "cd" "ef")     ; equals "abcdef"
(concatenate 'string "ab")               ; equals "ab"
(concatenate 'list '(a b) '(c d) '(e f)) ; equals '(a b c d e f)
(concatenate 'list)                      ; equals nil

General Form:
(concatenate result-type x1 x2 ... xn)
```

where `n >= 0` and either: `result-type` is `'string^` and each `xi` is a string; or `result-type` is `'list^` and each `xi` is a true list. `Concatenate` simply concatenates its arguments to form the result string or list. Also see Section 10.26 [APPEND], page 408 and see Section 10.221 [STRING-APPEND], page 494, though `concatenate` is probably preferable to `string-append^` for efficiency.

Note: We do *not* try to comply with the Lisp language's insistence that `concatenate` copies its arguments. Not only are we in an applicative setting, where this issue shouldn't matter for the logic, but also we do not actually modify the underlying lisp implementation of `concatenate`; we merely provide a definition for it.

`Concatenate` is a Common Lisp function. See any Common Lisp documentation for more information.

10.90 COND

COND conditional based on if-then-else

`Cond` is the construct for IF, THEN, ELSE IF, ... The test is against `nil`. The argument list for `cond` is a list of "clauses", each of which is a list. In ACL2, clauses must have length 1 or 2.

`Cond` is a Common Lisp macro. See any Common Lisp documentation for more information.

10.91 CONJUGATE

CONJUGATE complex number conjugate

`Conjugate` takes an ACL2 number as an argument, and returns its complex conjugate (i.e., the result of negating its imaginary part.).

`Conjugate` is a Common Lisp function. See any Common Lisp documentation for more information.

10.92 CONS

CONS pair and list constructor

`(cons x y)` is a pair whose first component is `x` and second component is `y`. If `y` is a list, then `(cons x y)` is a list that has an addtional element `x` on the front.

10.93 CONSP

CONSP recognizer for cons^ pairs

`(consp x)` is true if and only if `x` is a cons^ pair.

10.94 DECLARE

DECLARE declarations

```
Examples:
(declare (ignore x y z))
(declare (ignore x y z)
         (type integer i j k)
         (type (satisfies integerp) m1 m2))
(declare (xargs :guard (and (integerp i)
                            (<= 0 i))
                :guard-hints (("Goal" :use (:instance lemma3
                                                  (x (+ i j)))))))

General Form:
(declare d1 ... dn)
where, in ACL2, each di is of one of the following forms:

  (ignore v1 ... vn) -- where each vi is a variable introduced in
  the immediately superior lexical environment.

  (type type-spec v1 ... vn) -- where each vi is a variable
  introduced in the immediately superior lexical environment and
  type-spec is a type specifier (as described in the documentation
  for type-spec).

  (xargs :key1 val1 ... :keyn valn) -- where the legal values of the
  keyi's and their respective vali's are described in the
  documentation for xargs.
```

Declarations in ACL2 may occur only where `dcl` occurs below:

```
(DEFUN name args doc-string dcl ... dcl body)
(DEFMACRO name args doc-string dcl ... dcl body)
(LET ((v1 t1) ...) dcl ... dcl body)
(MV-LET (v1 ...) term dcl ... dcl body)
```

Of course, if a form macroexpands into one of these (as `let*^` expands into nested `let^`s and our `er-let*` expands into nested `mv-let^`s) then declarations are permitted as handled by the macros involved.

`Declare` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.95 DENOMINATOR

DENOMINATOR   divisor of a ratio in lowest terms

Basic axiom:

```
(equal (denominator x)
       (if (rationalp x)
           (denominator x)
         1))
```

Guard^ for `(denominator x)`:

```
(rationalp x)
```

10.96 DIGIT-CHAR-P

DIGIT-CHAR-P   the number, if any, corresponding to a given character

`(digit-char-p ch)` is the integer corresponding to the character `ch` in base `10`. For example, `(digit-char-p #\3)` is equal to the integer `3`. More generally, an optional second argument specifies the radix (default `10`, as indicated above).

The guard^ for `digit-char-p` (more precisely, for the function `our-digit-char-p` that calls of this macro expand to) requires its second argument to be an integer between 2 and 36, inclusive, and its first argument to be a character.

`Digit-char-p` is a Common Lisp function, though it is implemented in the ACL2 logic as an ACL2 macro. See any Common Lisp documentation for more information.

10.97 DIGIT-TO-CHAR

DIGIT-TO-CHAR map a digit to a character

```
Example:
ACL2 !>(digit-to-char 8)
#\8
```

For an integer `n` from 0 to 9, `(digit-to-char n)` is the character corresponding to `n` in decimal notation.

The guard^ for `digit-to-char` requires its argument to be an integer between 0 and 9, inclusive.

10.98 EIGHTH

EIGHTH eighth member of the list

See any Common Lisp documentation for details.

10.99 ENDP

ENDP recognizer for empty lists

In the ACL2 logic, `(endp x)` is the same as `(atom x)`. See Section 10.33 [ATOM], page 411.

Unlike `atom^`, the guard^ for `endp` requires that `x` is a `cons^` pair or is `nil`. Thus, `endp` is typically used as a termination test for functions that recur on a `true-listp^` argument. See Section 8.39 [GUARD], page 285 for general information about guard^s.

`Endp` is a Common Lisp function. See any Common Lisp documentation for more information.

10.100 EQ

EQ equality of symbols

`Eq` is the function for determining whether two objects are identical (i.e., have the exact same store address in the current von Neumann implementation of Common Lisp). It is the same as `equal^` in the ACL2 logic.

`Eq` is a Common Lisp function. In order to ensure conformance with Common Lisp, the ACL2 guard^ on `eq` requires at least one of the arguments to `eq` to be a symbol. Common Lisp guarantees that if `x` is a symbol, then `x` is `eq` to `y` if and only if `x` is `equal^` to `y`. Thus, the ACL2 user should think of `eq` as nothing besides a fast means for checking `equal^` when one argument is known to be a symbol. In particular, it is possible that an `eq` test will not even require the cost of a function call but will be as fast as a single machine instruction.

10.101 EQL

EQL test equality (of two numbers, symbols, or characters^)

`(eql x y)` is logically equivalent to `(equal x y)`.

Unlike `equal^`, `eql` has a guard^ requiring at least one of its arguments to be a number, a symbol, or a character. Generally, `eql` is executed more efficiently than `equal^`.

For a discussion of the various ways to test against 0, See Section 10.258 [ZERO-TEST-IDIOMS], page 508.

`Eql` is a Common Lisp function. See any Common Lisp documentation for more information.

10.102 EQLABLE-ALISTP

EQLABLE-ALISTP recognizer for a true list of pairs whose `car^`s are suitable for `eql^`

The predicate `eqlable-alistp` tests whether its argument is a `true-listp^` of `consp^` objects whose `car^`s all satisfy `eqlablep^`.

10.103 EQLABLE-LISTP

EQLABLE-LISTP recognizer for a true list of objects each suitable for `eql^`

The predicate `eqlable-listp` tests whether its argument is a `true-listp^` of objects satisfying `eqlablep^`.

10.104 EQLABLEP

EQLABLEP the guard^ for the function `eql^`

The predicate `eqlablep` tests whether its argument is suitable for `eql^`, at least one of whose arguments must satisfy this predicate in Common Lisp. `(Eqlablep x)` is true if and only if its argument is a number, a symbol, or a character.

10.105 EQUAL

EQUAL true equality

`(equal x y)` is equal to `t` or `nil`, according to whether or not `x` and `y` are the same value.

For a discussion of the various idioms for testing against 0, See Section 10.258 [ZERO-TEST-IDIOMS], page 508.

10.106 ER-PROGN

ER-PROGN perform a sequence of state-changing “error triples”

```
Example:
(er-progn (check-good-foo-p (f-get-global 'my-foo state) state)
          (value (* (f-get-global 'my-foo state)
                    (f-get-global 'bar state))))
```

This sequencing primitive is only useful when programming with state^, something that very few users will probably want to do. See Section 8.81 [STATE], page 344.

`Er-progn` is used much the way that `progn^` is used in Common Lisp, except that it expects each form within it to evaluate to an "error triple" of the form `(mv erp val state)`. The first such form, if any, that evaluates to such a triple where `erp` is not `nil` yields the error triple returned by the `er-progn`. If there is no such form, then the last form returns the value of the `er-progn` form.

We intend to write more about this topic, especially if there are requests to do so.

10.107 EVENP

EVENP test whether an integer is even

`(evenp x)` is true if and only if the integer `x` is even. Actually, in the ACL2 logic `(evenp x)` is defined to be true when `x/2` is an integer.

The guard^ for `evenp` requires its argument to be an integer.

`Evenp` is a Common Lisp function. See any Common Lisp documentation for more information.

10.108 EXPLODE-NONNEGATIVE-INTEGER

EXPLODE-NONNEGATIVE-INTEGER the list of characters^ in the decimal form of a number

```
Example:
ACL2 !>(explode-nonnegative-integer 325 nil)
(#3 #2 #5)
```

For a non-negative integer `n`, `(explode-nonnegative-integer n nil)` is the list of characters^ in the decimal representation of `n`.

The guard^ for `explode-nonnegative-integer` requires the first argument to be a nonnegative integer.

10.109 EXPT

EXPT exponential function

`(Expt r i)` is the result of raising the number `r` to the integer power `i`.

The guard^ for `(expt r i)` is that `r` is a number and `i` is an integer, and furthermore, if `r` is `0` then `i` is nonnegative. When the type requirements of the guard^ aren't met, `(expt r i)` first coerces `r` to a number and `i` to an integer.

`Expt` is a Common Lisp function. See any Common Lisp documentation for more information. Note that `r` can be a complex number; this is consistent with Common lisp.

10.110 FIFTH

FIFTH fifth member of the list

See any Common Lisp documentation for details.

10.111 FIRST

FIRST first member of the list

See any Common Lisp documentation for details.

10.112 FIX

FIX coerce to a number

`Fix` simply returns any numeric argument unchanged, returning `0` on a non-numeric argument. Also see Section 10.173 [NFIX], page 475, see Section 10.122 [IFIX], page 451, and see Section 10.208 [RFIX], page 490 for analogous functions that coerce to a natural number, an integer, and a rational number, respectively.

`Fix` has a guard^ of `t`.

10.113 FIX-TRUE-LIST

FIX-TRUE-LIST coerce to a true list

`Fix-true-list` is the identity function on `true-listp^` objects. It converts every list to a true list by dropping the final `cdr^`, and it converts every atom^ to `nil`.

10.114 FLOOR

FLOOR division returning an integer by truncating toward negative infinity

```
Example Forms:
ACL2 !>(floor 14 3)
4
ACL2 !>(floor -14 3)
-5
ACL2 !>(floor 14 -3)
-5
ACL2 !>(floor -14 -3)
4
ACL2 !>(floor -15 -3)
5
```

`(Floor i j)` returns the result of taking the quotient of `i` and `j` and returning the greatest integer not exceeding that quotient. For example, the quotient of `-14` by `3` is `-4 2/3`, and the largest integer not exceeding that rational number is `-5`.

The guard^ for `(floor i j)` requires that `i` and `j` are rational numbers and `j` is non-zero.

`Floor` is a Common Lisp function. See any Common Lisp documentation for more information.

10.115 FMS

FMS `:(str alist co-channel state evisc) => state`

See Section 10.116 [FMT], page 440 for further explanation.

```
The tilde-directives are
```

```
~px  pretty print vx (maybe after printing a newline)
~qx  pretty print vx starting in current column; end with newline
~Pxy like ~px but use vy as the evisceration tuple
~Qxy like ~qx but use vy as the evisceration tuple
~@x  if vx is a string, "str",  recursively format "str"
     if vx is ("str" . a), recursively format "str" under a+
~#x~[...~/...~/ ... ~/...~] cases on vx
     ^    ^     ...   ^  if 0<=vx<=k, choose vxth alternative
     0    1     ...   k  if vx is a list of length 1, case 0; else 1
~*x  iterator: vx must be of the form
     ("str0" "str1" "str2" "str3" lst . a);
     if lst is initially empty, format "str0" under a+; otherwise,
     bind #\* successively to the elements of lst and then
     recursively format "stri" under a+, where i=1 if there is one
     element left to process, i=2 if there are two left, and i=3
     otherwise.
~&x  print elements of vx with ~p, separated by commas and a
     final ``and''
~vx  print elements of vx with ~p, separated by commas and a
     final ``or''
~nx  if vx is a small positive integer, print it as a word, e.g.,
     seven;
     if vx is a singleton containing a small positive integer, print
       the corresponding ordinal as a word, e.g., seventh
~Nx  like ~nx but the word is capitalized, e.g., Seven or Seventh
~tx  tab out to column vx; newline first if at or past column vx
~cx  vx is (n . w), print integer n right justified in field of
     width w
~fx  print object vx flat over as many lines as necessary
~Fx  same as ~f, except that subsequent lines are indented to
     start one character to the right of the first character printed
~sx  if vx is a symbol, print vx, breaking on hyphens; if vx is a
     string, print the characters in it, breaking on hyphens
~    tilde space: print a space
~_x  print vx spaces
~
     tilde newline: skip following whitespace
~%   output a newline
~|   output a newline unless already on left margin
~~   print a tilde
~-   if close to rightmargin, output a hyphen and newline; else
     skip this char
```

10.116 FMT

FMT formatted printing

ACL2 provides the functions `fmt`, `fmt1^`, and `fms^` as substitutes for Common Lisp's `format` function.

All three print a given string under an alist pairing character objects with values, interpreting certain "tilde-directives" in the string. `Channel` must be a character output channel (e.g., `*standard-co*^`).

```
General Forms:                                              result
(fms string alist channel state evisc-tuple)         ; state
(fmt string alist channel state evisc-tuple)         ; (mv col state)
(fmt1 string alist column channel state evisc-tuple) ; (mv col state)
```

`Fms^` and `fmt` print an initial newline to put `channel` in column `0`; `Fmt1^` requires the current column as input. Columns are numbered from `0`. The current column is the column into which the next character will be printed. (Thus, the current column number is also the number of characters^ printed since the last newline.) The `col` returned by `fmt` and `fmt1^` is the current column at the conclusion of the formatting. `Evisc-tuple` must be either `nil` (meaning no abbreviations are used when objects are printed) or an "evisceration tuple" such as that returned by `(default-evisc-tuple state)`.

We list the tilde-directives below. The notation is explained after the chart.

```
~px  pretty print vx (maybe after printing a newline)
~qx  pretty print vx starting in current column; end with newline
~Pxy like ~px but use vy as the evisceration tuple
~Qxy like ~qx but use vy as the evisceration tuple
~@x  if vx is a string, "str",  recursively format "str"
     if vx is ("str" . a), recursively format "str" under a+
~#x~[...~/...~/ ... ~/...~] cases on vx
     ^    ^     ...   ^  if 0<=vx<=k, choose vxth alternative
     0    1     ...   k  if vx is a list of length 1, case 0; else 1
~*x  iterator: vx must be of the form
     ("str0" "str1" "str2" "str3" lst . a);
     if lst is initially empty, format "str0" under a+; otherwise,
     bind #\* successively to the elements of lst and then
     recursively format "stri" under a+, where i=1 if there is one
     element left to process, i=2 if there are two left, and i=3
     otherwise.
~&x  print elements of vx with ~p, separated by commas and a
     final ``and''
~vx  print elements of vx with ~p, separated by commas and a
     final ``or''
~nx  if vx is a small positive integer, print it as a word, e.g.,
     seven;
```

```
     if vx is a singleton containing a small positive integer, print
       the corresponding ordinal as a word, e.g., seventh
~Nx  like ~nx but the word is capitalized, e.g., Seven or Seventh
~tx  tab out to column vx; newline first if at or past column vx
~cx  vx is (n . w), print integer n right justified in field of
     width w
~fx  print object vx flat over as many lines as necessary
~Fx  same as ~f, except that subsequent lines are indented to
     start one character to the right of the first character printed
~sx  if vx is a symbol, print vx, breaking on hyphens; if vx is a
     string, print the characters in it, breaking on hyphens
~    tilde space: print a space
~_x  print vx spaces
~
     tilde newline: skip following whitespace
~%   output a newline
~|   output a newline unless already on left margin
~~   print a tilde
~-   if close to rightmargin, output a hyphen and newline; else
     skip this char
```

If `x` is a character, then `vx` is the value of `#\x` under the current alist. When we say "format `str` under `a+`" we mean recursively process the given string under an alist obtained by appending `a` to the current alist.

ACL2's formatting functions print to the indicated channel, keeping track of which column they are in. `Fmt1^` can be used if the caller knows which column the channel is in (i.e., how many characters^ have been printed since the last newline). Otherwise, `fmt` or `fms^` must be used, both of which output a newline so as to establish the column position at `0`. Unlike Common Lisp's `format` routine, `fmt` and its relatives break the output into lines so as to try to avoid printing past column 77. That number is built-into the definitions of ACL2's formatting functions. Line breaks are automatically inserted as necessary in place of spaces and after hyphens in the text being printed.

The formatting functions scan the string from left to right, printing each successive character unless it is a tilde `(~)`. Upon encountering tildes the formatters take action determined by the character or characters^ immediately following the tilde. The typical tilde-directive is a group of three successive characters^ from the string being printed. For example, `~p0` is a 3 character `tilde-directive`. The first character in a tilde-directive is always the tilde character itself. The next character is called the "command" character. The character after that is usually taken as the name of a "format variable" that is bound in the alist under which the string is being printed. Format variables are, by necessity, characters^. The objects actually printed by a tilde-directive are the objects obtained by looking up the command's format variables in the alist. Typical format variable names are `0`, `1`, 2, ..., 9, a, b, c, etc., and if a tilde-directive uses the format variable 0, as in ~p0, then the character #\0 must be bound in the alist. Some tilde commands take no arguments and others take more than one, so some directives are of length two and others are longer.

It should be noted that this use of characters^ in the string to denote arguments is another break from Common Lisp's format routine. In Common Lisp, the directives refer implicitly to the "next item to be printed." But in ACL2 the directives name each item explicitly with our format variables.

The following text contains examples that can be evaluated. To make this process easier, we use a macro which is defined as part of ACL2 just for this documentation^. The macro is named fmx and it takes up to eleven arguments, the first of which is a format string, str, and the others of which are taken as the values of format variables. The variables used are #\0 through #\9. The macro constructs an appropriate alist, a, and then evaluates (fmt str a *standard-co* state nil).

Thus,

```
(fmx "Here is v0, ~p0, and here is v1, ~p1."
     (cons 'value 0)
     (cons 'value 1))
```

is just an abbreviation for

```
(fmt "Here is v0, ~p0, and here is v1, ~p1."
     (list (cons #\0 (cons 'value 0))
           (cons #\1 (cons 'value 1)))
     *standard-co*
     state
     nil)
```

which returns (mv 53 state) after printing the line

```
Here is v0, (VALUE . 0), and here is v1, (VALUE . 1).
```

We now devote special attention to three of the tilde-directives whose use is non-obvious.

The Case Statement

~#x is essentially a "case statement" in the language of fmt. The proper form of the statement is

```
~#x~[case-0~/case-1~/ ... ~/case-k~],
```

where each of the `case-i` is a format string. In the most common use, the variable `x` has an integer value, `vx`, between `0` and `k`, inclusive. The effect of formatting the directive is to format `case-vx`.

For example

```
(fmx "Go ~#0~[North~/East~/South~/West~].~%" 1)
```

will print "Go East." followed by a newline and will return

`(mv 0 state)`, while if you change the `1` above to `3` (the maximum legal value), it will print "Go West."

In order to make it easier to print such phrases as "there are seven cases" requiring agreement between subject and verb based on the number of elements of a list, the case statement allows its variable to take a list as its value and selects `case-0` if the list has length `1` and `case-1` otherwise.

```
(let ((cases '(a b c)))
  (fmx "There ~#0~[is ~n1 case~/are ~n1 cases~]."
       cases
       (length cases)))
```

will print "There are three cases." but if you change the

`'(a b c)` above simply to `'(a)` it will print "There is one case." and if you change it to `nil` it will print "There are zero cases."

Indirection

Roughly speaking, `~@` will act as though the value of its argument is a format string and splice it into the current string at the current position. It is often used when the phrase to be printed must be computed. For example,

```
(let ((ev 'DEFUN))
 (fmx "~p0 is an event~@1."
      'foo
      (if (member-eq ev '(defun defstub encapsulate))
          " that may introduce a function symbol"
          "")))
```

will print "`foo` is an event that may introduce a function symbol," but if the value of `ev` is changed from `'defun^` to `'defthm^`, it prints "`foo` is an event." The `~@` directive "splices" in the computed phrase (which might be empty). Of course, this particular example could be done with the case statement

```
~#1~[~/ that may introduce a function symbol~]
```

where the value of `#\1` is appropriately computed to be `0` or `1`.

If the argument to `~@` is a pair, it is taken to be a format string `cons^`ed onto an alist, i.e., `("str" . a)`, and the alist, `a`, is used to extend the current one before `"str"` is recursively processed. This feature of `fmt` can be used to pass around "phrases" that contain computed contextual information in `a`. The most typical use is as "error messages." For example, suppose you are writing a function which does not have access to `state^` and so cannot print an error message. It may nevertheless be necessary for it to signal an error to its caller, say by returning two results, the first of which is interpreted as an error message if non-`nil`. Our convention is to use a `~@` pair to represent such messages. For example, the error value might be produced by the code:

```
(cons
  "Error:  The instruction ~p0 is illegal when the stack is ~p1.~%"
  (list (cons #\0 (current-instruction st))
        (cons #\1 (i-stack st))))
```

If the `current-instruction` and `i-stack` (whatever they are) are `'(popi 3)` and `'(a b)` when the `cons^` above is evaluated, then it produces

```
'("Error:  The instruction ~p0 is illegal when the stack is ~p1.~%"
  (#\0 POPI 3)
  (#\1 A B))
```

and if this pair is made the value of the `fmt` variable `0`, then `~@0` will print

```
   Error:  The instruction (POPI 3) is illegal when the stack is (A B).
```

For example, evaluate

```
(let
 ((pair
  '("Error:  The instruction ~p0 is illegal when the stack is ~p1.~%"
    (#\0 POPI 3)
    (#\1 A B))))
```

```
 (fmx "~@0" pair)).
```

Thus, even though the function that produced the "error" could not print it, it could specify exactly what error message and data are to be printed.

This example raises another issue. Sometimes it is desirable to break lines in your format strings so as to make your source code more attractive. That is the purpose of the `tilde-newline` directive. The following code produces exactly the same output as described above.

```
(let ((pair '("Error:  The instruction ~p0 ~
              is illegal when the stack is ~
              ~p1.~%"
              (#\0 POPI 3)
              (#\1 A B))))
 (fmx "~@0" pair)).
```

Finally, observe that when `~@0` extends the current alist, `alist`, with the one, `a`, in its argument, the bindings from `a` are added to the front of `alist`, overriding the current values of any shared variables. This insures that the variable values seen by the recursively processed string, `"str"`, are those from `a`, but if `"str"` uses variables not bound in `a`, their values are as specified in the original alist. Intuitively, variables bound in `a` are local to the processing of `("str" . a)` but `"str"` may use "global variables." The example above illustrates this because when the `~@0` is processed, `#\0` is bound to the error message pair. But when the `~p0` in the error string is processed, `#\0` is bound to the illegal instruction.

Iteration

The `~*` directive is used to process each element of a list. For example,

```
(let ((lst '(a b c d e f g h))) ; a true-list whose elements we exhibit
 (fmx "~*0"
      `("Whoa!"          ; what to print if there's nothing to print
        "~p*!"           ; how to print the last element
        "~p* and "       ; how to print the 2nd to last element
        "~p*, "          ; how to print all other elements
        ,lst)))          ; the list of elements to print
```

will print "`A, B, C, D, E, F, G and H!`". Try this example with other true list values of `lst`, such as `'(a b)`, `'(a)`, and `nil`. The tilde-directives `~&0` and `~v0`, which take a true list argument and display its elements separated by commas and a final "and" or "or," are implemented in terms of the more general `~*`.

The `~*` directive allows the 5-tuple to specify in its final `cdr^` an alist with which to extend the current one before processing the individual elements.

We often use `~*` to print a series of phrases, separated by suitable punctuation, whitespace and noise words. In such use, the `~*` handles the separation of the phrases and each phrase is generally printed by `~@`.

Here is a complex example. In the `let*^`, below, we bind phrases to a list of `~@` pairs and then we create a `~*` 5-tuple to print out the conjunction of the phrases with a parenthetical "finally!" if the series is longer than 3.

```
(let* ((phrases
        (list (list "simplifying with the replacement rules ~&0"
                    (cons #\0 '(rewrite-rule1
                                rewrite-rule2
                                rewrite-rule3)))
              (list "destructor elimination using ~p0"
                    (cons #\0 'elim-rule))
              (list "generalizing the terms ~&0"
                    (cons #\0 '((rev x) (app u v))))
              (list "inducting on ~p0"
                    (cons #\0 'I))))
       (5-tuple
        (list
         "magic"                              ; no phrases
         "~@*"                                ; last phrase
         "~@*, and~#f~[~/ (finally!)~] "      ; second to last phrase
         "~@*, "                              ; other phrases
         phrases                              ; the phrases themselves
         (cons #\f
               (if (>(length phrases) 3) 1 0))))) ;print ''finally''?
  (fmx "We did it by ~*0." 5-tuple))
```

This `let*^` prints

```
We did it by simplifying with the replacement rules REWRITE-RULE1,
REWRITE-RULE2 and REWRITE-RULE3, destructor elimination using ELIM-
RULE, generalizing the terms (REV X) and (APP U V), and (finally!)
inducting on I.
```

You might wish to try evaluating the `let*^` after removing elements of phrases.

Most of the output produced by ACL2 is produced via `fmt` statements. Thus, inspection of the source code will yield many examples. A complicated example is the code that explains the simplifier's work. See `:pc^ simplify-clause-msg1`. An ad hoc example is provided by the function `fmt-doc-example`, which takes two arguments: an arbitrary true list and `state^`. To see how `fmt-doc-example` works, `:pe^ fmt-doc-example`.

```
(fmt-doc-example '(a b c d e f g h i j k l m n o p) state)
```

will produce the output

```
Here is a true list:  (A B C D E F G H I J K L M N O P).  It has 16
elements, the third of which is C.

We could print each element in square brackets:
([A], [B], [C], [D], [E], [F], [G], [H], [I], [J], [K], [L], [M], [N],
[almost there: O], [the end: P]).  And if we wished to itemize them
into column 15 we could do it like this
0123456789012345
    0 (zeroth) A
    1 (first)  B
    2 (second) C
    3 (third)  D
    4 (fourth) E
    5 (fifth)  F
    6 (sixth)  G
    7 (seventh)
               H
    8 (eighth) I
    9 (ninth)  J
   10 (tenth)  K
   11 (eleventh)
               L
   12 (twelfth)
               M
   13 (thirteenth)
               N
   14 (14th)   O
   15 (15th)   P
End of example.
```

and return `(mv 15 state)`.

Finally, we should remind the reader that `fmt` and its subfunctions, most importantly `fmt0`, are written entirely in ACL2. We make this comment for two reasons. First, it illustrates the fact that quite low level code can be efficiently written in the language. Second, it means that as a last resort for documentation purposes you can read the source code without changing languages.

10.117 FMT1

FMT1 `:(str alist col co-channel state evisc) => (mv col state)`

See Section 10.116 [FMT], page 440 for further explanation.

```
The tilde-directives are
~px  pretty print vx (maybe after printing a newline)
~qx  pretty print vx starting in current column; end with newline
~Pxy like ~px but use vy as the evisceration tuple
~Qxy like ~qx but use vy as the evisceration tuple
~@x  if vx is a string, "str",  recursively format "str"
     if vx is ("str" . a), recursively format "str" under a+
~#x~[...~/...~/ ... ~/...~] cases on vx
     ^    ^     ...   ^  if 0<=vx<=k, choose vxth alternative
     0    1     ...   k  if vx is a list of length 1, case 0; else 1
~*x  iterator: vx must be of the form
     ("str0" "str1" "str2" "str3" lst . a);
     if lst is initially empty, format "str0" under a+; otherwise,
     bind #\* successively to the elements of lst and then
     recursively format "stri" under a+, where i=1 if there is one
     element left to process, i=2 if there are two left, and i=3
     otherwise.
~&x  print elements of vx with ~p, separated by commas and a
     final ``and''
~vx  print elements of vx with ~p, separated by commas and a
     final ``or''
~nx  if vx is a small positive integer, print it as a word, e.g.,
     seven;
     if vx is a singleton containing a small positive integer, print
       the corresponding ordinal as a word, e.g., seventh
~Nx  like ~nx but the word is capitalized, e.g., Seven or Seventh
~tx  tab out to column vx; newline first if at or past column vx
~cx  vx is (n . w), print integer n right justified in field of
     width w
~fx  print object vx flat over as many lines as necessary
~Fx  same as ~f, except that subsequent lines are indented to
     start one character to the right of the first character printed
~sx  if vx is a symbol, print vx, breaking on hyphens; if vx is a
     string, print the characters in it, breaking on hyphens
~    tilde space: print a space
~_x  print vx spaces
```

```
 ~
     tilde newline: skip following whitespace
 ~%  output a newline
 ~|  output a newline unless already on left margin
 ~~  print a tilde
 ~-  if close to rightmargin, output a hyphen and newline; else
     skip this char
```

10.118 FOURTH

FOURTH fourth member of the list

See any Common Lisp documentation for details.

10.119 IDENTITY

IDENTITY the identity function

`(Identity x)` equals `x`; what else can we say?

`Identity` is a Common Lisp function. See any Common Lisp documentation for more information.

10.120 IF

IF if-then-else function

`(if x y z)` is equal to `y` if `x` is any value other than `nil`, and is equal to `z` if `x` is `nil`.

Only one of `x`, `y` is evaluated when `(if x y z)` is evaluated.

`If` has a guard^ of `t`.

`If` is part of Common Lisp. See any Common Lisp documentation for more information.

10.121 IFF

IFF logical "if and only if"

Iff is the ACL2 biconditional, "if and only if". (iff P Q) means that either P and Q are both false (i.e., nil) or both true (i.e., not nil).

10.122 IFIX

IFIX coerce to an integer

Ifix simply returns any integer argument unchanged, returning 0 on a non-integer argument. Also see Section 10.173 [NFIX], page 475, see Section 10.208 [RFIX], page 490, and see Section 10.112 [FIX], page 438 for analogous functions that coerce to a natural number, a rational number, and a number, respectively.

Ifix has a guard^ of t.

10.123 ILLEGAL

ILLEGAL cause a hard error

(illegal ctx str alist) causes evaluation to halt with a short message using the "context" ctx. An error message is printed using the string str and alist alist that are of the same kind as expected by fmt^. See Section 10.116 [FMT], page 440.

10.124 IMAGPART

IMAGPART imaginary part of a complex number

Basic axiom:

```
(equal (imagpart x)
       (if (acl2-numberp x)
           (imagpart x)
```

```
            0))
```

Guard^ for `(imagpart x)`:

```
(acl2-numberp x)
```

10.125 IMPLIES

IMPLIES   logical implication

`Implies` is the ACL2 implication function. `(implies P Q)` means that either `P` is false (i.e., `nil`) or `Q` is true (i.e., not `nil`).

10.126 IMPROPER-CONSP

IMPROPER-CONSP   recognizer for improper (non-null-terminated) non-empty lists

`Improper-consp` is the function that checks whether its argument is a non-empty list that ends in other than `nil`. See Section 10.193 [PROPER-CONSP], page 482 and also see Section 10.249 [TRUE-LISTP], page 503.

10.127 INT=

INT=   test equality of two integers

`(int= x y)` is logically equivalent to `(equal x y)`.

Unlike `equal^`, `int=` requires its arguments to be numbers (or else causes a guard^ violation; see Section 8.39 [GUARD], page 285). Generally, `int=` is executed more efficiently than `equal^` or `=^` on integers.

10.128 INTEGER-LENGTH

INTEGER-LENGTH number of bits in two's complement integer representation

For non-negative integers, `(integer-length i)` is the minimum number of bits needed to represent the integer. Any integer can be represented as a signed two's complement field with a minimum of `(+ (integer-length i) 1)` bits.

The guard^ for `integer-length` requires its argument to be an integer. `Integer-length` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.129 INTEGER-LISTP

INTEGER-LISTP recognizer for a true list of integers

The predicate `integer-listp` tests whether its argument is a true list of integers.

10.130 INTEGERP

INTEGERP recognizer for whole numbers

`(integerp x)` is true if and only if `x` is an integer.

10.131 INTERN

INTERN create a new symbol in a given package

`(intern symbol-name symbol-package-name)` returns a symbol with the given `symbol-name^` and the given `symbol-package-name^`. We restrict Common Lisp's `intern` so that the second argument is either the symbol *main-lisp-package-name*, the value of that constant, or is one of "ACL2", "ACL2-INPUT-CHANNEL", "ACL2-OUTPUT-CHANNEL", or "KEYWORD".

In ACL2 `intern` is actually implemented as a macro that expands to a call of a similar function whose second argument is a symbol. Invoke `:pe intern` to see the definition, or see Section 10.132 [INTERN-IN-PACKAGE-OF-SYMBOL], page 454.

To see why is `intern` so restricted consider `(intern "X" "P")`. In particular, is it a symbol and if so, what is its `symbol-package-name^`? One is tempted to say "yes, it is a symbol in the package `"P"`." But if package `"P"` has not yet been defined, that would be premature because the imports to the package are unknown. For example, if `"P"` were introduced with

```
(defpkg "P" '(LISP::X))
```

then in Common Lisp `(symbol-package-name (intern "X" "P"))` returns `"LISP"`.

The obvious restriction on `intern` is that its second argument be the name of a package known to ACL2. We cannot express such a restriction (except, for example, by limiting it to those packages known at some fixed time, as we do). Instead, we provide `intern-in-package-of-symbol^` which requires a "witness symbol" for the package instead of the package. The witness symbol is any symbol (expressible in ACL2) and uniquely specifies a package necessarily known to ACL2.

10.132 INTERN-IN-PACKAGE-OF-SYMBOL

INTERN-IN-PACKAGE-OF-SYMBOL create a symbol with a given name

Basic axiom:

```
(equal (intern-in-package-of-symbol x y)
       (if (and (stringp x)
                (symbolp y))
           (intern-in-package-of-symbol x y)
         nil))
```

Guard^ for `(intern-in-package-of-symbol x y)`:

```
(and (stringp x) (symbolp y))
```

Intuitively, `(intern-in-package-of-symbol x y)` creates a symbol with `symbol-name^ x` intern^ed in the package containing `y`. More precisely, suppose `x` is a string, `y` is a symbol with `symbol-package-name^` pkg and that the `defpkg^` event creating pkg had the list of symbols imports as the value of its second argument. Then `(intern-in-package-of-symbol x y)` returns a symbol, ans, the `symbol-name^` of ans is `x`, and the `symbol-package-name^` of ans is pkg, unless `x` is the `symbol-name^` of some member of imports with `symbol-package-name^` ipkg, in which case the `symbol-package-name^` of ans is ipkg. Because `defpkg^` requires that there be no duplications among the `symbol-name^`s of the imports, `intern-in-package-of-symbol` is uniquely defined.

For example, suppose "MY-PKG" was created by

```
(defpkg "MY-PKG" '(ACL2::ABC LISP::CAR)).
```

Let w be 'my-pkg::witness. Observe that

```
(symbolp w) is t                       ; w is a symbol
(symbol-name w) is "WITNESS"           ; w's name is "WITNESS"
(symbol-package-name w) is "MY-PKG"    ; w is in the package "MY-PKG"
```

The construction of w illustrates one way to obtain a symbol in a given package: write it down as a constant using the double-colon notation.

But another way to obtain a symbol in a given package is to create it with intern-in-package-of-symbol.

```
(intern-in-package-of-symbol "XYZ" w) is MY-PKG::XYZ

(intern-in-package-of-symbol "ABC" w) is ACL2::ABC

(intern-in-package-of-symbol "CAR" w) is LISP::CAR

(intern-in-package-of-symbol "car" w) is MY-PKG::|car|
```

10.133 INTERSECTP-EQ

INTERSECTP-EQ test whether two lists of symbols intersect

See Section 10.134 [INTERSECTP-EQUAL], page 455, which is logically the same function.

(Intersectp-eq x y) has a guard^ that x and y are lists of symbols.

10.134 INTERSECTP-EQUAL

INTERSECTP-EQUAL test whether two lists intersect

`(Intersectp-equal x y)` returns `t` if `x` and `y` have a member in common, else it returns `nil`. Also see Section 10.133 [INTERSECTP-EQ], page 455, which is logically the same but can be more efficient since it uses `eq^` instead of `equal^` to look for members common to the two given lists.

`(Intersectp-equal x y)` has a guard^ that `x` and `y` are true lists.

10.135 IO

IO input/output facilities in ACL2

```
Example:
(mv-let
  (channel state)
  (open-input-channel "foo.lisp" :object state)
  (mv-let (eofp obj state)
          (read-object channel state)
          (.
            .
             (let ((state (close-input-channel channel state)))
                   (mv final-ans state))..)))
```

Also see Section 1.5.6.1 [FILE-READING-EXAMPLE], page 58.

ACL2 supports input and output facilities equivalent to a subset of those found in Common Lisp. ACL2 does not support random access files or bidirectional streams. In Common Lisp, input and output are to or from objects of type `stream`. In ACL2, input and output are to or from objects called "channels," which are actually symbols. Although a channel is a symbol, one may think of it intuitively as corresponding to a Common Lisp stream. Channels are in one of two ACL2 packages, `"ACL2-INPUT-CHANNEL"` and `"ACL2-OUTPUT-CHANNEL"`. When one "opens" a file one gets back a channel whose `symbol-name^` is the file name passed to "open," postfixed with `-n`, where `n` is a counter that is incremented every time an open or close occurs.

There are three channels which are open from the beginning and which cannot be closed:

```
acl2-input-channel::standard-character-input-0
acl2-input-channel::standard-object-input-0
acl2-input-channel::standard-character-output-0
```

All three of these are really Common Lisp's `*standard-input*` or `*standard-output*`, appropriately.

For convenience, three global variables are bound to these rather tedious channel names:

```
*standard-ci*
*standard-oi*
*standard-co*
```

Common Lisp permits one to open a stream for several different kinds of `io`, e.g. character or byte. ACL2 permits an additional type called "object". In ACL2 an "io-type" is a keyword, either `:character`, `:byte`, or `:object`. When one opens a file, one specifies a type, which determines the kind of io operations that can be done on the channel returned. The types `:character` and `:byte` are familiar. Type `:object` is an abstraction not found in Common Lisp. An `:object` file is a file of Lisp objects. One uses `read-object` to read from `:object` files and `print-object$` to print to `:object` files. (The reading and printing are really done with the Common Lisp `read` and `print` functions.)

File-names are strings. ACL2 does not support the Common Lisp type `pathname^`.

Here are the signature^s of the ACL2 io functions. See Section 8.78 [SIGNATURE], page 340.

```
Input Functions:
  (open-input-channel (file-name io-type state) (mv channel state))
  (open-input-channel-p (channel io-type state) boolean)
  (close-input-channel (channel state) state)
  (read-char$ (channel state) (mv char/nil state))
  (peek-char$ (channel state) boolean)
  (read-byte$ (channel state) (mv byte/nil state))
  (read-object (channel state) (mv eof-read-flg obj-read state))

Output Functions:
  (open-output-channel (file-name io-type state) (mv channel state))
  (open-output-channel-p (channel io-type state) boolean)
  (close-output-channel (channel state) state)
  (princ$ (obj channel state) state)
  (write-byte$ (byte channel state) state)
  (print-object$ (obj channel state) state)
  (fms (string alist channel state evisc-tuple) state)
  (fmt (string alist channel state evisc-tuple) (mv col state))
  (fmt1 (string alist col channel state evisc-tuple) (mv col state))
```

The "formatting" functions are particularly useful; see Section 10.116 [FMT], page 440.

When one enters ACL2 with `(lp)`, input and output are taken from `*standard-oi*^` to `*standard-co*^`. Because these are synonyms for `*standard-input*` and `*standard-output*`, one can drive

ACL2 io off of arbitrary Common Lisp streams, bound to `*standard-input*` and `*standard-output*` before entry to ACL2.

10.136 IRRELEVANT-FORMALS

IRRELEVANT-FORMALS formals that are used but only insignificantly

Let `fn` be a function of `n` arguments. Let `x` be the `i`th formal of `fn`. We say `x` is "irrelevant in `fn`" if `x` is not involved in either the guard^ or the measure for `fn`, `x` is used in the body, but the value of `(fn a1...ai...an)` is independent of `ai`.

The easiest way to define a function with an irrelevant formal is simply not to use the formal in the guard^, measure, or body of the function. Such formals are said to be "ignored" by Common Lisp and a special declaration is provided to allow ignored formals. ACL2 makes a distinction between ignored and irrelevant formals.

An example of an irrelevant formal is `x` in the definition of `fact` below.

```
(defun fact (i x)
  (declare (xargs :guard (and (integerp i) (<= 0 i))))
  (if (zerop i) 0 (* i (fact (1- i) (cons i x))))).
```

Observe that `x` is only used in recursive calls of `fact`; it never "gets out" into the result. ACL2 can detect some irrelevant formals by a closure analysis on how the formals are used. For example, if the `i`th formal is only used in the `i`th argument position of recursive calls, then it is irrelevant. This is how `x` is used above.

It is possible for a formal to appear only in recursive calls but still be relevant. For example, `x` is **not** irrelevant below, even though it only appears in the recursive call.

```
(defun fn (i x) (if (zerop i) 0 (fn x (1- i))))
```

The key observation above is that while `x` only appears in a recursive call, it appears in an argument position, namely `i`'s, that is relevant. (The function above can be admitted with a `:guard`^ requiring both arguments to be nonnegative integers and the `:measure (+ i x)`.)

Establishing that a formal is irrelevant, in the sense defined above, can be an arbitrarily hard problem because it requires theorem proving. For example, is `x` irrelevant below?

```
(defun test (i j k x) (if (cond i j k) x 0))
```

Note that the value of `(test i j k x)` is independent of `x` — thus making `x` irrelevant — precisely if `(cond i j k)` is a theorem. ACL2's syntactic analysis of a definition does not guarantee to notice all irrelevant formals.

We regard the presence of irrelevant formals as an indication that something is wrong with the definition. We cause an error on such definitions and suggest that you recode the definition so as to eliminate the irrelevant formals. If you must have an irrelevant formal, one way to "trick" ACL2 into accepting the definition, without slowing down the execution of your function, is to use the formal in an irrelevant way in the guard^. For example, to admit fact, above, with its irrelevant `x` one might use

```
(defun fact (i x)
  (declare (xargs :guard (and (integerp i) (<= 0 i) (equal x x))))
  (if (zerop i) 0 (* i (fact (1- i) (cons i x)))))
```

For those who really want to turn off this feature, we have provided a way to use the `acl2-defaults-table`^ for this purpose; see Section 6.31 [SET-IRRELEVANT-FORMALS-OK], page 190.

If you need to introduce a function with an irrelevant formal, please explain to the authors why this should be allowed.

10.137 KEYWORD-VALUE-LISTP

KEYWORD-VALUE-LISTP   recognizer for true lists whose even-position elements are keywords

`(keyword-value-listp l)` is true if and only if `l` is a list of even length of the form `(k1 a1 k2 a2 ... kn an)`, where each `ki` is a keyword.

10.138 KEYWORDP

KEYWORDP   recognizer for keywords

`(Keywordp x)` is true if and only if `x` is a keyword, i.e., a symbol in the "KEYWORD" package. Such symbols are typically printed using a colon (:) followed by the `symbol-name`^ of the symbol.

Keywordp has a guard^ of t.

Keywordp is a Common Lisp function. See any Common Lisp documentation for more information.

10.139 LAST

LAST the last cons^ (not element) of a list

(Last l) is the last cons^ of a list. Here are examples.

```
ACL2 !>(last '(a b . c))
(B . C)
ACL2 !>(last '(a b c))
(C)
```

(Last l) has a guard^ of (listp l); thus, l need not be a true-listp^.

Last is a Common Lisp function. See any Common Lisp documentation for more information. Unlike Common Lisp, we do not allow an optional second argument for last.

10.140 LENGTH

LENGTH length of a string or proper list

Length is the function for determining the length of a sequence. In ACL2, the argument is required to be either a true-listp^ or a string.

Length is a Common Lisp function. See any Common Lisp documentation for more information.

10.141 LET

LET binding of lexically scoped (local) variables

```
Example LET Form:
(let ((x (* x x))
```

```
        (y (* 2 x)))
  (list x y))
```

If the form above is executed in an environment in which `x` has the value `-2`, then the result is `'(4 -4)`.

`Let` expressions bind variables so that their "local" values, the values they have when the "body" of the `let` is evaluated, are possibly different than their "global" values, the values they have in the context in which the `let` expression appears. In the `let` expression above, the local variables bound by the `let` are `x` and `y`. They are locally bound to the values delivered by the two forms `(* x x)` and `(* 2 x)`, respectively, that appear in the "bindings" of the `let`. The body of the `let` is `(list x y)`.

Suppose that the `let` expression above occurs in a context in which `x` has the value `-2`. (The global value of `y` is irrelevant to this example.) For example, one might imagine that the `let` form above occurs as the body of some function, `fn`, with the formal parameter `x` and we are evaluating `(fn -2)`.

To evaluate the `let` above in a context in which `x` is `-2`, we first evaluate the two forms specifying the local values of the variables. Thus, `(* x x)` is evaluated and produces `4` (because `x` is `-2`) and `(* 2 x)` is evaluated and produces `-4` (because `x` is `-2`). Then `x` and `y` are bound to these values and the body of the `let` is evaluated. Thus, when the body, `(list x y)` is evaluated, `x` is `4` and `y` is `-4`. Thus, the body produces `'(4 -4)`.

Note that the binding of `y`, which is written after the binding of `x` and which mentions `x`, nevertheless uses the global value of `x`, not the new local value. That is, the local variables of the `let` are bound "in parallel" rather than "sequentially." In contrast, if the

```
Example LET* Form:
(let* ((x (* x x))
       (y (* 2 x)))
 (list x y))
```

is evaluated when the global value of `x` is `-2`, then the result is `'(4 8)`, because the local value of `y` is computed after `x` has been bound to `4`. `Let*^` binds its local variables "sequentially."

```
General LET Forms:
(let ((var1 term1) ... (varn termn)) body)
and
(let ((var1 term1) ... (varn termn))
 (declare ...) ... (declare ...)
```

```
   body)
```

where the vari are distinct variables, the termi are terms involving only variables bound in the environment containing the let, and body is a term involving only the vari plus the variables bound in the environment containing the let. Each vari must be used in body or else declare^d ignored.

A let form is evaluated by first evaluating each of the termi, obtaining for each a vali. Then, each vari is bound to the corresponding vali and body is evaluated.

Actually, let forms are just abbreviations for certain uses of lambda notation. In particular

```
(let ((var1 term1) ... (varn termn)) (declare ...) body)
```

is equivalent to

```
((lambda (var1 ... varn)
   (declare ...)
   body)
 term1 ... termn).
```

Let*^ forms are used when it is desired to bind the vari sequentially, i.e., when the local values of preceding varj are to be used in the computation of the local value for vari.

```
General LET* Forms:
(let* ((var1 term1) ... (varn termn)) body)
```

where the vari are distinct variables, the termi are terms involving only variables bound in the environment containing the let*^ and those varj such that j<i, and body is a term involving only the vari plus the variables bound in the environment containing the let*^. Each vari must be used in either some subsequent termj or in body. Note that let*^ does not permit the inclusion of declare^ forms.

The above let*^ is equivalent to

```
(let ((var1 term1))
 ...
 (let ((varn termn)) body)...)
```

Thus, the `termi` are evaluated successively and after each evaluation the corresponding `vali` is bound to the value of `termi`.

Each `(vari termi)` pair in a `let` or `let*^` form is called a "binding" of `vari` and the `vari` are called the "local variables" of the `let` or `let*^`. The common use of `let` and `let*^` is to save the values of certain expressions (the `termi`) so that they may be referenced several times in the body without suggesting their recomputation.

`Let` is part of Common Lisp. See any Common Lisp documentation for more information.

10.142 LET*

LET* binding of lexically scoped (local) variables

```
Example LET* Form:
(let* ((x (* x x))
       (y (* 2 x)))
 (list x y))
```

If the form above is executed in an environment in which `x` has the value `-2`, then the result is `'(4 8)`. See Section 10.141 [LET], page 460 for a discussion of both `let^` and `let*`, or read on for a briefer discussion.

The difference between `let^` and `let*` is that the former binds its local variables in parallel while the latter binds them sequentially. Thus, in `let*`, the term evaluated to produce the local value of one of the locally bound variables is permitted to reference any locally bound variable occurring earlier in the binding list and the value so obtained is the newly computed local value of that variable. See Section 10.141 [LET], page 460.

`Let*` is a Common Lisp macro. See any Common Lisp documentation for more information.

10.143 LIST

LIST build a list

`List` is the macro for building a list of objects. For example, `(list 5 6 7)` returns a list of length 3 whose elements are `5`, `6`, and `7` respectively. Also see Section 10.144 [LIST*], page 464.

`List` is a Common Lisp function. See any Common Lisp documentation for more information.

10.144 LIST*

LIST* build a list

`List*` is the Common Lisp macro for building a list of objects from given elements and a tail. For example, `(list* 5 6 '(7 8 9))` equals the list `'(5 6 7 8 9)`. Also see Section 10.143 [LIST], page 463.

`List*` is a Common Lisp function. See any Common Lisp documentation for more information.

10.145 LISTP

LISTP recognizer for (not necessarily proper) lists

`(listp x)` is true when `x` is either a `cons^` pair or is `nil`.

`Listp` has no guard^, i.e., its guard^ is `t`.

`Listp` is a Common Lisp function. See any Common Lisp documentation for more information.

10.146 LOGAND

LOGAND bitwise logical 'and' of two integers

When integers are viewed in their two's complement representation, `logand` returns their bitwise logical 'and'.

The guard^ for `logand` requires its arguments to be integers. `Logand` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.147 LOGANDC1

LOGANDC1 bitwise logical 'and' of two ints, complementing the first

When integers are viewed in their two's complement representation, `logandc1` returns the bitwise logical 'and' of the second with the bitwise logical 'not' of the first.

The guard^ for `logandc1` requires its arguments to be integers. `Logandc1` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.148 LOGANDC2

LOGANDC2 bitwise logical 'and' of two ints, complementing the second

When integers are viewed in their two's complement representation, `logandc2` returns the bitwise logical 'and' of the first with the bitwise logical 'not' of the second.

The guard^ for `logandc2` requires its arguments to be integers. `Logandc2` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.149 LOGBITP

LOGBITP the `i`th bit of an integer

For a nonnegative integer `i` and an integer `j`, `(logbitp i j)` is the value of the `i`th bit in the two's complement representation of `j`.

`(Logbitp i j)` has a guard^ that `i` is a nonnegative integer and `j` is an integer.

`Logbitp` is a Common Lisp function. See any Common Lisp documentation for more information.

10.150 LOGCOUNT

LOGCOUNT number of "on" bits in a two's complement number

`(Logcount x)` is the number of "on" bits in the two's complement representation of `x`.

`(Logcount x)` has a guard^ of `(integerp x)`.

`Logcount` is a Common Lisp function. See any Common Lisp documentation for more information.

10.151 LOGEQV

LOGEQV bitwise logical equivalence of two integers

When integers are viewed in their two's complement representation, `logeqv` returns the bitwise logical equivalence of the first with the second.

The guard^ for `logeqv` requires its arguments to be integers. `Logeqv` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.152 LOGIOR

LOGIOR bitwise logical inclusive or of two integers

When integers are viewed in their two's complement representation, `logior` returns their bitwise logical inclusive or.

The guard^ for `logior` requires its arguments to be integers. `Logior` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.153 LOGNAND

LOGNAND bitwise logical 'nand' of two integers

When integers are viewed in their two's complement representation, `lognand` returns their bitwise logical 'nand'.

The guard^ for `lognand` requires its arguments to be integers. `Lognand` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.154 LOGNOR

LOGNOR bitwise logical 'nor' of two integers

When integers are viewed in their two's complement representation, `lognor` returns the bitwise logical 'nor' of the first with the second.

The guard^ for `lognor` requires its arguments to be integers. `Lognor` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.155 LOGNOT

LOGNOT bitwise not of a two's complement number

`(lognot i)` is the two's complement bitwise `'not'` of the integer `i`.

`Lognot` is actually defined by coercing its argument to an integer (see Section 10.122 [IFIX], page 451), negating the result, and then subtracting `1`.

The guard^ for `lognot` requires its argument to be an integer.

`Lognot` is a Common Lisp function. See any Common Lisp documentation for more information.

10.156 LOGORC1

LOGORC1 bitwise logical inclusive or of two ints, complementing the first

When integers are viewed in their two's complement representation, `logorc1` returns the bitwise logical inclusive or of the second with the bitwise logical 'not' of the first.

The guard^ for `logorc1` requires its arguments to be integers. `Logorc1` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.157 LOGORC2

LOGORC2 bitwise logical inclusive or of two ints, complementing the second

When integers are viewed in their two's complement representation, `logorc2` returns the bitwise logical inclusive or of the first with the bitwise logical 'not' of the second.

The guard^ for `logorc2` requires its arguments to be integers. `Logorc2` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.158 LOGTEST

LOGTEST test if two integers share a '1' bit

When integers `x` and `y` are viewed in their two's complement representation, `(logtest x y)` is true if and only if there is some position for which both `x` and `y` have a '1' bit in that position.

The guard^ for `logtest` requires its arguments to be integers. `Logtest` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.159 LOGXOR

LOGXOR bitwise logical exclusive or of two integers

When integers are viewed in their two's complement representation, `logxor` returns the bitwise logical exclusive or of the first with the second.

The guard^ for `logxor` requires its arguments to be integers. `Logxor` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.160 LOWER-CASE-P

LOWER-CASE-P recognizer for lower case characters

`(Lower-case-p x)` is true if and only if `x` is a lower case character, i.e., a member of the list `#A`, `#B`, ..., `#Z`.

The guard^ for `lower-case-p` requires its argument to be a character.

`Lower-case-p` is a Common Lisp function. See any Common Lisp documentation for more information.

10.161 MAKE-CHARACTER-LIST

MAKE-CHARACTER-LIST coerce^ to a list of characters

Non-characters in the given list are coerce^d to the character with code 0.

10.162 MAKE-LIST

MAKE-LIST make a list of a given size

For a nonnegative integer `size`, `(Make-list size)` is a list of elements of length `size`, each of which is initialized to the `:initial-element` (which defaults to `nil`).

`Make-list` is a macro in ACL2, defined in terms of a tail recursive function `make-list-ac` whose guard^ requires `size` to be a nonnegative integer. `Make-list` is a Common Lisp function. See any Common Lisp documentation for more information.

10.163 MAX

MAX the larger of two rational numbers

`(Max x y)` is the larger of the numbers `x` and `y`.

The guard^ for `max` requires its arguments to be rational numbers.

`Max` is a Common Lisp function. See any Common Lisp documentation for more information.

10.164 MEMBER

MEMBER membership predicate, using `eql^` as test

`(Member x l)` equals the longest tail of `l` that begins with `x`, or else `nil` if no such tail exists.

`(Member x l)` is provably the same in the ACL2 logic as `(member-equal x l)`. It has a stronger guard^ than `member-equal^` because uses `eql^` to test for whether `x` is equal to a given member of `l`. Its guard^ requires that `l` is a true list, and moreover, either `(eqlablep x)` or all members of `l` are `eqlablep^`. See Section 10.166 [MEMBER-EQUAL], page 470 and see Section 10.165 [MEMBER-EQ], page 470.

`Member` is a Common Lisp function. See any Common Lisp documentation for more information. Since ACL2 functions cannot take keyword arguments (though macros can), the ACL2 functions `member-equal^` and `member-eq^` are defined to correspond to calls of the Common Lisp function `member` whose keyword argument `:test` is `equal^` or `eq^`, respectively.

10.165 MEMBER-EQ

MEMBER-EQ membership predicate, using `eq^` as test

`(Member-eq x lst)` equals the longest tail of `lst` that begins with `x`, or else `nil` if no such tail exists.

`(Member-eq x lst)` is provably the same in the ACL2 logic as `(member x lst)` and `(member-equal x lst)`, but it has a stronger guard^ because it uses `eq^` for a more efficient test for whether `x` is equal to a given member of `lst`. Its guard^ requires that `lst` is a true list, and moreover, either `x` is a symbol or `lst` is a list of symbols. See Section 10.166 [MEMBER-EQUAL], page 470 and see Section 10.164 [MEMBER], page 470.

10.166 MEMBER-EQUAL

MEMBER-EQUAL membership predicate

`(Member-equal x lst)` equals the longest tail of `lst` that begins with `x`, or else `nil` if no such tail exists.

`(Member-equal x lst)` has a guard^ of `(true-listp lst)`. `Member-equal` has the same functionality as the Common Lisp function `member^`, except that it uses the `equal^` function to test whether `x` is the same as each successive element of `lst`. See Section 10.164 [MEMBER], page 470 and see Section 10.165 [MEMBER-EQ], page 470.

10.167 MIN

MIN the smaller of two rational numbers

`(Min x y)` is the smaller of the numbers `x` and `y`.

The guard^ for `min` requires its arguments to be rational numbers.

`Min` is a Common Lisp function. See any Common Lisp documentation for more information.

10.168 MINUSP

MINUSP test whether a rational number is negative

`(Minusp x)` is true if and only if `x < 0`.

The guard^ of `minusp` requires its argument to be a rational number.

`Minusp` is a Common Lisp function. See any Common Lisp documentation for more information.

10.169 MOD

MOD remainder using `floor^`

```
ACL2 !>(mod 14 3)
2
ACL2 !>(mod -14 3)
1
ACL2 !>(mod 14 -3)
-1
ACL2 !>(mod -14 -3)
```

```
-2
ACL2 !>(mod -15 -3)
0
ACL2 !>
```

`(Mod i j)` is that number `k` that `(* j (floor i j))` added to `k` equals `i`.

The guard^ for `(mod i j)` requires that `i` and `j` are rational numbers and `j` is non-zero.

`Mod` is a Common Lisp function. See any Common Lisp documentation for more information.

10.170 MV

MV returning multiple values

`Mv` is the mechanism provided by ACL2 for returning two or more values. Logically, `(mv x1 x2 ... xn)` is the same as `(list x1 x2 ... xn)`, a list of the indicated values. However, ACL2 avoids the cost of building this list structure, with the cost that `mv` may only be used in a certain style in definitions: if a function ever returns using `mv` (either directly, or by calling another function that returns multiple values), then this function must always return the same number of multiple values.

For more explanation of the multiple value mechanism, see Section 10.171 [MV-LET], page 472.

ACL2 does not support the Common Lisp construct `values`, whose logical meaning seems difficult to characterize. `Mv` is the ACL2 analogue of that construct.

10.171 MV-LET

MV-LET calling multi-valued ACL2 functions

```
Example Form:
(mv-let (x y z)              ; local variables
        (mv 1 2 3)           ; multi-valued expression
        (declare (ignore y)) ; optional declarations
        (cons x z))          ; body
```

The form above binds the three "local variables," x, y, and z, to the three results returned by the multi-valued expression and then evaluates the body. The result is '(1 . 3). The second local, y, is declare^d ignored. The multi-valued expression can be any ACL2 expression that returns k results, where k is the number of local variables listed. Often however it is simply the application of a k-valued function. Mv-let is the standard way to invoke a multi-valued function when the caller must manipulate the vector of results returned.

```
General Form:
(mv-let (var1 ... vark)
        term
        body)
or
(mv-let (var1 ... vark)
        term
        (declare ...) ... (declare ...)
        body)
```

where the vari are distinct variables, term is a term that returns k results and mentions only variables bound in the environment containing the mv-let expression, and body is a term mentioning only the vari and variables bound in the environment containing the mv-let. Each vari must occur in body unless it is declare^d ignored in one of the optional declare^ forms, unless this requirement is turned off; see Section 6.28 [SET-IGNORE-OK], page 188. The value of the mv-let term is the result of evaluating body in an environment in which the vari are bound, in order, to the k results obtained by evaluating term in the environment containing the mv-let.

Here is an extended example that illustrates both the definition of a multi-valued function and the use of mv-let to call it. Consider a simple binary tree whose interior nodes are cons^es and whose leaves are non-cons^es. Suppose we often need to know the number, n, of interior nodes of such a tree; the list, syms, of symbols that occur as leaves; and the list, ints, of integers that occur as leaves. (Observe that there may be leaves that are neither symbols nor integers.) Using a multi-valued function we can collect all three results in one pass.

Here is the first of two definitions of the desired function. This definition is "primitive recursive" in that it has only one argument and that argument is reduced in size on every recursion.

```
(defun count-and-collect (x)

; We return three results, (mv n syms ints) as described above.

  (cond ((atom x)

; X is a leaf.  Thus, there are 0 interior nodes, and depending on
```

```
; whether x is a symbol, an integer, or something else, we return
; the list containing x in as the appropriate result.

          (cond ((symbolp x) (mv 0 (list x) nil))
                ((integerp x)(mv 0 nil      (list x)))
                (t           (mv 0 nil      nil))))
         (t

; X is an interior node.  First we process the car, binding n1, syms1, and
; ints1 to the answers.

            (mv-let (n1 syms1 ints1)
                    (count-and-collect (car x))

; Next we process the cdr, binding n2, syms2, and ints2.

                    (mv-let (n2 syms2 ints2)
                            (count-and-collect (car x))

; Finally, we compute the answer for x from those obtained for its car
; and cdr, remembering to increment the node count by one for x itself.

                            (mv (1+ (+ n1 n2))
                                (append syms1 syms2)
                                (append ints1 ints2)))))))
```

This use of multiple values to "do several things at once" is very common in ACL2. However, the function above is inefficient because it append^s `syms1` to `syms2` and `ints1` to `ints2`, copying the list structures of `syms1` and `ints1` in the process. By adding "accumulators" to the function, we can make the code more efficient.

```
(defun count-and-collect1 (x n syms ints)
  (cond ((atom x)
         (cond ((symbolp x) (mv n (cons x syms) ints))
               ((integerp x) (mv n syms (cons x ints)))
               (t (mv n syms ints))))
        (t (mv-let (n2 syms2 ints2)
                   (count-and-collect1 (cdr x) (1+ n) syms ints)
                   (count-and-collect1 (car x) n2 syms2 ints2)))))
```

We claim that `(count-and-collect x)` returns the same triple of results as `(count-and-collect1 x 0 nil nil)`. The reader is urged to study this claim until convinced that it is true and that the latter method of computing the results is more efficient. One might try proving the theorem

```
(defthm count-and-collect-theorem
```

```
(equal (count-and-collect1 x 0 nil nil) (count-and-collect x))).
```

Hint: the inductive proof requires attacking a more general theorem.

ACL2 does not support the Common Lisp construct `multiple-value-bind`, whose logical meaning seems difficult to characterize. `Mv-let` is the ACL2 analogue of that construct.

10.172 MV-NTH

MV-NTH the mv-nth element (zero-based) of a list

`(Mv-nth n l)` is the `n`th element of `l`, zero-based. If `n` is greater than or equal to the length of `l`, then `mv-nth` returns `nil`.

`(Mv-nth n l)` has a guard^ that `n` is a non-negative integer and `l` is a `true-listp^`.

`Mv-nth` is equivalent to the Common Lisp function `nth^`, but is used by ACL2 to access the nth value returned by a multiply valued expression. For an example of the use of `mv-nth`, try

```
ACL2 !>:trans1 (mv-let (erp val state)
                       (read-object ch state)
                       (value (list erp val)))
```

10.173 NFIX

NFIX coerce to a natural number

`Nfix` simply returns any natural number argument unchanged, returning `0` on an argument that is not a natural number. Also see Section 10.173 [NFIX], page 475, see Section 10.208 [RFIX], page 490, and see Section 10.112 [FIX], page 438 for analogous functions that coerce to an integer, a rational number, and a number, respectively.

`Nfix` has a guard^ of `t`.

10.174 NINTH

NINTH ninth member of the list

See any Common Lisp documentation for details.

10.175 NO-DUPLICATESP

NO-DUPLICATESP check for duplicates in a list

`(no-duplicatesp l)` is true if and only if no member of `l` occurs twice in `l`.

`(no-duplicatesp l)` has a guard^ of `(true-listp l)`. Membership is tested using `member-equal`^, hence using `equal`^ as the test.

10.176 NONNEGATIVE-INTEGER-QUOTIENT

NONNEGATIVE-INTEGER-QUOTIENT natural number division function

```
Example Forms:
(nonnegative-integer-quotient 14 3) ; equals 4
(nonnegative-integer-quotient 15 3) ; equals 5
```

`(nonnegative-integer-quotient i j)` returns the integer quotient of the integers `i` and (non-zero) `j`, i.e., the largest `k` such that `(* j k)` is less than or equal to `i`. Also see Section 10.114 [FLOOR], page 439, see Section 10.72 [CEILING], page 422 and see Section 10.250 [TRUNCATE], page 504, which are derived from this function and apply to rational numbers.

The guard^ of `(nonnegative-integer-quotient i j)` requires that `i` is a nonnegative integer and `j` is an integer.

10.177 NOT

NOT logical negation

`Iff^` is the ACL2 negation function. The negation of `nil` is `t` and the negation of anything else is `nil`.

`Not` is a Common Lisp function. See any Common Lisp documentation for more information.

10.178 NTH

NTH the nth element (zero-based) of a list

`(Nth n l)` is the `n`th element of `l`, zero-based. If `n` is greater than or equal to the length of `l`, then `nth` returns `nil`.

`(Nth n l)` has a guard^ that `n` is a non-negative integer and `l` is a `true-listp^`.

`Nth` is a Common Lisp function. See any Common Lisp documentation for more information.

10.179 NTHCDR

NTHCDR final segment of a list

`(Nthcdr n l)` removes the first `n` elements from the list `l`.

The following is a theorem.

```
(implies (and (integerp n)
              (<= 0 n)
              (true-listp l))
         (equal (length (nthcdr n l))
                (if (<= n (length l))
                    (- (length l) n)
                  0)))
```

For related functions, see Section 10.244 [TAKE], page 502 and see Section 10.39 [BUTLAST], page 413.

The guard^ of `(nthcdr n l)` requires that `n` is a nonnegative integer and `l` is a true list.

`Nthcdr` is a Common Lisp function. See any Common Lisp documentation for more information.

10.180 NULL

NULL recognizer for the empty list

`Null` is the function that checks whether its argument is `nil`. For recursive definitions it is often preferable to test for the end of a list using `endp^` instead of `null`; see Section 10.99 [ENDP], page 434.

`Null` is a Common Lisp function. See any Common Lisp documentation for more information.

10.181 NUMERATOR

NUMERATOR dividend of a ratio in lowest terms

Basic axiom:

```
(equal (numerator x)
       (if (rationalp x)
           (numerator x)
         0))
```

Guard^ for `(numerator x)`:

```
(rationalp x)
```

10.182 ODDP

ODDP test whether an integer is odd

`(oddp x)` is true if and only if `x` is odd, i.e., not even in the sense of `evenp^`.

The guard^ for `oddp` requires its argument to be an integer.

`Oddp` is a Common Lisp function. See any Common Lisp documentation for more information.

10.183 OR

OR conjunction

`Or` is the macro for disjunctions. `Or` takes any number of arguments and returns the first that is non-`nil`, or `nil` if there is no non-`nil` element.

`Or` is a Common Lisp macro. See any Common Lisp documentation for more information.

10.184 PAIRLIS

PAIRLIS see `pairlis$^`

The Common Lisp language allows its `pairlis` function to construct an alist in any order! So we have to define our own version: See Section 10.185 [PAIRLIS$], page 479.

10.185 PAIRLIS$

PAIRLIS$ zipper together two lists

The Common Lisp language allows its `pairlis^` function to construct an alist in any order! So we have to define our own version, `pairlis$`. It returns the list of pairs obtained by `cons^`ing together successive respective members of the given lists.

The guard^ for `pairlis$` requires that its arguments are true lists.

10.186 PLUSP

PLUSP test whether a rational number is positive

`(Plusp x)` is true if and only if `x > 0`.

The guard^ of `plusp` requires its argument to be a rational number.

`Plusp` is a Common Lisp function. See any Common Lisp documentation for more information.

10.187 POSITION

POSITION position of an item in a string or a list, using `eql^` as test

`(Position item seq)` is the least index (zero-based) of the element `item` in the string or list `seq`, if in fact `item` is an element of `seq`. Otherwise `(position item seq)` is `nil`.

`(Position item lst)` is provably the same in the ACL2 logic as `(position-equal item lst)`. It has a stronger guard^ than `position-equal^` because uses `eql^` to test equality of `item` with members of `lst`. Its guard^ requires that `lst` is a true list, and moreover, either `(eqlablep item)` or all members of `lst` are `eqlablep^`. See Section 10.189 [POSITION-EQUAL], page 481 and see Section 10.188 [POSITION-EQ], page 480.

`Position` is a Common Lisp function. See any Common Lisp documentation for more information. Since ACL2 functions cannot take keyword arguments (though macros can), the ACL2 functions `position-equal^` and `position-eq^` are defined to correspond to calls of the Common Lisp function `position` whose keyword argument `:test` is `equal^` or `eq^`, respectively.

10.188 POSITION-EQ

POSITION-EQ position of an item in a string or a list, using `eq^` as test

`(Position-eq item seq)` is the least index (zero-based) of the element `item` in the string or list `seq`, if in fact `item` is an element of `seq`. Otherwise `(position item seq)` is `nil`.

`(Position-eq item lst)` is provably the same in the ACL2 logic as `(position item lst)` and `(position-equal item lst)`, but it has a stronger guard^ because it uses `eq^` for a more efficient test for whether `item` is equal to a given member of `lst`. Its guard^ requires that `lst` is a true list, and moreover, either `item` is a symbol or `lst` is a list of symbols. See Section 10.189 [POSITION-EQUAL], page 481 and see Section 10.187 [POSITION], page 480.

10.189 POSITION-EQUAL

POSITION-EQUAL position of an item in a string or a list

`(Position item seq)` is the least index (zero-based) of the element `item` in the string or list `seq`, if in fact `item` is an element of `seq`. Otherwise `(position item seq)` is `nil`.

`(Position-equal item lst)` has a guard^ of `(true-listp lst)`. `Position-equal` has the same functionality as the Common Lisp function `position^`, except that it uses the `equal^` function to test whether `item` is the same as each successive element of `lst`. See Section 10.187 [POSITION], page 480 and see Section 10.188 [POSITION-EQ], page 480.

10.190 PPROGN

PPROGN evaluate a sequence of forms that return state^

```
Example Form:
(pprogn
 (cond ((or (equal (car l) #\) (equal (car l) slash-char))
        (princ$ #\ channel state))
       (t state))
 (princ$ (car l) channel state)
 (mv (cdr l) state))
```

The convention for `pprogn` usage is to give it a non-empty sequence of forms, each of which (except possibly for the last) returns state (see Section 8.81 [STATE], page 344) as its only value. The state^ returned by each but the last is passed on to the next. The value or values of the last form are returned as the value of the `pprogn`.

General Form:

```
(PPROGN form1
        form2
        ...
        formk
        result-form)
```

This general form is equivalent, via macro expansion, to:

```
(LET ((STATE form1))
```

```
(LET ((STATE form2))
     ...
     (LET ((STATE formk))
          result-form)))
```

10.191 PROGN

PROGN see the documentation for `er-progn^`

ACL2 does not allow the use of `progn` in definitions. Instead, a function `er-progn^` can be used for sequencing state^-oriented operations; see Section 10.106 [ER-PROGN], page 436 and see Section 8.81 [STATE], page 344.

`Progn` is a Common Lisp function. See any Common Lisp documentation for more information.

10.192 PROOFS-CO

PROOFS-CO the proofs character output channel

`Proofs-co` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(proofs-co state)` and the updater is `(set-proofs-co val state)`. `Proofs-co` must be an open character output channel. It is to this channel that `defun^`, `defthm^`, and the other event command^s print their commentary.

“Proofs-co” stands for “proofs character output.” The initial value of `proofs-co` is the same as the value of `*standard-co*^` (see Section 10.3 [*STANDARD-CO*], page 399).

10.193 PROPER-CONSP

PROPER-CONSP recognizer for proper (null-terminated) non-empty lists

`Proper-consp` is the function that checks whether its argument is a non-empty list that ends in `nil`. Also see Section 10.249 [TRUE-LISTP], page 503.

10.194 PUT-ASSOC-EQ

PUT-ASSOC-EQ modify an association list by associating a value with a key

`(Put-assoc-eq name val alist)` returns an alist that is the same as the list `alist`, except that the first pair in `alist` with a `car^` of `name` is replaced by `(cons name val)`, if there is one. If there is no such pair, then `(cons name val)` is added at the end.

The guard^ of `(put-assoc-eq name val alist)` requires that `alist` is an `alistp^`, and moreover, either `name` is a symbol or `alist` is a `symbol-alistp^`.

10.195 PUT-ASSOC-EQUAL

PUT-ASSOC-EQUAL modify an association list by associating a value with a key

`(Put-assoc-equal name val alist)` returns an alist that is the same as the list `alist`, except that the first pair in `alist` with a `car^` of `name` is replaced by `(cons name val)`, if there is one. If there is no such pair, then `(cons name val)` is added at the end.

The guard^ of `(put-assoc-equal name val alist)` requires that `alist` is an `alistp^`.

10.196 RASSOC

RASSOC look up value in association list, using `eql^` as test

`(Rassoc x alist)` is similar to `(assoc x alist)`, the difference being that it looks for the first pair in the given alist whose `cdr^`, rather than `car^`, is `eql^` to `x`. See Section 10.28 [ASSOC], page 409.

The guard^ of `rassoc` requires its second argument to be an alist, and in addition, that either its first argument is `eqlablep^` or else all second components of pairs belonging to the second argument are `eqlablep^`. See Section 10.104 [EQLABLEP], page 436.

`Rassoc` is a Common Lisp function. See any Common Lisp documentation for more information.

10.197 RATIONAL-LISTP

RATIONAL-LISTP recognizer for a true list of rational numbers

The predicate `rational-listp` tests whether its argument is a true list of rational numbers.

10.198 RATIONALP

RATIONALP recognizer for whole numbers

`(rationalp x)` is true if and only if `x` is an rational number.

10.199 REALPART

REALPART real part of a complex number

Basic axiom:

```
(equal (realpart x)
       (if (acl2-numberp x)
           (realpart x)
         0))
```

Guard^ for `(realpart x)`:

```
(acl2-numberp x)
```

10.200 REDEFINING-PROGRAMS

REDEFINING-PROGRAMS an explanation of why we restrict redefinitions

ACL2 does not in general allow the redefinition of functions because logical inconsistency can result: previously stored theorems can be rendered invalid if the axioms defining the functions involved are changed. However, to permit prototyping of both `:program^` and `:logic^` mode systems, ACL2 permits redefinition if the user has accepted logical responsibility for the consequences by setting `ld-redefinition-action^` to an appropriate non-`nil` value. The refusal of ACL2 to support the unrestricted redefinition of `:program^` mode functions may appear somewhat capricious. After all, what are the logical consequences of changing a definition if no axioms are involved?

Three important points should be made before we discuss redefinition further.

The first is that ACL2 does support redefinition (of both `:program^` and `:logic^` functions) when `ld-redefinition-action^` is non-`nil`.

The second is that a "redefinition" that does not change the mode, formals or body of a function is considered redundant and is permitted even when `ld-redefinition-action^` is `nil`. We recognize and permit redundant definitions because it is not uncommon for two distinct books^ to share identical function definitions. When determining whether the body of a function is changed by a proposed redefinition, we actually compare the untranslated versions of the two bodies. See Section 8.85 [TERM], page 352. For example, redundancy is not recognized if the old body is `(list a b)` and the new body is `(cons a (cons b nil))`. We use the untranslated bodies because of the difficulty of translating the new body in the presence of the old syntactic information, given the possibility that the redefinition might attempt to change the signature^ of the function, i.e., the number of formals, the number of results, or the position of `STATE^` in either.

The third important point is that a "redefinition" that preserves the formals and body but changes the mode from `:program^` to `:logic^` is permitted even when `ld-redefinition-action^` is `nil`. That is what `verify-termination^` does.

This note addresses the temptation to allow redefiniton of `:program^` functions in situations other than the three described above. Therefore, suppose `ld-redefinition-action^` is `nil` and consider the cases.

Case 1. Suppose the new definition attempts to change the formals or more generally the signature^ of the function. Accepting such a redefinition would render ill-formed other `:program^` functions which call the redefined function. Subsequent attempts to evaluate those callers could arbitrarily damage the Common Lisp image. Thus, redefinition of `:program^` functions under these circumstances requires the user's active approval, as would be sought with `ld-redefinition-action^ '(:query . :overwrite)`.

Case 2. Suppose the new definition attempts to change the body (even though it preserves the signature^). At one time we believed this was acceptable and ACL2 supported the quiet redefinition of `:program^` mode functions in this circumstance. However, because such functions can be used in macros and redundancy checking is based on untranslated bodies, this turns out to be unsound!

It is therefore now prohibited. We illustrate such an unsoundness below. Let `foo-thm1.lisp` be a book with the following contents.

```
(in-package "ACL2")
(defun p1 (x) (declare (xargs :mode :program)) (list 'if x 't 'nil))
(defmacro p (x) (p1 x))
(defun foo (x) (p x))
(defthm foo-thm1 (iff (foo x) x) :rule-classes nil)
```

Note that the macro form `(p x)` translates to `(if x t nil)`. The `:program^` function `p1` is used to generate this translation. The function `foo` is defined so that `(foo x)` is `(p x)` and a theorem about `foo` is proved, namely, that `(foo x)` is true iff `x` is true.

Now let `foo-thm2.lisp` be a book with the following contents.

```
(in-package "ACL2")
(defun p1 (x) (declare (xargs :mode :program)) (list 'if x 'nil 't))
(defmacro p (x) (p1 x))
(defun foo (x) (p x))
(defthm foo-thm2 (iff (foo x) (not x)) :rule-classes nil)
```

In this book, the `:program^` function `p1` is defined so that `(p x)` means just the negation of what it meant in the first book, namely, `(if x nil t)`. The function `foo` is defined identically — more precisely, the *untranslated* body of `foo` is identical in the two books^, but because of the difference between the two versions of the the `:program^` function `p1` the axioms defining the two `foo`s are different. In the second book we prove the theorem that `(foo x)` is true iff `x` is nil.

Now consider what would happen if the signature^-preserving redefinition of `:program^` functions were permitted and these two books^ were included. When the second book is included the redefinition of `p1` would be permitted since the signature^ is preserved and `p1` is just a `:program^`. But then when the redefinition of `foo` is processed it would be considered redundant and thus be permitted. The result would be a logic in which it was possible to prove that `(foo x)` is equivalent to both `x` and `(not x)`. In particular, the following sequence leads to a proof of nil:

```
(include-book "foo-thm1")
(include-book "foo-thm2")
(thm nil :hints (("Goal" :use (foo-thm1 foo-thm2))))
```

It might be possible to loosen the restrictions on the redefinition of `:program^` functions by allowing signature^-preserving redefinition of `:program^` functions not involved in macro definitions. Alternatively, we could implement definition redundancy checking based on the translated bodies of functions (though that is quite problematic). Barring those two changes, we believe it is necessary simply to impose the same restrictions on the redefinition of `:program^` mode functions as we do on `:logic^` mode functions.

10.201 REM

REM remainder using `truncate^`

```
ACL2 !>(rem 14 3)
2
ACL2 !>(rem -14 3)
-2
ACL2 !>(rem 14 -3)
2
ACL2 !>(rem -14 -3)
-2
ACL2 !>(rem -15 -3)
0
ACL2 !>
```

`(Rem i j)` is that number `k` that `(* j (truncate i j))` added to `k` equals `i`.

The guard^ for `(rem i j)` requires that `i` and `j` are rational numbers and `j` is non-zero.

`Rem` is a Common Lisp function. See any Common Lisp documentation for more information.

10.202 REMOVE

REMOVE remove all occurrences, testing using `eql^`

`(remove x l)` is `l` if `x` is not a member of `l`, else is the result of removing all occurrences of `x` from `l`.

The guard^ for `(remove x l)` requires `l` to be a true list and moreover, either `x` is `eqlablep^` or all elements of `l` are `eqlablep^`.

`Remove` is a Common Lisp function. See any Common Lisp documentation for more information. Note that we do not allow keyword arguments (such as `test`) in ACL2 functions, in particular, in `remove`.

10.203 REMOVE-DUPLICATES

REMOVE-DUPLICATES remove duplicates from a string or (using `eql^`) a list

`Remove-duplicates` returns the result of deleting duplicate elements from the beginning of the given string or true list, i.e., leaving the last element in place. For example,

```
(remove-duplicates '(1 2 3 2 4))
```

is equal to `'(1 3 2 4)`.

The guard^ for `Remove-duplicates` requires that its argument is a string or a true-list of `eqlablep^` objects. It uses the function `eql^` to test for equality between elements of its argument.

`Remove-duplicates` is a Common Lisp function. See any Common Lisp documentation for more information. Note that we do not allow keyword arguments (such as `test`) in ACL2 functions, in particular, in `remove-duplicates`. But see Section 10.204 [REMOVE-DUPLICATES-EQUAL], page 488, which is similar but uses the function `equal^` to test for duplicate elements.

10.204 REMOVE-DUPLICATES-EQUAL

REMOVE-DUPLICATES-EQUAL remove duplicates from a list

`Remove-duplicates-equal` is the same as `remove-duplicates^`, except that its argument must be a true list (not a string), and `equal^` is used to check membership rather than `eql^`. See Section 10.203 [REMOVE-DUPLICATES], page 488.

The guard^ for `Remove-duplicates-equal` requires that its argument is a true list. Note that unlike `remove-duplicates^`, it does not allow string arguments.

10.205 REST

REST rest (`cdr^`) of the list

In the logic, `rest` is just a macro for `cdr^`.

`Rest` is a Common Lisp function. See any Common Lisp documentation for more information.

10.206 REVAPPEND

REVAPPEND concatentate the reverse^ of one list to another

`(Revappend x y)` concatenate^s the reverse^ of the list `x` to `y`, which is also typically a list.

The following theorem characterizes this English description.

```
(equal (revappend x y)
       (append (reverse x) y))
```

Hint: This lemma follows immediately from the definition of `reverse^` and the following lemma.

```
(defthm revappend-append
  (equal (append (revappend x y) z)
         (revappend x (append y z))))
```

The guard^ for `(revappend x y)` requires that `x` is a true list.

`Revappend` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.207 REVERSE

REVERSE reverse a list

`(Reverse x)` is the result of reversing the order of the elements of the list `x`.

The guard^ for `reverse` requires that its argument is a true list.

`Reverse` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.208 RFIX

RFIX coerce to a rational number

`Rfix` simply returns any rational number argument unchanged, returning `0` on a non-rational argument. Also see Section 10.173 [NFIX], page 475, see Section 10.122 [IFIX], page 451, and see Section 10.112 [FIX], page 438 for analogous functions that coerce to a natural number, an integer, and a number, respectively.

`Rfix` has a guard^ of `t`.

10.209 ROUND

ROUND division returning an integer by rounding off

```
Example Forms:
ACL2 !>(round 14 3)
5
ACL2 !>(round -14 3)
-5
ACL2 !>(round 14 -3)
-5
ACL2 !>(round -14 -3)
5
ACL2 !>(round 13 3)
4
ACL2 !>(round -13 3)
-4
ACL2 !>(round 13 -3)
-4
ACL2 !>(round -13 -3)
4
ACL2 !>(round -15 -3)
5
ACL2 !>(round 15 -2)
-8
```

`(Round i j)` is the result of taking the quotient of `i` and `j` and rounding off to the nearest integer. When the quotient is exactly halfway between consecutive integers, it rounds to the even one.

The guard^ for `(round i j)` requires that `i` and `j` are rational numbers and `j` is non-zero.

`Round` is a Common Lisp function. See any Common Lisp documentation for more information.

10.210 SECOND

SECOND second member of the list

See any Common Lisp documentation for details.

10.211 SET-DIFFERENCE-EQUAL

SET-DIFFERENCE-EQUAL elements of one list that are not elements of another

`(Set-difference l1 l2)` equals a list whose members (see Section 10.166 [MEMBER-EQUAL], page 470) contains the members of `x` that are not members of `y`. More precisely, the resulting list is the same as one gets by deleting the members of `y` from `x`, leaving the remaining elements in the same order as they had in `x`.

The guard^ for `set-difference-equal` requires both arguments to be true lists. Essentially, `set-difference-equal` has the same functionality as the Common Lisp function `set-difference`, except that it uses the `equal^` function to test membership rather than `eql^`. However, we do not include the function `set-difference` in ACL2, because the Common Lisp language does not specify the order of the elements in the list that it returns.

10.212 SEVENTH

SEVENTH seventh member of the list

See any Common Lisp documentation for details.

10.213 SIGNUM

SIGNUM indicator for positive, negative, or zero

`(Signum x)` is `0` if `x` is `0`, `-1` if `x` is negative, and is `1` otherwise.

The guard^ for `signum` requires its argument to be rational.

`Signum` is a Common Lisp function. See any Common Lisp documentation for more information.

From "Common Lisp the Language" page 206, we see a definition of `signum` in terms of `abs^`. As explained elsewhere (see Section 10.16 [ABS], page 405), the guard^ for `abs^` requires its argument to be a rational number; hence, we make the same restriction for `signum`.

10.214 SIXTH

SIXTH sixth member of the list

See any Common Lisp documentation for details.

10.215 STANDARD-CHAR-LISTP

STANDARD-CHAR-LISTP recognizer for standard characters

`(standard-char-listp x)` is true if and only if `x` is a null-terminated list all of whose members are standard characters^. See Section 10.216 [STANDARD-CHAR-P], page 492.

`Standard-char-listp` has a guard^ of `t`.

10.216 STANDARD-CHAR-P

STANDARD-CHAR-P recognizer for standard characters

`(Standard-char-p x)` is true if and only if `x` is a "standard" character, i.e., a member of the list `*standard-chars*`. This list includes `#Newline` and `#Space` characters^, as well as the usual punctuation and alphanumeric characters^.

`Standard-char-p` has a guard^ requiring its argument to be a character.

`Standard-char-p` is a Common Lisp function. See any Common Lisp documentation for more information.

10.217 STANDARD-CO

STANDARD-CO the character output channel to which `ld^` prints

`Standard-co` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(standard-co state)` and the updater is `(set-standard-co val state)`. `Standard-co` must be an open character output channel. It is to this channel that `ld^` prints the prompt^, the form to be evaluated, and the results. The event command^s such as `defun^`, `defthm^`, etc., which print extensive commentary do not print to `standard-co` but rather to a different channel, `proofs-co^`, so that you may redirect this commentary while still interacting via `standard-co`. See Section 10.192 [PROOFS-CO], page 482.

"Standard-co" stands for "standard character output." The initial value of `standard-co` is the same as the value of `*standard-co*^` (see Section 10.3 [*STANDARD-CO*], page 399).

10.218 STANDARD-OI

STANDARD-OI the standard object input "channel"

`Standard-oi` is an `ld^` special (see Section 9.7 [LD], page 380). The accessor is `(standard-oi state)` and the updater is `(set-standard-oi val state)`. `Standard-oi` must be an open object input channel, a true list of objects, or a list of objects whose last `cdr^` is an open object input channel. It is from this source that `ld^` takes the input forms to process. When `ld^` is called, if the value specified for `standard-oi` is a string or a list of objects whose last `cdr^` is a string, then `ld^` treats the string as a file name and opens an object input channel from that file. The channel opened by `ld^` is closed by `ld^` upon termination.

"Standard-oi" stands for "standard object input." The read-eval-print loop in `ld^` reads the objects in `standard-oi` and treats them as forms to be evaluated. The initial value of `standard-oi` is the same as the value of `*standard-oi*^` (see Section 10.4 [*STANDARD-OI*], page 400).

10.219 STRING

STRING coerce^ to a string

`(String x)` coerce^s `x` to a string. If `x` is already a string, then it is returned unchanged; if `x` is a symbol, then its `symbol-name^` is returned; and if `x` is a character, the corresponding one-character string is returned.

The guard^ for `string` requires its argument to be a string, a symbol, or a character.

`String` is a Common Lisp function. See any Common Lisp documentation for more information.

10.220 STRING-ALISTP

STRING-ALISTP recognizer for association lists with strings as keys

`(String-alistp x)` is true if and only if `x` is a list of pairs of the form `(cons key val)` where `key` is a string.

`String-alistp` has a guard^ of `t`.

10.221 STRING-APPEND

STRING-APPEND concatenate^ two strings

NOTE: It is probably more efficient to use the Common Lisp function `concatenate^` in place of `string-append`. That is,

```
(string-append str1 str2)
```

is equal to

```
(concatenate 'string str1 str2).
```

At any rate, `string-append` takes two arguments, which are both strings (if the guard^ is to be met), and returns a string obtained concatenating together the characters^ in the first string followed by those in the second. See Section 10.89 [CONCATENATE], page 430.

10.222 STRING-DOWNCASE

STRING-DOWNCASE in a given string, turn upper-case characters^ into lower-case

For a string `x`, `(string-downcase x)` is the result of applying `char-downcase`^ to each character in `x`.

The guard^ for `string-downcase` requires its argument to be a string.

`String-downcase` is a Common Lisp function. See any Common Lisp documentation for more information.

10.223 STRING-EQUAL

STRING-EQUAL string equality without regard to case

For strings `str1` and `str2`, `(string-equal str1 str2)` is true if and only `str1` and `str2` are the same except perhaps for the cases of their characters^.

The guard^ on `string-equal` requires that its arguments are strings.

`String-equal` is a Common Lisp function. See any Common Lisp documentation for more information.

10.224 STRING-LISTP

STRING-LISTP recognizer for a true list of strings

The predicate `string-listp` tests whether its argument is a `true-listp`^ of strings.

10.225 STRING-UPCASE

STRING-UPCASE in a given string, turn lower-case characters^ into upper-case

For a string `x`, `(string-upcase x)` is the result of applying `char-upcase`^ to each character in `x`.

The guard^ for `string-upcase` requires its argument to be a string.

`String-upcase` is a Common Lisp function. See any Common Lisp documentation for more information.

10.226 STRING<

STRING< less-than test for strings

`(String< str1 str2)` is non-`nil` if and only if the string `str1` precedes the string `str2` lexicographically, where character inequalities are tested using `char<`^. When non-`nil`, `(string< str1 str2)` is the first position (zero-based) at which the strings differ. Here are some examples.

```
ACL2 !>(string< "abcd" "abu")
2
ACL2 !>(string< "abcd" "Abu")
NIL
ACL2 !>(string< "abc" "abcde")
3
ACL2 !>(string< "abcde" "abc")
NIL
```

The guard^ for `string<` specifies that its arguments are strings.

`String<` is a Common Lisp function. See any Common Lisp documentation for more information.

10.227 STRING<=

STRING<= less-than-or-equal test for strings

(String<= str1 str2) is non-nil if and only if the string str1 precedes the string str2 lexicographically or the strings are equal. When non-nil, (string<= str1 str2) is the first position (zero-based) at which the strings differ, if they differ, and otherwise is their common length. See Section 10.226 [STRING<], page 496.

The guard^ for string<= specifies that its arguments are strings.

String<= is a Common Lisp function. See any Common Lisp documentation for more information.

10.228 STRING>

STRING> greater-than test for strings

(String> str1 str2) is non-nil if and only if str2 precedes str1 lexicographically. When non-nil, (string> str1 str2) is the first position (zero-based) at which the strings differ. See Section 10.226 [STRING<], page 496.

String> is a Common Lisp function. See any Common Lisp documentation for more information.

10.229 STRING>=

STRING>= less-than-or-equal test for strings

(String>= str1 str2) is non-nil if and only if the string str2 precedes the string str1 lexicographically or the strings are equal. When non-nil, (string>= str1 str2) is the first position (zero-based) at which the strings differ, if they differ, and otherwise is their common length. See Section 10.228 [STRING>], page 497.

The guard^ for string>= specifies that its arguments are strings.

String>= is a Common Lisp function. See any Common Lisp documentation for more information.

10.230 STRINGP

STRINGP recognizer for strings

`(stringp x)` is true if and only if `x` is a string.

10.231 STRIP-CARS

STRIP-CARS collect up all first components of pairs in a list

`(strip-cars x)` is the list obtained by walking through the list `x` and collecting up all first components (`car`^s).

`(strip-cars x)` has a guard^ of `(alistp x)`.

10.232 STRIP-CDRS

STRIP-CDRS collect up all second components of pairs in a list

`(strip-cdrs x)` has a guard^ of `(alistp x)`, and returns the list obtained by walking through the list `x` and collecting up all second components (`cdr`^s).

10.233 SUBLIS

SUBLIS substitute an alist into a tree

`(Sublis alist tree)` is obtained by replacing every leaf of `tree` with the result of looking that leaf up in the association list `alist`. However, a leaf is left unchanged if it is not found as a key in `alist`.

Leaves are lookup up using the function `assoc^`. The guard^ for `(subsetp alist tree)` requires `(eqlable-alistp alist)`. This guard^ ensures that the guard^ for `assoc^` will be met for each lookup generated by `sublis`.

`Sublis` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.234 SUBSEQ

SUBSEQ subsequence of a string or list

For any natural numbers `start` and `end`, where `start <= end <= (length seq)`, `(subseq start end)` is the subsequence of `seq` from index `start` up to, but not including, index `end`. `End` may be `nil`, which which case it is treated as though it is `(length seq)`, i.e., we obtain the subsequence of `seq` from index `start` all the way to the end.

The guard^ for `(subseq seq start end)` is that `seq` is a true list or a string, `start` and `end` are integers (except, `end` may be `nil`, in which case it is treated as `(length seq)` for ther rest of this discussion), and `0 <= start <= end <= (length seq)`.

`Subseq` is a Common Lisp function. See any Common Lisp documentation for more information. Note: In Common Lisp the third argument of `subseq` is optional, but in ACL2 it is required, though it may be `nil` as explained above.

10.235 SUBSETP

SUBSETP test if every `member^` of one list is a `member^` of the other

`(Subsetp x y)` is true if and only if every member of the list `x` is a member of the list `y`.

Membership is tested using the function `member^`. Thus, the guard^ for `subsetp` requires that its arguments are true lists, and moreover, at least one satisfies `eqlable-listp^`. This guard^ ensures that the guard^ for `member^` will be met for each call generated by `subsetp`.

`Subsetp` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.236 SUBSETP-EQUAL

SUBSETP-EQUAL check if all members of one list are members of the other

`(Subsetp-equal x y)` returns `t` if every member of `x` is a member of `y`, where membership is tested using `member-equal^`.

The guard^ for `subsetp-equal` requires both arguments to be true lists. `Subsetp-equal` has the same functionality as the Common Lisp function `subsetp^`, except that it uses the `equal^` function to test membership rather than `eql^`.

10.237 SUBST

SUBST   a single substitution into a tree

`(Subst new old tree)` is obtained by substituting `new` for every occurence of `old` in the given tree.

Equality to `old` is determined using the function `eql^`. The guard^ for `(subst new old tree)` requires `(eqlablep old)`, which ensures that the guard^ for `eql^` will be met for each comparison generated by `subst`.

`Subst` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.238 SYMBOL-<

SYMBOL-<   less-than test for symbols

`(symbol-< x y)` is non-`nil` if and only if either the `symbol-name^` of the symbol `x` lexicographially precedes the `symbol-name^` of the symbol `y` (in the sense of `string<^`) or else the `symbol-name^`s are equal and the `symbol-package-name^` of `x` lexicographically precedes that of `y` (in the same sense). So for example, `(symbol-< 'abcd 'abce)` and `(symbol-< 'acl2::abcd 'foo::abce)` are true.

The guard^ for `symbol` specifies that its arguments are symbols.

10.239 SYMBOL-ALISTP

SYMBOL-ALISTP   recognizer for association lists with symbols as keys

`(Symbol-alistp x)` is true if and only if `x` is a list of pairs of the form `(cons key val)` where `key` is a `symbolp^`.

10.240 SYMBOL-LISTP

SYMBOL-LISTP   recognizer for a true list of symbols

The predicate `symbol-listp` tests whether its argument is a true list of symbols.

10.241 SYMBOL-NAME

SYMBOL-NAME   the name of a symbol (a string)

Basic axiom:

```
(equal (symbol-name x)
       (if (symbolp x)
           (symbol-name x)
         ""))
```

Guard^ for `(symbol-name x)`:

```
(symbolp x)
```

10.242 SYMBOL-PACKAGE-NAME

SYMBOL-PACKAGE-NAME   the name of the package of a symbol (a string)

Basic axiom:

```
(equal (symbol-package-name x)
       (if (symbolp x)
           (symbol-package-name x)
         ""))
```

Guard^ for `(symbol-package-name x)`:

```
(symbolp x)
```

10.243 SYMBOLP

SYMBOLP recognizer for symbols

`(symbolp x)` is true if and only if `x` is a symbol.

10.244 TAKE

TAKE initial segment of a list

For any natural number `n` not exceeding the length of `l`, `(take n l)` collects the first `n` elements of the list `l`.

The following is a theorem (though it takes some effort, including lemmas, to get ACL2 to prove it):

```
(equal (length (take n l)) (nfix n))
```

If `n` is is an integer greater than the length of `l`, then `take` pads the list with the appropriate number of `nil` elements. Thus, the following is also a theorem.

```
(implies (and (integerp n)
              (true-listp l)
              (<= (length l) n))
         (equal (take n l)
                (append l (make-list (- n (length l))))))
```

For related functions, see Section 10.179 [NTHCDR], page 477 and see Section 10.39 [BUTLAST], page 413.

The guard^ for `(take n l)` is that `n` is a nonnegative integer and `l` is a true list.

10.245 TENTH

TENTH tenth member of the list

See any Common Lisp documentation for details.

10.246 THE

THE run-time type check

`(The typ val)` checks that `val` satisfies the type specification `typ` (see Section 10.251 [TYPE-SPEC], page 504). An error is caused if the check fails, and otherwise, `val` is the value of this expression. Here are some examples.

```
(the integer 3)       ; returns 3
(the (integer 0 6) 3) ; returns 3
(the (integer 0 6) 7) ; causes an error
```

See Section 10.251 [TYPE-SPEC], page 504 for a discussion of the legal type specifications.

`The` is defined in Common Lisp. See any Common Lisp documentation for more information.

10.247 THIRD

THIRD third member of the list

See any Common Lisp documentation for details.

10.248 TRUE-LIST-LISTP

TRUE-LIST-LISTP recognizer for true (proper) lists of true lists

`True-list-listp` is the function that checks whether its argument is a list that ends in, or equals, `nil`, and furthermore, all of its elements have that property. Also see Section 10.249 [TRUE-LISTP], page 503.

10.249 TRUE-LISTP

TRUE-LISTP recognizer for proper (null-terminated) lists

`True-listp` is the function that checks whether its argument is a list that ends in, or equals, `nil`.

10.250 TRUNCATE

TRUNCATE division returning an integer by truncating toward 0

```
Example Forms:
ACL2 !>(truncate 14 3)
4
ACL2 !>(truncate -14 3)
-4
ACL2 !>(truncate 14 -3)
-4
ACL2 !>(truncate -14 -3)
4
ACL2 !>(truncate -15 -3)
5
```

`(Truncate i j)` is the result of taking the quotient of `i` and `j` and dropping the fraction. For example, the quotient of `-14` by `3` is `-4 2/3`, so dropping the fraction `2/3`, we obtain a result for `(truncate -14 3)` of `-4`.

The guard^ for `(truncate i j)` requires that `i` and `j` are rational numbers and `j` is non-zero.

`Truncate` is a Common Lisp function. See any Common Lisp documentation for more information.

10.251 TYPE-SPEC

TYPE-SPEC type specifiers in declarations

```
Examples:
The symbol INTEGER in (declare (type INTEGER i j k)) is a type-spec.  Other
type-specs supported by ACL2 include RATIONAL, COMPLEX, (INTEGER 0 127),
(RATIONAL 1 *), CHARACTER, and ATOM.
```

The type-specs and their meanings (when applied to the variable `x` as in `(declare (type type-spec x))` are given below.

```
type-spec                meaning
ATOM                  (ATOM X)
BIT                   (OR (EQUAL X 1) (EQUAL X 0))
CHARACTER             (CHARACTERP X)
COMPLEX,              (AND (COMPLEXP X)
```

```
(COMPLEX RATIONAL)       (RATIONALP (REALPART X))
                         (RATIONALP (IMAGPART X)))
(COMPLEX type)      (AND (COMPLEXP X) (p (REALPART X)) (p (IMAGPART X)))
                         where (p x) is the meaning for type-spec type
CONS                (CONSP X)
INTEGER             (INTEGERP X)
(INTEGER i j)       (AND (INTEGERP X)   ; See notes below
                         (<= i X)
                         (<= X j))
(MEMBER x1 ... xn)  (MEMBER X '(x1 ... xn))
(MOD i)             same as (INTEGER 0 i-1)
NIL                 NIL
NULL                (EQ X NIL)
RATIO               (AND (RATIONALP X) (NOT (INTEGERP X)))
RATIONAL            (RATIONALP X)
(RATIONAL i j)      (AND (RATIONALP X)  ; See notes below
                         (<= i X)
                         (<= X j))
(SATISFIES pred)    (pred X)
SIGNED-BYTE         (INTEGERP X)
(SIGNED-BYTE i)     same as (INTEGER -2**i-1 (2**i-1)-1)
STANDARD-CHAR       (STANDARD-CHARP X)
STRING              (STRINGP X)
(STRING max)        (AND (STRINGP X) (EQUAL (LENGTH X) max))
SYMBOL              (SYMBOLP X)
T                   T
UNSIGNED-BYTE       same as (INTEGER 0 *)
(UNSIGNED-BYTE i)   same as (INTEGER 0 (2**i)-1)
```

Notes:

In general, `(integerp i j)` means

```
(AND (INTEGERP X)
     (<= i X)
     (<= X j)).
```

But if `i` is the symbol `*`, the first inequality is omitted. If `j` is the symbol `*`, the second inequality is omitted. If instead of being an integer, the second element of the type specification is a list containing an integer, `(i)`, then the first inequality is made strict. An analogous remark holds for the `(j)` case. The `RATIONAL` type specifier is similarly generalized.

10.252 UNARY–

UNARY– arithmetic negation function

Basic axiom:

```
(equal (unary-- x)
       (if (acl2-numberp x)
           (unary-- x)
         0))
```

Guard^ for (unary-- x):

```
(acl2-numberp x)
```

Notice that like all arithmetic functions, `unary--` treats non-numeric inputs as `0`.

Calls of the macro `-^` on one argument expand to calls of `unary--`; see Section 10.6 [-], page 401.

10.253 UNARY-/

UNARY-/ reciprocal function

Basic axiom:

```
(equal (unary-/ x)
       (if (and (acl2-numberp x)
                (not (equal x 0)))
           (unary-/ x)
         0))
```

Guard^ for (unary-/ x):

```
(and (acl2-numberp x)
     (not (equal x 0)))
```

Notice that like all arithmetic functions, `unary-/` treats non-numeric inputs as `0`.

Calls of the macro `/^` on one argument expand to calls of `unary-/`; see Section 10.7 [/], page 402.

10.254 UNION-EQ

UNION-EQ union of two lists of symbols

`(Union-eq x y)` equals a list whose members (see Section 10.165 [MEMBER-EQ], page 470) contains the members of `x` and the members of `y`. More precisely, the resulting list is the same as one would get by first deleting the members of `y` from `x`, and then concatenating the result to the front of `y`.

The guard^ for `union-eq` requires both arguments to be true lists, but in fact further requires the first list to contain only symbols, as the function `member-eq^` is used to test membership (with `eq^`). See Section 10.255 [UNION-EQUAL], page 507.

10.255 UNION-EQUAL

UNION-EQUAL union of two lists

`(Union-equal x y)` equals a list whose members (see Section 10.166 [MEMBER-EQUAL], page 470) contains the members of `x` and the members of `y`. More precisely, the resulting list is the same as one would get by first deleting the members of `y` from `x`, and then concatenating the result to the front of `y`.

The guard^ for `union-equal` requires both arguments to be true lists. Essentially, `union-equal` has the same functionality as the Common Lisp function `union`, except that it uses the `equal^` function to test membership rather than `eql^`. However, we do not include the function `union` in ACL2, because the Common Lisp language does not specify the order of the elements in the list that it returns.

10.256 UPDATE-NTH

UPDATE-NTH modify a list by putting the given value at the given position

`(Update-nth key val l)` returns a list that is the same as the list `l`, except that the value at the `0`-based position `key` (a natural number) is `val`.

If `key` is an integer at least as large as the length of `l`, then `l` will be padded with the appropriate number of `nil` elements, as illustrated by the following example.

```
ACL2 !>(update-nth 8 'z '(a b c d e))
(A B C D E NIL NIL NIL Z)
```

We have the following theorem.

```
(implies (and (true-listp l)
              (integerp key)
              (<= 0 key))
         (equal (length (update-nth key val l))
                (if (< key (length l))
                    (length l)
                  (+ 1 key))))
```

The guard^ of `update-nth` requires that its first (position) argument is a natural number and its last (list) argument is a true list.

10.257 UPPER-CASE-P

UPPER-CASE-P recognizer for upper case characters

`(Upper-case-p x)` is true if and only if `x` is an upper case character, i.e., a member of the list `#A`, `#B`, ..., `#Z`.

The guard^ for `upper-case-p` requires its argument to be a character.

`Upper-case-p` is a Common Lisp function. See any Common Lisp documentation for more information.

10.258 ZERO-TEST-IDIOMS

ZERO-TEST-IDIOMS how to test for 0

Below are six commonly used idioms for testing whether `x` is `0`. `Zip^` and `zp^` are the preferred termination tests for recursions down the integers and naturals, respectively.

idiom	logical meaning	guard	primary compiled code*
(equal x 0)	(equal x 0)	t	(equal x 0)
(eql x 0)	(equal x 0)	t	(eql x 0)
(zerop x)	(equal x 0)	x is a number	(= x 0)
(= x 0)	(equal x 0)	x is a number	(= x 0)
(zip x)	(equal (ifix x) 0)	x is an integer	(= x 0)
(zp x)	(equal (nfix x) 0)	x is a natural	(= x 0)

*See Section 8.39.4 [GUARDS-AND-EVALUATION], page 290, especially the subsection titled "Guards and evaluation V: efficiency issues". Primary code is relevant only if guard^s are verified. The "compiled code" shown is only suggestive.

The first four idioms all have the same logical meaning and differ only with respect to their executability and efficiency. In the absence of compiler optimizing, `(= x 0)` is probably the most efficient, `(equal x 0)` is probably the least efficient, and `(eql x 0)` is in between. However, an optimizing compiler could always choose to compile `(equal x 0)` as `(eql x 0)` and, in situations where `x` is known at compile-time to be numeric, `(eql x 0)` as `(= x 0)`. So efficiency considerations must, of course, be made in the context of the host compiler.

Note also that `(zerop x)` and `(= x 0)` are indistinguishable. They have the same meaning and the same guard^, and can reasonably be expected to generate equally efficient code.

Note that `(zip x)` and `(zp x)` do not have the same logical meanings as the others or each other. They are not simple tests for equality to `0`. They each coerce `x` into a restricted domain, `zip^` to the integers and `zp^` to the natural numbers, choosing `0` for `x` when `x` is outside the domain. Thus, `1/2`, `#c(1 3)`, and `'abc`, for example, are all "recognized" as zero by both `zip^` and `zp^`. But `zip^` reports that `-1` is different from `0` while `zp^` reports that `-1` "is" `0`. More precisely, `(zip -1)` is `nil` while `(zp -1)` is `t`.

Note that the last four idioms all have guard^s that restrict their Common Lisp executability. If these last four are used in situations in which guard^s are to be verified, then proof obligations are incurred as the price of using them. If guard verification is not involved in your project, then the first four can be thought of as synonymous.

`Zip^` and `zp^` are not provided by Common Lisp but are ACL2-specific functions. Why does ACL2 provide these functions? The answer has to do with the admission of recursively defined functions and efficiency. `Zp^` is provided as the zero-test in situations where the controlling formal parameter is understood to be a natural number. `Zip^` is analogously provided for the integer case. We illustrate below.

Here is an admissible definition of factorial

```
(defun fact (n) (if (zp n) 1 (* n (fact (1- n)))))
```

Observe the classic recursion scheme: a test against `0` and recursion by `1-^`. Note however that the test against `0` is expressed with the `zp^` idiom. Note also the absence of a guard^ making explicit our intention that `n` is a natural number.

This definition of factorial is readily admitted because when `(zp n)`

is false (i.e., `nil`) then `n` is a natural number other than `0` and so `(1- n)` is less than `n`. The base case, where `(zp n)` is true, handles all the "unexpected" inputs, such as arise with `(fact -1)` and `(fact 'abc)`. When calls of `fact` are evaluated, `(zp n)` checks `(integerp n)` and `(> n 0)`. Guard^ verification is unsuccessful for this definition of `fact` because `zp^` requires its argument to be a natural number and there is no guard^ on `fact`, above. Thus the primary raw lisp for `fact` is inaccessible and only the `:logic^` definition (which does runtime "type" checking) is used in computation. In summary, this definition of factorial is easily admitted and easily manipulated by the prover but is not executed as efficiently as it could be.

Runtime efficiency can be improved by adding a guard^ to the definition.

```
(defun fact (n)
  (declare (xargs :guard (and (integerp n) (>= n 0))))
  (if (zp n) 1 (* n (fact (1- n)))))
```

This guard^ed definition has the same termination conditions as before – termination is not sensitive to the guard^. But the guard^s can be verified. This makes the primary raw lisp definition accessible during execution. In that definition, the `(zp n)` above is compiled as `(= n 0)`, because `n` will always be a natural number when the primary code is executed. Thus, by adding a guard^ and verifying it, the elegant and easily used definition of factorial is also efficiently executed on natural numbers.

Now let us consider an alternative definition of factorial in which `(= n 0)` is used in place of `(zp n)`.

```
(defun fact (n) (if (= n 0) 1 (* n (fact (1- n)))))
```

This definition does not terminate. For example `(fact -1)` gives rise to a call of `(fact -2)`, etc. Hence, this alternative is inadmissible. A plausible response is the addition of a guard^ restricting `n` to the naturals:

```
(defun fact (n)
 (declare (xargs :guard (and (integerp n) (>= n 0))))
 (if (= n 0) 1 (* n (fact (1- n)))))
```

But because the termination argument is not sensitive to the guard^, it is still impossible to admit this definition. To influence the termination argument one must change the conditions tested. Adding a runtime test that `n` is a natural number would suffice and allow both admission and guard^ verification. But such a test would slow down the execution of the compiled function.

The use of `(zp n)` as the test avoids this dilemma. `Zp^` provides the logical equivalent of a runtime test that `n` is a natural number but the execution efficiency of a direct `=^` comparison with `0`, at the expense of a guard^ conjecture to prove. In addition, if guard^ verification and most-efficient execution are not needed, then the use of `(zp n)` allows the admission of the function without a guard^ or other extraneous verbiage.

While general rules are made to be broken, it is probably a good idea to get into the habit of using `(zp n)` as your terminating "`0` test" idiom when recursing down the natural numbers. It provides the logical power of testing that `n` is a non-`0` natural number and allows efficient execution.

We now turn to the analogous function, `zip^`. `Zip^` is the preferred `0`-test idiom when recursing through the integers toward `0`. `Zip^` considers any non-integer to be `0` and otherwise just recognizes `0`. A typical use of `zip^` is in the definition of `integer-length^`, shown below. (ACL2 can actually accept this definition, but only after appropriate lemmas have been proved.)

```
(defun integer-length (i)
  (declare (xargs :guard (integerp i)))
  (if (zip i)
      0
    (if (= i -1)
      0
      (+ 1 (integer-length (floor i 2))))))
```

Observe that the function recurses by `(floor i 2)`. Hence, calling the function on `25` causes calls on `12`, `6`, `3`, `1`, and `0`, while calling it on `-25` generates calls on `-13`, `-7`, `-4`, `-2`, and `-1`. By making `(zip i)` the first test, we terminate the recursion immediately on non-integers. The guard^, if present, can be verified and allows the primary raw lisp definition to check `(= i 0)` as the first terminating condition (because the primary code is executed only on integers).

10.259 ZEROP

ZEROP test an acl2-number against 0

`(zerop x)` is `t` if `x` is `0` and is `nil` otherwise. Thus, it is logically equivalent to `(equal x 0)`.

`(Zerop x)` has a guard^ requiring `x` to be numeric and can be expected to execute more efficiently than `(equal x 0)` in properly guard^ed compiled code.

In recursions down the natural numbers, `(zp x)` is preferred over `(zerop x)` because the former coerces `x` to a natural and allows the termination proof. In recursions through the integers, `(zip x)` is preferred. See Section 10.258 [ZERO-TEST-IDIOMS], page 508.

`Zerop` is a Common Lisp function. See any Common Lisp documentation for more information.

10.260 ZIP

ZIP testing an "integer" against 0

`(Zip i)` is logically equivalent to `(equal (ifix i) 0)` and is the preferred termination test for recursion through the integers. `(Zip i)` returns `t` if `i` is `0` or not an integer; it returns `nil` otherwise. Thus,

```
  i       (zp i)
 3          nil
 0          t
 -2         nil
 5/2        t
 #c(1 3)    t
 'abc       t
```

`(Zip i)` has a guard^ requiring `i` to be an integer.

For a discussion of the various idioms for testing against `0`, see Section 10.258 [ZERO-TEST-IDIOMS], page 508.

`Zip` is typically used as the termination test in recursions through the integers. It has the advantage of "coercing" its argument to an integer and hence allows the definition to be admitted without an explicit type check in the body. Guard^ verification allows `zip` to be compiled as a direct `=`^-comparision with `0`.

10.261 ZP

ZP testing a "natural" against 0

`(Zp n)` is logically equivalent to `(equal (nfix n) 0)` and is the preferred termination test for recursion down the natural numbers. `(Zp n)` returns `t` if `n` is `0` or not a natural number; it returns `nil` otherwise. Thus, in the ACL2 logic (ignoring the issue of guard^s):

```
 n        (zp n)
3          nil
0          t
-1         t
5/2        t
#c(1 3)    t
'abc       t
```

`(Zp n)` has a guard^ requiring `n` to be a natural number.

For a discussion of the various idioms for testing against `0`, see Section 10.258 [ZERO-TEST-IDIOMS], page 508.

`Zp` is typically used as the termination test in recursions down the natural numbers. It has the advantage of "coercing" its argument to a natural and hence allows the definition to be admitted without an explicit type check in the body. Guard^ verification allows `zp` to be compiled as a direct `=`^-comparision with `0`.

11 PROOF-CHECKER

PROOF-CHECKER   support for low-level interaction

Call this up with `(verify ...)`.

* ACL2-PC::=:: (atomic macro)
attempt an equality (or equivalence) substitution
* ACL2-PC::ACL2-WRAP:: (macro)
same as `(lisp x)`
* ACL2-PC::ADD-ABBREVIATION:: (primitive)
add an abbreviation
* ACL2-PC::BASH:: (atomic macro)
call the ACL2 theorem prover's simplifier
* ACL2-PC::BDD:: (atomic macro)
prove the current goal using bdds
* ACL2-PC::BK:: (atomic macro)
move backward one argument in the enclosing term
* ACL2-PC::BOOKMARK:: (macro)
insert matching "bookends" comments
* ACL2-PC::CASESPLIT:: (primitive)
split into two cases
* ACL2-PC::CG:: (macro)
change to another goal.
* ACL2-PC::CHANGE-GOAL:: (primitive)
change to another goal.
* ACL2-PC::CLAIM:: (atomic macro)
add a new hypothesis
* ACL2-PC::COMM:: (macro)
display instructions from the current interactive session
* ACL2-PC::COMMANDS:: (macro)
display instructions from the current interactive session
* ACL2-PC::COMMENT:: (primitive)
insert a comment
* ACL2-PC::CONTRADICT:: (macro)
same as `contrapose`
* ACL2-PC::CONTRAPOSE:: (primitive)
switch a hypothesis with the conclusion, negating both
* ACL2-PC::DEMOTE:: (primitive)
move top-level hypotheses to the conclusion
* ACL2-PC::DIVE:: (primitive)
move to the indicated subterm
* ACL2-PC::DO-ALL:: (macro)
run the given instructions
* ACL2-PC::DO-ALL-NO-PROMPT:: (macro)
run the given instructions, halting once there is a "failure"

* ACL2-PC::DO-STRICT:: (macro)
run the given instructions, halting once there is a “failure”
* ACL2-PC::DROP:: (primitive)
drop top-level hypotheses
* ACL2-PC::DV:: (atomic macro)
move to the indicated subterm
* ACL2-PC::ELIM:: (atomic macro)
call the ACL2 theorem prover’s elimination process
* ACL2-PC::EQUIV:: (primitive)
attempt an equality (or congruence-based) substitution
* ACL2-PC::EX:: (macro)
exit after possibly saving the state
* ACL2-PC::EXIT:: (meta)
exit the interactive proof-checker
* ACL2-PC::EXPAND:: (primitive)
expand the current function call without simplification
* ACL2-PC::FAIL:: (macro)
cause a failure
* ACL2-PC::FORWARDCHAIN:: (atomic macro)
forward chain from an implication in the hyps
* ACL2-PC::FREE:: (atomic macro)
create a “free variable”
* ACL2-PC::GENERALIZE:: (primitive)
perform a generalization
* ACL2-PC::GOALS:: (macro)
list the names of goals on the stack
* ACL2-PC::HELP:: (macro)
proof-checker help facility
* ACL2-PC::HELP!:: (macro)
proof-checker help facility
* ACL2-PC::HELP-LONG:: (macro)
same as `help!`
* ACL2-PC::HYPS:: (macro)
print the hypotheses
* ACL2-PC::ILLEGAL:: (macro)
illegal instruction
* ACL2-PC::IN-THEORY:: (primitive)
set the current proof-checker theory
* ACL2-PC::INDUCT:: (atomic macro)
generate subgoals using induction
* ACL2-PC::LISP:: (meta)
evaluate the given form in Lisp
* ACL2-PC::MORE:: (macro)
proof-checker help facility
* ACL2-PC::MORE!:: (macro)
proof-checker help facility
* ACL2-PC::NEGATE:: (macro)
run the given instructions, and “succeed” if and only if they “fail”

* ACL2-PC::NIL:: (macro)
used for interpreting `control-d`
* ACL2-PC::NOISE:: (meta)
run instructions with output
* ACL2-PC::NX:: (atomic macro)
move forward one argument in the enclosing term
* ACL2-PC::ORELSE:: (macro)
run the first instruction; if (and only if) it "fails", run the second
* ACL2-PC::P:: (macro)
prettyprint the current term
* ACL2-PC::P-TOP:: (macro)
prettyprint the conclusion, highlighting the current term
* ACL2-PC::PP:: (macro)
prettyprint the current term
* ACL2-PC::PRINT:: (macro)
print the result of evaluating the given form
* ACL2-PC::PRINT-ALL-GOALS:: (macro)
print all the (as yet unproved) goals
* ACL2-PC::PRINT-MAIN:: (macro)
print the original goal
* ACL2-PC::PRO:: (atomic macro)
repeatedly apply promote
* ACL2-PC::PROMOTE:: (primitive)
move antecedents of conclusion's `implies` term to top-level hypotheses
* ACL2-PC::PROTECT:: (macro)
run the given instructions, reverting to existing state upon failure
* ACL2-PC::PROVE:: (primitive)
call the ACL2 theorem prover to prove the current goal
* ACL2-PC::PUT:: (macro)
substitute for a "free variable"
* ACL2-PC::QUIET:: (meta)
run instructions without output
* ACL2-PC::R:: (macro)
same as rewrite
* ACL2-PC::REDUCE:: (atomic macro)
call the ACL2 theorem prover's simplifier
* ACL2-PC::REDUCE-BY-INDUCTION:: (macro)
call the ACL2 prover without induction, after going into induction
* ACL2-PC::REMOVE-ABBREVIATIONS:: (primitive)
remove one or more abbreviations
* ACL2-PC::REPEAT:: (macro)
repeat the given instruction until it "fails"
* ACL2-PC::REPEAT-REC:: (macro)
auxiliary to `repeat`

* ACL2-PC::REPLAY:: (macro)
replay one or more instructions
* ACL2-PC::RESTORE:: (meta)
remove the effect of an UNDO command
* ACL2-PC::RETAIN:: (atomic macro)
drop all **but** the indicated top-level hypotheses
* ACL2-PC::RETRIEVE:: (macro)
re-enter the proof-checker
* ACL2-PC::REWRITE:: (primitive)
apply a rewrite rule
* ACL2-PC::RUN-INSTR-ON-GOAL:: (macro)
auxiliary toxae THEN
* ACL2-PC::RUN-INSTR-ON-NEW-GOALS:: (macro)
auxiliary to `then`
* ACL2-PC::S:: (primitive)
simplify the current subterm
* ACL2-PC::S-PROP:: (atomic macro)
simplify propositionally
* ACL2-PC::SAVE:: (macro)
save the proof-checker state (state-stack)
* ACL2-PC::SEQUENCE:: (meta)
run the given list of instructions according to a multitude of options
* ACL2-PC::SHOW-ABBREVIATIONS:: (macro)
display the current abbreviations
* ACL2-PC::SHOW-REWRITES:: (macro)
display the applicable rewrite rules
* ACL2-PC::SKIP:: (macro)
"succeed" without doing anything
* ACL2-PC::SL:: (atomic macro)
simplify with lemmas
* ACL2-PC::SPLIT:: (atomic macro)
split the current goal into cases
* ACL2-PC::SR:: (macro)
same as SHOW-REWRITES
* ACL2-PC::SUCCEED:: (macro)
run the given instructions, and "succeed"
* ACL2-PC::TH:: (macro)
print the top-level hypotheses and the current subterm
* ACL2-PC::THEN:: (macro)
apply one instruction to current goal and another to new subgoals
* ACL2-PC::TOP:: (atomic macro)
move to the top of the goal
* ACL2-PC::TYPE-ALIST:: (macro)
display the type-alist from the current context
* ACL2-PC::UNDO:: (meta)
undo some instructions
* ACL2-PC::UNSAVE:: (macro)

remove a proof-checker state
* ACL2-PC::UP:: (primitive)
move to the parent (or some ancestor) of the current subterm
* ACL2-PC::USE:: (atomic macro)
use a lemma instance
* ACL2-PC::X:: (atomic macro)
expand and (maybe) simplify function call at the current subterm
* ACL2-PC::X-DUMB:: (atomic macro)
expand function call at the current subterm, without simplifying
* DEFINE-PC-HELP:: define a macro command whose purpose is to print something
* INSTRUCTIONS:: instructions to the proof checker
* MACRO-COMMAND:: compound command for the proof-checker
* RETRIEVE:: re-enter a (specified) proof-checker^ state
* TOGGLE-PC-MACRO:: change an ordinary macro command to an atomic macro, or vice-versa
* UNSAVE:: remove a proof-checker^ state
* VERIFY:: enter the interactive proof checker

This is an interactive system for checking theorems in ACL2. One enters it using VERIFY; see Section 11.101 [VERIFY], page 577. The result of an interactive session is a `defthm^` event with an `:instructions^` parameter supplied. Such an event can be replayed later in a new ACL2 session with the "proof-checker" loaded.

Individual proof-checker commands have already been listed above. Among them are proof-checker commands `help^`, `help-long`, `more^`, and `more!^`, which behave respectively like the ACL2 functions `:doc^`, `:doc!^`, `:more^`, and `:more!^`.

11.1 ACL2-PC::=

ACL2-PC::= (atomic macro)
attempt an equality (or equivalence) substitution

```
Examples:
=     -- replace the current subterm by a term equated to it in
         one of the hypotheses (if such a term exists)
(= x) -- replace the current subterm by x, assuming that the prover
         can show that they are equal
(= (+ x y) z)
      -- replace the term (+ x y) by the term z inside the current
         subterm, assuming that the prover can prove
         (equal (+ x y) z) from the current top-level hypotheses
         or that this term or (equal z (+ x y)) is among the
         current top-level hypotheses or the current governors
(= & z)
```

```
        -- exactly the same as above, if (+ x y) is the current
           subterm
   (= (+ x y) z :hints :none)
        -- same as (= (+ x y) z), except that a new subgoal is
           created with the current goal's hypotheses and governors
           as its top-level hypotheses and (equal (+ x y) z) as its
           conclusion
   (= (+ x y) z 0)
        -- exactly the same as immediately above
   (= (p x)
      (p y)
      :equiv iff
      :otf-flg t
      :hints (("Subgoal 2" :BY FOO) ("Subgoal 1" :use bar)))
        -- same as (= (+ x y) z), except that the prover uses
           the indicated values for otf-flg and hints, and only
           propositional (iff) equivalence is used (however, it
           must be that only propositional equivalence matters at
           the current subterm)

   General Form:
   (= &optional x y &rest keyword-args)
```

If terms `x` and `y` are supplied, then replace `x` by `y` inside the current subterm if they are "known" to be "equal". Here "known" means the following: the prover is called as in the `prove` command (using `keyword-args`) to prove `(equal x y)`, except that a keyword argument `:equiv` is allowed, in which case `(equiv x y)` is proved instead, where `equiv` is that argument. (See below for how governors are handled.)

Actually, `keyword-args` is either a single non-keyword or is a list of the form `((kw-1 x-1) ... (kw-n x-n))`, where each `kw-i` is one of the keywords `:equiv`, `:otf-flg`, `:hints`. Here `:equiv` defaults to `equal` if the argument is not supplied or is `nil`, and otherwise should be the name of an ACL2 equivalence relation. `:Otf-flg` and `:hints` give directives to the prover, as explained above and in the documentation for the `prove` command; however, no prover call is made if `:hints` is a non-`nil` atom or if `keyword-args` is a single non-keyword (more on this below).

Remarks on defaults

(1) If there is only one argument, say `a`, then `x` defaults to the current subterm, in the sense that `x` is taken to be the current subterm and `y` is taken to be `a`.

(2) If there are at least two arguments, then `x` may be the symbol `&`, which then represents the current subterm. Thus, `(= a)` is equivalent to `(= & a)`. (Obscure point: actually, `&` can be in any package, except the keyword package.)

(3) If there are no arguments, then we look for a top-level hypothesis or a governor of the form `(equal c u)` or `(equal u c)`, where `c` is the current subterm. In that case we replace the current subterm by `u`.

As with the `prove` command, we allow goals to be given "bye"s in the proof, which may be generated by a `:hints` keyword argument in `keyword-args`. These result in the creation of new subgoals.

A proof is attempted unless the `:hints` argument is a non-`nil` atom other than :`none`, or unless there is one element of `keyword-args` and it is not a keyword. In that case, if there are any hypotheses in the current goal, then what is attempted is a proof of the implication whose antecedent is the conjunction of the current hypotheses and governors and whose conclusion is the appropriate `equal` term.

Notes: (1) It is allowed to use abbreviations in the hints. (2) The keyword :`none` has the special role as a value of :`hints` that is shown clearly in an example above. (3) If there are governors, then the new subgoal has as additional hypotheses the current governors.

11.2 ACL2-PC::ACL2-WRAP

ACL2-PC::ACL2-WRAP (macro)
same as `(lisp x)`

```
Example:
(acl2-wrap (pe :here))

General Form:
(acl2-wrap form)
```

Same as `(lisp form)`. This is provided for interface tools that want to be able to execute the same form in raw Lisp, in the proof-checker, or in the ACL2 top-level loop `(lp)`.

11.3 ACL2-PC::ADD-ABBREVIATION

ACL2-PC::ADD-ABBREVIATION (primitive)
add an abbreviation

Example: `(add-abbreviation v (* x y))` causes future occurrences of `(* x y)` to be printed as `(? v)`, until (unless) a corresponding invocation of `remove-abbreviations` occurs. In this case we say that `v` "abbreviates" `(* x y)`.

```
General Form:
(add-abbreviation var &optional raw-term)
```

Let `var` be an abbreviation for `raw-term`, if `raw-term` is supplied, else for the current subterm. Note that `var` must be a variable that does not already abbreviate some term.

A way to think of abbreviations is as follows. Imagine that whenever an abbreviation is added, say `v` abbreviates `expr`, an entry associating `v` to `expr` is made in an association list, which we will call "`*abbreviations-alist*`". Then simply imagine that `?` is a function defined by something like:

```
(defun ? (v)
  (let ((pair (assoc v *abbreviations-alist*)))
    (if pair (cdr pair)
      (error ...))))
```

Of course the implementation isn't exactly like that, since the "constant" `*abbreviations-alist*` actually changes each time an `add-abbreviation` instruction is successfully invoked. Nevertheless, if one imagines an appropriate redefinition of the "constant" `*abbreviations-alist*` each time an `add-abbreviation` is invoked, then one will have a clear model of the meaning of such an instruction.

The effect of abbreviations on output is that before printing a term, each subterm that is abbreviated by a variable `v` is first replaced by `(? v)`.

The effect of abbreviations on input is that every built-in proof-checker command accepts abbreviations wherever a term is expected as an argument, i.e., accepts the syntax `(? v)` whenever `v` abbreviates a term. For example, the second argument of `add-abbreviation` may itself use abbreviations that have been defined by previous `add-abbreviation` instructions.

See also `remove-abbreviations` and `show-abbreviations`.

11.4 ACL2-PC::BASH

ACL2-PC::BASH (atomic macro)
call the ACL2 theorem prover's simplifier

```
Examples:
bash -- attempt to prove the current goal by simplification alone
(bash ("Subgoal 2" :by foo) ("Subgoal 1" :use bar))
     -- attempt to prove the current goal by simplification alone,
        with the indicated hints

General Form:
(bash &rest hints)
```

Call the theorem prover's simplifier, creating a subgoal for each resulting goal.

Notice that unlike `prove`, the arguments to `bash` are spread out, and are all hints.

Note: All forcing rounds will be skipped (unless there are more than 15 subgoals generated in the first forcing round, an injustice that should be rectified by the next release).

11.5 ACL2-PC::BDD

ACL2-PC::BDD (atomic macro)
prove the current goal using bdds

```
Examples:
bdd
(bdd :vars nil :bdd-constructors (cons) :prove t :literal :all)
```

The general form is as shown in the latter example above, but with any keyword-value pairs omitted and with values as described for the `:bdd` hint; see Section 8.41 [HINTS], page 297.

This command simply calls the theorem prover with the indicated bdd hint for the top-level goal. Note that if `:prove` is `t` (the default), then the proof will succeed entirely using bdds or else it will fail immediately. See Chapter 2 [BDD], page 69.

11.6 ACL2-PC::BK

ACL2-PC::BK (atomic macro)
move backward one argument in the enclosing term

```
Example and General Form:
bk
```

For example, if the conclusion is `(= x (* (- y) z))` and the current subterm is `(* (- y) z)`, then after executing `bk`, the current subterm will be `x`.

Move to the previous argument of the enclosing term.

This is the same as `up` followed by `(dive n-1)`, where `n` is the position of the current subterm in its parent term in the conclusion. Thus in particular, the `nx` command fails if one is already at the top of the conclusion.

See also `up`, `dive`, `top`, and `bk`.

11.7 ACL2-PC::BOOKMARK

ACL2-PC::BOOKMARK (macro)
insert matching "bookends" comments

```
Example:
(bookmark final-goal)

General Form:
(bookmark name &rest instruction-list)
```

Run the instructions in `instruction-list` (as though this were a call of `do-all`; see the documentation for `do-all`), but first insert a begin bookend with the given name and then, when the instructions have been completed, insert an end bookend with that same name. See the documentation of `comm` for an explanation of bookends and how they can affect the display of instructions.

11.8 ACL2-PC::CASESPLIT

ACL2-PC::CASESPLIT (primitive)
split into two cases

```
Example:
(casesplit (< x y)) -- assuming that we are at the top of the
                       conclusion, add (< x y) as a new top-level
                       hypothesis in the current goal, and create a
                       subgoal identical to the current goal except
                       that it has (not (< x y)) as a new top-level
                       hypothesis

General Form:
(casesplit expr &optional use-hyps-flag do-not-flatten-flag)
```

When the current subterm is the entire conclusion, this instruction adds `expr` as a new top-level hypothesis, and create a subgoal identical to the existing current goal except that it has the negation of `expr` as a new top-level hypothesis. See also `claim`. The optional arguments control the use of governors and the "flattening" of new hypotheses, as we now explain.

The argument `use-hyps-flag` is only of interest when there are governors. (To read about governors, see the documentation for the command `hyps`). In that case, if `use-hyps-flag` is not supplied or is `nil`, then the description above is correct; but otherwise, it is not `expr` but rather it is `(implies govs expr)` that is added as a new top-level hypothesis (and whose negation is added as a top-level hypothesis for the new goal), where `govs` is the conjunction of the governors.

If `do-not-flatten-flag` is supplied and not `nil`, then that is all there is to this command. Otherwise (thus this is the default), when the claimed term (first argument) is a conjunction (`and`) of terms and the `claim` instruction succeeds, then each (nested) conjunct of the claimed term is added as a separate new top-level hypothesis. Consider the following example, assuming there are no governors.

```
(casesplit (and (and (< x y) (integerp a)) (equal r s)) t)
```

Three new top-level hypotheses are added to the current goal, namely `(< x y)`, `(integerp a)`, and `(equal r s)`. In that case, only one hypothesis is added to create the new goal, namely the negation of `(and (< x y) (integerp a) (equal r s))`. If the negation of this term had been `claim`ed, then it would be the other way around: the current goal would get a single new hypothesis while the new goal would be created by adding three hypotheses.

Note: It is allowed to use abbreviations in the hints.

11.9 ACL2-PC::CG

ACL2-PC::CG (macro)
change to another goal.

```
Examples:
(cg (main . 1)) -- change to the goal (main . 1)
cg              -- change to the next-to-top goal

General Form:
(CG &OPTIONAL goal-name)
```

Same as `(change-goal goal-name t)`, i.e. change to the indicated and move the current goal to the end of the goal stack.

11.10 ACL2-PC::CHANGE-GOAL

ACL2-PC::CHANGE-GOAL (primitive)
change to another goal.

```
Examples:
(change-goal (main . 1)) -- change to the goal (main . 1)
change-goal              -- change to the next-to-top goal

General Form:
(change-goal &optional goal-name end-flg)
```

Change to the goal with the name `goal-name`, i.e. make it the current goal. However, if `goal-name` is `nil` or is not supplied, then it defaults to the next-to-top goal, i.e., the second goal in the stack of goals. If `end-flg` is supplied and not `nil`, then move the current goal to the end of the goal stack; else merely swap it with the next-to-top goal. Also see documentation for `cg`.

11.11 ACL2-PC::CLAIM

ACL2-PC::CLAIM (atomic macro)
add a new hypothesis

```
Examples:
(claim (< x y))   -- attempt to prove (< x y) from the current
                     top-level hypotheses and if successful, then
                     add (< x y) as a new top-level hypothesis in
                     the current goal
```

```
(claim (< x y)
       :otf-flg t
       :hints (("Goal" :induct t)))
                   -- as above, but call the prover using the
                      indicated values for the otf-flg and hints
(claim (< x y) 0) -- as above, except instead of attempting to
                      prove (< x y), create a new subgoal with the
                      same top-level hypotheses as the current goal
                      that has (< x y) as its conclusion
(claim (< x y) :hints :none)
                   -- same as immediately above

General Form:
(claim expr &rest rest-args)
```

This command creates a new subgoal with the same top-level hypotheses as the current goal but with a conclusion of `expr`. If `rest-args` is a non-empty list headed by a non-keyword, then there will be no proof attempted for the new subgoal. With that possible exception, `rest-args` should consist of keyword arguments. The keyword argument `:do-not-flatten` controls the "flattening" of new hypotheses, just as with the `casesplit` command (as described in its documentation). The remaining `rest-args` are used with a call the `prove` command on the new subgoal, except that if `:hints` is a non-`nil` atom, then the prover is not called — rather, this is the same as the situation described above, where `rest-args` is a non-empty list headed by a non-keyword.

Notes: (1) Unlike the `casesplit` command, the `claim` command is completely insensitive to governors. (2) It is allowed to use abbreviations in the hints. (3) The keyword :`none` has the special role as a value of :`hints` that is shown clearly in an example above.

11.12 ACL2-PC::COMM

ACL2-PC::COMM (macro)
display instructions from the current interactive session

```
Examples:
comm
(comm 10)

General Form:
(comm &optional n)
```

Prints out instructions in reverse order. This is actually the same as `(commands n t)` — or, `(commands nil t)` if `n` is not supplied. As explained in the documentation for `commands`, the final argument of `t` causes suppression of instructions occurring between so-called "matching bookends," which we now explain.

A "begin bookend" is an instruction of the form

```
(COMMENT :BEGIN x . y).
```

Similarly, an "end bookend" is an instruction of the form

```
(COMMENT :END x' . y').
```

The "name" of the first bookend is `x` and the "name" of the second bookend is `x'`. When such a pair of instructions occurs in the current state-stack, we call them "matching bookends" provided that they have the same name (i.e. `x` equals `x'`) and if no other begin or end bookend with name `x` occurs between them. The idea now is that `comm` hides matching bookends together with the instructions they enclose. Here is a more precise explanation of this "hiding"; probably there is no value in reading on!

A `comm` instruction hides bookends in the following manner. (So does a `comment` instruction when its second optional argument is supplied and non-`nil`.) First, if the first argument `n` is supplied and not `nil`, then we consider only the last `n` instructions from the state-stack; otherwise, we consider them all. Now the resulting list of instructions is replaced by the result of applying the following process to each pair of matching bookends: the pair is removed, together with everything in between the begin and end bookend of the pair, and all this is replaced by the "instruction"

```
("***HIDING***" :COMMENT :BEGIN name ...)
```

where `(comment begin name ...)` is the begin bookend of the pair. Finally, after applying this process to each pair of matching bookends, each begin bookend of the form `(comment begin name ...)` that remains is replaced by

```
("***UNFINISHED***" :COMMENT :BEGIN name ...) .
```

11.13 ACL2-PC::COMMANDS

ACL2-PC::COMMANDS (macro)
display instructions from the current interactive session

```
Examples:
commands
(commands 10 t)

General Forms:

commands or (commands nil)
Print out all the instructions (in the current state-stack) in
reverse order, i.e. from the most recent instruction to the starting
instruction.

(commands n) [n a positive integer]
Print out the most recent n instructions (in the current
state-stack), in reverse order.

(commands x abbreviate-flag)
Same as above, but if abbreviate-flag is non-NIL, then do not
display commands between ``matching bookends''.  See documentation
for comm for an explanation of matching bookends.
```

Note: If there are more than `n` instructions in the state-stack, then `(commands n)` is the same as `commands` (and also, `(commands n abb)` is the same as `(commands nil abb)`).

11.14 ACL2-PC::COMMENT

ACL2-PC::COMMENT (primitive)
insert a comment

```
Example:
(comment now begin difficult final goal)

General Form:
(comment &rest x)
```

This instruction makes no change in the state except to insert the `comment` instruction.

Some comments can be used to improve the display of commands; see documentation for `comm`.

11.15 ACL2-PC::CONTRADICT

ACL2-PC::CONTRADICT (macro)

same as `contrapose`

see documentation for `contrapose`

11.16 ACL2-PC::CONTRAPOSE

ACL2-PC::CONTRAPOSE (primitive)
switch a hypothesis with the conclusion, negating both

```
Example:
(contrapose 3)

General Form:
(contrapose &optional n)
```

The (optional) argument `n` should be a positive integer that does not exceed the number of hypotheses. Negate the current conclusion and make it the `n`th hypothesis, while negating the current `n`th hypothesis and making it the current conclusion. If no argument is supplied then the effect is the same as for `(contrapose 1)`.

Note: By "negate" we mean an operation that replaces `nil` by `t`, `x` by `nil` for any other explicit value `x`, `(not x)` by `x`, and any other `x` by `(not x)`.

11.17 ACL2-PC::DEMOTE

ACL2-PC::DEMOTE (primitive)
move top-level hypotheses to the conclusion

```
Examples:
demote          -- demote all top-level hypotheses
(demote 3 5)  -- demote hypotheses 3 and 5
```

For example, if the top-level hypotheses are `x` and `y` and the conclusion is `z`, then after execution of `demote`, the conclusion will be `(implies (and x y) z)` and there will be no (top-level) hypotheses.

```
General Form:
(demote &rest hyps-indices)
```

Eliminate the indicated (top-level) hypotheses, but replace the conclusion `conc` with `(implies hyps conc)` where `hyps` is the conjunction of the hypotheses that were eliminated. If no arguments are supplied, then all hypotheses are demoted, i.e. `demote` is the same as `(demote 1 2 ... n)` where `n` is the number of top-level hypotheses.

Note: You must be at the top of the conclusion in order to use this command. Otherwise, first invoke `top`. Also, `demote` fails if there are no top-level hypotheses or if indices are supplied that are out of range.

11.18 ACL2-PC::DIVE

ACL2-PC::DIVE (primitive)
move to the indicated subterm

```
Examples:
(DIVE 1)    -- assign the new current subterm to be the first
               argument of the existing current subterm
(DIVE 1 2)  -- assign the new current subterm to be the result of
               first taking the 1st argument of the existing
               current subterm, and then the 2nd argument of that
```

For example, if the current subterm is

```
(* (+ a b) c),
```

then after `(dive 1)` it is

```
(+ a b).
```

If after that, then `(dive 2)` is invoked, the new current subterm will be

```
b.
```

Instead of `(dive 1)` followed by `(dive 2)`, the same current subterm could be obtained by instead submitting the single instruction `(dive 1 2)`.

```
General Form:
(dive &rest naturals-list)
```

If `naturals-list` is a non-empty list `(n_1 ... n_k)` of natural numbers, let the new current subterm be the result of selecting the `n_1`-st argument of the current subterm, and then the `n_2`-th subterm of that, ..., finally the `n_k`-th subterm.

Note: `Dive` is related to the command `pp`, in that the diving is done according to raw (translated, internal form) syntax. Use the command `dv` if you want to dive according to the syntax displayed by the command `p`. Note that `(dv n)` can be abbreviated by simply `n`.

11.19 ACL2-PC::DO-ALL

ACL2-PC::DO-ALL (macro)
run the given instructions

```
Example:
(do-all induct p prove)

General Form:
(do-all &rest instruction-list)
```

Run the indicated instructions until there is a hard "failure". The instruction "succeeds" if and only if each instruction in `instruction-list` does. (See the documentation for `sequence` for an explanation of "success" and "failure.") As each instruction is executed, the system will print the usual prompt followed by that instruction, unless the global state variable `print-prompt-and-instr-flg` is `nil`.

Note: If `do-all` "fails", then the failure is hard if and only if the last instruction it runs has a hard "failure".

Obscure point: For the record, `(do-all ins_1 ins_2 ... ins_k)` is the same as `(sequence (ins_1 ins_2 ... ins_k))`.

11.20 ACL2-PC::DO-ALL-NO-PROMPT

ACL2-PC::DO-ALL-NO-PROMPT (macro)
run the given instructions, halting once there is a "failure"

```
Example:
(do-all-no-prompt induct p prove)
```

```
General Form:
(do-all-no-prompt &rest instruction-list)
```

Do-all-no-prompt is the same as do-all, except that the prompt and instruction are not printed each time, regardless of the value of print-prompt-and-instr-flg. Also, restoring is disabled. See the documentation for do-all.

11.21 ACL2-PC::DO-STRICT

ACL2-PC::DO-STRICT (macro)
run the given instructions, halting once there is a "failure"

```
Example:
(do-strict induct p prove)

General Form:
(do-strict &rest instruction-list)
```

Run the indicated instructions until there is a (hard or soft) "failure". In fact do-strict is identical in effect to do-all, except that do-all only halts once there is a hard "failure". See the documentation for do-all.

11.22 ACL2-PC::DROP

ACL2-PC::DROP (primitive)
drop top-level hypotheses

```
Examples:
(drop 2 3) -- drop the second and third hypotheses
drop       -- drop all top-level hypotheses

General Forms:
(drop n1 n2 ...) -- Drop the hypotheses with the indicated indices.

drop             -- Drop all the top-level hypotheses.
```

Note: If there are no top-level hypotheses, then the instruction drop will fail. If any of the indices is out of range, i.e. is not an integer between one and the number of top-level hypotheses (inclusive), then (drop n1 n2 ...) will fail.

11.23 ACL2-PC::DV

ACL2-PC::DV (atomic macro)
move to the indicated subterm

```
Examples:
(dv 1)    -- assign the new current subterm to be the first argument
             of the existing current subterm
(dv 1 2)  -- assign the new current subterm to be the result of
             first taking the 1st argument of the existing
             current subterm, and then the 2nd argument of that
```

For example, if the current subterm is

```
(* (+ a b) c),
```

then after `(dv 1)` it is

```
(+ a b).
```

If after that, then `(dv 2)` is invoked, the new current subterm will be

```
b.
```

Instead of `(dv 1)` followed by `(dv 2)`, the same current subterm could be obtained by instead submitting the single instruction `(dv 1 2)`.

```
General Form:
(dv &rest naturals-list)
```

If `naturals-list` is a non-empty list `(n_1 ... n_k)` of natural numbers, let the new current subterm be the result of selecting the `n_1`-st argument of the current subterm, and then the `n_2`-th subterm of that, ..., finally the `n_k`-th subterm.

Note: `(dv n)` may be abbreviated by simply `n`, so we could have typed `1` instead of `(dv 1)` in the first example above.

Note: See also `dive`, which is related to the command `pp`, in that the diving is done according to raw (translated, internal form) syntax. Use the command `dv` if you want to dive according to the syntax displayed by the command `p`.

11.24 ACL2-PC::ELIM

ACL2-PC::ELIM (atomic macro)
call the ACL2 theorem prover's elimination process

```
Example and General Form:
elim
```

Upon running the `elim` command, the system will create a subgoal will be created for each goal that would have been pushed for proof by induction in an ordinary proof, where **only** elimination is used; not even simplification is used!

11.25 ACL2-PC::EQUIV

ACL2-PC::EQUIV (primitive)
attempt an equality (or congruence-based) substitution

```
Examples:
(equiv (* x y) 3) -- replace (* x y) by 3 everywhere inside the
                     current subterm, if their equality is among the
                     top-level hypotheses or the governors
(equiv x t iff)   -- replace x by t everywhere inside the current
                     subterm, where only propositional equivalence
                     needs to be maintained at each occurrence of x

General form:
(equiv old new &optional relation)
```

Substitute new for old everywhere inside the current subterm, provided that either (relation old new) or (relation new old) is among the top-level hypotheses or the governors (possibly by way of backchaining and/or refinement; see below). If relation is `nil` or is not supplied, then it defaults to `equal`. See also the command `=`, which is much more flexible. Note that this command fails if no substitution is actually made.

Note: No substitution takes place inside explicit values. So for example, the instruction `(equiv 3 x)` will cause `3` to be replaced by `x` if the current subterm is, say, `(* 3 y)`, but not if the current subterm is `(* 4 y)` even though `4 = (1+ 3)`.

The following remarks are quite technical and mostly describe a certain weak form of "backchaining" that has been implemented for `equiv` in order to support the `=` command. In fact neither the term `(relation old new)` nor the term `(relation new old)` needs to be **explicitly** among the current "assumptions", i.e., the top-level hypothesis or the governors. Rather, there need only be such an assumption that "tells us" `(r old new)` or `(r new old)`, for **some** equivalence relation `r` that **refines** `relation`. Here, "tells us" means that either one of the indicated terms is among those assumptions, or else there is an assumption that is an implication whose conclusion is one of the indicated terms and whose hypotheses (gathered up by appropriately flattening the first argument of the `implies` term) are all among the current assumptions.

11.26 ACL2-PC::EX

ACL2-PC::EX (macro)
exit after possibly saving the state

```
Example and General Form:
ex
```

Same as `exit`, except that first the instruction `save` is executed.

If `save` queries the user and is answered negatively, then the exit is aborted.

11.27 ACL2-PC::EXIT

ACL2-PC::EXIT (meta)
exit the interactive proof-checker

```
Examples:
exit                         -- exit the interactive proof-checker
(exit append-associativity) -- exit and create a defthm
                               event named append-associativity

General Forms:

exit --  Exit without storing an event.

(exit event-name &optional rule-classes do-it-flg)
Exit, and store an event.
```

The command `exit` returns you to the ACL2 loop. At a later time, `(verify)` may be executed to get back into the same proof-checker state, as long as there hasn't been an intervening use of the proof-checker (otherwise see `save`).

When given one or more arguments as shown above, `exit` still returns you to the ACL2 loop, but first, if the interactive proof is complete, then it attempts create a `defthm` event with the specified `event-name` and `rule-classes` (which defaults to `(:rewrite)` if not supplied). The event will be printed to the terminal, and then normally the user will be queried whether an event should really be created. However, if the final optional argument `do-it-flg` is supplied and not `nil`, then an event will be made without a query.

For example, the form

```
(exit top-pop-elim (:elim :rewrite) t)
```

causes a `defthm` event named `top-pop-elim` to be created with rule-classes `(:elim :rewrite)`, without a query to the user (because of the argument `t`).

Note: it is permitted for `event-name` to be `nil`. In that case, the name of the event will be the name supplied during the original call of `verify`. (See the documentation for `verify` and `commands`.) Also in that case, if `rule-classes` is not supplied then it defaults to the rule-classes supplied in the original call of `verify`.

`Comments` on "success" and "failure". An `exit` instruction will always "fail", so for example, if it appears as an argument of a `do-strict` instruction then none of the later (instruction) arguments will be executed. Moreover, the "failure" will be "hard" if an event is successfully created or if the instruction is simply `exit`; otherwise it will be "soft". See the documentation for `sequence` for an explanation of hard and soft "failures". An obscure but potentially important fact is that if the "failure" is hard, then the error signal is a special signal that the top-level interactive loop can interpret as a request to exit. Thus for example, a sequencing command that turns an error triple `(mv erp val state)` into `(mv t val state)` would never cause an exit from the interactive loop.

If the proof is not complete, then `(exit event-name ...)` will not cause an exit from the interactive loop. However, in that case it will print out the original user-supplied goal (the one that was supplied with the call to `verify`) and the current list of instructions.

11.28 ACL2-PC::EXPAND

ACL2-PC::EXPAND (primitive)
expand the current function call without simplification

```
Examples:
expand -- expand and do not simplify.
```

For example, if the current subterm is `(append a b)`, then after `(expand t)` the current subterm will be the term:

```
(if (true-listp x)
    (if x
        (cons (car x) (append (cdr x) y))
      y)
  (apply 'binary-append (list x y)))
```

regardless of the top-level hypotheses and the governors.

```
General Form:
(expand &optional
        do-not-expand-lambda-flg new-goals-flg keep-all-guards-flg)
```

Expand the function call at the current subterm, and do not simplify. The options have the following meanings:

```
do-not-expand-lambda-flg:   default is nil; otherwise, the result
                            should be a lambda expression

new-goals-flg:       default of nil means to introduce APPLY for guards

keep-all-guards-flg: default of nil means that the system should make
                     a weak attempt to prove the guards from the
                     current context
```

See also `x`, which allows simplification.

11.29 ACL2-PC::FAIL

ACL2-PC::FAIL (macro)
cause a failure

```
Examples:
```

```
fail
(fail t)

General Form:
(fail &optional hard)
```

This is probably only of interest to writers of macro commands. The only function of `fail` is to fail to "succeed".

The full story is that `fail` and `(fail nil)` simply return `(mv nil nil state)`, while `(fail hard)` returns `(mv hard nil state)` if `hard-flag` is not `nil`. See also `do-strict`, `do-all`, and `sequence`.

11.30 ACL2-PC::FORWARDCHAIN

ACL2-PC::FORWARDCHAIN (atomic macro)
forward chain from an implication in the hyps

```
Example:
(forwardchain 2) ; Second hypothesis should be of the form
                 ; (IMPLIES hyp concl), and the result is to replace
                 ; that hypothesis with concl.


General Forms:
(forwardchain hypothesis-number)
(forwardchain hypothesis-number hints)
(forwardchain hypothesis-number hints quiet-flg)
```

This command replaces the hypothesis corresponding to given index, which should be of the form `(IMPLIES hyp concl)`, with its consequent `concl`. In fact, the given hypothesis is dropped, and the replacement hypothesis will appear as the final hypothesis after this command is executed.

The prover must be able to prove the indicated hypothesis from the other hypotheses, or else the command will fail. The `:hints` argument is used in this prover call, and should have the usual syntax of hints to the prover.

Output is suppressed if `quiet-flg` is supplied and not `nil`.

11.31 ACL2-PC::FREE

ACL2-PC::FREE (atomic macro)
create a "free variable"

```
Example:
(free x)

General Form:
(free var)
```

Mark `var` as a "free variable". Free variables are only of interest for the `put` command; see its documentation for an explanation.

11.32 ACL2-PC::GENERALIZE

ACL2-PC::GENERALIZE (primitive)
perform a generalization

```
Example:
(generalize
 ((and (true-listp x) (true-listp y)) 0)
 ((append x y) w))

General Form:
(generalize &rest substitution)
```

Generalize using the indicated substitution, which should be a non-empty list. Each element of that list should be a two-element list of the form `(term variable)`, where `term` may use abbreviations. The effect of the instruction is to replace each such term in the current goal by the corresponding variable. This replacement is carried out by a parallel substitution, outside-in in each hypothesis and in the conclusion. More generally, actually, the "variable" (second) component of each pair may be `nil` or a number, which causes the system to generate a new name of the form `_` or `_n`, with `n` a natural number; more on this below. However, when a variable is supplied, it must not occur in any goal of the current proof-checker state.

When the "variable" above is `nil`, the system will treat it as the variable `|_|` if that variable does not occur in any goal of the current proof-checker state. Otherwise it treats it as `|_0|`, or `|_1|`, or `|_2|`, and so on, until one of these is not among the variables of the current proof-checker state. If the "variable" is a non-negative integer `n`, then the system treats it as `|_n|` unless that variable already occurs among the current goals, in which case it increments n just as above until it obtains a new variable.

Note: The same variable may not occur as the variable component of two different arguments (though `nil` may occur arbitrarily many times, as may a positive integer).

11.33 ACL2-PC::GOALS

ACL2-PC::GOALS (macro)
list the names of goals on the stack

```
Example and General Form:
goals
```

`Goals` lists the names of all goals that remain to be proved. They are listed in the order in which they appear on the stack of remaining goals, which is relevant for example to the effect of a `change-goal` instruction.

11.34 ACL2-PC::HELP

ACL2-PC::HELP (macro)
proof-checker help facility

```
Examples:

(help rewrite) -- partial documentation on the rewrite command; the
                  rest is available using more or more!

(help! rewrite) -- full documentation on the rewrite command

help, help!     -- this documentation (in part, or in totality,
                   respectively)

General Forms:
(help &optional command)
(help! &optional command)
more
more!
```

The proof checker supports the same kind of documentation as does ACL2 proper. The main difference is that you need to type

```
(help command)
```

in a list rather than `:doc command`. So, to get all the documentation on `command`, type `(help! command)` inside the interactive loop, but to get only a one-line description of the command together with some examples, type `(help command)`. In the latter case, you can get the rest of the help by typing `more!`; or type `more` if you don't necessarily want all the rest of the help at once. (Then keep typing `more` if you want to keep getting more of the help for that command.)

To summarize: as with ACL2, you can type either of the following:

```
more, more!
-- to obtain more (or, all the rest of) the documentation last
   requested by help (or, outside the proof-checker's loop, :doc)
```

It has been arranged that the use of `(help command)` will tell you just about everything you could want to know about `command`, almost always by way of examples. For more details about a command, use `help!`, `more`, or `more!`.

We use the word "command" to refer to the name itself, e.g. `rewrite`. We use the word "instruction" to refer to an input to the interactive system, e.g. `(rewrite foo)` or `(help split)`. Of course, we allow commands with no arguments as instructions in many cases, e.g. `rewrite`. In such cases, `command` is treated identically to `(command)`.

11.35 ACL2-PC::HELP!

ACL2-PC::HELP! (macro)
proof-checker help facility

Same as `help`, except that the entire help message is printed without any need to invoke `more!` or `more`.

Invoke `help` for documentation about the proof-checker help facility.

11.36 ACL2-PC::HELP-LONG

ACL2-PC::HELP-LONG (macro)
same as `help!`

See the documentation for `help!`.

`Help-long` has been included in addition to `help!` for historical reasons. (Such a command is included in Pc-Nqthm).

11.37 ACL2-PC::HYPS

ACL2-PC::HYPS (macro)
print the hypotheses

```
Examples:
hyps                -- print all (top-level) hypotheses
(hyps (1 3) (2 4)) -- print hypotheses 1 and 3 and governors 2 and 4
(hyps (1 3) t)     -- print hypotheses 1 and 3 and all governors

General Form:
(hyps &optional hyps-indices govs-indices)
```

Print the indicated top-level hypotheses and governors. (The notion of "governors" is defined below.) Here, `hyps-indices` and `govs-indices` should be lists of indices of hypotheses and governors (respectively), except that the atom `t` may be used to indicate that one wants all hypotheses or governors (respectively).

The list of "governors" is defined as follows. Actually, we define here the notion of the governors for a pair of the form `<term`, address>]; we're interested in the special case where the term is the conclusion and the address is the current address. If the address is `nil`, then there are no governors, i.e., the list of governors is `nil`. If the term is of the form `(if x y z)` and the address is of the form `(2 . rest)` or `(3 . rest)`, then the list of governors is the result of `cons`ing `x` or its negation (respectively) onto the list of governors for the pair `<y, rest>` or the pair `<z, rest>` (respectively). If the term is of the form `(implies x y)` and the address is of the form `(2 . rest)`, then the list of governors is the result of `cons`ing `x` onto the list of governors for the pair `<y, rest>`. Otherwise, the list of governors for the pair `<term, (n . rest)>` is exactly the list of governors for the pair `<argn, rest>` where `argn` is the `n`th argument of `term`.

If all goals have been proved, a message saying so will be printed. (as there will be no current hypotheses or governors!).

The hyps command never causes an error. It "succeeds" (in fact its value is t) if the arguments (when supplied) are appropriate, i.e. either t or lists of indices of hypotheses or governors, respectively. Otherwise it "fails" (its value is nil).

11.38 ACL2-PC::ILLEGAL

ACL2-PC::ILLEGAL (macro)
illegal instruction

```
Example:
(illegal -3)

General Form:
(illegal instruction)
```

Probably not of interest to most users; always "fails" since it expands to the fail command.

The illegal command is used mainly in the implementation. For example, the instruction 0 is "read" as (illegal 0), since dive expects positive integers.

11.39 ACL2-PC::IN-THEORY

ACL2-PC::IN-THEORY (primitive)
set the current proof-checker theory

```
Example:
(in-theory
   (union-theories *s-prop-theory* '(true-listp binary-append)))

General Form:
(in-theory &optional atom-or-theory-expression)
```

If the argument is not supplied, then this command sets the current proof-checker theory (see below for explanation) to agree with the current ACL2 theory. Otherwise, the argument should be a theory expression, and in that case the proof-checker theory is set to the value of that theory expression.

The current proof-checker theory is used in all calls to the ACL2 theorem prover and rewriter from inside the proof-checker. Thus, the most recent in-theory instruction in the current state-stack has an effect in the proof-checker totally analogous to the effect caused by an `in-theory` hint or event in ACL2. However, `in-theory` instructions in the proof-checker have no effect outside the proof-checker's interactive loop.

If the most recent `in-theory` instruction in the current state of the proof-checker has no arguments, or if there is no `in-theory` instruction in the current state of the proof-checker, then the proof-checker will use the current ACL2 theory. This is true even if the user has interrupted the interactive loop by exiting and changing the global ACL2 theory. However, if the most recent `in-theory` instruction in the current state of the proof-checker had an argument, then global changes to the current theory will have no effect on the proof-checker state.

11.40 ACL2-PC::INDUCT

ACL2-PC::INDUCT (atomic macro)
generate subgoals using induction

```
Examples:
induct, (induct t)
   -- induct according to a heuristically-chosen scheme, creating
      a new subgoal for each base and induction step
(induct (append (reverse x) y))
   -- as above, but choose an induction scheme based on the term
      (append (reverse x) y) rather than on the current goal

General Form:
(induct &optional term)
```

Induct as in the corresponding `:induct` hint given to the theorem prover, creating new subgoals for the base and induction steps. If term is `t` or is not supplied, then use the current goal to determine the induction scheme; otherwise, use that term.

Note: As usual, abbreviations are allowed in the term.

Note: `Induct` actually calls the `prove` command with all processes turned off. Thus, you must be at top of the goal for an `induct` instruction.

11.41 ACL2-PC::LISP

ACL2-PC::LISP (meta)
evaluate the given form in Lisp

```
Example:
(lisp (assign xxx 3))

General Form:
(lisp form)
```

Evaluate `form`. The `lisp` command is mainly of interest for side effects. See also `print`, `skip`, and `fail`.

The rest of the documentation for `lisp` is of interest only to those who use it in macro commands. If the Lisp evaluation (by `trans-eval`) of form returns an "error triple" of the form `(mv erp (cons 3 val) &)`, then `val` is of the form `(erp-1 val-1 &)`, and the `lisp` command returns the appropriate error triple

```
(mv (or erp erp-1)
    val-1
    state) .
```

Otherwise, the `trans-eval` of form must return an "error triple" of the form `(mv erp (list* state-out mult val) &)`, and the `lisp` command returns the appropriate error triple

```
(mv erp
    val
    state) .
```

11.42 ACL2-PC::MORE

ACL2-PC::MORE (macro)
proof-checker help facility

Continues documentation of last proof-checker command visited with `help`.

Invoke `help` for documentation about the proof-checker help facility.

11.43 ACL2-PC::MORE!

ACL2-PC::MORE! (macro)
proof-checker help facility

Continues documentation of last proof-checker command visited with `help`, until all documentation on that command is printed out.

Invoke `help` for documentation about the proof-checker help facility.

11.44 ACL2-PC::NEGATE

ACL2-PC::NEGATE (macro)
run the given instructions, and "succeed" if and only if they "fail"

Example: (negate prove)

```
General form:
(negate &rest instruction-list)
```

Run the indicated instructions exactly in the sense of `do-all`, and "succeed" if and only if they "fail".

Note: `Negate` instructions will never produce hard "failures".

11.45 ACL2-PC::NIL

ACL2-PC::NIL (macro)
used for interpreting `control-d`

```
Example and General form:
nil
```

(or, `control-d`).

The whole point of this command is that in some Lisps (including akcl), if you type `control-d` then it seems, on occasion, to get interpreted as `nil`. Without this command, one seems to get into an infinite loop.

11.46 ACL2-PC::NOISE

ACL2-PC::NOISE (meta)
run instructions with output

```
Example:
(noise induct prove)

General Form:
(noise &rest instruction-list)
```

Run the `instruction-list` through the top-level loop with output.

In fact, having output is the default. `Noise` is useful inside a surrounding call of `quiet`, when one temporarily wants output. For example, if one wants to see output for a `prove` command immediately following an `induct` command but before an `s` command, one may want to submit an instruction like `(quiet induct (noise prove) s)`. See also `quiet`.

11.47 ACL2-PC::NX

ACL2-PC::NX (atomic macro)
move forward one argument in the enclosing term

```
Example and General Form:
nx
```

For example, if the conclusion is `(= x (* (- y) z))` and the current subterm is `x`, then after executing `nx`, the current subterm will be `(* (- y) z)`.

This is the same as `up` followed by `(dive n+1)`, where `n` is the position of the current subterm in its parent term in the conclusion. Thus in particular, the `nx` command fails if one is already at the top of the conclusion.

See also `up`, `dive`, `top`, and `bk`.

11.48 ACL2-PC::ORELSE

ACL2-PC::ORELSE (macro)

run the first instruction; if (and only if) it "fails", run the
second

```
Example:
(orelse top (print "Couldn't move to the top"))

General form:
(orelse instr1 instr2)
```

Run the first instruction. Then if it "fails", run the second instruction also; otherwise, stop after the first.

This instruction "succeeds" if and only if either `instr1` "succeeds", or else `instr2` "succeeds". If it "fails", then the failure is soft.

11.49 ACL2-PC::P

ACL2-PC::P (macro)
prettyprint the current term

```
Example and General Form:
p
```

Prettyprint the current term. The usual user syntax is used, so that for example one would see `(and x y)` rather than `(if x y 'nil)`. (See also `pp`.) Also, abbreviations are inserted where appropriate; see `add-abbreviation`.

The "current term" is the entire conclusion unless `dive` commands have been given, in which case it may be a subterm of the conclusion.

If all goals have been proved, a message saying so will be printed (as there will be no current `term`!).

11.50 ACL2-PC::P-TOP

ACL2-PC::P-TOP (macro)
prettyprint the conclusion, highlighting the current term

```
Example and General Form:
```

```
p-top
```

For example, if the conclusion is `(equal (and x (p y)) (foo z))` and the current subterm is `(p y)`, then `p-top` will print `(equal (and x (*** (p y) ***)) (foo z))`.

Prettyprint the the conclusion, highlighting the current term. The usual user syntax is used, as with the command `p` (as opposed to `pp`). This is illustrated in the example above, where one would `*not*` see `(equal (if x (*** (p y) ***) 'nil) (foo z))`.

Note (obscure): In some situations, a term of the form `(if x t y)` occurring inside the current subterm will not print as `(or x y)`, when `x` isn't a call of a boolean primitive. There's nothing incorrect about this, however.

11.51 ACL2-PC::PP

ACL2-PC::PP (macro)
prettyprint the current term

```
Example and General Form:
pp
```

This is the same as `p` (see its documentation), except that raw syntax (internal form) is used. So for example, one would see `(if x y 'nil)` rather than `(and x y)`. Abbreviations are however still inserted, as with `p`.

11.52 ACL2-PC::PRINT

ACL2-PC::PRINT (macro)
print the result of evaluating the given form

```
Example:
(print (append '(a b) '(c d)))
Print the list (a b c d) to the terminal

General Form:
(print form)
```

Prettyprints the result of evaluating form. The evaluation of `form` should return a single value that is not `state`.

If the form you want to evaluate does not satisfy the criterion above, you should create an appropriate call of the `lisp` command instead. Notice that this command always returns `(mv nil state state)`.

11.53 ACL2-PC::PRINT-ALL-GOALS

ACL2-PC::PRINT-ALL-GOALS (macro)
print all the (as yet unproved) goals

Example and General Form: print-all-goals

Prints all the goals that remain to be proved, in a pleasant format.

11.54 ACL2-PC::PRINT-MAIN

ACL2-PC::PRINT-MAIN (macro)
print the original goal

```
Example and General Form:
print-main
```

Prints the goal as originally entered.

11.55 ACL2-PC::PRO

ACL2-PC::PRO (atomic macro)
repeatedly apply promote

```
Example and General Form:
pro
```

Apply the `promote` command until there is no change. This command "succeeds" exactly when at least one call of `promote` "succeeds". In that case, only a single new proof-checker state will be created.

11.56 ACL2-PC::PROMOTE

ACL2-PC::PROMOTE (primitive)
move antecedents of conclusion's `implies` term to top-level hypotheses

```
Examples:
promote
(promote t)
```

For example, if the conclusion is `(implies (and x y) z)`, then after execution of `promote`, the conclusion will be `z` and the terms `x` and `y` will be new top-level hypotheses.

```
General Form:
(promote &optional do-not-flatten-flag)
```

Replace conclusion of `(implies hyps exp)` or `(if hyps exp t)` with simply `exp`, adding `hyps` to the list of top-level hypotheses. Moreover, if `hyps` is viewed as a conjunction then each conjunct will be added as a separate top-level hypothesis. An exception is that if `do-not-flatten-flag` is supplied and not `nil`, then only one top-level hypothesis will be added, namely `hyps`.

Note: You must be at the top of the conclusion in order to use this command. Otherwise, first invoke `top`.

11.57 ACL2-PC::PROTECT

ACL2-PC::PROTECT (macro)
run the given instructions, reverting to existing state upon failure

```
Example:
(protect induct p prove)

General Form:
(protect &rest instruction-list)
```

Protect is the same as do-strict, except that as soon as an instruction "fails", the state-stack reverts to what it was before the protect instruction began, and restore is given the same meaning that it had before the protect instruction began. See the documentation for do-strict.

11.58 ACL2-PC::PROVE

ACL2-PC::PROVE (primitive)
call the ACL2 theorem prover to prove the current goal

```
Examples:
prove -- attempt to prove the current goal
(prove :otf-flg t
       :hints (("Subgoal 2" :by foo) ("Subgoal 1" :use bar)))
      -- attempt to prove the current goal, with the indicated hints
         and with OTF-FLG set

General Form:
(prove &rest rest-args)
```

Attempt to prove the current goal, where rest-args is as in the keyword arguments to defthm except that only :hints and :otf-flg are allowed. The command succeeds exactly when the corresponding defthm would succeed, except that it is all right for some goals to be given "bye"s. Each goal given a "bye" will be turned into a new subgoal. (See Section 8.41 [HINTS], page 297 for an explanation of :by hints.)

Note: Use (= t) instead if you are not at the top of the conclusion. Also note that if there are any hypotheses in the current goal, then what is actually attempted is a proof of (implies hyps conc), where hyps is the conjunction of the top-level hypotheses and conc is the goal's conclusion.

Note: It is allowed to use abbreviations in the hints.

11.59 ACL2-PC::PUT

ACL2-PC::PUT (macro)
substitute for a "free variable"

```
Example:
(put x 17)
```

```
General Form:
(put var expr)
```

Substitute `expr` for the "free variable" `var`, as explained below.

A "free variable" is, for our purposes, a variable `var` such that the instruction `(free var)` has been executed earlier in the state-stack. What `(free var)` really does is to let `var` be an abbreviation for the term `(hide var)` (see documentation for `add-abbreviation`). What `(put var expr)` really does is to unwind the state-stack, replacing that `free` instruction with the instruction `(add-abbreviation var expr)`, so that future references to `(? var)` become reference to `expr` rather than to `(hide var)`, and then to replay all the other instructions that were unwound. Because `hide` was used, the expectation is that in most cases, the instructions will replay successfully and `put` will "succeed". However, if any replayed instruction "fails", then the entire replay will abort and "fail", and the state-stack will revert to its value before the `put` instruction was executed.

If `(put var expr)` "succeeds", then `(remove-abbreviation var)` will be executed at the end.

Note: The `restore` command will revert the state-stack to its value present before the `put` instruction was executed.

11.60 ACL2-PC::QUIET

ACL2-PC::QUIET (meta)
run instructions without output

```
Example:
(quiet induct prove)

General Form:
(quiet &rest instruction-list)
```

Run the `instruction-list` through the top-level loop with no output.

See also `noise`.

11.61 ACL2-PC::R

ACL2-PC::R (macro)
same as rewrite

```
Example:
(r 3)

General Form:
(rewrite &optional rule-id substitution
         ;; below are rare arguments, used for disambiguation:
         target-lhs target-rhs target-hyps target-equiv)
```

See the documentation for `rewrite`, as `r` and `rewrite` are identical.

11.62 ACL2-PC::REDUCE

ACL2-PC::REDUCE (atomic macro)
call the ACL2 theorem prover's simplifier

```
Examples:
reduce -- attempt to prove the current goal without using induction
(reduce ("Subgoal 2" :by foo) ("Subgoal 1" :use bar))
       -- attempt to prove the current goal by without using
          induction, with the indicated hints

General Form:
(reduce &rest hints)
```

Attempt to prove the current goal without using induction, using the indicated hints (if any). A subgoal will be created for each goal that would have been pushed for proof by induction in an ordinary proof.

Notice that unlike `prove`, the arguments to `reduce` are spread out, and are all hints.

Note: Induction will be used to the extent that it is ordered explicitly in the hints.

11.63 ACL2-PC::REDUCE-BY-INDUCTION

ACL2-PC::REDUCE-BY-INDUCTION (macro)
call the ACL2 prover without induction, after going into
induction

```
Examples:
reduce-by-induction
  -- attempt to prove the current goal after going into induction,
     with no further inductions

(reduce-by-induction ("Subgoal 2" :by foo) ("Subgoal 1" :use bar))
  -- attempt to prove the current goal after going into induction,
     with no further inductions, using the indicated hints

General Form:
(reduce-by-induction &rest hints)
```

A subgoal will be created for each goal that would have been pushed for proof by induction in an ordinary proof, except that the proof begins with a top-level induction.

Notice that unlike `prove`, the arguments to `reduce-by-induction` are spread out, and are all hints. See also `prove`, `reduce`, and `bash`.

Note: Induction and the various processes will be used to the extent that they are ordered explicitly in the `:induct` and `:do-not` hints.

11.64 ACL2-PC::REMOVE-ABBREVIATIONS

ACL2-PC::REMOVE-ABBREVIATIONS (primitive)
remove one or more abbreviations

```
Examples:
remove-abbreviations -- remove all abbreviations
(remove-abbreviations v w)
                     -- assuming that V and W currently abbreviate
                        terms, then they are ``removed'' in the
                        sense that they are no longer considered to
                        abbreviate those terms

General Forms:
(remove-abbreviations &rest vars)
```

If vars is not empty (i.e., not `nil`), remove the variables in `vars` from the current list of abbreviations, in the sense that each variable in `vars` will no longer abbreviate a term.

Note: The instruction fails if at least one of the arguments fails to be a variable that abbreviates a term.

See also the documentation for `add-abbreviation`, which contains a discussion of abbreviations in general, and `show-abbreviations`.

11.65 ACL2-PC::REPEAT

ACL2-PC::REPEAT (macro)
repeat the given instruction until it "fails"

```
Example:
(repeat promote)

General Form:
(repeat instruction)
```

The given `instruction` is run repeatedly until it "fails".

Note: There is nothing here in general to prevent the instruction from being run after all goals have been proved, though this is indeed the case for primitive instructions.

11.66 ACL2-PC::REPEAT-REC

ACL2-PC::REPEAT-REC (macro)
auxiliary to `repeat`

See documentation for `repeat`.

11.67 ACL2-PC::REPLAY

ACL2-PC::REPLAY (macro)
replay one or more instructions

```
Examples:
REPLAY     -- replay all instructions in the current session
```

```
                (i.e., state-stack)
(REPLAY 5) -- replay the most recent 5 instructions
(REPLAY 5
        (COMMENT deleted dive command here))
           -- replace the 5th most recent instruction with the
              indicated comment instruction, and then replay it
              followed by the remaining 4 instructions

General Form:
(REPLAY &OPTIONAL n replacement-instruction)
```

Replay the last `n` instructions if `n` is a positive integer; else `n` should be `nil` or not supplied, and replay all instructions. However, if `replacement-instruction` is supplied and not `nil`, then before the replay, replace the `nth` instruction (from the most recent, as shown by `commands`) with `replacement-instruction`.

If this command "fails", then the `restore` command will revert the state-stack to its value present before the `replay` instruction was executed.

11.68 ACL2-PC::RESTORE

ACL2-PC::RESTORE (meta)
remove the effect of an UNDO command

```
Example and General Form:
restore
```

`Restore` removes the effect of an `undo` command. This always works as expected if `restore` is invoked immediately after `undo`, without intervening instructions. However, other commands may also interact with `restore`, notably "sequencing" commands such as `do-all`, `do-strict`, `protect`, and more generally, `sequence`.

Note: Another way to control the saving of proof-checker state is with the `save` command; see the documentation for `save`.

The `restore` command always "succeeds"; it returns `(mv nil t state)`.

11.69 ACL2-PC::RETAIN

ACL2-PC::RETAIN (atomic macro)
drop all **but** the indicated top-level hypotheses

```
Example:
(RETAIN 2 3) -- keep the second and third hypotheses, and drop
                the rest

General Form:
(retain &rest args)
```

Drop all top-level hypotheses **except** those with the indicated indices.

There must be at least one argument, and all must be in range (i.e. integers between one and the number of top-level hypotheses, inclusive).

11.70 ACL2-PC::RETRIEVE

ACL2-PC::RETRIEVE (macro)
re-enter the proof-checker

```
Examples:
(retrieve associativity-of-permutationp)
retrieve

General Form:
(retrieve &optional name)
```

Must be used from `outside` the interactive proof-checker loop. If name is supplied and not `nil`, this causes re-entry to the interactive proof-checker loop in the state at which `save` was last executed for the indicated name. (See documentation for `save`.) If `name` is `nil` or is not supplied, then the user is queried regarding which proof-checker state to re-enter. The query is omitted, however, if there only one proof-checker state is present that was saved with `save`, in which case that is the one that is used. See also `unsave`.

11.71 ACL2-PC::REWRITE

ACL2-PC::REWRITE (primitive)
apply a rewrite rule

```
Examples:
(rewrite reverse-reverse)
   -- apply the rewrite rule `reverse-reverse'
(rewrite (:rewrite reverse-reverse))
   -- same as above
(rewrite 2)
   -- apply the second rewrite rule, as displayed by show-rewrites
rewrite
   -- apply the first rewrite rule, as displayed by show-rewrites
(rewrite transitivity-of-< ((y 7)))
   -- apply the rewrite rule transitivity-of-< with the substitution
      that associates 7 to the ``free variable'' y

General Form:
(rewrite &optional rule-id substitution)
```

Replace the current subterm with a new term by applying a rewrite rule. If `rule-id` is a positive integer `n`, then the `n`th rewrite rule as displayed by `show-rewrites` is the one that is applied. If `rule-id` is `nil` or is not supplied, then it is treated as the number 1. Otherwise, `rule-id` should be either a rune of or name of a rewrite rule. If a name is supplied, then any rule of that name may be used. More explanation of all of these points follows below.

Consider first the following example. Suppose that the current subterm is `(reverse (reverse y))` and that there is a rewrite rule called `reverse-reverse` of the form

```
(implies (true-listp x)
         (equal (reverse (reverse x)) x)) .
```

Then the instruction `(rewrite reverse-reverse)` would cause the current subterm to be replaced by `y` and would create a new goal with conclusion `(true-listp y)`. An exception is that if the top-level hypotheses imply `(true-listp y)` using only "trivial reasoning" (more on this below), then no new goal is created.

A rather important point is that if the `rule-id` argument is a number or is not supplied, then the system will store an instruction of the form `(rewrite name ...)`, where `name` is the name of a rewrite rule; this is in order to make it easier to replay instructions when there have been changes to the history. Actually, instead of the name (whether the name is supplied or calculated), the system stores the rune if there is any chance of ambiguity. (Formally, "ambiguity" here means that the rune being applied is of the form `(:rewrite name . index)`, where index is not `nil`.)

Speaking in general, then, a `rewrite` instruction works as follows:

First, a rewrite rule is selected according to the arguments of the `rewrite` instruction. The selection is made as explained above under "General Form" above. The "disambiguating rare arguments" will rarely be of interest to the user; as explained just above, the stored instruction always contains the name of the rewrite rule, so if there is more than one rule of that name then the system creates and stores these extra arguments in order to make the resulting instruction unambiguous, i.e., so that only one rewrite rule applies. For what it's worth, they correspond respectively to the fields of a rewrite rule record named `lhs`, `rhs`, `hyps`, and `equiv`.

Next, the left-hand side of the rule is matched with the current subterm, i.e., a substitution `unify-subst` is found such that if one instantiates the left-hand side of the rule with `unify-subst`, then one obtains the current subterm. If this matching fails, then the instruction fails.

Now an attempt is made to relieve the hypotheses, in much the same sense as the theorem prover relieves hypotheses except that there is no call to the rewriter. Essentially, this means that the substitution `unify-subst` is applied to the hypotheses and the system then checks whether all hypotheses are "clearly" true in the current context. If there are variables in the hypotheses of the rewrite rule that do not occur in the left-hand side of the conclusion even after the user-supplied substitution (default: `nil`) is applied, then a weak attempt is made to extend that substitution so that even those hypotheses can be relieved. However, if even one hypothesis remains unrelieved, then no automatic extension of the substitution is made, and in fact hypotheses that contain even one uninstantiated variable will remain unrelieved.

Finally, the instruction is applied as follows. The current subterm is replaced by applying the final substitution, i.e., the extension of `unify-subst` by the user-supplied substitution which may in turn be extended by the system (as explained above) in order to relieve all hypotheses, to the right-hand side of the selected rewrite rule. And, one new subgoal is created for each unrelieved hypothesis of the rule, whose top-level hypotheses are the governors and top-level hypotheses of the current goal and whose conclusion and current subterm are the instance, by that same final substitution, of that unrelieved hypothesis.

Note: The substitution argument should be a list whose elements have the form `(variable term)`, where `term` may contain abbreviations.

11.72 ACL2-PC::RUN-INSTR-ON-GOAL

ACL2-PC::RUN-INSTR-ON-GOAL (macro)
auxiliary toxae THEN

See documentation for `then`.

11.73 ACL2-PC::RUN-INSTR-ON-NEW-GOALS

ACL2-PC::RUN-INSTR-ON-NEW-GOALS (macro)
auxiliary to `then`

See documentation for `then`.

11.74 ACL2-PC::S

ACL2-PC::S (primitive)
simplify the current subterm

```
Examples:
S  -- simplify the current subterm
(S :backchain-limit 2 :normalize t :expand (append x z))
   -- simplify the current subterm, but during the rewriting
      process first ``normalize'' it by pushing IFs to the
      top-level, and also force the term (append x z) to be
      expanded during the rewriting process

General Form:
(s &key rewrite normalize backchain-limit repeat in-theory hands-off
        expand)
```

Simplify the current subterm according to the keyword parameters supplied. First if-normalization is applied (unless the `normalize` argument is `nil`), i.e., each subterm of the form `(f ... (if test x y) ...)` is replaced by the term `(if test (f ... x ...) (f ... y ...))` except, of course, when `f` is `if` and the indicated `if` subterm is in the second or third argument position. Then rewriting is applied (unless the rewrite argument is `nil`). Finally this pair of actions is repeated — until the rewriting step causes no change in the term. A description of each parameter follows.

```
:rewrite -- default t
```

When non-`nil`, instructs the system to use ACL2's rewriter (or, something close to it) during simplification.

```
:normalize -- default t
```

When non-nil, instructs the system to use if-normalization (as described above) during simplification.

```
:backchain-limit -- default 0
```

Sets the number of recursive calls to the rewriter that are allowed for backchaining. Even with the default of 0, some reasoning is allowed (technically speaking, type-set reasoning is allowed) in the relieving of hypotheses.

```
:repeat -- default 0
```

Sets the number of times the current term is to be rewritten. If this value is t, then the default is used (as specified by the constant *default-s-repeat-limit*).

```
:in-theory, :hands-off, :expand
```

These have their usual meaning; see Section 8.41 [HINTS], page 297.

Note: if conditional rewrite rules are used that cause case splits because of the use of force, then appropriate new subgoals will be created, i.e., with the same current subterm (and address) but with each new (forced) hypothesis being negated and then used to create a corresponding new subgoal. In that case, the current goal will have all such new hypotheses added to the list of top-level hypotheses.

11.75 ACL2-PC::S-PROP

ACL2-PC::S-PROP (atomic macro)
simplify propositionally

```
Example:
s-prop

General Form:
(s-prop &rest names)
```

Simplify, using the default settings for s (which include if-normalization and rewriting without real backchaining), but with respect to a theory in which only basic functions and rules (the ones in *s-prop-theory*), together with the names (or parenthesized names) in the &rest argument names, are enabled.

See also the documentation for `s`.

11.76 ACL2-PC::SAVE

ACL2-PC::SAVE (macro)
save the proof-checker state (state-stack)

```
Example:
(save lemma3-attempt

General Form:
(save &optional name do-it-flg)
```

Saves the current proof-checker state by "associating" it with the given name. Submit `(retrieve name)` to Lisp to get back to this proof-checker state. If `verify` was originally supplied with an event name, then the argument can be omitted in favor of that name as the default.

Note that if a `save` has already been done with the indicated name (or the default event name), then the user will be queried regarding whether to go ahead with the save — except, if `do-it-flg` is supplied and not `nil`, then there will be no query and the `save` will be effected.

See also the documentation for `retrieve` and `unsave`.

11.77 ACL2-PC::SEQUENCE

ACL2-PC::SEQUENCE (meta)
run the given list of instructions according to a multitude of
options

```
Example:
(sequence (induct p prove) t)
```

See also the definitions of commands `do-all`, `do-strict`, `protect`, and `succeed`.

```
General Form:
(sequence instruction-list
          &optional
          strict-flg protect-flg success-expr no-prompt-flg)
```

Each instruction in the list `instruction-list` is run, and the instruction "succeeds" if every instruction in `instruction-list` "succeeds". However, it might "succeed" even if some instructions in the list "fail"; more generally, the various arguments control a number of aspects of the running of the instructions. All this is explained in the paragraphs below. First we embark on a general discussion of the instruction interpreter, including the notions of "succeed" and "fail".

Note: The arguments are **not** evaluated, except (in a sense) for `success-expr`, as described below.

Each primitive and meta instruction can be thought of as returning an error triple (in the standard ACL2 sense), say `(erp val state)`. An instruction (primitive or meta) "succeeds" if `erp` is `nil` and `val` is not `nil`; otherwise it "fails". (When we use the words "succeed" or "fail" in this technical sense, we'll always include them in double quotes.) If an instruction "fails," we say that that the failure is "soft" if `erp` is `nil`; otherwise the failure is "hard". The `sequence` command gives the user control over how to treat "success" and "failure" when sequencing instructions, though we have created a number of handy macro commands for this purpose, notably `do-all`, `do-strict` and `protect`.

Here is precisely what happens when a `sequence` instruction is run. The instruction interpreter is run on the instructions supplied in the argument `instruction-list` (in order). The interpreter halts the first time there is a hard "failure." except that if `strict-flg` is supplied and not `nil`, then the interpreter halts the first time there is any "failure." The error triple `(erp val state)` returned by the `sequence` instruction is the triple returned by the last instruction executed (or, the triple `(nil t state)` if `instruction-list` is `nil`), except for the following provision. If `success-expr` is supplied and not `nil`, then it is evaluated with the state global variables `erp` and `val` (in ACL2 package) bound to the corresponding components of the error triple returned (as described above). At least two values should be returned, and the first two of these will be substituted for `erp` and `val` in the triple finally returned by `sequence`. For example, if `success-expr` is `(mv erp val)`, then no change will be made to the error triple, and if instead it is `(mv nil t)`, then the `sequence` instruction will "succeed".

That concludes the description of the error triple returned by a `sequence` instruction, but it remains to explain the effects of the arguments `protect-flg` and `no-prompt-flg`.

If `protect-flg` is supplied and not `nil` and if also the instruction "fails" (i.e., the error component of the triple is not `nil` or the value component is `nil`), then the state is reverted so that the proof-checker's state (including the behavior of `restore`) is set back to what it was before the `sequence` instruction was executed. Otherwise, unless `no-restore-flg` is set, the state is changed so that the `restore` command will now undo the effect of this `sequence` instruction (even if there were nested calls to `sequence`).

Finally, as each instruction in `instruction-list` is executed, the prompt and that instruction will be printed, unless the global state variable `print-prompt-and-instr-flg` is unbound or `nil` and the parameter `no-prompt-flg` is supplied and not `nil`.

11.78 ACL2-PC::SHOW-ABBREVIATIONS

ACL2-PC::SHOW-ABBREVIATIONS (macro)
display the current abbreviations

```
Examples:
(show-abbreviations v w)
   -- assuming that v and w currently abbreviate terms,
      then this instruction displays them together with
      the terms they abbreviate
show-abbreviations
   -- display all abbreviations
```

See also `add-abbreviation` and `remove-abbreviations`. In particular, the documentation for `add-abbreviation` contains a general discussion of abbreviations.

```
General Form:
(show-abbreviations &rest vars)
```

Display each argument in `vars` together with the term it abbreviates (if any). If there are no arguments, i.e. the instruction is simply `show-abbreviations`, then display all abbreviations together with the terms they abbreviate.

If the term abbreviated by a variable, say `v`, contains a proper subterm that is also abbreviate by (another) variable, then both the unabbreviated term and the abbreviated term (but not using `(? v)` to abbreviate the term) are displayed with together with `v`.

11.79 ACL2-PC::SHOW-REWRITES

ACL2-PC::SHOW-REWRITES (macro)
display the applicable rewrite rules

```
Example:
show-rewrites
```

```
General Form:
(show-rewrites &optional rule-id enabled-only-flg)
```

Display rewrite rules whose left-hand side matches the current subterm. This command is useful in conjunction with `rewrite`. If `rule-id` is supplied and is a name (non-`nil` symbol) or a rune, then only the corresponding rewrite rule(s) will be displayed, while if `rule-id` is a positive integer `n`, then only the `n`th rule that would be in the list is displayed. In each case, the display will point out when a rule is currently disabled (in the interactive environment), except that if `enabled-only-flg` is supplied and not `nil`, then disabled rules will not be displayed at all. Finally, the free variables of any rule (those occurring in the rule that do not occur in the left-hand side of its conclusion) will be displayed. See also the documentation for `rewrite`.

11.80 ACL2-PC::SKIP

ACL2-PC::SKIP (macro)
"succeed" without doing anything

```
Example and General Form:
skip
```

Make no change in the state-stack, but "succeed". Same as `(sequence nil)`.

11.81 ACL2-PC::SL

ACL2-PC::SL (atomic macro)
simplify with lemmas

```
Examples:
sl
(sl 3)

General Form:
(sl &optional backchain-limit)
```

Simplify, but with all function definitions disabled (see Section 15.5 [FUNCTION-THEORY], page 672 in the top-level ACL2 loop), except for a few basic functions (the ones in `*s-prop-theory*`). If `backchain-limit` is supplied and not `nil`, then it should be a nonnegative integer; see `(help s)`.

11.82 ACL2-PC::SPLIT

ACL2-PC::SPLIT (atomic macro)
split the current goal into cases

```
Example:
split
```

For example, if the current goal has one hypothesis `(or x y)` and a conclusion of `(and a b)`, then `split` will create four new goals:

```
one with hypothesis X and conclusion A
one with hypothesis X and conclusion B
one with hypothesis Y and conclusion A
one with hypothesis Y and conclusion B.

General Form:
SPLIT
```

Replace the current goal by subgoals whose conjunction is equivalent (primarily by propositional reasoning) to the original goal, where each such goal cannot be similarly split.

Note: The new goals will all have their hypotheses promoted; in particular, no conclusion will have a top function symbol of `implies`. Also note that `split` will fail if there is exactly one new goal created and it is the same as the existing current goal.

The way `split` really works is to call the ACL2 theorem prover with only simplification (and preprocessing) turned on, and with only a few built-in functions (especially, propositional ones) enabled, namely, the ones in the list `*s-prop-theory*`. However, because the prover is called, type-set reasoning can be used to eliminate some cases. For example, if `(true-listp x)` is in the hypotheses, then probably `(true-listp (cdr x))` will be reduced to `t`.

11.83 ACL2-PC::SR

ACL2-PC::SR (macro)
same as SHOW-REWRITES

```
Example:
sr
```

```
General Form:
(sr &optional rule-id)
```

See the documentation for `show-rewrites`, as `sr` and `show-rewrites` are identical.

11.84 ACL2-PC::SUCCEED

ACL2-PC::SUCCEED (macro)
run the given instructions, and "succeed"

```
Example:
(succeed induct p prove)

General Form:
(succeed &rest instruction-list)
```

Run the indicated instructions until there is a hard "failure", and "succeed". (See the documentation for `sequence` for an explanation of "success" and "failure".)

11.85 ACL2-PC::TH

ACL2-PC::TH (macro)
print the top-level hypotheses and the current subterm

```
Examples:
th               -- print all (top-level) hypotheses and the current
                    subterm
(th (1 3) (2 4)) -- print hypotheses 1 and 3 and governors 2 and 4,
                    and the current subterm
(th (1 3) t)     -- print hypotheses 1 and 3 and all governors, and
                    the current subterm

General Form:
(th &optional hyps-indices govs-indices)
```

Print hypotheses and the current subterm. The printing of hypotheses (and perhaps governors) are controlled as in the `hyps` command; see its documentation.

Historical note: The name `th` is adapted from the Gypsy Verification Environment, where `th` abbreviates the command `theorem`, which says to print information on the current goal.

11.86 ACL2-PC::THEN

ACL2-PC::THEN (macro)
apply one instruction to current goal and another to new subgoals

```
Example:
(then induct prove)

General Form:
(then first-instruction &optional completion must-succeed-flg)
```

Run `first-instruction`, and then run `completion` (another instruction) on each subgoal created by `first-instruction`. If `must-succeed-flg` is supplied and not `nil`, then immediately remove the effects of each invocation of `completion` that "fails".

11.87 ACL2-PC::TOP

ACL2-PC::TOP (atomic macro)
move to the top of the goal

```
Example and General Form:
top
```

For example, if the conclusion is `(= x (* (- y) z))` and the current subterm is `y`, then after executing `top`, the current subterm will be the same as the conclusion, i.e., `(= x (* (- y) z))`.

`Top` is the same as `(up n)`, where `n` is the number of times one needs to execute `up` in order to get to the top of the conclusion. The `top` command fails if one is already at the top of the conclusion.

See also `up`, `dive`, `nx`, and `bk`.

11.88 ACL2-PC::TYPE-ALIST

ACL2-PC::TYPE-ALIST (macro)
display the type-alist from the current context

```
Example and General Form:
type-alist
```

Display the current assumptions as a type-alist. Note that this display includes the result of forward chaining.

11.89 ACL2-PC::UNDO

ACL2-PC::UNDO (meta)
undo some instructions

```
Examples:
(undo 7)
undo

General Forms:

(undo n) -- Undo the last n instructions.  The argument n should be
            a positive integer.

undo     -- Same as (undo 1).
```

Note: To remove the effect of an `undo` command, use `restore`. See the documentation for details.

Note: If the argument `n` is greater than the total number of interactive instructions in the current session, then `(undo n)` will simply take you back to the start of the session.

The `undo` meta command always "succeeds"; it returns `(mv nil t state)` unless its optional argument is supplied and of the wrong type (i.e. not a positive integer) or there are no instructions to undo.

11.90 ACL2-PC::UNSAVE

ACL2-PC::UNSAVE (macro)
remove a proof-checker state

```
Example:
(unsave assoc-of-append)

General Form:
(unsave &optional name)
```

Eliminates the association of a proof-checker state with `name`, if `name` is supplied and not `nil`. The name may be `nil` or not supplied, in which case it defaults to the event name supplied with the original call to `verify` (if there is one — otherwise, the instruction "fails" and there is no change). The ACL2 function `unsave` may also be executed outside the interactive loop, with the same syntax.

See also documentation for `save` and `retrieve`.

11.91 ACL2-PC::UP

ACL2-PC::UP (primitive)
move to the parent (or some ancestor) of the current subterm

```
Examples:  if the conclusion is (= x (* (- y) z)) and the
           current subterm is y, then we have:
up or (up 1) -- the current subterm becomes (- y)
(up 2)       -- the current subterm becomes (* (- y) z)
(up 3)       -- the current subterm becomes the entire conclusion
(up 4)       -- no change; can't go up that many levels

General Form:
(up &optional n)
```

Move up `n` levels in the conclusion from the current subterm, where `n` is a positive integer. If `n` is not supplied or is `nil`, then move up 1 level, i.e., treat the instruction as `(up 1)`.

See also `dive`, `top`, `nx`, and `bk`.

11.92 ACL2-PC::USE

ACL2-PC::USE (atomic macro)
use a lemma instance

```
Example:
(USE true-listp-append
     (:instance assoc-of-append (x a) (y b) (z c)))
-- Add two top-level hypotheses, one the lemma called
   true-listp-append, and the other an instance of the lemma called
   assoc-of-append by the substitution in which x is assigned a, y
```

```
   is assigned b, and z is assigned c.

General Form:
(use &rest args)
```

Add the given lemma instances to the list of top-level hypotheses. See Section 8.41 [HINTS], page 297 for the syntax of `:use` hints in `defthm`, which is essentially the same as the syntax here (see the example above).

This command calls the `prove` command, and hence should only be used at the top of the conclusion.

11.93 ACL2-PC::X

ACL2-PC::X (atomic macro)
expand and (maybe) simplify function call at the current subterm

```
Examples:
x --  expand and simplify.
```

For example, if the current subterm is (append a b), then after `x` the current subterm will probably be (cons (car a) (append (cdr a) b)) if (consp a) and (true-listp a) are among the top-level hypotheses and governors. If there are no top-level hypotheses and governors, then after `x` the current subterm will probably be:

```
(if (true-listp x)
    (if x
        (cons (car x) (append (cdr x) y))
      y)
  (apply 'binary-append (list x y))).

General Form:
(X &key
   rewrite normalize backchain-limit in-theory hands-off expand)
```

Expand the function call at the current subterm, and simplify using the same conventions as with the `s` command (see documentation for `s`).

Unlike `s`, it is permitted to set both `:rewrite` and `:normalize` to `nil`, which will result in no simplification; see `x-dumb`.

Note (obscure): On rare occasions the current address may be affected by the use of `x`. For example, suppose we have the definition

```
(defun g (x) (if (consp x) x 3))
```

and then we enter the proof-checker with

```
(verify (if (integerp x) (equal (g x) 3) t)) .
```

Then after invoking the instruction `(dive 2 1)`, so that the current subterm is `(g x)`, followed by the instruction `x`, we would expect the conclusion to be `(if (integerp x) (equal 3 3) t)`. However, the system actually replaces `(equal 3 3)` with `t` (because we use the ACL2 term-forming primitives), and hence the conclusion is actually `(if (integerp x) (equal 3 3) t)`. Therefore, the current address is put at `(2)` rather than `(2 1)`. In such cases, a warning "`NOTE`" will be printed to the terminal.

The other primitive commands to which the above "truncation" note applies are `equiv`, `rewrite`, and `s`.

11.94 ACL2-PC::X-DUMB

ACL2-PC::X-DUMB (atomic macro)
expand function call at the current subterm, without simplifying

```
Example:
x-dumb:  expand without simplification.

General Form:
(x-dumb &optional new-goals-flg keep-all-guards-flg)
```

Same as `(expand t new-goals-flg keep-all-guards-flg)`. See documentation for `expand`.

See also `x`, which allows simplification.

11.95 DEFINE-PC-HELP

DEFINE-PC-HELP define a macro command whose purpose is to print something

```
Example:
(define-pc-help pp ()
  (if (goals t)
      (fms0 "~|~q0~|"
            (list (cons #0
                        (fetch-term (conc t)
                                    (current-addr t)))))
    (print-all-goals-proved-message state)))

General Form:
(define-pc-help name args &rest body)
```

This defines a macro command named `name`, as explained further below. The `body` should (after removing optional declarations) have multiplicity 1 and `state-out t`, i.e. it should return a single value of `state`. Typically, it will just print something.

What `(define-pc-help name args &rest body)` really does is to create a call of `define-pc-macro` that defines `name` to take arguments `args`, to have the declarations indicated by all but the last form in `body`, and to have a body that (via `pprogn`) first executes the form in the last element of body and then returns a call to the command `skip` (which will return `(mv nil t state)`).

11.96 INSTRUCTIONS

INSTRUCTIONS instructions to the proof checker

```
Example:
:instructions (induct prove promote (dive 1) x
                      (dive 2) = top (drop 2) prove)
```

See Section 6.15 [DEFTHM], page 168 for the role of `:instructions` in place of `:hints^`. As illustrated by the example above, the value associated with `:instructions` is a list of proof-checker^ commands. At the moment the best way to understand the idea of the interactive proof-checker (see Chapter 11 [PROOF-CHECKER], page 515 and see Section 11.101 [VERIFY], page 577) is probably to read the first 11 pages of CLI Technical Report 19, which describes the corresponding facility for Nqthm.

When inside the interactive loop (i.e., after executing `verify^`), use `help^` to get a list of legal instructions and `(help instr)` to get help for the instruction `instr`.

11.97 MACRO-COMMAND

MACRO-COMMAND compound command for the proof-checker

The proof-checker (see Chapter 11 [PROOF-CHECKER], page 515) allows the user to supply interactive commands. Compound commands, called macro commands, may be defined; these expand into zero or more other commands. Some of these are "atomic" macro commands; these are viewed as a single command step when completed successfully.

More documentation^ will be written on the proof-checker^. For now, we simply point out that there are lots of examples of the use of `define-pc-macro` and `define-pc-atomic-macro` in the ACL2 source file `"proof-checker-b.lisp"`. The former is used to create macro commands, which can be submitted to the interactive loop (see Section 11.101 [VERIFY], page 577) and will "expand" into zero or more commands. The latter is similar, except that the undoing mechanism (see Section 11.89 [ACL2-PC::UNDO], page 571) understands atomic macro commands to represent single interactive commands. Also see Section 11.12 [ACL2-PC::COMM], page 527 and see Section 11.13 [ACL2-PC::COMMANDS], page 528 for a discussion of the display of interactive commands.

Also see Section 11.99 [TOGGLE-PC-MACRO], page 576 for how to change a macro command to an atomic macro command, and vice versa.

11.98 RETRIEVE

RETRIEVE re-enter a (specified) proof-checker^ state

```
Examples:
(retrieve associativity-of-permutationp)
retrieve

General Form:
(retrieve &optional name)
```

See Section 11.70 [ACL2-PC::RETRIEVE], page 559, or use `(help retrieve)` inside the interactive proof-checker^ loop. Also see Section 11.100 [UNSAVE], page 577.

11.99 TOGGLE-PC-MACRO

TOGGLE-PC-MACRO change an ordinary macro command to an atomic macro, or vice-versa

```
Example:
(toggle-pc-macro pro)
```

Change `pro` from an atomic macro command to an ordinary one (or vice-versa, if `pro` happens to be an ordinary macro command)

```
General Form:
(toggle-pc-macro name &optional new-tp)
```

If name is an atomic macro command then this turns it into an ordinary one, and vice-versa. However, if `new-tp` is supplied and not `nil`, then it should be the new type, or else there is no change.

11.100 UNSAVE

UNSAVE remove a proof-checker^ state

```
Example:
(unsave assoc-of-append)

General Form:
(unsave name)
```

Eliminates the association of a proof-checker^ state with `name`. See Section 11.100 [UNSAVE], page 577 or see Section 11.90 [ACL2-PC::UNSAVE], page 571.

Also see Section 11.76 [ACL2-PC::SAVE], page 564 and see Section 11.98 [RETRIEVE], page 576.

11.101 VERIFY

VERIFY enter the interactive proof checker

For proof-checker command summaries, see Chapter 11 [PROOF-CHECKER], page 515.

```
Examples:
(VERIFY (implies (and (true-listp x) (true-listp y))
                 (equal (append (append x y) z)
                        (append x (append y z)))))
```

```
   -- Attempt to prove the given term interactively.

(VERIFY (p x)
        :event-name p-always-holds
        :rule-classes (:rewrite :generalize)
        :instructions ((rewrite p-always-holds-lemma)
                       change-goal))
   -- Attempt to prove (p x), where the intention is to call the
      resulting DEFTHM event by the name p-always-holds, with
      rule-classes as indicated.  The two indicated instructions
      will be run immediately to start the proof.

(VERIFY)
   -- Re-enter the proof-checker in the state at which is was last
      left.

General Form:
(VERIFY &OPTIONAL raw-term
        &KEY
        event-name
        rule-classes
        instructions)
```

`Verify` is the function used for entering the proof-checker^'s interactive loop.

12 PROOF-TREE

PROOF-TREE proof tree displays

A view of ACL2 proofs may be obtained by way of "proof tree displays." Environments such as emacs may be customized to provide window-based versions of proof tree displays. (See for example documentation of the emacs function start-proof-tree^, co-written with Michael K. Smith, that the implementors are happy to make available.) Proof tree displays may be turned on with the command `:start-proof-tree^` and may be turned off with the command `:start-proof-tree^`; see Section 12.4 [START-PROOF-TREE], page 588 and see Section 12.5 [STOP-PROOF-TREE], page 588.

Here is an example of a proof tree display, with comments. Lines marked with "c" are considered "checkpoints," i.e., goals whose scrutiny may be of particular value.

```
( DEFTHM PLUS-TREE-DEL ...)    ;currently proving PLUS-TREE-DEL
   1 Goal preprocess   ;"Goal" creates 1 subgoal by preprocessing
   2 |  Goal' simp     ;"Goal'" creates 2 subgoals by simplification
c  0 |  |  Subgoal 2 PUSH *1   ;"Subgoal 2" pushes "*1" for INDUCT
++++++++++++++++++++++++++++++ ;first pass thru waterfall completed
c  6 *1 INDUCT                 ;Proof by induction of "*1" has
     |  <5 more subgoals>      ; created 6 top-level subgoals.  At
                               ; this point, one of those 6 has been
                               ; proved, and 5 remain to be proved.
                               ; We are currently working on the
                               ; first of those 5 remaining goals.
```

See Section 12.3 [PROOF-TREE-EXAMPLES], page 582 for many examples that contain proof tree displays. But first, we summarize the kinds of lines that may appear in a proof tree display. The simplest form of a proof tree display is a header showing the current event, followed by list of lines, each having one of the following forms.

```
n <goal> <process> ...
```

Says that the indicated goal created `n` subgoals using the indicated process. Here "..." refers to possible additional information.

```
c   n <goal> <process> ...
```

As above, but calls attention to the fact that this goal is a "checkpoint" in the sense that it may be of particular interest. Some displays may overwrite "c" with ">" to indicate the current checkpoint being shown in the proof transcript.

```
|  <goal> ...
|  |  <k subgoals>
```

Indicates that the goal just above this line, which is pointed to by the rightmost vertical bar ("`|`"), has `k` subgoals, none of which have yet been processed.

```
|  <goal> ...
|  |  <k more subgoals>
```

As above, except that some subgoals have already been processed.

```
++++++++++++++++++++++++++++++
```

Separates successive passes through the "waterfall". Thus, this "fencepost" mark indicates the start of a new proof by induction or of a new forcing round.

See Section 12.3 [PROOF-TREE-EXAMPLES], page 582 for detailed examples. To learn how to turn off proof tree displays or to turn them back on again, see Section 12.5 [STOP-PROOF-TREE], page 588 and see Section 12.4 [START-PROOF-TREE], page 588, respectively. See Section 12.1 [CHECKPOINT-FORCED-GOALS], page 580 to learn how to mark goals as checkpoints that force^ the creation of goals in forcing rounds. Finally, see Section 12.2 [PROOF-TREE-DETAILS], page 581 for some points not covered elsewhere.

12.1 CHECKPOINT-FORCED-GOALS

CHECKPOINT-FORCED-GOALS Cause forcing goals to be checkpointed in proof trees

```
Example forms:
(checkpoint-forced-goals t)
```

```
(checkpoint-forced-goals nil)
```

Also see Chapter 12 [PROOF-TREE], page 579.

By default, goals are not marked as checkpoints by a proof tree display (as described in the documentation for proof-tree^) merely because they force^ some hypotheses, thus possibly contributing to a forcing round. However, some users may want such behavior, which will occur once the command `(checkpoint-forced-goals) t`) has been executed. To return to the default behavior, use the command `(checkpoint-forced-goals nil)`.

12.2 PROOF-TREE-DETAILS

PROOF-TREE-DETAILS   proof tree details not covered elsewhere

See Chapter 12 [PROOF-TREE], page 579 for an introduction to proof trees, and for a list of related topics. Here we present some details not covered elsewhere.

1. When proof tree display is enabled (because the command `:stop-proof-tree^` has not been executed, or has been superseded by a later `:start-proof-tree^` command), then time summaries will include the time for proof tree display. This time includes the time spent computing with proof trees, such as the pruning process described briefly above. Even when proof trees are not displayed, such as when their display is turned off in the middle of a proof, this time will be printed if it is not 0.

2. When a goal is given a `:bye` in a proof (see Section 8.41 [HINTS], page 297), it is treated for the purpose of proof tree display just as though it had been proved.

3. Several state^ global variables affect proof tree display. `(@ proof-tree-indent)` is initially the string `"| "`: it is the string that is laid down the appropriate number of times to effect indentation. `(@ proof-tree-buffer-width)` is initially the value of `(fmt-soft-right-margin state)`, and is used to prevent printing of the annotation "(force^d ...)" in any greater column than this value. However, `(assign proof-tree-buffer-width nil)` to avoid any such suppression. Finally, `(@ checkpoint-processors)` is a list of processors from the constant list `*preprocess-clause-ledge*`, together with `:induct`. You may remove elements of `(@ checkpoint-processors)` to limit which processes are considered checkpoints.

4. When `:otf-flg^` is not set to `t` in a proof, and the prover then decides to revert to the original goal and prove it by induction, the proof tree display will reflect this fact as shown here:

```
c  0 |  |  Subgoal 2 PUSH (reverting)
```

5. `Proof-tree^` display is turned off during calls of `certify-book^`.

6. The usual failure^ message is printed as part of the prooftree display when a proof has failed.

12.3 PROOF-TREE-EXAMPLES

PROOF-TREE-EXAMPLES   proof tree example

See Chapter 12 [PROOF-TREE], page 579 for an introduction to proof trees, and for a list of related topics. Here we present a detailed example followed by a shorter example that illustrates proof by induction.

Consider the guard^ proof for the definition of a function `cancel_equal_plus`; the body of this definition is of no importance here. The first proof tree display is:

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
     |  <18 subgoals>
```

This is to be read as follows.

> At this stage of the proof we have encountered the top-level goal, named "Goal", which generated 18 subgoals using the “preprocess” process. We have not yet begun to work on those subgoals.

The corresponding message from the ordinary prover output is:

> By case analysis we reduce the conjecture to the following 18 conjectures.

Note that the field just before the name of the goal (`"Goal"`), which here contains the number 18, indicates the number of cases (children) created by the goal using the indicated process. This number will remain unchanged as long as this goal is displayed.

The next proof tree display is:

```
( DEFUN CANCEL_EQUAL_PLUS ...)
```

```
  18 Goal preprocess
   1 |  Subgoal 18 simp
     |  |  <1 subgoal>
     |  <17 more subgoals>
```

which indicates that at this point, the prover has used the simplification ("simp") process on Subgoal 18 to create one subgoal ("<1 subgoal>"). The vertical bar ("|") below "Subgoal 18", accompanied by the line below it, signifies that there are 17 siblings of Subgoal 18 that remain to be processed.

The next proof tree displayed is:

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
   1 |  Subgoal 18 simp
c  2 |  |  Subgoal 18' ELIM
     |  |  |  <2 subgoals>
     |  <17 more subgoals>
```

Let us focus on the fourth line of this display:

```
c  2 |  |  Subgoal 18' ELIM
```

The "c" field marks this goal as a "checkpoint", i.e., a goal worthy of careful scrutiny. In fact, any goal that creates children by a process other than "preprocess" or "simp" is marked as a checkpoint. In this case, the destructor-elimination ("ELIM^") process has been used to create subgoals of this goal. The indentation shows that this goal, Subgoal 18', is a child of Subgoal 18. The number "2" indicates that 2 subgoals have been created (by ELIM^). Note that this information is consistent with the line just below it, which says "<2 subgoals>".

Finally, the last line of this proof tree display,

```
     |  <17 more subgoals>
```

is connected by vertical bars ("|") up to the string `"Subgoal 18"`, which suggests that there are 17 immediate subgoals of Goal remaining to process after Subgoal 18. Note that this line is indented one level from the second line, which is the line for the goal named `"Goal"`. The display is intended to suggest that the subgoals of Goal that remain to be proved consist of Subgoal 18 together with 17 more subgoals.

The next proof tree display differs from the previous one only in that now, Subgoal 18' has only one more subgoal to be processed.

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
   1 |  Subgoal 18 simp
c  2 |  |  Subgoal 18' ELIM
     |  |  |  <1 more subgoal>
     |  <17 more subgoals>
```

Note that the word "more" in "<1 more subgoal>" tells us that there was originally more than one subgoal of Subgoal 18. In fact that information already follows from the line above, which (as previously explained) says that Subgoal 18' originally created 2 subgoals.

The next proof tree display occurs when the prover completes the proof of that "1 more subgoal" referred to above.

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
     |  <17 more subgoals>
```

Then, Subgoal 17 is processed and creates one subgoal, by simplification:

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
   1 |  Subgoal 17 simp
     |  |  <1 subgoal>
     |  <16 more subgoals>
```

... and so on.

Later in the proof one might find the following successive proof tree displays.

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
     |  <9 more subgoals>

( DEFUN CANCEL_EQUAL_PLUS ...)

  18 Goal preprocess
   0 |  Subgoal 9 simp (FORCED)
     |  <8 more subgoals>
```

These displays tell us that Subgoal 9 simplified to `t` (note that the “0” shows clearly that no subgoals were created), but that some rule’s hypotheses were force^d. Although this goal is not checkpointed (i.e., no “c” appears on the left margin), one can cause such goals to be checkpointed; see Section 12.1 [CHECKPOINT-FORCED-GOALS], page 580.

In fact, the proof tree displayed at the end of the “main proof” (the 0-th forcing round) is as follows.

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
   0 |  Subgoal 9 simp (FORCED)
   0 |  Subgoal 8 simp (FORCED)
   0 |  Subgoal 7 simp (FORCED)
   0 |  Subgoal 6 simp (FORCED)
   0 |  Subgoal 4 simp (FORCED)
   0 |  Subgoal 3 simp (FORCED)
```

This is followed by the following proof tree display at the start of the forcing round.

```
  18 Goal preprocess
   0 |  Subgoal 9 simp (FORCED [1]Subgoal 4)
   0 |  Subgoal 8 simp (FORCED [1]Subgoal 6)
   0 |  Subgoal 7 simp (FORCED [1]Subgoal 1)
   0 |  Subgoal 6 simp (FORCED [1]Subgoal 3)
   0 |  Subgoal 4 simp (FORCED [1]Subgoal 5)
   0 |  Subgoal 3 simp (FORCED [1]Subgoal 2)
++++++++++++++++++++++++++++++
   6 [1]Goal FORCING-ROUND
   2 |  [1]Subgoal 6 preprocess
     |  |  <2 subgoals>
     |  <5 more subgoals>
```

This display shows which goals to “blame” for the existence of each goal in the forcing round. For example, Subgoal 9 is to blame for the creation of [1]Subgoal 4.

Actually, there is no real goal named `"[1]Goal"`. However, the line

```
   6 [1]Goal FORCING-ROUND
```

appears in the proof tree display to suggest a “parent” of the six top-level goals in that forcing round. As usual, the numeric field before the goal name contains the original number of children of that (virtual, in this case) goal — in this case, 6.

In our example proof, Subgoal 6 eventually gets proved, without doing any further forcing. At that point, the proof tree display looks as follows.

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
   0 |  Subgoal 9 simp (FORCED [1]Subgoal 4)
   0 |  Subgoal 7 simp (FORCED [1]Subgoal 1)
   0 |  Subgoal 6 simp (FORCED [1]Subgoal 3)
   0 |  Subgoal 4 simp (FORCED [1]Subgoal 5)
   0 |  Subgoal 3 simp (FORCED [1]Subgoal 2)
++++++++++++++++++++++++++++++
   6 [1]Goal FORCING-ROUND
     |  <5 more subgoals>
```

Notice that the line for Subgoal 8,

```
   0 |  Subgoal 8 simp (FORCED [1]Subgoal 6)
```

no longer appears. That is because the goal [1]Subgoal 6 has been proved, along with all its children; and hence, the proof of Subgoal 8 no longer depends on any further reasoning.

The final two proof tree displays in our example are as follows.

```
( DEFUN CANCEL_EQUAL_PLUS ...)
  18 Goal preprocess
   0 |  Subgoal 7 simp (FORCED [1]Subgoal 1)
++++++++++++++++++++++++++++++
   6 [1]Goal FORCING-ROUND
   2 |  [1]Subgoal 1 preprocess
   1 |  |  [1]Subgoal 1.1 preprocess
   1 |  |  |  [1]Subgoal 1.1' simp
c  3 |  |  |  |  [1]Subgoal 1.1'' ELIM
     |  |  |  |  |  <1 more subgoal>

( DEFUN CANCEL_EQUAL_PLUS ...)
<<PROOF TREE IS EMPTY>>
```

The explanation for the empty proof tree is simple: once [1]Subgoal 1.1.1 was proved, nothing further remained to be proved. In fact, the much sought-after "Q.E.D." appeared shortly after the final proof tree was displayed.

Let us conclude with a final, brief example that illustrates proof by induction. Partway through the proof one might come across the following proof tree display.

```
( DEFTHM PLUS-TREE-DEL ...)
   1 Goal preprocess
   2 |  Goal' simp
c  0 |  |  Subgoal 2 PUSH *1
     |  |  <1 more subgoal>
```

This display says that in the attempt to prove a theorem called `plus-tree-del`, preprocessing created the only child Goal' from Goal, and Goal' simplified to two subgoals. Subgoal 2 is immediately pushed for proof by induction, under the name "*1". In fact if Subgoal 1 simplifies to `t`, then we see the following successive proof tree displays after the one shown above.

```
( DEFTHM PLUS-TREE-DEL ...)
   1 Goal preprocess
   2 |  Goal' simp
c  0 |  |  Subgoal 2 PUSH *1

( DEFTHM PLUS-TREE-DEL ...)
   1 Goal preprocess
   2 |  Goal' simp
c  0 |  |  Subgoal 2 PUSH *1
++++++++++++++++++++++++++++++
c  6 *1 INDUCT
     |  <5 more subgoals>
```

The separator "+++++..." says that we are beginning another trip through the waterfall. In fact this trip is for a proof by induction (as opposed to a forcing round), as indicated by the word "INDUCT". Apparently *1.6 was proved immediately, because it was not even displayed; a goal is only displayed when there is some work left to do either on it or on some goal that it brought (perhaps indirectly) into existence.

Once a proof by induction is completed, the "PUSH" line that refers to that proof is eliminated ("pruned"). So for example, when the present proof by induction is completed, the line

```
c  0 |  |  Subgoal 2 PUSH *1
```

is eliminated, which in fact causes the lines above it to be eliminated (since they no longer refer to unproved children). Hence, at that point one might expect to see:

```
( DEFTHM PLUS-TREE-DEL ...)
<<PROOF TREE IS EMPTY>>
```

However, if the proof by induction of *1 necessitates further proofs by induction or a forcing round, then this "pruning" will not yet be done.

12.4 START-PROOF-TREE

START-PROOF-TREE start displaying proof trees during proofs

Also see Chapter 12 [PROOF-TREE], page 579 and see Section 12.5 [STOP-PROOF-TREE], page 588. Note that `:start-proof-tree` works by removing `'proof-tree^` from the `inhibit-output-lst`; see Section 9.13 [SET-INHIBIT-OUTPUT-LST], page 387.

Proof tree displays are explained in the documentation for proof-tree^. `:start-proof-tree` causes proof tree display to be turned on, once it has been turned off by `:stop-proof-tree^`.

Do not attempt to invoke `start-proof-tree` during an interrupt in the middle of a proof.

12.5 STOP-PROOF-TREE

STOP-PROOF-TREE stop displaying proof trees during proofs

Also see Chapter 12 [PROOF-TREE], page 579 and see Section 12.4 [START-PROOF-TREE], page 588. Note that `:stop-proof-tree` works by adding `'proof-tree^` to the `inhibit-output-lst`; see Section 9.13 [SET-INHIBIT-OUTPUT-LST], page 387.

Proof tree displays are explained in the documentation for proof-tree^. `:Stop-proof-tree` causes proof tree display to be turned off.

It is permissible to submit the form `(stop-proof-tree)` during a break. Thus, you can actually turn off proof tree display in the middle of a proof by interrupting ACL2 and submitting the form `(stop-proof-tree)` in raw Lisp.

13 RELEASE-NOTES

RELEASE-NOTES pointers to what has changed

* NOTE1:: Acl2 Version 1.1 Notes
* NOTE2:: Acl2 Version 1.2 Notes
* NOTE3:: Acl2 Version 1.3 Notes
* NOTE4:: Acl2 Version 1.4 Notes
* NOTE5:: Acl2 Version 1.5 Notes
* NOTE6:: Acl2 Version 1.6 Notes
* NOTE7:: ACL2 Version 1.7 (released October 1994) Notes
* NOTE8:: ACL2 Version 1.8 (May, 1995) Notes
* NOTE8-UPDATE:: ACL2 Version 1.8 (Summer, 1995) Notes

This section of the online documentation^ contains notes on the changes that distinguish successive released versions of ACL2.

The current version of ACL2 is the value of the constant `*acl2-version*`.

13.1 NOTE1

NOTE1 Acl2 Version 1.1 Notes

The new features are extensively documented. The relevant topics are:

Related topics other than immediate subtopics:
* BOOKS:: files of ACL2 event forms
* GUARD:: restricting the domain of a function
* MORE:: your response to `:doc^` or `:more-doc^`'s "`(cont'd)`"
* REDUNDANT-EVENTS:: allowing a name to be introduced "twice"

It is especially important to read all of of the documentation^ for books^ before trying to use books. However, the new `:more^` keyword command is so handy for reading long documentation^ strings that we recommend you start with `:doc^` more if reading at the terminal. Some documentation has been written for guard^s which you might find interesting.

13.2 NOTE2

NOTE2 Acl2 Version 1.2 Notes

Hacker mode has been eliminated and programming^ mode has been added. Programming^ mode is unsound but does syntax checking and permits redefinitions of names. See `:doc^ load-mode` and `:doc^ g-mode`.

The arguments to `ld^` have changed. `Ld^` is now much more sophisticated. See Section 9.7 [LD], page 380.

For those occasions on which you wish to look at a large list structure that you are afraid to print, try `(walkabout x state)`, where `x` is an Acl2 expression that evaluates to the structure in question. I am afraid there is no documentation^ yet, but it is similar in spirit to the Interlisp structure editor. You are standing on an object and commands move you around in it. E.g., 1 moves you to its first element, 2 to its second, etc.; 0 moves you up to its parent; `nx` and `bk` move you to its next sibling and previous sibling; `pp` prettyprints it; `q^` exits returning `nil`; `=^` exits returning the thing you're standing on; `(= symb)` assigns the thing you're standing on to the state^ global variable `symb`.

Several new hints^ have been implemented, including `:by` and `:do-not`. The old `:do-not-generalize` has been scrapped in favor of such new hints^ as `:do-not (generalize elim)`. `:By` lets you say "this goal is subsumed by" a given lemma instance. The `:by` hint also lets you say "this goal can't be proved yet but skip it and see how the rest of the proof goes." See Section 8.41 [HINTS], page 297.

13.3 NOTE3

NOTE3 Acl2 Version 1.3 Notes

Programming^ mode has been eliminated. Instead, all functions have a "color" which indicates what can be done with the function. For example, `:red` functions can be executed but have no axioms describing them. Thus, `:red` functions can be introduced after passing a simple syntactic check and they can be redefined without undoing. But nothing of consequence can be proved about them. At the other extreme are `:gold` functions which can be executed and which also have passed both the termination and the guard^ verification proofs. The color of a function can be specified with the new `xargs^` keyword, `:color`, which, if omitted defaults to the global setting of `ld-color`. `Ld-color` replaces `load-mode`. Setting `ld-color` to `:red` causes behavior similar to the old `:g-mode`. Setting `ld-color` to `:gold` causes behavior similar to the old `:v-mode`. It is possible to prototype your system in `:red` and then convert `:red` functions to `:blue` individually by calling `verify-termination^` on them. They can then be converted to `:gold` with `verify-guards^`. This allows us to undertake to verify the termination and guard^s of system functions. See `:doc^` color for an introduction to the use of colors.

Type prescription rules have been added. Recall that in Nqthm, some `rewrite^` rules were actually stored as "type-prescription^s." Such rules allow the user to inform Nqthm's primitive type mechanism as to the kinds of shells returned by a function. Earlier versions of Acl2 did not have an analogous kind of rule because Acl2's type mechanism is complicated by guard^s. Version 1.3 supports `type-prescription^` rules. See Section 14.15 [TYPE-PRESCRIPTION], page 659.

Three more new rule-classes^ implement congruence-based rewriting. It is possible to identify a binary relation as an equivalence relation (see Section 14.6 [EQUIVALENCE], page 632), to show that one equivalence relation refines another (see Section 14.13 [REFINEMENT], page 656) and to show that a given equivalence relation is maintained when rewriting a given function call, e.g., `(fn ...xk...)`, by maintaining another equivalence relation while rewriting the `k`th argument (see Section 14.3 [CONGRUENCE], page 627). If `r` has been shown to be an equivalence^ relation and then `(implies hyps (r (foo x) (bar x)))` is proved as a `:rewrite^` rule, then instances of `(foo x)` will be replaced by corresponding instances of `(bar x)` provided the instance occurs in a slot where the maintainence of `r-equivalence` is known to be sufficient and `hyps` can be established as usual.

In Version 1.2, rule-classes^ were simple keywords, e.g., `:rewrite^` or `:elim^`. In Version 1.3, rule-classes^ have been elaborated to allow you to specify how the theorem ought to be used as a rule. That is, the new rule-classes^ allows you to separate the mathematical statement of the formula from its interpretation as a rule. See Chapter 14 [RULE-CLASSES], page 617.

Rules used to be named by symbols, e.g., `car^` and `car-cons` were the names of rules. Unfortunately, this was ambiguous because there are three rules associated with function symbols: the symbolic definition, the executable counterpart, and the type-prescription^; many different rules might be associated with theorems, depending on the rule classes. In Version 1.3 rules are named by "rune^s" (which is just short hand for "rule names"). Example rune^s are `(:definition car)`, `(:executable-counterpart car)`, and `(:type-prescription car . 1)`. Every rule added by an event has a different name and you can enable^ and disable^ them independently. See Section 15.10 [RUNE], page 675 and see Chapter 15 [THEORIES], page 667.

The identity function `force^`, of one argument, has been added and given a special interpretation by the functions responsible for establishing hypotheses in backchaining: When the system fails to establish some hypothesis of the form `(force term)`, it simply assumes it is true and goes on, delaying until later the establishment of term. In particular, pushes a new subgoal to prove term in the current context. When that subgoal is attacked, all of the resources of the theorem prover, not just rewriting, are brought to bear. Thus, for example, if you wish to prove the rule `(implies (good-statep s) (equal (exec s n) s'))` and it is your expectation that every time `exec` appears its first argument is a `good-statep` then you might write the rule as `(implies (force (good-statep s)) (equal (exec s n) s'))`. This rule is essentially an unconditional rewrite of `(exec s n)` to `s'` that spawns the new goal `(good-statep s)`. See Section 8.35 [FORCE], page 277. Because you can now specify independently how a theorem is used as a rule, you need not write the `force^` in the actual theorem proved. See Chapter 14 [RULE-CLASSES], page 617.

Version 1.3 supports a facility similar to Nqthm's `break-lemma^`. See Chapter 4 [BREAK-REWRITE], page 113. You can install "monitor^s" on rune^s that will cause interactive breaks under certain conditions.

Acl2 also provides "wormhole^s" which allow you to write functions that cause interaction with the user but which do not require that you have access to `state^`. See Section 8.90 [WORMHOLE], page 364.

The rewriter now automatically backchains to stronger recognizers. There is no user hook to this feature but it may simplify some proofs with which older versions of Acl2 had trouble. For example, if the rewriter is trying to prove `(rationalp (foo a b c))` it is now smart enough to try lemmas that match with `(integerp (foo a b c))`.

13.4 NOTE4

NOTE4 Acl2 Version 1.4 Notes

Once again `ld^` only takes one required argument, as the `bind-flg` has been deleted.

Three commands have been added in the spirit of `:pe^`. `:Pe!^` is similar to `:pe^` but it prints all events^ with the given name, rather than just the most recent. The command `:pf^` prints the corollary formula corresponding to a name or rune^. The command `:pl^` (print lemmas) prints rules whose top function symbol is the given name. See Section 7.8 [PE!], page 210, see Section 7.9 [PF], page 210, and see Section 7.10 [PL], page 211.

Book naming conventions have been changed somewhat. The once-required `.lisp` extension is now prohibited! Directories are supported, including a notion of "connected book directory". See Section 3.3 [BOOK-NAME], page 95. Also, the second argument of `certify-book^` is now optional, defaulting to `0`.

Compilation^ is now supported inside the Acl2 loop. See Section 9.5 [COMP], page 379 and see Section 6.27 [SET-COMPILE-FNS], page 187.

The default color is now part of the Acl2 world^; see `:doc^ default-color`. `Ld-color` is no longer an `ld^` special. Instead, colors are events^; see the documentation for `red`, `pink`, `blue`, and `gold`.

A table^ exists for controlling whether Acl2 prints comments when it force^s hypotheses of rules; see `:doc^ force-table`. Also, it is now possible to turn off the forcing of assumptions by disabling the definition of force^; see Section 8.35 [FORCE], page 277.

The event `defconstant` is no longer supported, but a very similar event, `defconst^`, has been provided in its place. See Section 6.5 [DEFCONST], page 158.

The event for defining congruence^ relations is now `defcong^` (formerly, `defcon`).

Patterns are now allowed in `:expand` hints^. See the documentation for `:expand` inside the documentation for hints^.

We have improved the way we report rules used by the simplifier. All rune^s of the same type are reported together in the running commentary associated with each goal, so that for example, executable counterparts are listed separately from definitions, and rewrite rules are listed separately from linear^ rules. The preprocessor now mentions "simple" rules; see Section 8.79 [SIMPLE], page 340.

The mechanism for printing warning messages for new rewrite rules, related to subsumption, now avoids worrying about nonrecursive function symbols when those symbols are disable^d. These messages have also been eliminated for the case where the old rule is a `:definition^` rule.

Backquote has been modified so that it can usually provide predictable results when used on the left side of a rewrite rule.

Time statistics are now printed even when an event fails.

The Acl2 trace package has been modified so that it prints using the values of the Lisp globals `*print-level*` and `*print-length*` (respectively).

Table^ has been modified so that the `:clear` option lets you replace the entire table^ with one that satisfies the `val` and key guards (if any); see Section 6.35 [TABLE], page 193.

We have relaxed the translation rules for `:measure` hints^ to `defun^`, so that the the same rules apply to these terms that apply to terms in `defthm^` events^. In particular, in `:measure` hints^ `mv^` is treated just like `list^`, and `state^` receives no special handling.

The loop-stopper^ test has been relaxed. The old test required that every new argument be strictly less than the corresponding old argument in a certain term-order^. The new test uses a lexicographic order on term lists instead. For example, consider the following rewrite rule.

```
(equal
 (variable-update var1
                  val1 (variable-update var2 val2 vs))
 (variable-update var2
                  val2 (variable-update var1 val1 vs)))
```

This rule is permutative. Now imagine that we want to apply this rule to the term

```
(variable-update u y (variable-update u x vs)).
```

Since the actual corresponding to both `var1` and `var2` is `u`, which is not strictly less than itself in the term-order^, this rule would fail to be applied in this situation when using the old test. However, since the pair `(u x)` is lexicographically less than the pair `(u y)` with respect to our term-order^, the rule is in fact applied using our new test.

Messages about events^ now contain a space after certain left parentheses, in order to assist emacs users. For example, the event

```
(defthm abc (equal (+ (len x) 0) (len x)))
```

leads to a summary containing the line

```
Form:  ( DEFTHM ABC ...)
```

and hence, if you search backwards for "`(defthm abc`", you won't stop at this message.

More tautology checking is done during a proof; in fact, no goal printed to the screen, except for the results of applying `:use` and `:by` hints^ or the top-level goals from an induction proof, are known to Acl2 to be tautologies.

The `ld-query-control-alist^` may now be used to suppress printing of queries; see Section 8.55 [LD-QUERY-CONTROL-ALIST], page 313.

Warning messages are printed with short summary strings, for example the string "`Use`" in the following message.

```
Acl2 Warning [Use] in DEFTHM:  It is unusual to :USE an enabled
:REWRITE or :DEFINITION rule, so you may want to consider
disabling FOO.
```

At the end of the event, just before the time is printed, all such summary strings are printed out.

The keyword command `:u` has been introduced as an abbreviation for `:ubt^ :max^`. Printing of query messages is suppressed by `:u`.

The keyword `:cheat` is no longer supported by any event form.

Some irrelevant formals are detected; see Section 10.136 [IRRELEVANT-FORMALS], page 458.

A bug in the application of metafunctions was fixed: now if the output of a metafunction is equal to its input, the application of the metafunction is deemed unsuccessful and the next metafunction is tried.

An example has been added to the documentation for equivalence^ to suggest how to make use of equivalence^ relations in rewriting.

The following Common Lisp functions have been added to Acl2: `alpha-char-p^`, `upper-case-p^`, `lower-case-p^`, `char-upcase^`, `char-downcase^`, `string-downcase^`, `string-upcase^`, and `digit-charp-p`.

A documentation section called `proof-checker^` has been added for the interactive facility, whose documentation has been slightly improved. See in particular the documentation for proof-checker^, `verify^`, and macro-command^.

A number of events^ that had been inadvertently disallowed in books^ are now permitted in books^. These are: `defcong^`, `defcor`, `defequiv^`, `defrefinement^`, `defstub^`, and `verify-termination^`.

13.5 NOTE5

NOTE5 Acl2 Version 1.5 Notes

Acl2 now allows "complex rationals," which are complex numbers whose real parts are rationals and whose imaginary parts are non-zero rationals. See Section 10.87 [COMPLEX], page 429.

A new way of handling `force`^d hypotheses has been implemented. Rather than cause a case split at the time the `force`^ occurs, we complete the main proof and then embark on one or more "forcing rounds" in which we try to prove the force^d hypotheses. See Section 8.36 [FORCING-ROUND], page 279. To allow us to compare the new handling of `force`^ with the old, Version 1.5 implements both and uses a flag in `state`^ to determine which method should be used. Do `(assign old-style-forcing t)` if you want `force`^ to be handled as it was in Version 1.4. However, we expect to eliminate the old-style forcing eventually because we think the new style is more effective. To see the difference between the two approaches to forcing, try proving the associativity of append^ under both settings of `old-style-forcing`. To get the new behavior invoke:

```
(thm (implies (and (true-listp a) (true-listp b))
              (equal (append (append a b) c)
                     (append a (append b c)))))
```

Then `(assign old-style-forcing t)` and invoke the `thm` command^ above again.

A new `:cases` hints^ allows proof by cases. See Section 8.41 [HINTS], page 297.

`Include-book`^ and `encapsulate`^ now restore the `acl2-defaults-table`^ when they complete. See Section 6.20 [INCLUDE-BOOK], page 180 and see Section 6.18 [ENCAPSULATE], page 176.

The guard^s on many Acl2 primitives defined in `axioms.lisp` have been weakened to permit them to be used in accordance with lisp custom and tradition.

It is possible to attach heuristic filters to `:rewrite`^ rules to limit their applicability. See Section 8.84 [SYNTAXP], page 350.

A tutorial has been added; see Chapter 1 [ACL2-TUTORIAL], page 3.

Events^ now print the Summary paragraph listing rune^s used, time, etc., whether they succeed or fail. The format of the "failure^ banner" has been changed but still has multiple asterisks in it. `Thm` also prints a Summary, whether it succeeds or fails; but `thm` is not an event.

A new event form `skip-proofs^` has been added; see Section 9.14 [SKIP-PROOFS], page 388.

A user-specific customization facility has been added in the form of a book that is automatically included, if it exists on the current directory. See Section 8.4 [ACL2-CUSTOMIZATION], page 225.

A facility for conditional metalemmas has been implemented; see Section 14.12 [META], page 652.

The acceptable values for `ld-skip-proofsp^` have changed. In the old version (Version 1.4), a value of `t` meant that proofs and `local^` events^ are to be skipped. In Version 1.5, a value of `t` means proofs (but not `local^` events^) are to be skipped. A value of `'include-book^` means proofs and `local^` events^ are to be skipped. There are two other, more obscure, acceptable values. See Section 8.57 [LD-SKIP-PROOFSP], page 317.

In order to turn off the forcing of assumptions, one should now disable^ the `:executable-counterpart^` of `force^` (rather than the `:definition^` of `force^`, as in the previous release); see Section 8.35 [FORCE], page 277.

The macros `enable-forcing^` and `disable-forcing^` make it convenient to enable^ or disable^ forcing. See Section 8.26 [ENABLE-FORCING], page 269 and see Section 8.21 [DISABLE-FORCING], page 261.

The new commands `:pr^` and `:pr!^` print the rules created by an event or command. See Section 7.11 [PR], page 211 and see Section 7.12 [PR!], page 212.

The new history^ command^s `:puff^` and `:puff*^` will replace a compound command^ such as an `encapsulate^` or `include-book^` by the sequence of events^ in it. That is, they "puff^ up" or "lift" the subevents of a command^ to the command^ level, eliminating the formerly superior command^ and lengthening the history^. This is useful if you want to "partially undo" an `encapsulate^` or book or other compound command^ so you can experiment. See Section 7.13 [PUFF], page 212 and see Section 7.14 [PUFF*], page 215.

Theory expressions now are allowed to use the free variable `world^` and prohibited from using the free variable `state^`. See Chapter 15 [THEORIES], page 667, although it is essentially the same as before except it mentions `world^` instead of `state^`. See Section 8.89 [WORLD], page 362 for a discussion of the Acl2 logical world^. Allowing `in-theory^` events^ to be state-sensitive violated an important invariant about how books^ behaved.

`Table^` keys and values now are allowed to use the free variable `world^` and prohibited from using the free variable `state^`. See the note above about theory expressions for some explanation.

The macro for minus, `-^`, used to expand `(- x 3)` to `(+ x -3)` and now expands it to `(+ -3 x)` instead. The old macro, if used in the left-hand sides of rewrite rules, produced inapplicable rules because the constant occurs in the second argument of the `+^`, but potential target terms generally had the constant in the first argument position because of the effect of `commutativity-of-+`.

A new class of rule, `:linear-alias^` rules, allows one to implement the nqthm package and similar hacks in which a disable^d function is to be known equivalent to an arithmetic function. See Section 14.11 [LINEAR-ALIAS], page 646.

A new class of rule, `:built-in-clause` rules, allows one to extend the set of clauses proved silently by `defun^` during measure and guard^ processing. See Section 14.1 [BUILT-IN-CLAUSES], page 621.

The new command `pcb!^` is like `pcb^` but sketches the command^ and then prints its subsidiary events^ in full. See Section 7.5 [PCB!], page 209.

`:Rewrite^` class rules may now specify the `:loop-stopper^` field. See Chapter 14 [RULE-CLASSES], page 617 and see Section 8.62 [LOOP-STOPPER], page 324.

The rules for how loop-stopper^s control permutative rewrite rules have been changed. One effect of this change is that now when the built-in commutativity rules for `+^` are used, the terms `a` and `(- a)` are permuted into adjacency. For example, `(+ a b (- a))` is now normalized by the commutativity rules to `(+ a (- a) b)`; in Version 1.4, `b` was considered syntactically smaller than `(- a)` and so `(+ a b (- a))` is considered to be in normal form. Now it is possible to arrange for unary functions be be considered "invisible" when they are used in certain contexts. By default, `unary--^` is considered invisible when its application appears in the argument list of `binary-+^`. See Section 8.62 [LOOP-STOPPER], page 324 and see Section 6.30 [SET-INVISIBLE-FNS-ALIST], page 189.

Extensive documentation has been provided on the topic of Acl2's "term ordering." See Section 8.86 [TERM-ORDER], page 355.

Calls of `ld^` now default `ld-error-action^` to `:return` rather than to the current setting.

The command^ descriptor `:x` has been introduced and is synonymous with `:max^`, the most recently executed command^. History^ command^s such as `:pbt^` print a `:x` beside the most recent command^, simply to indicate that it `is` the most recent one.

The command^ descriptor `:x-23` is synonymous with `(:x -23)`. More generally, every symbol in the keyword package whose first character is `#\x` and whose remaining characters^ parse as a negative integer is appropriately understood. This allows `:pbt^ :x-10` where `:pbt^ (:max -10)` or `:pbt^ (:here -10)` were previously used. The old forms are still legal.

The order of the arguments to `defcong^` has been changed.

The simplifier now reports the use of unspecified built-in type information about the primitives with the phrase "primitive type reasoning." This phrase may sometimes occur in situations where "propositional calculus" was formerly credited with the proof.

The function `pairlis^` has been replaced in the code by a new function `pairlis$^`, because Common Lisp does not adequately specify its `pairlis^` function.

Some new Common Lisp functions have been added, including `logtest^`, `logcount^`, `integer-length^`, `make-list^`, `remove-duplicates^`, `string^`, and `concatenate^`. The source file `/slocal/src/acl2/axioms.lisp` is the ultimate reference regarding Common Lisp functions in Acl2.

The functions `defuns^` and `theory-invariant^` have been documented. See Section 8.20 [DEFUNS], page 261 and see Section 6.37 [THEORY-INVARIANT], page 197.

A few symbols have been added to the list `*acl2-exports*`.

A new key has been implemented for the `acl2-defaults-table^`, `:irrelevant-formals-ok`. See Section 6.31 [SET-IRRELEVANT-FORMALS-OK], page 190.

The connected book directory, `cbd^`, must be nonempty and begin and end with a slash. It is set (and displayed) automatically upon your first entry to `lp^`. You may change the setting with `set-cbd^`. See Section 3.4 [CBD], page 97.

`:oops^` will undo the last `:ubt^`. See Section 7.1 [OOPS], page 203.

Documentation has been written about the ordinals. See Section 8.24 [E0-ORDINALP], page 263 and see Section 8.23 [E0-ORD-<], page 262.

The color events^ — (red), (pink), (blue), and (gold) — may no longer be enclosed inside calls of `local^`, for soundness reasons. In fact, neither may any event that sets the `acl2-defaults-table^`. See Section 8.25 [EMBEDDED-EVENT-FORM], page 267.

See Section 8.50 [LD-KEYWORD-ALIASES], page 309 for an example of how to change the exit keyword from `:q^` to something else.

The attempt to install a monitor^ on `:rewrite^` rules stored as simple abbreviations now causes an error because the application of abbreviations is not tracked.

A new message is sometimes printed by the theorem prover, indicating that a given simplification is "specious" because the subgoals it produces include the input goal. In Version 1.4 this was detected but not reported, causing behavior some users found bizarre. See Section 8.80 [SPECIOUS-SIMPLIFICATION], page 341.

`:Definition^` rules are no longer always required to specify the `:clique` and `:controller-alist` fields; those fields can be defaulted to system-determined values in many common instances. See Section 14.4 [DEFINITION], page 628.

A warning is printed if a macro form with keyword arguments is given duplicate keyword values. Execute `(thm t :doc nil :doc "ignored")` and read the warning printed.

A new restriction has been placed on `encapsulate^`. Non-`local^` recursive definitions inside the `encapsulate^` may not use, in their tests and recursive calls, the constrained functions introduced by the `encapsulate^`. See Section 8.82 [SUBVERSIVE-INDUCTIONS], page 347.

The events^ `defequiv^`, `defcong^`, `defrefinement^`, and `defevaluator^` have been reimplemented so that they are just macros that expand into appropriate `defthm^` or `encapsulate^` events^; they are no longer primitive events^. See the documentation^ of each affected event.

The `defcor` event, which was a shorthand for a `defthm^` that established a corollary^ of a named, previously proved event, has been eliminated because its implementation relied on a technique we have decided to ban from our code. If you want the effect of a `defcor` in Version 1.5 you must submit the corresponding `defthm^` with a `:by` hint naming the previously proved event.

Error reporting has been improved for inappropriate `in-theory^` hints^ and events^, and for syntax errors in rule classes, and for non-existent filename arguments to `ld^`.

Technical Note: We now maintain the Third Invariant on `type-alists`, as described in the Essay on the Invariants on Type-alists, and Canonicality. This change will affect some proofs, for example, by causing a to rewrite more quickly to `c` when `(equiv a b)` and `(equiv b c)` are both known and `c` is the canonical representative of the three.

13.6 NOTE6

NOTE6 Acl2 Version 1.6 Notes

A new key has been implemented for the `acl2-defaults-table^`, `:ignore-ok`. See Section 6.28 [SET-IGNORE-OK], page 188.

It is now legal to have color events^, such as `(red)`, in the portcullis^ of a book. More generally, it is legal to set the `acl2-defaults-table^` in the portcullis^ of a book. For example, if you execute `:red` and then certify a book, the event `(red)` will show up in the portcullis^ of that book, and hence the definitions in that book will all be red (except when overridden by appropriate declarations or events^). When that book is included, then as always, its portcullis^ must first be “raised,” and that will cause the default color to become red before the events^ in the book are executed. As always, the value of `acl2-defaults-table^` immediately after execution of an `include-book^`, `certify-book^`, or `encapsulate^` form will be the same as it was immediately before execution (and hence, so will the default color). See Section 3.10 [PORTCULLIS], page 108 and, for more about books, see Chapter 3 [BOOKS], page 89.

A theory `ground-zero^` has been defined to contain exactly those rules that are enable^d when Acl2 starts up. See Section 15.6 [GROUND-ZERO], page 673.

The function `nth^` is now enable^d, correcting an oversight from Version 1.5.

Customization files no longer need to meet the syntactic restrictions put on books^; rather, they can contain arbitrary Acl2 forms. See Section 8.4 [ACL2-CUSTOMIZATION], page 225.

Structured directory names and structured file names are supported; see especially the documentation for pathname^, book-name^, and `cbd^`.

Acl2 now works with some Common Lisp implementations other than akcl, including Lucid, Allegro, and MCL.

A facility has been added for displaying proof trees, especially using emacs; see Chapter 12 [PROOF-TREE], page 579.

There is a considerable amount of new documentation^, in particular for the printing functions `fmt^`, `fmt1^`, and `fms^`, and for the notion of Acl2 term (see Section 8.85 [TERM], page 352).

It is possible to introduce new well-founded relations, to specify which relation should be used by `defun^`, and to set a default relation. See Section 14.17 [WELL-FOUNDED-RELATION], page 662.

It is possible to make functions suggest new inductions. See Section 14.9 [INDUCTION], page 639.

It is possible to change how Acl2 expresses type-set^ information; in particular, this affects what clauses are proved when force^d assumptions are generated. See Section 14.16 [TYPE-SET-INVERTER], page 661.

A new restriction has been added to `defpkg^`, having to do with undoing. If you undo a `defpkg^` and define the same package name again, the imports list must be identical to the previous imports or else an explanatory error will occur. See Section 8.68 [PACKAGE-REINCARNATION-IMPORT-RESTRICTIONS], page 330.

`Theory-invariant^` and `set-irrelevant-formals-ok^` are now embedded event forms.

The command `:good-bye^` may now be used to quit entirely out of Lisp, thus losing your work forever. This command works in akcl but may not work in every Common Lisp.

A theory `ground-zero^` has been added that contains exactly the enable^d rules in the startup^ theory. See Section 15.6 [GROUND-ZERO], page 673.

`Define-pc-macro` and `define-pc-atomic-macro` now automatically define `:red` functions. (It used to be necessary, in general, to change color to `:red` before invoking these.)

For a proof of the well-foundedness of `e0-ord-<^` on the `e0-ordinalps`, see Section 8.71 [PROOF-OF-WELL-FOUNDEDNESS], page 332.

Free variables are now handled properly for hypotheses of `:type-prescription^` rules.

When the system is loaded or saved, `state^` is now bound to `*the-live-state*`.

Certify-book^ has been modified so that when it compiles a file, it loads that object file.

Defstub^ has been modified so that it works when the color is hot (:red or :pink).

Several basic, but not particularly commonly used, events^ have been added or changed. The obscure axiom symbol-name-intern has been modified. The definition of firstn has been changed. Butlast^ is now defined. The definition of integer-length^ has been modified. The left-hand side of the rewrite rule rational-implies2 has been changed from (* (numerator x) (/ (denominator x))) to (* (/ (denominator x)) (numerator x)), in order to respect the fact that unary-/^ is invisible with respect to binary-*^. See Section 8.62 [LOOP-STOPPER], page 324.

The 'preprocess' process in the waterfall (see Section 8.41 [HINTS], page 297 for a discussion of the :do-not hint) has been changed so that it works to avoid case-splitting. The 'simplify' process refuses to force (see Section 8.35 [FORCE], page 277) when there are if^ terms, including and^ and or^ terms, in the goal being simplified.

The function apply is no longer introduced automatically by translation of user input to internal form when functions are called on inappropriate explicit values, e.g., (car 3).

The choice of which variable to use as the measured variable in a recursive definition has been very slightly changed.

13.7 NOTE7

NOTE7 ACL2 Version 1.7 (released October 1994) Notes

Include-book^ now takes (optionally) an additional keyword argument, indicating whether a compiled file is to be loaded. The default behavior is unchanged, except that a warning is printed when a compiled file is not loaded. See Section 6.20 [INCLUDE-BOOK], page 180.

A markup^ language for documentation^ strings has been implemented, and many of the source files have been marked up using this language (thanks largely to the efforts of Laura Lawless). See Section 5.8 [MARKUP], page 140. Moreover, there are translators that we have used to provide versions of the ACL2 documentation^ in info (for use in emacs), html (for Mosaic), and tex (for hardcopy) formats.

A new event `defdoc^` has been implemented. It is like `deflabel^`, but allows redefinition of doc^ strings and has other advantages. See Section 6.6 [DEFDOC], page 158.

We used to ignore corollaries when collecting up the axioms introduced about constrained functions. That bug has been fixed. We thank John Cowles for bringing this bug to our attention.

The macro `defstub^` now allows a `:doc^` keyword argument, so that documentation^ may be attached to the name being introduced.

A new command `nqthm-to-acl2^` has been added to help Nqthm users to make the transition to ACL2. See Section 5.12 [NQTHM-TO-ACL2], page 148, which also includes a complete listing of the relevant tables.

Many function names, especially of the form "foo-`lst`", have been changed in order to support the following convention, for any "foo":

`(foo-listp lst)` represents the notion `(for x in lst always foop x)`.

A complete list of these changes may be found at the end of this note. All of them except `symbolp-listp` and `list-of-symbolp-listp` have the string "`-lst`" in their names. Note also that `keyword-listp` has been renamed `keyword-value-listp^`.

Accumulated persistence has been implemented. It is not connected to `:brr^` or rule monitoring. See Section 8.1 [ACCUMULATED-PERSISTENCE], page 221.

`:Trigger-terms` has been added for `:linear^` rule classes, so you can hang a linear^ rule under any addend you want. See Section 14.10 [LINEAR], page 643, which has been improved and expanded.

ACL2 now accepts `256` characters^ and includes the Common Lisp functions `code-char^` and `char-code^`. However, ACL2 controls the lisp reader so that `#\c` may only be used when `c` is a single standard character or one of `Newline`, `Space`, `Page`, `Rubout`, `Tab`. If you want to enter other characters^ use `code-char^`, e.g., `(coerce (list (code-char 7) (code-char 240) #a) 'string)`. See Section 10.83 [CHARACTERS], page 426. Note: our current handling of characters^ makes the set of theorems different under Macintosh Common Lisp (MCL) than under other Common Lisps. We hope to rectify this situation before the final release of ACL2.

A new table^, `macro-aliases-table^`, has been implemented, that associates macro names with function names. So for example, since `append^` is associated with `binary-append^`, the form `(disable append)` it is interpreted as though it were `(disable binary-append)`. See Section 6.23

[MACRO-ALIASES-TABLE], page 183, see Section 6.1 [ADD-MACRO-ALIAS], page 154 and see Section 6.26 [REMOVE-MACRO-ALIAS], page 187.

The implementation of conditional metalemmas has been modified so that the metafunction is applied before the hypothesis metafunction is applied. See Section 14.12 [META], page 652.

The Common Lisp functions `acons^` and `endp^` have been defined in the ACL2 logic.

We have added the symbol `declare^` to the list `*acl2-exports*`, and hence to the package `"ACL2-USER"`.

A new hint, `:restrict`, has been implemented. See Section 8.41 [HINTS], page 297.

It used to be that if `:ubt^` were given a number that is greater than the largest current command^ number, it treated that number the same as `:max^`. Now, an error is caused.

The table^ `:force-table` has been eliminated.

A command `:disabledp^` (and macro `disabledp^`) has been added; see Section 8.22 [DISABLEDP], page 262.

Compilation^ via `:set-compile-fns^` is now suppressed during `include-book^`. In fact, whenever the state^ global variable `ld-skip-proofsp^` has value `'include-book^`.

Here are some less important changes, additions, and so on.

Unlike previous releases, we have not proved all the theorems in `axioms.lisp`; instead we have simply assumed them. We have deferred such proofs because we anticipate a fairly major changed in Version 1.8 in how we deal with guard^s.

We used to (accidentally) prohibit the "redefinition" of a table^ as a function. That is no longer the case.

The check for whether a corollary^ follows tautologically has been sped up, at the cost of making the check less "smart" in the following sense: no longer do we expand primitive functions such as `implies^` before checking this propositional implication.

The command^ ubt!^ has been modified so that it never causes or reports an error. See Section 7.17 [UBT!], page 217.

ACL2 now works in Harlequin Lispworks.

The user can now specify the :trigger-terms for :linear^ rules. See Section 14.10 [LINEAR], page 643.

The name of the system is now “ACL2”; no longer is it “Acl2”.

The raw lisp counterpart of theory-invariant^ is now defined to be a no-op as is consistent with the idea that it is just a call of table^.

A bug was fixed that caused proof-checker^ instructions^ to be executed when ld-skip-proofsp^ was t.

The function rassoc^ has been added, along with a corresponding function used in its guard^, r-eqlable-alistp.

The in-theory^ event and hint now print a warning not only when certain “primitive” :definition^ rules are disable^d, but also when certain “primitive” :executable-counterpart^ rules are disable^d.

The modified version of trace provided by ACL2, for use in raw Lisp, has been modified so that the lisp special variable *trace-alist* is consulted. This alist associates, using eq^, values with their print representations. For example, initially *trace-alist* is a one-element list containing the pair (cons state '|*the-live-state*|).

The system now prints an observation when a form is skipped because the default color is :red or :pink. (Technically: when-cool has been modified.)

Additional protection exists when you submit a form to raw Common Lisp that should only be submitted inside the ACL2 read-eval-print loop.

Here is a complete list of the changes in function names described near the top of this note, roughly of the form

```
foo-lst --> foo-listp
```

meaning: the name "`foo-lst`" has been changed to "`foo-listp`."

```
symbolp-listp    --> symbol-listp
list-of-symbolp-listp  --> symbol-list-listp
                       {for consistency with change to symbol-listp}
rational-lst     --> rational-listp
                     {which in fact was already defined as well}
integer-lst      --> integer-listp
character-lst    --> character-listp
stringp-lst      --> string-listp
32-bit-integer-lst   --> 32-bit-integer-listp
typed-io-lst     --> typed-io-listp
open-channel-lst --> open-channel-listp
readable-files-lst   --> readable-files-listp
written-file-lst --> written-file-listp
read-file-lst    --> read-file-listp
writeable-file-lst   --> writable-file-listp
                     {note change in spelling of ``writable''}
writeable-file-lst1  --> writable-file-listp1
pseudo-termp-lst     --> pseudo-term-listp
hot-termp-lst --> hot-term-listp {by analogy with pseudo-term-listp}
weak-termp-lst   --> weak-term-listp
weak-termp-lst-lst   --> weak-termp-list-listp
ts-builder-case-lstp -> ts-builder-case-listp
quotep-lst       --> quote-listp
termp-lst        --> term-listp
instr-lst        --> instr-listp
spliced-instr-lst    --> spliced-instr-listp
rewrite-fncallp-lst  --> rewrite-fncallp-listp
every-occurrence-equiv-hittablep1-lst -->
            every-occurrence-equiv-hittablep1-listp
some-occurrence-equiv-hittablep1-lst  -->
            some-occurrence-equiv-hittablep1-listp
            {by analogy with the preceding, even though it's a
             ``some'' instead of ``all'' predicate]
almost-quotep1-lst   --> almost-quotep1-listp
ffnnames-subsetp-lst --> ffnnames-subsetp-listp
boolean-lstp     --> boolean-listp
subst-expr1-lst-okp  --> subst-expr1-ok-listp
```

13.8 NOTE8

NOTE8 ACL2 Version 1.8 (May, 1995) Notes

See Section 13.9 [NOTE8-UPDATE], page 614 for yet more recent changes.

Guard^s have been eliminated from the ACL2 logic. A summary is contained in this brief note. Also see Section 8.18 [DEFUN-MODE], page 256 and see Section 9.12 [SET-GUARD-CHECKING], page 386.

Guard^s may be included in defuns^ as usual but are ignored from the perspective of admission to the logic: functions must terminate on all arguments.

As in Nqthm, primitive functions, e.g., `+^` and `car^`, logically default unexpected arguments to convenient values. Thus, `(+ 'abc 3)` is 3 and `(car 'abc)` is `nil`. See Chapter 10 [PROGRAMMING], page 393, and see the documentation^ for the individual primitive functions.

In contrast to earlier versions of ACL2, Version 1.8 logical functions are executed at Nqthm speeds even when guard^s have not been verified. In versions before 1.8, such functions were interpreted by ACL2.

Colors have been eliminated. Two "defun-mode^s" are supported, `:program^` and `:logic^`. Roughly speaking, `:program^` does what `:red` used to do, namely, allow you to prototype functions for execution without any proof burdens. `:Logic^` mode does what `:blue` used to do, namely, allow you to add a new definitional axiom to the logic. A global default-defun-mode^ is comparable to the old default color. The system comes up in `:logic^` mode. To change the global defun-mode^, type `:program^` or `:logic^` at the top-level. To specify the defun-mode^ of a `defun^` locally use

```
(declare (xargs :mode mode)).
```

The prompt^ has changed. The initial prompt^, indicating `:logic^` mode, is

```
ACL2 !>
```

If you change to `:program^` mode the prompt^ becomes

```
ACL2 p!>
```

Guard^s can be seen as having either of two roles: (a) they are a specification device allowing you to characterize the kinds of inputs a function "should" have, or (b) they are an efficiency device allowing logically defined functions to be executed directly in Common Lisp. If a guard^ is specified, as with `xargs^ :guard^`, then it is "verified" at defun-time (unless you also specify `xargs^ :verify-guards nil`). Guard^ verification means what it always has: the input guard^ is shown to imply the guard^s on all subroutines in the body. If the guard^s of a function are verified, then a call of the function on inputs satisfying the guard^ can be computed directly by Common Lisp. Thus, verifying the guard^s on your functions will allow them to execute more efficiently. But it does not affect their logical behavior and since you will automatically get Nqthm speeds on unverified logical definitions, most users will probably use guard^s either as a specification device or only use them when execution efficiency is extremely important.

Given the presence of guard^s in the system, two issues are unavoidable. Are guard^s verified as part of the `defun^` process? And are guard^s checked when terms are evaluated? We answer both of those questions below.

Roughly speaking, in its initial state^ the system will try to verify the guard^s of a `defun^` if a `:guard^` is supplied in the `xargs^` and will not try otherwise. However, guard^ verification in `defun^` can be inhibited "locally" by supplying the `xargs^` `:verify-guards^` `nil`. "Global" inhibition can be obtained via the `:set-verify-guards-eagerness^`. If you do not use the `:guard^` `xargs^`, you will not need to think about guard^ verification.

We now turn to the evaluation of expressions. Even if your functions contain no guard^s, the primitive functions do and hence you have the choice: when you submit an expression for evaluation do you mean for guard^s to be checked at runtime or not? Put another way, do you mean for the expression to be evaluated in Common Lisp (if possible) or in the logic? Note: If Common Lisp delivers an answer, it will be the same as in the logic, but it might be erroneous to execute the form in Common Lisp. For example, should `(car 'abc)` cause a guard^ violation error or return `nil`?

The top-level ACL2 loop has a variable which controls which sense of execution is provided. To turn "guard^ checking on," by which we mean that guard^s are checked at runtime, execute the top-level form `:set-guard-checking t`. To turn it off, do `:set-guard-checking nil`. The status of this variable is reflected in the prompt^.

```
ACL2 !>
```

means guard^ checking is on and

```
ACL2 >
```

means guard^ checking is off. The exclamation mark can be thought of as "barring" certain computations. The absence of the mark suggests the absence of error messages or unbarred access to the logical axioms. Thus, for example

```
ACL2 !>(car 'abc)
```

will signal an error, while

```
ACL2 >(car 'abc)
```

will return `nil`.

Note that whether or not guard^s are checked at runtime is independent of whether you are operating in `:program^` mode or `:logic^` mode and whether theorems are being proved or not. (Although it must be added that functions defined in `:program^` mode cannot help but check their guard^s because no logical definition exists.)

Version 1.8 permits the verification of the guard^s of theorems, thus insuring that all instances of the theorem will evaluate without error in Common Lisp. To verify the guard^s of a theorem named `name` execute the event

```
(verify-guards name).
```

If a theorem's guard^s have been verified, the theorem is guaranteed to evaluate without error to non-`nil` in Common Lisp (provided resource errors do not arise).

Caveat about `verify-guards^`: `implies^` is a function symbol, so in the term `(implies p q)`, `p` cannot be assumed true when `q` is evaluated; they are both evaluated "outside." Hence, you cannot generally verify the guard^s on a theorem if `implies^` is used to state the hypotheses. Use `if^` instead. In a future version of ACL2, `implies^` will likely be a macro.

See sum-list-example.lisp for a nice example of the use of Version 1.8. This is roughly the same as the documentation for guard-example^.

We have removed the capability to do "old-style-forcing" as existed before Version 1.5. See Section 13.5 [NOTE5], page 596.

NOTE: Some low level details have, of course, changed. One such change is that there are no longer two distinct type prescriptions stored when a function is admitted with its guard^s verified. So for example, the type prescription rune^ for `binary-append^` is now

```
(:type-prescription binary-append)
```

while in Versions 1.7 and earlier, there were two such rune^s:

```
(:type-prescription binary-append . 1)
(:type-prescription binary-append . 2)
```

Nqthm-style forcing on linear^ arithmetic assumptions is no longer executed when forcing is disable^d.

Functional instantiation now benefits from a trick also used in Nqthm: once a constraint^ generated by a `:functional-instance` lemma instance (see Section 8.59 [LEMMA-INSTANCE], page 319) has been proved on behalf of a successful event, it will not have to be re-proved on behalf of a later event.

`1+^` and `1-^` are now macros in the logic, not functions. Hence, for example, it is "safe" to use them on left-hand sides of rewrite rules, without invoking the common warning about the presence of nonrecursive function symbols.

A new documentation^ section file-reading-example^ illustrates how to process forms in a file.

A new proof-checker^ command `forwardchain` has been added; see Section 11.30 [ACL2-PC::FORWARDCHAIN], page 539.

It is now possible to use quantifiers. See Section 6.17 [DEFUN-SK], page 173 and see Section 6.3 [DEFCHOOSE], page 155.

There is a new event `set-inhibit-warnings^`, which allows the user to turn off warnings of various types. see Section 6.29 [SET-INHIBIT-WARNINGS], page 189.

An unsoundness relating `encapsulate^` and `:functional-instance` hints^ has been remedied, with a few small effects visible at the user level. The main observable effect is that `defaxiom^` and non-local `include-book^` events^ are no longer allowed in the scope of any `encapsulate^` event that has a non-empty signature^.

When `certify-book^` is called, we now require that the default defun-mode^ (see Section 8.16 [DEFAULT-DEFUN-MODE], page 254) be `:logic^`. On a related note, the default defun-mode^ is irrelevant to `include-book^`; the mode is always set to `:logic^` initially, though it may be changed within the book and reverts to its original value at the conclusion of the `include-book^`. A bug in `include-book^` prevented it from acting this way even though the documentation^ said otherwise.

The documentation^ has been substantially improved. A new section "Programming" contains documentation^ of many useful functions provided by ACL2; see Chapter 10 [PROGRAMMING], page 393. Also, the documentation^ has been "marked up" extensively. Thus in particular, users of Mosaic will find many links in the documentation^.

The symbols `force^`, `mv-nth^`, and `acl2-count` have been added to the list `*acl2-exports*`.

We now permit most names from the main Lisp package to be used as names, except for names that define functions, macros, or constants. See Section 8.65 [NAME], page 329.

We have changed the list of imports from the Common Lisp package to ACL2, i.e., the list `*common-lisp-symbols-from-main-lisp-package*`, to be exactly those external symbols of the Common Lisp package as specified by the draft Common Lisp standard. In order to accommodate this change, we have renamed some ACL2 functions as shown below, but these and other ramifications of this change should be transparent to most ACL2 users.

```
warning      --> warning$
print-object --> print-object$
```

Proof trees are no longer enabled by default. To start them up, `:start-proof-tree^`.

We have added the capability of building smaller images. The easiest way to do this on a Unix (trademark of AT&T) system is: `make small`.

Here we will put some less important changes, additions, and so on.

We have added definitions for the Common Lisp function `position^` (for the test `eql^`), as well as corresponding versions `position-equal^` and `position-eq^` that use tests `equal^` and `eq^`, respectively. See Section 10.187 [POSITION], page 480, see Section 10.189 [POSITION-EQUAL], page 481, and see Section 10.188 [POSITION-EQ], page 480.

The `defthm^` event `rational-listp-implies-rationalp-car` no longer exists.

We fixed a bug in the hint mechanism that applied `:by`, `:cases`, and `:use` hints^ to the first induction goal when the prover reverted to proving the original goal by induction.

We fixed a bug in the handling of `(set-irrelevant-formals-ok :warn)`.

In support of removing the old-style forcing capability, we deleted the initialization of state^ global `old-style-forcing` and deleted the definitions of `recover-assumptions`, `recover-assumptions-from-goal`, `remove-assumptions1`, `remove-assumptions`, and `split-on-assumptions`, and we renamed `split-on-assumptions1` to `split-on-assumptions`.

The special value `'none` in the proof-checker^ commands `claim` and `=^` has been replaced by `:none`.

A bug in the handling of hints^ by subgoals has been fixed. For example, formerly a `:do-not` hint could be "erased" by a `:use` hint on a subgoal. Thanks go to Art Flatau for noticing the bug.

The functions `weak-termp` and `weak-term-listp` have been deleted, and their calls have been replaced by corresponding calls of `pseudo-termp^` and `pseudo-term-listp`. The notion of `pseudo-termp^` has been slightly strenthened by requiring that terms of the form `(quote ...)` have length 2.

Performance has been improved in various ways. At the prover level, backchaining through the recognizer alist has been eliminated in order to significantly speed up ACL2's rewriter. Among the other prover changes (of which there are several, all technical): we no longer clausify the input term when a proof is interrupted in favor of inducting on the input term. At the IO^ level, we have improved performance somewhat by suitable declarations and proclamations. These include technical modifications to the macros `mv^` and `mv-let^`, and introduction of a macro `the-mv` analogous to the macro `the^` but for forms returning multiple values.

The function `spaces` now takes an extra argument, the current column.

A bug in the proof-checker^ `equiv` command was fixed.

The function `intersectp` has been deleted, because it was essentially duplicated by the function `intersectp-equal^`.

We now proclaim functions in AKCL and GCL before compiling books^. This should result in somewhat increased speed.

The function `repeat` has been eliminated; use `make-list^` instead.

The proof-checker^ command `expand` has been fixed so that it eliminates `let^` (lambda) expressions when one would expect it to.

A new primitive function, `mv-nth^`, has been introduced. `Mv-nth^` is equivalent to `nth^` and is used in place of `nth^` in the translation of `mv-let^` expressions. This allows the user to control the simplification of `mv-let^` expressions without affecting how `nth^` is treated. In that spirit, the rewriter has been modified so that certain `mv-nth^` expressions, namely those produced in the translation of `(mv-let (a b c)(mv x y z) p)`, are given special treatment.

A minor bug in `untranslate` has been fixed, which for example will fix the printing of conjunctions.

`Translate` now takes a `logicp` argument, which indicates whether it enforces the restriction that `:program^` mode functions do not occur in the result.

The modified version of `trace` provided by ACL2, for use in raw Lisp, has been modified so that the lisp special variable `*trace-alist*` has a slightly different functionality. This alist associates, using `eq^`, symbols with the print representations of their values. For example, initially `*trace-alist*` is a one-element list containing the pair `(cons 'state '|*the-live-state*|)`. Thus, one may cons the pair `(cons '*foo* "It's a FOO!")` on to `*trace-alist*`; then until `*foo*` is defined, this change will have no effect, but after for example

```
(defconst *foo* 17)
```

then `trace` will print 17 as `"It's a FOO!"`.

`Trace` also traces the corresponding logic function.

Proof-tree^ display has been improved slightly in the case of successful proofs and certain event failures.

The function `positive-integer-log2` has been deleted.

The macro `skip-proofs^` now prints a warning message when it is encountered in the context of an `encapsulate^` event or a book. See Section 9.14 [SKIP-PROOFS], page 388.

Some functions related to `the-fn` and `wormhole1` now have defun-mode^ `:program^`, but this change is almost certain to be inconsequential to all users.

13.9 NOTE8-UPDATE

NOTE8-UPDATE ACL2 Version 1.8 (Summer, 1995) Notes

ACL2 can now use Ordered Binary Decision Diagram technology. See Chapter 2 [BDD], page 69. There is also a proof-checker^ `bdd` command.

ACL2 is now more respectful of the intention of the function `hide^`. In particular, it is more careful not to dive inside any call of `hide^` during equality substitution and case splitting.

The `ld^` special (see Section 9.7 [LD], page 380) `ld-pre-eval-print^` may now be used to turn off printing of input forms during processing of `encapsulate^` and `certify-book^` forms, by setting it to the value `:never`, i.e., `(set-ld-pre-eval-print :never state)`. See Section 8.53 [LD-PRE-EVAL-PRINT], page 311.

The TUTORIAL documentation section has, with much help from Bill Young, been substantially improved to a bona fide introduction, and has been renamed acl2-tutorial^.

The term pretty-printer has been modified to introduce `(<= X Y)` as an abbreviation for `(not (< Y X))`.

Forward chaining and linear arithmetic now both benefit from the evaluation of ground subterms.

A new macro `set-inhibit-output-lst^` has been defined. This should be used when setting the state^ global `inhibit-output-lst`; see Section 9.13 [SET-INHIBIT-OUTPUT-LST], page 387 and see Chapter 12 [PROOF-TREE], page 579.

The test for redundancy in definitions includes the guard^ and type declarations. See Section 8.76 [REDUNDANT-EVENTS], page 337.

See Section 8.37 [GENERALIZED-BOOLEANS], page 281 for a discussion of a potential soundness problem for ACL2 related to the question: Which Common Lisp functions are known to return Boolean values?

Here we will put some less important changes, additions, and so on.

A bug has been fixed so that now, execution of `:comp t` (see Section 9.5 [COMP], page 379) correctly handles non-standard characters.

A bug in `digit-char-p^` has been fixed, so that the “default” is `nil` rather than `0`.

`True-listp^` now tests the final `cdr^` against `nil` using `eq^` instead of `equal^`, for improved efficiency. The logical meaning is, however, unchanged.

`Put-assoc-equal^` has been added to the logic (it used to have `:defun-mode^ :program^`, and has been documented.

14 RULE-CLASSES

RULE-CLASSES adding rules to the data base

```
General Form:
a true list of rule class objects as defined below

Special Cases:
a symbol abbreviating a single rule class object
```

ACL2 provides users with the ability to create a number of different kinds of rules, including (conditional) rewrite rules but also including others. Don't be put off by the long description to follow; usually, you'll probably want to use rewrite rules. More on this below.

A rule class object is either one of the `:class` keywords or else is a list of the form shown below. Those fields marked with "(!)" are required when the `:class` is as indicated.

```
(:class
  :COROLLARY term
  :TRIGGER-FNS (fn1 ... fnk) ; provided :class = :META (!)
  :TRIGGER-TERMS (t1 ... tk) ; provided :class = :FORWARD-CHAINING
                             ;       or :class = :LINEAR
  :TYPE-SET n                ; provided :class = :TYPE-SET-INVERTER
  :TYPED-TERM term           ; provided :class = :TYPE-PRESCRIPTION
  :CLIQUE (fn1 ... fnk)      ; provided :class = :DEFINITION (!)
  :CONTROLLER-ALIST alist    ; provided :class = :DEFINITION (!)
  :LOOP-STOPPER alist        ; provided :class = :REWRITE
  :PATTERN term              ; provided :class = :INDUCTION (!)
  :CONDITION term            ; provided :class = :INDUCTION
  :SCHEME term               ; provided :class = :INDUCTION (!)
  :HINTS hints               ; provided instrs = nil
  :INSTRUCTIONS instrs       ; provided hints = nil
  :OTF-FLG flg)
```

When rule class objects are provided by the user, most of the fields are optional and their values are computed in a context sensitive way. When a `:class` keyword is used as a rule class object, all relevant fields are determined contextually. Each rule class object in `:rule-classes` causes one or more rules to be added to the data base. The `:class` keywords are documented individually under the following names. Note that when one of these names is used as a `:class`, it is expected to be in the keyword package (i.e., the names below should be preceded by a colon but the ACL2 documentation^ facilities do not permit us to use keywords below).

Before we get into the discussion of rule classes, let us return to an important point. In spite of the large variety of rule classes available, at present we recommend that new ACL2 users rely almost exclusively on (conditional) rewrite rules. A reasonable but slightly bolder approach is to use `:type-prescription^` and `:forward-chaining^` rules for "type-theoretic" rules, especially ones whose top-level function symbol is a common one like `true-listp^` or `consp^`; see Section 14.15 [TYPE-PRESCRIPTION], page 659 and see Section 14.7 [FORWARD-CHAINING], page 638. However, the rest of the rule classes are really not intended for widespread use, but rather are mainly for experts.

We expect that we will write more about the question of which kind of rule to use. For now: when in doubt, use a `:rewrite^` rule.

`:Rule-classes` is an optional keyword argument of the `defthm^` (and `defaxiom^`) event. In the following, let `name` be the name of the event and let `thm` be the formula to be proved or added as an axiom.

If `:rule-classes` is not specified in a `defthm^` (or `defaxiom^`) event, it is as though `:rule-classes ((:rewrite))` had been used. Use `:rule-classes nil` to specify that no rules are to be generated.

If `:rule-classes` class is specified, where class is a non-`nil` symbol, it is as though `:rule-classes ((class))` had been used. Thus, `:rule-classes :forward-chaining^` is equivalent to `:rule-classes ((:forward-chaining))`.

We therefore now consider `:rule-classes` as a true list. If any element of that list is a keyword, replace it by the singleton list containing that keyword. Thus, `:rule-classes (:rewrite :elim)` is the same as `:rule-classes ((:rewrite) (:elim))`.

Each element of the expanded value of `:rule-classes` must be a true list whose `car^` is one of the rule class keyword tokens listed above, e.g., `:rewrite^`, `:elim^`, etc., and whose `cdr^` is a "keyword alist" alternately listing keywords and values. The keywords in this alist must be taken from those shown below. They may be listed in any order and most may be omitted, as specified below.

> `:Corollary^` — its value, `term`, must be a term. If omitted, this field defaults to `thm`. The `:corollary^` of a rule class object is the formula actually used to justify the rule created and thus determines the form of the rule. Nqthm provided no similar capability: each rule was determined by `thm`, the theorem or axiom added. ACL2 permits `thm` to be stated "elegantly" and then allows the `:corollary^` of a rule class object to specify how that elegant statement is to be interpreted as a rule. For the rule class object to be well-formed, its (defaulted) `:corollary^`, `term`, must follow from `thm`. Unless `term` is trivially implied by `thm`, using little more than propositional logic, the formula `(implies thm term)` is submitted to the theorem prover and the proof attempt must be successful. During that proof attempt the values of `:hints^`, `:instructions^`, and `:otf-flg^`, as provided in the rule class object, are provided as arguments to the prover. Such auxiliary proofs give the sort of output that one expects from the prover. However, as noted above, corollaries that follow trivially are not submitted to the prover; thus, such corollaries cause no prover output.
>
> Note that before `term` is stored, all calls of macros in it are expanded away. See Section 9.16 [TRANS], page 390.
>
> `:Hints^`, `:instructions^`, `:otf-flg^` — the values of these fields must satisfy the same restrictions placed on the fields of the same names in `defthm^`. These values are passed to the recursive call of the prover used to establish that the `:corollary^` of the rule class object follows from the theorem or axiom `thm`.
>
> `:Type-set^` — this field may be supplied only if the `:class` is `:type-set-inverter^`. When provided, the value must be a type-set, an integer in a certain range. If not provided, an attempt is made to compute it from the corollary. See Section 14.16 [TYPE-SET-INVERTER], page 661.
>
> `:Typed-term` — this field may be supplied only if the `:class` is `:type-prescription^`. When provided, the value is the term for which the `:corollary^` is a type-prescription lemma. If no `:typed-term` is provided in a `:type-prescription^` rule class object, we try to compute heuristically an acceptable term. See Section 14.15 [TYPE-PRESCRIPTION], page 659.

`:Trigger-terms` — this field may be supplied only if the `:class` is `:forward-chaining^` or `:linear^`. When provided, the value is a list of terms, each of which is to trigger the attempted application of the rule. If no `:trigger-terms` is provided, we attempt to compute heuristically an appropriate set of triggers. See Section 14.7 [FORWARD-CHAINING], page 638 or see Section 14.10 [LINEAR], page 643.

`:Trigger-fns` — this field must (and may only) be supplied if the `:class` is `:meta^`. Its value must be a list of function symbols. Terms with these symbols trigger the application of the rule. See Section 14.12 [META], page 652.

`:Clique` and `:controller-alist` — these two fields must (and may only) be supplied if the `:class` is `:definition^`. Suppose the `:corollary^` of the rule is `(implies hyp (equiv (fn a1 ... an) body))`. The value of the `:clique` field should be a true list of function symbols, and if non-`nil` must include `fn`. These symbols are all the members of the mutually recursive clique containing this definition of `fn`. That is, a call of any function in `:clique` is considered a "recursive call" for purposes of the expansion heuristics. The value of the `:controller-alist` field should be an alist that maps each function symbol in the `:clique` to a list of `t`'s and `nil`'s of length equal to the arity of the function. For example, if `:clique` consists of just two symbols, `fn1` and `fn2`, of arities 2 and 3 respectively, then `((fn1 t nil) (fn2 nil t t))` is a legal value of `:controller-alist`. The value associated with a function symbol in this alist is a "mask" specifying which argument slots of the function "control" the recursion for heuristic purposes. Sloppy choice of `:clique` or `:controller-alist` can result in infinite expansion and stack overflow.

`:loop-stopper^` — this field may only be supplied if the class is `:rewrite^`. Its value must be a list of entries each consisting of two variables followed by a (possibly empty) list of functions, for example `((x y binary-+) (u v foo bar))`. It will be used to restrict application of rewrite rules by requiring that the list of instances of the second variables must be "smaller" than the list of instances of the first variables in a sense related to the corresponding functions listed; see Section 8.62 [LOOP-STOPPER], page 324. The list as a whole is allowed to be `nil`, indicating that no such restriction shall be made. Note that any such entry that contains a variable not being instantiated, i.e., not occurring on the left side of the rewrite rule, will be ignored. However, for simplicity we merely require that every variable mentioned should appear somewhere in the corresponding `:corollary^` formula.

`:pattern`, `:condition`, `:scheme` — the first and last of these fields must (and may only) be supplied if the class is `:induction^`. `:Condition` is optional but may only be supplied if the class is `:induction^`. The values must all be terms and indicate, respectively, the pattern to which a new induction scheme is to be attached, the condition under which the suggestion is to be made, and a term which suggests the new scheme. See Section 14.9 [INDUCTION], page 639.

Once `thm` has been proved (in the case of `defthm^`) and each rule class object has been checked for well-formedness (which might require additional proofs), we consider each rule class object in turn to generate and add rules. Let `:class` be the class keyword token of the `i`th class object (counting from left to right). Generate the rune^ `(:class name . x)`, where `x` is `nil` if there is only one class and otherwise `x` is `i`. Then, from the `:corollary^` of that object, generate one or more rules, each of which has the name `(:class name . x)`. See the `:doc^` entry for each rule class to see how formulas determine rules. Note that it is in principle possible for several rules to share the same name; it happens whenever a `:corollary^` determines more than one rule. This in fact only occurs for `:rewrite^`, `:linear^`, and `:forward-chaining^` class rules and only then if the `:corollary^` is essentially a conjunction. (See the documentation for rewrite^, linear^, or forward-chaining^ for details.)

14.1 BUILT-IN-CLAUSES

BUILT-IN-CLAUSES to build in a clause for measure and guard processing

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. A `:built-in-clause` rule can be built from any formula other than propositional tautologies. Roughly speaking, the system uses the list of built-in clauses as the first method of proof when attacking a new goal. Any goal that is subsumed by a built in clause is proved "silently."

ACL2 maintains a set of "built-in" clauses that are used to short-circuit certain theorem proving tasks. We discuss this at length below. When a theorem is given the rule class `:built-in-clause` ACL2 flattens the `implies^` and `and^` structure of the `:corollary^` formula so as to obtain a set of formulas whose conjunction is equivalent to the given corollary. It then converts each of these to clausal form and adds each clause to the set of built-in clauses.

For example, the following `:corollary^` (regardless of the definition of `abl`)

```
(and (implies (and (true-listp x)
                   (not (equal x nil)))
              (< (acl2-count (abl x))
                 (acl2-count x)))
     (implies (and (true-listp x)
                   (not (equal nil x)))
              (< (acl2-count (abl x))
                 (acl2-count x))))
```

will build in two clauses,

```
{(not (true-listp x))
 (equal x nil)
 (< (acl2-count (abl x)) (acl2-count x))}
```

and

```
{(not (true-listp x))
 (equal nil x)
 (< (acl2-count (abl x)) (acl2-count x))}.
```

We now give more background.

Recall that a clause is a set of terms, implicitly representing the disjunction of the terms. Clause `c1` is "subsumed" by clause `c2` if some instance of `c2` is a subset `c1`.

For example, let `c1` be

```
{(not (consp l))
 (equal a (car l))
 (< (acl2-count (cdr l)) (acl2-count l))}.
```

Then `c1` is subsumed by `c2`, shown below,

```
{(not (consp x))
 ; second term omitted here
 (< (acl2-count (cdr x)) (acl2-count x))}
```

because we can instantiate `x` in `c2` with `l` to obtain a subset of `c1`.

Observe that `c1` is the clausal form of

```
(implies (and (consp l)
              (not (equal a (car l))))
         (< (acl2-count (cdr l)) (acl2-count l))),
```

`c2` is the clausal form of

```
(implies (consp l)
         (< (acl2-count (cdr l)) (acl2-count l)))
```

and the subsumption property just means that `c1` follows trivially from `c2` by instantiation.

The set of built-in clauses is just a set of known theorems in clausal form. Any formula that is subsumed by a built-in clause is thus a theorem. If the set of built-in theorems is reasonably small, this little theorem prover is fast. ACL2 uses the "built-in clause check" in three places: at the entrance to the general-purpose theorem prover, as a filter on the clauses generated to prove the termination of recursively `defun^`'d functions and as a filter on the clauses generated to verify the guards of a function.

The latter two uses are the ones that most often motivate an extension to the set of built-in clauses. Frequently a given formalization problem requires the definition of many functions which require virtually identical termination and/or guard proofs. These proofs can be short-circuited by extending the set of built-in clauses to contain the most general forms of the clauses generated by the definitional schemes in use.

The attentive user might have noticed that there are some recursive schemes, e.g., recursion by `cdr^` after testing `consp^`, that ACL2 just seems to "know" are ok, while for others it generates measure clauses to prove. Actually, it always generates measure clauses but then filters out any that pass the built-in clause check. When ACL2 is initialized, the clause justifying `cdr^` recursion after a `consp^` test is added to the set of built-in clauses. (That clause is `c2` above.)

Note that only a subsumptin check is made; no rewriting or simplification is done. Thus, if we want the system to "know" that `cdr^` recursion is ok after a negative `atom^` test (which, by the definition of `atom^`, is the same as a `consp^` test), we have to build in a second clause. The subsumption algorithm does not "know" about commutative functions. Thus, for predictability, we have built in commuted versions of each clause involving commutative functions. For example, we build in both

```
{(not (integerp x))
 (< 0 x)
 (= x 0)
 (< (acl2-count (+ -1 x)) (acl2-count x))}
```

and the commuted version

```
{(not (integerp x))
 (< 0 x)
 (= 0 x)
 (< (acl2-count (+ -1 x)) (acl2-count x))}
```

so that the user need not worry whether to write `(= x 0)` or `(= 0 x)` in definitions.

`:built-in-clause` rules added by the user can be enabled and disabled.

14.2 COMPOUND-RECOGNIZER

COMPOUND-RECOGNIZER make a rule used by the typing mechanism

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:compound-recognizer` rule might be built is:

```
Example:
(implies (alistp x)          When (alistp x) is assumed true,
         (true-listp x))     add the additional hypothesis that x
                             is of primitive type true-listp.

General Forms:
(implies (fn x) concl)
(implies (not (fn x)) concl)
(and (implies (fn x) concl1)
     (implies (not (fn x)) concl2))
(iff (fn x) concl)
(equal (fn x) concl)
```

where `fn` is a Boolean valued function of one argument, `x` is a variable symbol, and the system can deduce some restriction on the primitive type of `x` from the assumption that `concl` holds. The third restriction is vague but one way to understand it is to weaken it a little to "and `concl` is a non-tautological conjunction or disjunction of the primitive type recognizers listed below."

The primitive ACL2 types and a suitable primitive recognizing expression for each are listed below.

```
type                suitable primitive recognizer

zero                (equal x 0)
negative integers   (and (integerp x) (< x 0))
positive integers   (and (integerp x) (> x 0))
negative ratio      (and (rationalp x)
                         (not (integerp x))
                         (< x 0))
positive ratio      (and (rationalp x)
                         (not (integerp x))
                         (> x 0))
complex rational    (complex-rationalp x)
```

```
nil                 (equal x nil)
t                   (equal x t)
other symbols       (and (symbolp x)
                         (not (equal x nil))
                         (not (equal x t)))
proper conses       (and (consp x)
                         (true-listp x))
improper conses     (and (consp x)
                         (not (true-listp x)))
strings             (stringp x)
characters          (characterp x)
```

Thus, a suitable `concl` to recognize the naturals would be `(or (equal x 0) (and (integerp x) (> x 0)))` but it turns out that we also permit `(and (integerp x) (>= x 0))`. Similarly, the true-lists could be specified by

```
(or (equal x nil) (and (consp x) (true-listp x)))
```

but we in fact allow `(true-listp x)`. When time permits we will document more fully what is allowed or implement a macro that permits direct specification of the desired type in terms of the primitives.

There are essentially four forms of `:compound-recognizer` rules, the last two general forms above being equivalent. We explain how such rules are used by considering the individual forms.

Consider a rule of the first form, `(implies (fn x) concl)`. The effect of such a rule is that when the rewriter assumes `(fn x)` true, as it would while diving through `(if (fn x) xxx ...)` to rewrite `xxx`, it restricts the type of `x` as specified by `concl`. However, when assuming `(fn x)` false, as necessary in `(if (fn x) ... xxx)`, the rule permits no additional assumptions about the type of `x`. For example, if `fn` is `primep`, i.e., the predicate that recognizes prime numbers, then `(implies (primep x) (and (integerp x) (>= x 0)))` is a compound recognizer rule of the first form. When `(primep x)` is assumed true, the rewriter gains the additional information that `x` is a natural number. When `(primep x)` is assumed false, no additional information is gained — since `x` may be a non-prime natural or may not even be a natural.

The second general form addresses itself to the symmetric case, when assuming `(fn x)` false permits type restrictions on `x` but assuming `(fn x)` true permits no such restrictions. For example, if we defined `exprp` to be the recognizer for well-formed expressions for some language in which all symbols, numbers, character objects and strings were well-formed — e.g., the well-formedness rules only put restrictions on expressions represented by `consp`^s — then the theorem `(implies (not` `(exprp x)) (consp x))` is a rule of the second form. Assuming `(exprp x)` true tells us nothing about the type of `x`; assuming it false tells us `x` is a `consp^`.

The third general form addresses itself to the case where one type may be deduced from `(fn x)` and a generally unrelated type may be deduced from its negation. If we modified the expression recognizer above so that character objects are illegal, then a rule of the third form is

```
(and (implies (exprp x) (not (characterp x)))
     (implies (not (exprp x)) (or (consp x) (characterp x)))).
```

Finally, rules of the fourth class address the case where `fn` recognizes all and only the objects whose type is described. In this case, `fn` is really just a new name for some "compound recognizers." The classic example is `(booleanp x)`, which is just a handy combination of two primitive types:

```
(iff (booleanp x) (or (equal x t) (equal x nil))).
```

Often it is best to disable `fn` after proving that it is a compound recognizer, since `(fn x)` will not be recognized as a compound recognizer once it has been expanded.

Every time you prove a new compound recognizer rule about `fn` it overrides all previously proved compound recognizer rules about `fn`. Thus, if you want to establish the type implied by `(fn x)` and you want to establish the type implied by `(not (fn x))`, you must prove a compound recognizer rule of the third or fourth forms. Proving a rule of the first form followed by one of the second only leaves the second fact in the data base.

Compound recognizer rules can be disabled with the effect that older rules about `fn`, if any, are exposed.

If you prove multiple compound recognizer rules for a function, you may see a **warning** message to the effect that the new rule is not as "restrictive" as the old. That is, the new rules do not give the rewriter strictly more type information than it already had. The new rule is stored anyway, overriding the old, if enabled. You may be playing subtle games with enabling or rewriting. But two other interpretions are more likely, we think. One is that you have forgotten about an earlier rule and should merely print it out to make sure it says what you know and then forget your new rule. The other is that you meant to give the system more information and the system has simply been unable to extract the intended type information from the term you placed in the conclusion of the new rule. Given our lack of specificity in saying how type information is extracted from rules, you can hardly blame yourself for this problem. Sorry. If you suspect you've been burned this way, you should rephrase the new rule in terms of the primitive recognizing expressions above and see if the warning is still given. It would also be helpful to let us see your example so we can consider it as we redesign this stuff.

Compound recognizer rules are similar to `:forward-chaining^` rules in that the system deduces new information from the act of assuming something true or false. If a compound recognizer rule were stored as a forward chaining rule it would have essentially the same effect as described, when it has any effect at all. The important point is that `:forward-chaining^` rules, because of their more general and expensive form, are used "at the top level" of the simplification process: we forward chain from assumptions in the goal being proved. But compound recognizer rules are built in at the bottom-most level of the simplifier, where type reasoning is done.

All that said, compound recognizer rules are a rather fancy, specialized mechanism. It may be more appropriate to create `:forward-chaining^` rules instead of `:compound-recognizer` rules.

14.3 CONGRUENCE

CONGRUENCE the relations to maintain while simplifying arguments

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:congruence` rule might be built is:

```
Example:
(implies (set-equal x y)
         (iff (memb e x) (memb e y))).
```

Also see Section 6.4 [DEFCONG], page 157 and see Section 14.6 [EQUIVALENCE], page 632.

```
General Form:
(implies (equiv1 xk xk-equiv)
         (equiv2 (fn x1... xk       ...xn)
                 (fn x1... xk-equiv ...xn)))
```

where `equiv1` and `equiv2` are known equivalence relations, `fn` is an `n-ary` function symbol and the `xi` and `xk-equiv` are all distinct variables. The effect of such a rule is to record that the `equiv2`-equivalence of `fn`-expressions can be maintained if, while rewriting the `kth` argument position, `equiv1`-equivalence is maintained. See Section 14.6 [EQUIVALENCE], page 632 for a general discussion of the issues. We say that `equiv2`, above, is the "outside equivalence" in the rule and `equiv1` is the "inside equivalence for the `k`th argument"

The macro form `(defcong equiv1 equiv2 (fn x1 ... x1) k)` is an abbreviation for a `defthm^` of rule-class `:congruence` that attempts to establish that `equiv2` is maintained by maintaining `equiv1` in `fn`'s `k`th argument. The `defcong^` macro automatically generates the general formula shown above. See Section 6.4 [DEFCONG], page 157.

The `memb` example above tells us that `(memb e x)` is propositionally equivalent to `(memb e y)`, provided `x` and `y` are `set-equal`. The outside equivalence is `iff^` and the inside equivalence for the second argument is `set-equal`. If we see a `memb` expression in a propositional context, e.g., as a literal of a clause or test of an `if^` (but not, for example, as an argument to `cons^`), we can rewrite its second argument maintaining `set-equality`. For example, a rule stating the commutativity of `append^` (modulo set-equality) could be applied in this context. Since equality is a refinement of all equivalence relations, all equality rules are always available. See Section 14.13 [REFINEMENT], page 656.

All known `:congruence` rules about a given outside equivalence and `fn` can be used independently. That is, consider two `:congruence` rules with the same outside equivalence, `equiv`, and about the same function `fn`. Suppose one says that `equiv1` is the inside equivalence for the first argument and the other says `equiv2` is the inside equivalence for the second argument. Then `(fn a b)` is `equiv (fn a' b')` provided `a` is `equiv1` to `a'` and `b` is `equiv2` to `b'`. This is an easy consequence of the transitivity of `equiv`. It permits you to think independently about the inside equivalences.

Furthermore, it is possible that more than one inside equivalence for a given argument slot will maintain a given outside equivalence. For example, `(length a)` is equal to `(length a')` if `a` and `a'` are related either by `list-equal` or by `string-equal^`. You may prove two (or more) `:congruence` rules for the same slot of a function. The result is that the system uses a new, "generated" equivalence relation for that slot with the result that rules of both (or all) kinds are available while rewriting.

`:congruence` rules can be disabled. For example, if you have two different inside equivalences for a given argument position and you find that the `:rewrite^` rules for one are unexpectedly preventing the application of the desired rule, you can disable the rule that introduced the unwanted inside equivalence.

More will be written about this as we develop the techniques.

14.4 DEFINITION

DEFINITION make a rule that acts like a function definition

```
Example:
(implies (true-listp x)
         (equal (len x)
                (if (null x)
                    0
                    (if (null (cdr x))
                        1
                        (len (cddr x))))))

General Form:
(implies hyp (equiv (fn a1 ... an) body))
```

where `equiv` is an equivalence relation and `fn` is a function symbol other than `if^`, `hide^` or `force^`. Such rules allow "alternative" definitions of `fn` to be proved as theorems but used as definitions. These rules are not true "definitions" in the sense that they (a) cannot introduce new function symbols and (b) do not have to be terminating recursion schemes. They are just conditional rewrite rules that are controlled the same way we control recursive definitions. We call these "definition rules" or "generalized definitions".

Consider the general form above. Generalized definitions are stored among the `:rewrite^` rules for the function "defined," `fn` above, but the procedure for applying them is a little different. During rewriting, instances of `(fn a1 ... an)` are replaced by corresponding instances of `body` provided the `hyp`s can be established as for a `:rewrite^` rule and the result of rewriting `body` satisfies the criteria for function expansion. There are two primary criteria, either of which permits expansion. The first is that the "recursive" calls of `fn` in the rewritten body have arguments that already occur in the goal conjecture. The second is that the "controlling" arguments to `fn` are simpler in the rewritten body.

The notions of "recursive call" and "controllers" are complicated by the provisions for mutually recursive definitions. Consider a "clique" of mutually recursive definitions. Then a "recursive call" is a call to any function defined in the clique and an argument is a "controller" if it is involved in the measure that decreases in all recursive calls. These notions are precisely defined by the definitional principle and do not necessarily make sense in the context of generalized definitional equations as implemented here.

But because the heuristics governing the use of generalized definitions require these notions, it is generally up to the user to specify which calls in body are to be considered recursive and what the controlling arguments are. This information is specified in the `:clique` and `:controller-alist` fields of the `:definition` rule class.

The `:clique` field is the list of function symbols to be considered recursive calls of `fn`. In the case of a non-recursive definition, the `:clique` field is empty; in a singly recursive definition, it should consist of the singleton list containing `fn`; otherwise it should be a list of all of the functions in the mutually recursive clique with this definition of `fn`.

If the `:clique` field is not provided it defaults to `nil` if `fn` does not occur as a function symbol in `body` and it defaults to the singleton list containing `fn` otherwise. Thus, `:clique` must be supplied by the user only when the generalized definition rule is to be treated as one of several in a mutually recursive clique.

The `:controller-alist` is an alist that maps each function symbol in the `:clique` to a mask specifying which arguments are considered controllers. The mask for a given member of the clique, `fn`, must be a list of `t`'s and `nil`'s of length equal to the arity of `fn`. A `t` should be in each argument position that is considered a "controller" of the recursion. For a function admitted under the principle of definition, an argument controls the recursion if it is one of the arguments measured in the termination argument for the function. But in generalized definition rules, the user is free to designate any subset of the arguments as controllers. Failure to choose wisely may result in the "infinite expansion" of definitional rules but cannot render ACL2 unsound since the rule being misused is a theorem.

If the `:controller-alist` is omitted it can sometimes be defaulted automatically by the system. If the `:clique` is `nil`, the `:controller-alist` defaults to `nil`. If the `:clique` is a singleton containing `fn`, the `:controller-alist` defaults to the controller alist computed by `(defun fn args body)`. If the `:clique` contains more than one function, the user must supply the `:controller-alist` specifying the controllers for each function in the clique. This is necessary since the system cannot determine and thus cannot analyze the other definitional equations to be included in the clique.

For example, suppose `fn1` and `fn2` have been defined one way and it is desired to make "alternative" mutually recursive definitions available to the rewriter. Then one would prove two theorems and store each as a `:definition` rule. These two theorems would exhibit equations "defining" `fn1` and `fn2` in terms of each other. No provision is here made for exhibiting these two equations as a system of equations. One is proved and then the other. It just so happens that the user intends them to be treated as mutually recursive definitions. To achieve this end, both `:definition` rules should specify the `:clique (fn1 fn2)` and should specify a suitable `:controller-alist`. If, for example, the new definition of `fn1` is controlled by its first argument and the new definition of `fn2` is controlled by its second and third (and they each take three arguments) then a suitable `:controller-alist` would be `((fn1 t nil nil) (fn2 nil t t))`. The order of the pairs in the alist is unimportant, but there must be a pair for each function in the clique.

Inappropriate heuristic advice via `:clique` and `:controller-alist` can cause "infinite expansion" of generalized definitions, but cannot render ACL2 unsound.

Note that the actual definition of `fn1` has the runic name `(:definition fn1)`. The runic name of the alternative definition is `(:definition lemma)`, where `lemma` is the name given to the event that created the generalized `:definition` rule. This allows theories to switch between various "definitions" of the functions.

The definitional principle, `defun^`, actually adds `:definition` rules. Thus the handling of generalized definitions is exactly the same as for "real" definitions because no distinction is made in the implementation. Suppose `(fn x y)` is `defun^`'d to be `body`. Note that `defun^` (or `defuns^` or `mutual-recursion^`) can compute the clique for `fn` from the syntactic presentation and it can compute the controllers from the termination analysis. Provided the definition is admissible, `defun^` adds the `:definition` rule `(equal (fn x y) body)`. Thus, `(fn a b)` will not be expanded unless `(g a b)` can be established.

14.5 ELIM

ELIM make a destructor elimination rule

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which an `:elim` rule might be built is:

```
Example:
(implies (consp x)                       when (car v) or (cdr v) appears
         (equal (cons (car x) (cdr x))   in a conjecture, and v is a
                x))                      variable, consider replacing v by
                                         (cons a b), for two new variables
                                         a and b.

General Form:
(implies hyp (equal lhs x))
```

where `x` is a variable symbol and `lhs` contains one or more terms (called "destructor terms") of the form `(fn v1 ... vn)`, where `fn` is a function symbol and the `vi` are distinct variable symbols, `v1`, ..., `vn` include all the variable symbols in the formula, no `fn` occurs in `lhs` in more than one destructor term, and all occurrences of `x` in `lhs` are inside destructor terms.

To use an `:elim` rule, the theorem prover waits until a conjecture has been maximally simplified. If it then finds an instance of some destructor term `(fn v1 ... vn)` in the conjecture, where the instance for `x` is some variable symbol, `vi`, it instantiates the `:elim` formula as indicated by the destructor term matched, splits the conjecture into two goals, according to whether the instantiated hypothesis, `hyp`, holds, and in the case that it does, generalizes all the instantiated destructor terms in the conjecture to new variables and then replaces `vi` in the conjecture by the generalized instantiated `lhs`.

At the moment, the best description of how ACL2 `:elim` rules are used may be found in the discussion of "ELIM Rules," pp 247 of A Computational Logic Handbook.

14.6 EQUIVALENCE

EQUIVALENCE mark a relation as an equivalence relation

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:equivalence` rule might be built is as follows. (We assume that `r-equal` has been defined.)

```
Example:
(and (booleanp (r-equal x y))
     (r-equal x x)
     (implies (r-equal x y) (r-equal y x))
     (implies (and (r-equal x y)
                   (r-equal y z))
              (r-equal x z))).
```

Also see Section 6.7 [DEFEQUIV], page 160.

```
General Form:
(and (booleanp (equiv x y))
     (equiv x x)
     (implies (equiv x y) (equiv y x))
     (implies (and (equiv x y)
                   (equiv y z))
              (equiv x z)))
```

except that the order of the conjuncts and terms and the choice of variable symbols is unimportant. The effect of such a rule is to identify `equiv` as an equivalence relation. Note that only Boolean 2-

place function symbols can be treated as equivalence relations. See Section 14.3 [CONGRUENCE], page 627 and see Section 14.13 [REFINEMENT], page 656 for closely related concepts.

The macro form `(defequiv equiv)` is an abbreviation for a `defthm^` of rule-class `:equivalence` that establishes that `equiv` is an equivalence relation. It generates the formula shown above. See Section 6.7 [DEFEQUIV], page 160.

When `equiv` is marked as an equivalence relation, its reflexivity, symmetry, and transitivity are built into the system in a deeper way than via `:rewrite^` rules. More importantly, after `equiv` has been shown to be an equivalence relation, lemmas about `equiv`, e.g.,

```
(implies hyps (equiv lhs rhs)),
```

when stored as `:rewrite^` rules, cause the system to rewrite certain occurrences of (instances of) `lhs` to (instances of) `rhs`. Roughly speaking, an occurrence of `lhs` in the `kth` argument of some `fn`-expression, `(fn ... lhs' ...)`, can be rewritten to produce `(fn ... rhs' ...)`, provided the system "knows" that the value of `fn` is unaffected by `equiv`-substitution in the `kth` argument. Such knowledge is communicated to the system via "congruence lemmas."

For example, suppose that `r-equal` is known to be an equivalence relation. The `:congruence^` lemma

```
(implies (r-equal s1 s2)
         (equal (fn s1 n) (fn s2 n)))
```

informs the rewriter that, while rewriting the first argument of `fn`-expressions, it is permitted to use `r-equal` rewrite-rules. See Section 14.3 [CONGRUENCE], page 627 for details about `:congruence^` lemmas. Interestingly, congruence lemmas are automatically created when an equivalence relation is stored, saying that either of the equivalence relation's arguments may be replaced by an equivalent argument. That is, if the equivalence relation is `fn`, we store congruence rules that state the following fact:

```
(implies (and (fn x1 y1)
              (fn x2 y2))
         (iff (fn x1 x2) (fn y1 y2)))
```

Another aspect of equivalence relations is that of "refinement." We say `equiv1` "refines" `equiv2` iff `(equiv1 x y)` implies `(equiv2 x y)`. `:refinement^` rules permit you to establish such connections between your equivalence relations. The value of refinements is that if the system is trying to rewrite something while maintaining `equiv2` it is permitted to use as a `:rewrite^` rule any refinement of `equiv2`. Thus, if `equiv1` is a refinement of `equiv2` and there are `equiv1` rewrite-rules available, they can be brought to bear while maintaining `equiv2`. See Section 14.13 [REFINEMENT], page 656.

The system initially has knowledge of two equivalence relations, equality, denoted by the symbol `equal^`, and propositional equivalence, denoted by `iff^`. `Equal^` is known to be a refinement of all equivalence relations and to preserve equality across all arguments of all functions.

Typically there are five steps involved in introducing and using a new equivalence relation, equiv.

> (1) Define `equiv`,
>
> (2) prove the `:equivalence` lemma about `equiv`,
>
> (3) prove the `:congruence^` lemmas that show where `equiv` can be used to maintain known relations,
>
> (4) prove the `:refinement^` lemmas that relate `equiv` to known relations other than equal, and
>
> (5) develop the theory of conditional `:rewrite^` rules that drive equiv rewriting.

More will be written about this as we develop the techniques. For now, here is an example that shows how to make use of equivalence relations in rewriting.

Among the theorems proved below is

```
(defthm insert-sort-is-id
  (perm (insert-sort x) x))
```

Here `perm` is defined as usual with `delete` and is proved to be an equivalence relation and to be a congruence relation for `cons^` and `member^`.

Then we prove the lemma

```
(defthm insert-is-cons
  (perm (insert a x) (cons a x)))
```

which you must think of as you would `(insert a x) = (cons a x)`.

Now prove `(perm (insert-sort x) x)`. The base case is trivial. The induction step is

```
    (consp x)
  & (perm (insert-sort (cdr x)) (cdr x))

 -> (perm (insert-sort x) x).
```

Opening `insert-sort` makes the conclusion be

```
    (perm (insert (car x) (insert-sort (cdr x))) x).
```

Then apply the induction hypothesis (rewriting `(insert-sort (cdr x))` to `(cdr x)`), to make the conclusion be

```
(perm (insert (car x) (cdr x)) x)
```

Then apply `insert-is-cons` to get `(perm (cons (car x) (cdr x)) x)`. But we know that `(cons (car x) (cdr x))` is `x`, so we get `(perm x x)` which is trivial, since `perm` is an equivalence relation.

Here are the events.

```
(encapsulate ((lt (x y) t))
  (local (defun lt (x y) (declare (ignore x y)) nil))
  (defthm lt-non-symmetric (implies (lt x y) (not (lt y x)))))

(defun insert (x lst)
  (cond ((atom lst) (list x))
        ((lt x (car lst)) (cons x lst))
        (t (cons (car lst) (insert x (cdr lst))))))

(defun insert-sort (lst)
  (cond ((atom lst) nil)
        (t (insert (car lst) (insert-sort (cdr lst))))))

(defun del (x lst)
  (cond ((atom lst) nil)
        ((equal x (car lst)) (cdr lst))
        (t (cons (car lst) (del x (cdr lst))))))

(defun mem (x lst)
  (cond ((atom lst) nil)
        ((equal x (car lst)) t)
        (t (mem x (cdr lst)))))
```

```
(defun perm (lst1 lst2)
  (cond ((atom lst1) (atom lst2))
        ((mem (car lst1) lst2)
         (perm (cdr lst1) (del (car lst1) lst2)))
        (t nil)))

(defthm perm-reflexive
  (perm x x))

(defthm perm-cons
  (implies (mem a x)
           (equal (perm x (cons a y))
                  (perm (del a x) y)))
  :hints (("Goal" :induct (perm x y))))

(defthm perm-symmetric
  (implies (perm x y) (perm y x)))

(defthm mem-del
  (implies (mem a (del b x)) (mem a x)))

(defthm perm-mem
  (implies (and (perm x y)
                (mem a x))
           (mem a y)))

(defthm mem-del2
  (implies (and (mem a x)
                (not (equal a b)))
           (mem a (del b x))))

(defthm comm-del
  (equal (del a (del b x)) (del b (del a x))))

(defthm perm-del
  (implies (perm x y)
           (perm (del a x) (del a y))))

(defthm perm-transitive
  (implies (and (perm x y) (perm y z)) (perm x z)))

(defequiv perm)

(in-theory (disable perm
                    perm-reflexive
                    perm-symmetric
                    perm-transitive))
```

```
(defcong perm perm (cons x y) 2)

(defcong perm iff (mem x y) 2)

(defthm atom-perm
  (implies (not (consp x)) (perm x nil))
  :rule-classes :forward-chaining
  :hints (("Goal" :in-theory (enable perm))))

(defthm insert-is-cons
  (perm (insert a x) (cons a x)))

(defthm insert-sort-is-id
  (perm (insert-sort x) x))

(defun app (x y) (if (consp x) (cons (car x) (app (cdr x) y)) y))

(defun rev (x)
  (if (consp x) (app (rev (cdr x)) (list (car x))) nil))

(defcong perm perm (app x y) 2)

(defthm app-cons
  (perm (app a (cons b c)) (cons b (app a c))))

(defthm app-commutes
  (perm (app a b) (app b a)))

(defcong perm perm (app x y) 1
  :hints (("Goal" :induct (app y x))))

(defthm rev-is-id (perm (rev x) x))

(defun == (x y)
  (if (consp x)
      (if (consp y)
          (and (equal (car x) (car y))
               (== (cdr x) (cdr y)))
          nil)
      (not (consp y))))

(defthm ==-symmetric (== x x))

(defthm ==-reflexive (implies (== x y) (== y x)))

(defequiv ==)

(in-theory (disable ==-symmetric ==-reflexive))
```

```
(defcong == == (cons x y) 2)

(defcong == iff (consp x) 1)

(defcong == == (app x y) 2)

(defcong == == (app x y) 1)

(defthm rev-rev (== (rev (rev x)) x))
```

14.7 FORWARD-CHAINING

FORWARD-CHAINING make a rule to forward chain when a certain trigger arises

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:forward-chaining` rule might be built is:

```
Example:
(implies (and (p x) (r x))        when (p a) appears in a formula to be
         (q (f x)))               simplified, try to establish (r a) and
                                  if successful, add (q (f a)) to the
                                  known assumptions
```

To specify the triggering terms provide a non-empty list of terms as the value of the `:trigger-terms` field of the rule class object.

```
General Form:
Any theorem, provided an acceptable triggering term exists.
```

Forward chaining rules are generated from the corollary term as follows. If that term has the form `(implies hyp concl)`, then the `let^` expressions in `concl` (formally, lambda expressions) are expanded away, and the result is treated as a conjunction, with one forward chaining rule with hypothesis hyp created for each conjunct. In the other case, where the corollary term is not an `implies^`, we process it as we process the conclusion in the first case.

The `:trigger-terms` field of a `:forward-chaining` rule class object should be a non-empty list of terms, if provided, and should have certain properties described below. If the `:trigger-terms` field is not provided, it defaults to the singleton list containing the "atom" of the first hypothesis of the formula. (The atom of `(not x)` is `x`; the atom of any other term is the term itself.) If there are no hypotheses and no `:trigger-terms` were provided, an error is caused.

A triggering term is acceptable if it is not a variable, a quoted constant, a lambda application, a `let^`-expression, or a `not^`-expression, and every variable symbol in the conclusion of the theorem either occurs in the hypotheses or occurs in the trigger.

`:Forward-chaining` rules are used by the simplifier before it begins to rewrite the literals of the goal. If any term in the goal is an instance of a trigger of some forward chaining rule, we try to establish the hypotheses of that forward chaining theorem (from the negation of the goal). To relieve a hypotheses we only use type reasoning, evaluation of ground terms, and presence among our known assumptions. We do not use rewriting. If all hypotheses are relieved, we add the instantiated conclusion to our known assumptions.

14.8 GENERALIZE

GENERALIZE make a rule to restrict generalizations

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:generalize` rule might be built is:

```
Example:
any theorem

General Form:
any theorem
```

To use such a `:generalize` rule, the system waits until it has decided to generalize some term, `term`, by replacing it with some new variable `v`. If any `:generalize` formula can be instantiated so that some non-variable subterm becomes `term`, then that instance of the formula is added as a hypothesis.

At the moment, the best description of how ACL2 `:generalize` rules are used may be found in the discussion of "Generalize Rules," pp 248 of A Computational Logic Handbook.

14.9 INDUCTION

INDUCTION make a rule that suggests a certain induction

```
Example:
(:induction :corollary t  ; the theorem proved is irrelevant!
            :pattern (* 1/2 i)
            :condition (and (integerp i) (>= i 0))
            :scheme (recursion-by-sub2 i))
```

In ACL2, as in Nqthm, the functions in a conjecture "suggest" the inductions considered by the system. Because every recursive function must be admitted with a justification in terms of a measure that decreases in a well-founded way on a given set of "controlling" arguments, every recursive function suggests a dual induction scheme that "unwinds" the function from a given application.

For example, since `append^` (actually `binary-append^`, but we'll ignore the distinction here) decomposes its first argument by successive `cdr^`s as long as it is a non-`nil` true list, the induction scheme suggested by `(append x y)` has a base case supposing `x` to be either not a true list or to be `nil` and then has an induction step in which the induction hypothesis is obtained by replacing `x` by `(cdr x)`. This substitution decreases the same measure used to justify the definition of `append^`. Observe that an induction scheme is suggested by a recursive function application only if the controlling actuals are distinct variables, a condition that is sufficient to insure that the "substitution" used to create the induction hypothesis is indeed a substitution and that it drives down a certain measure. In particular, `(append (foo x) y)` does not suggest an induction unwinding `append^` because the induction scheme suggested by `(append x y)` requires that we substitute `(cdr x)` for `x` and we cannot do that if `x` is not a variable symbol.

Once ACL2 has collected together all the suggested induction schemes it massages them in various ways, combining some to simultaneously unwind certain cliques of functions and vetoing others because they "flaw" others. We do not further discuss the induction heuristics here; the interested reader should see Chapter XIV of A Computational Logic (Boyer and Moore, Academic Press, 1979) which represents a fairly complete description of the induction heuristics of ACL2.

However, unlike Nqthm, ACL2 provides a means by which the user can elaborate the rules under which function applications suggest induction schemes. Such rules are called `:induction` rules. The definitional principle automatically creates an `:induction` rule, named `(:induction fn)`, for each admitted recursive function, `fn`. It is this rule that links applications of `fn` to the induction scheme it suggests. Disabling `(:induction fn)` will prevent `fn` from suggesting the induction scheme derived from its recursive definition. It is possible for the user to create additional `:induction` rules by using the `:induction` rule class in `defthm^`.

Technically we are "overloading" `defthm^` by using it in the creation of `:induction` rules because no theorem need be proved to set up the heuristic link represented by an `:induction` rule. However, since `defthm^` is generally used to create rules and rule-class objects are generally used to specify the exact form of each rule, we maintain that convention and introduce the notion of an `:induction` rule. An `:induction` rule can be created from any lemma whatsoever.

```
General Form of an :induction Lemma or Corollary:
T

General Form of an :induction rule-class:
(:induction :pattern pat-term
            :condition cond-term
            :scheme scheme-term)
```

where `pat-term`, `cond-term`, and `scheme-term` are all terms, `pat-term` is the application of a function symbol, `fn`, `scheme-term` is the application of a function symbol, `rec-fn`, that suggests an induction, and, finally, every free variable of `cond-term` and `scheme-term` is a free variable of `pat-term`. We actually check that `rec-fn` is either recursively defined — so that it suggests the induction that is intrinsic to its recursion — or else that another `:induction` rule has been proved linking a call of `rec-fn` as the `:pattern` to some scheme.

The induction rule created is used as follows. When an instance of the `:pattern` term occurs in a conjecture to be proved by induction and the corresponding instance of the `:condition` term is known to be non-`nil` (by type reasoning alone), the corresponding instance of the `:scheme` term is created and the rule "suggests" the induction, if any, suggested by that term. If `rec-fn` is recursive, then the suggestion is the one that unwinds that recursion.

Consider, for example, the example given above,

```
(:induction :pattern (* 1/2 i)
            :condition (and (integerp i) (>= i 0))
            :scheme (recursion-by-sub2 i)).
```

In this example, we imagine that `recursion-by-sub2` is the function:

```
(defun recursion-by-sub2 (i)
  (if (and (integerp i)
           (< 1 i))
      (recursion-by-sub2 (- i 2))
      t))
```

Observe that this function recursively decomposes its integer argument by subtracting 2 from it repeatedly and stops when the argument is 1 or less. The value of the function is irrelevant; it is its induction scheme that concerns us. The induction scheme suggested by (recursion-by-sub2 i) is

```
(and (implies (not (and (integerp i) (< 1 i)))   ; base case
              (:p i))
     (implies (and (and (integerp i) (< 1 i))    ; induction step
                   (:p (- i 2)))
              (:p i)))
```

We can think of the base case as covering two situations. The first is when i is not an integer. The second is when the integer i is 0 or 1. In the base case we must prove (:p i) without further help. The induction step deals with those integer i greater than 1, and inductively assumes the conjecture for i-2 while proving it for i. Let us call this scheme "induction on i by twos."

Suppose the above :induction rule has been added. Then an occurrence of, say, (* 1/2 k) in a conjecture to be proved by induction would suggest, via this rule, an induction on k by twos, provided k was known to be a nonnegative integer. This is because the induction rule's :pattern is matched in the conjecture, its :condition is satisfied, and the :scheme suggested by the rule is that derived from (recursion-by-sub2 k), which is induction on k by twos. Similarly, the term (* 1/2 (length l)) would suggest no induction via this rule, even though the rule "fires" because it creates the :scheme (recursion-by-sub2 (length l)) which suggests no inductions unwinding recursion-by-sub2 (since the controlling argument of recursion-by-sub2 in this :scheme is not a variable symbol).

Continuing this example one step further illustrates the utility of :induction rules. We could define the function recursion-by-cddr that suggests the induction scheme decomposing its consp^ argument two cdr^s at a time. We could then add the :induction rule linking (* 1/2 (length x)) to (recursion-by-cddr x) and arrange for (* 1/2 (length l)) to suggest induction on l by cddr^.

Observe that :induction rules require no proofs to be done. Such a rule is merely a heuristic link between the :pattern term, which may occur in conjectures to be proved by induction, and the :scheme term, from which an induction scheme may be derived. Hence, when an :induction rule-class is specified in a defthm^ event, the theorem proved is irrelevant. The easiest theorem to prove is, of course, t. Thus, we suggest that when an :induction rule is to be created, the following form be used:

```
(defthm name T
```

```
:rule-classes ((:induction :pattern pat-term
                           :condition cond-term
                           :scheme scheme-term)))
```

The name of the rule created is `(:induction name)`. When that rune is disabled the heuristic link between `pat-term` and `scheme-term` is broken.

14.10 LINEAR

LINEAR make some arithmetic inequality rules

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:linear` rule might be built is:

```
Example:
(implies (and (eqlablep e)           if inequality reasoning begins to
              (true-listp x))        consider how (length (member a b))
         (>= (length x)              compares to any other term, add to
             (length (member e x)))) set of known inequalities the fact
                                     that it is no larger than (length b),
                                     provided (eqlablep a) and (true-listp b)
                                     rewrite to t

General Form:
(and ...
     (implies (and ...hi...)
              (implies (and ...hk...)
                       (and ...
                            (rel lhs rhs)
                            ...)))
     ...)
```

Note: One `:linear` rule class object might create many linear arithmetic rules from the `:corollary^` formula. To create the rules, we first flatten the `and^` and `implies^` structure of the formula, transforming it into a conjunction of formulas, each of the form

```
(implies (and h1 ... hn) (rel lhs rhs))
```

where no hypothesis is a conjunction and `rel` is one of the inequality relations `<^`, `<=^`, `=^`, `/=^`, `>^`, or `>=^`. If necessary, the hypothesis of such a conjunct may be vacuous.

We create a `:linear` rule for each such conjunct, if possible, and otherwise cause an error. Each rule has one or more "trigger terms" which may be specified by the user using the `:trigger-terms` field of the rule class or which may be defaulted to values chosen by the system. We discuss the determination of trigger terms after discussing how linear rules are used.

`:linear` rules are used by a linear arithmetic decision procedure during rewriting. We describe the procedure very roughly here. Fundamental to the procedure is the notion of a linear polynomial inequality. A "linear polynomial" is a sum of terms, each of which is the product of a rational constant and an "unknown." The "unknown" is permitted to be `1` simply to allow a term in the sum to be constant. Thus, an example linear polynomial is `3*x + 7*a + 2`; here `x` and `a` are the (interesting) unknowns. In our case, the unknowns need not be variable symbols. For example, `(length x)` might be used as an unknown in a linear polynomial. Thus, another linear polynomial is `3*(length x) + 7*a`. A "linear polynomial inequality" is an inequality (either `<^` or `<=^`) relation between two linear polynomials. Certain linear polynomial inequalities can be combined by cross-multiplication and addition to permit the deduction of a third linear inequality polynomial with fewer unknowns. If this deduced inequality is manifestly false, a contradiction has been deduced from the assumed inequalities.

For example, suppose we have three assumptions:

```
p1:  3*x + 7*a <  4
p2:          3 <  2*x
p3:          0 <= a.
```

By cross-multiplying and adding the first two (that is, multiplying `p1` by `2`, `p2` by `3` and adding the respective sides), we deduce the intermediate result

```
p4:  6*x + 14*a + 9 < 8 + 6*x
```

which, after cancellation, is

```
p4:        14*a + 1 <  0.
```

If we then cross-multiply and add `p3` to `p4`, we get

```
p5:                1 <= 0,
```

a contradiction. Thus, we have proved that `p1` and `p2` imply the negation of `p3`.

All of the unknowns of a polynomial must be eliminated by cancellation in order to produce a constant inequality. We can choose to eliminate the unknowns in any order. We eliminate them in term-order, largest unknowns first. (See Section 8.86 [TERM-ORDER], page 355.) Now suppose that this procedure does not produce a contradiction but instead yields a set of nontrivial inequalities. A contradiction might still be deduced if we could add to the set some additional inequalities allowing additional cancellation. That is where `:linear` lemmas come in. When the set of inequalities has stabilized under cross-multiplication and addition and no contradiction is produced, we search the data base of `:linear` rules for rules about the unknowns that are candidates for cancellation (i.e., are the largest unknowns in their respective inequalities).

If the trigger term of some `:linear` rule can be instantiated to yield a candidate for cancellation, we so instantiate that rule, attempt to relieve the hypotheses with general-purpose rewriting, and, if successful, convert the concluding inequality into a polynomial inequality and add it to the set. This may let cancellation continue. See the discussion of "Linear Arithmetic Rules" pp 242 of A Computational Logic Handbook for more details.

We now describe how the trigger terms are determined. Most of the time, the trigger terms are not specified by the user and are instead selected by the system. However, the user may specify the terms by including an explicit `:trigger-terms` field in the rule class, e.g.,

```
General Form of a Linear Rule Class:
(:LINEAR :COROLLARY formula
         :TRIGGER-TERMS (term1 ... termk))
```

Each `termi` must be a term and must not be a variable, quoted constant, lambda application, `let-expression` or `if-expression`. In addition, each `termi` must be such that if all the variables in the term are instantiated and then the hypotheses of the corollary formula are relieved (possibly instantiating additional free variables), then all the variables in the concluding inequality are instantiated. We generate a linear rule for each conjuctive branch through the corollary and store each rule under each of the specified triggers. Thus, if the corollary formula contains several conjuncts, the variable restrictions on the `termi` must hold for each conjunct.

Note: Problems may arise if you explicitly store a linear lemma under a trigger term that, when instantiated, is not the largest unknown in the instantiated concluding inequality. Suppose for example you store the linear rule `(<= (fn i j) (/ i (* j j)))` under the trigger term `(fn i j)`. Then when the system "needs" an inequality about `(fn a b)`, i.e., because `(fn a b)` is the largest unknown in some inequality in the set of assumed inequalities, it will appeal to the rule and deduce `(<= (fn a b) (/ a (* b b)))`. However, the largest unknown in this inequality is `(/ a (* b b))` and hence this inequality will not be used until that term is eliminated by cancellation with some other inequality. It is generally best to specify as a trigger term one of the "maximal" terms of the polynomial, as described below.

If `:trigger-terms` is omitted the system computes a set of trigger terms. Each conjunct of the corollary formula may be given a unique set of triggers depending on the variables that occur in the conjunct and the addends that occur in the concluding inequality. In particular, the trigger terms for a conjunct is the list of all "maximal addends" in the concluding inequality.

The "addends" of `(+ x y)` and `(- x y)` are the union of the addends of `x` and `y`. The addends of `(- x)` and `(* n x)`, where `n` is a numeric constant, is in all cases just `{x}`. The addends of an inequality are the union of the addends of the left- and right-hand sides. The addends of any other term, `x`, is `{x}`.

A term is maximal for a conjunct `(implies hyps concl)` of the corollary if (a) the term is a non-variable, non-quote, non-lambda application, non-`let^` and non-`if^` expression, (b) the term contains enough variables so that when they are instantiated and the hypotheses are relieved then all the variables in `concl` are instantiated, and (c) any instantiation of the term is always at least as "large" (see Section 8.86 [TERM-ORDER], page 355 for a description of the order used) as the corresponding instantiations of any other addend in `concl`.

14.11 LINEAR-ALIAS

LINEAR-ALIAS make a rule to extend the applicability of linear arithmetic

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:linear-alias` rule might be built is as follows. (Of course, the functions `fix`, `plus`, `diff`, and `lessp` would have to be defined first.)

```
Example:
(and (equal (fix (plus x y)) (plus x y))
     (equal (fix (diff x y)) (diff x y))
     (equal (fix (fix x)) (fix x))
     (equal (lessp x y) (< (fix x) (fix y)))
     (equal (plus x y) (+ (fix x) (fix y)))
     (implies (not (lessp x y))
              (equal (diff x y) (- (fix x) (fix y)))))
```

`:linear-alias` rules are essentially a hack to allow the creation of the `"NQTHM"` package. We will be surprised if the casual user of ACL2 ever finds a use for them. They are not documented very thoroughly.

```
General Forms:
(equal lhs rhs)
(implies hyps (equal lhs rhs))
```

where every variable that occurs in the rule must occur in the lhs. As for `:rewrite^` rules, we strip the conjuncts out of the `:corollary^` formula and generate one `:linear-alias` rule for each. Thus, it is possible to package together a complete set of `:linear-alias` rules under a single name.

When linear arithmetic is applied to a term that is not "linearizable," i.e., a term that does not start with `<^`, `+^`, or `-^`, it attempts to transduce the term to a linearizable one by applying all known `:linear-alias` rules as rewrite rules. This transduction is performed by the simple rewrite function `rewrite-with-linear-aliases`. Like the standard ACL2 rewriter, the linear alias rewriter applies linear alias rules exhaustively in the sense that if a rule applies then the `rhs` of the rule is recursively rewritten. Unlike `:rewrite^` rules, `:linear-alias` rules are applied from outside in (!) and hypotheses are not relieved but merely collected.

For example, rewriting `(lessp (diff (plus a b) c) (diff u v))` with the rules generated from the conjunction above produces

```
(< (+ (+ (fix a) (fix b))
      (- (fix c)))
   (+ (fix u) (- (fix v))))
```

under the hypothesis that `(not (lessp (plus a b) c))` and `(not (lessp u v))`. (Actually, the `+^` and `-^` operators shown above are really calls of `binary-+^` and `unary--^`.)

Think of `:linear-alias` rules as transforming a term from one theory to another, generating hypotheses in the first theory. Call the initial theory "nqthm" and the final one "acl2." One should strive to create a set of rules that transform from the nqthm to the acl2 theory while only generating hypotheses in the first theory. The restriction is motivated by the desire not to surprise the nqthm user by suddently introducing conditions that are not phrased in the nqthm theory. The purpose of the nqthm package is to emulate nqthm, so it simply represents a failure if acl2 suddenly produces a case split, say, in which some non-nqthm term appears, such as `(rationalp (plus a b))`. To this end, we offer some advice below. However, it is especially helpful to study how the outside-in rewrite strategy uses the rules above to achieve this goal.

For example,

```
(lessp (plus a b) (diff u v))
```

first steps to

```
(< (fix (plus a b)) (fix (diff u v))),
```

unconditionally. But `plus` and `diff` don't need `fix^`ing:

```
(< (plus a b) (diff u v)).
```

Rewriting further finishes with

```
(< (+ (fix a) (fix b)) (+ (fix u) (- (fix v)))).
```

Observe that had we rewritten inside-out the sequence would have been:

```
(lessp (+ (fix a) (fix b)) (+ (fix u) (- (fix v))))
```

and then

```
(< (fix (+ (fix a) (fix b))) (fix  (+ (fix u) (- (fix v))))).
```

Note that we now need to know that `+^` doesn't need `fix^`ing if its arguments have been. But that is a more complicated rule than the simple one that `plus` doesn't need `fix^`ing. Perhaps a different set of rules would work well for inside-out rewriting, but I just used the ones that first leapt to mind. It is interesting to note that linearization proceeds outside in too.

Here then is the advice. Rules should either transform nqthm terms to nqthm terms, e.g., `(fix (plus x y))` to `(plus x y)`, or should transform nqthm terms to acl2 terms, e.g., `(lessp x y)` to `(< (fix x) (fix y))`. When the `rhs` introduces an acl2 function symbol, the symbol should be outside of all nqthm terms in the `rhs`. Hypotheses should be expressed in the nqthm theory. No rule should rewrite an acl2 function symbol. If the transformation involves "fixers" then each operator should have two rules about it. The first, like `(plus x y) = (+ (fix x) (fix y))`, explains the nqthm function in terms of an acl2 function with fixed arguments. The second, like `(fix (plus x y)) = (plus x y)` eliminates the interior fixes generated by the first rule. These conventions mean that the rewriter, by going from outside in, always sees nqthm terms and instantiates the hypotheses of rules with nqthm terms. To violate these conventions might produce a set of rules that would cause a non-nqthm hypothesis to be generated.

Finally, observe that the rules probably will introduce a "fixer", which is the function `fix^` in the NQTHM package. The fixer tends to settle around the nonlinear leaves of the resultant linearizable term. It is important to have a `:type-prescription^` rule establishing that the fixer produces a rational.

When a `:linear^` arithmetic rule is proved its conclusion is linearized to obtain the relevant polynomials. `:linear-alias` rules may kick in there; thus, a conclusion of the form `(not (lessp (length str) (strpos pat str)))` may be a legal `:linear^` rule. Note however that its linearization is `(not (< (fix (length str)) (fix (strpos pat str))))`. The maximal term of this rule is `(fix (strpos pat str))`. However, we do not want to store such a linear rule under `fix^` because that would result in all of the Nqthm proveall's linear rules being stored under the same symbol. We therefore note when the maximal term does not actually occur in the conclusion, is of the form `(fix term)`, for some unary function `fix^`, and term does occur in the conclusion. In that case, we call `fix^` a "fixer" and we store a `t` under the property `'linear-alias-fixerp`. We then store the rule with the maximal term `term` instead of `(fix term)`. Later, when we are adding lemmas to the linear pot, we note that if we seek linear lemmas about `(fix term)`, where `fix^` is a fixer, we look for lemmas both stored under the function symbol of term and stored under the function symbol `fix^`. In all cases, the maximal term of the rule contains as its top-most function symbol the symbol under which the rule is stored. Thus, if we seek lemmas about `(fix (strpos pat str))` we will go to the property list of `strpos` and there find rules about `(strpos pat str)`. We will instantiate their conclusions and linearize them, producing polynomials about `(fix (strpos pat str))`. This raises a subtle point. It could be that there is more than one fixer used in connection with `strpos`. Thus, some lemmas could linearize to polynomials about `(rat (strpos pat str))`. Therefore, when we create the linear rule with `(strpos pat str)` as its (fixed) maximal term, we store in the `:fixer` field of the rule the symbol `fix^` to indicate that the linearization of this conclusion will (probably) produce `(fix (strpos pat str))` terms. When looking for lemmas about `(fix (strpos pat str))` we actually look for lemmas about `strpos` that have `:fixer fix^`.

Here is an extended example of the use of linear aliasing. First, we define some nqthm-level functions:

```
(defun fix (x) (if (and (integerp x) (>= x 0)) x 0))
(defun lessp (x y) (< (fix x) (fix y)))
(defun plus (x y) (+ (fix x) (fix y)))
(defun diff (x y) (fix (- (fix x) (fix y))))
```

Note that (fix x) gets a :type-prescription^ rule that it is a nonnegative integer.

Now we prove a :linear-alias rule, actually a package of the six rules above.

```
(defthm nqthm-linear-arithmetic
  (and (equal (fix (plus x y)) (plus x y))
       (equal (fix (diff x y)) (diff x y))
       (equal (fix (fix x)) (fix x))
       (equal (lessp x y) (< (fix x) (fix y)))
       (equal (plus x y) (+ (fix x) (fix y)))
       (implies (not (lessp x y))
                (equal (diff x y) (- (fix x) (fix y)))))
  :rule-classes :linear-alias)
```

To mimic Nqthm behavior, we need a :rewrite^ rule to dispatch the cases generated by the case split caused by linearizing diff-expressions. This is not, strictly speaking, part of linear aliasing.

```
(defthm diff-0 (implies (lessp x y) (equal (diff x y) 0)))
```

Now we disable the nqthm-level functions. Henceforth, we do not expect to see ACL2-level functions in proofs about these concepts.

```
(in-theory (disable fix lessp plus diff))
```

Note that we can now prove such facts as

```
(thm (implies (and (lessp a b) (lessp b c)) (lessp a c)))
(thm (implies (lessp a (diff b c)) (lessp a b)))
```

and that the proofs appeal to linear arithmetic!

We can add new aliases (although they get new names). Note that we have to (a) define the function, (b) prove the :linear-alias rule — probably enabling nqthm-level functions to do it, and (c) disable the new function so it survives rewriting and is around for linearization to de-alias.

```
(defun add1 (x) (+ (fix x) 1))                    ; (a)
(defthm add1-linear                               ; (b) - see below
  (and (equal (add1 x) (+ (fix x) 1))
       (equal (fix (add1 x)) (add1 x)))
  :rule-classes :linear-alias
  :hints (("Goal" :in-theory (enable fix))))
(in-theory (disable add1))                        ; (c)
```

```
(thm (lessp x (add1 x)))
```

Observe that in (b) we prove two rules about `add1`: the one that relates it to `+^` and the one that removes the `fix^`es inserted by the first rule.

Next we illustrate the use of `:linear^` lemmas in the context of linear aliasing. Consider the function:

```
(defun strpos (pat str)
; Find the position of pat in str.
  (cond ((atom str) 0)
        ((equal pat (car str)) 0)
        (t (add1 (strpos pat (cdr str))))))

(defun strlen (str)
  (cond ((atom str) 0)
        (t (add1 (strlen (cdr str))))))
```

We can prove and store the following `:linear^` lemma about `strpos`.

```
(defthm strpos-length
  (not (lessp (strlen str) (strpos pat str)))
  :rule-classes :linear)
```

This event is interesting for two reasons. The proof illustrates linear aliasing. But the fact that this event is stored as a `:linear^` lemma — even though the conclusion is not about `<^` — also illustrates linear aliasing. In fact, this lemma is stored under `(strpos pat str)` with `:fixer` `fix^`.

We can illustrate the storage and retrieval of the lemma directly by proving:

```
(thm (not (< (fix (strlen str)) (fix (strpos pat str)))))
```

But more commonly we will be interested in proofs that reach out for facts about `strpos` (implicitly `fix^`ing them for us) as in:

```
(thm (implies (lessp (strlen str) mx)
              (lessp (strpos pat str) mx))).
```

This ends the extended example of linear aliasing lemmas.

14.12 META

META make a `:meta` rule (a hand-written simplifier)

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. Meta rules extend the ACL2 simplifier with hand-written code to transform certain terms to equivalent ones. To add a meta rule, the `:corollary^` formula must establish that the hand-written "metafunction" preserves the meaning of the transformed term.

Example `:corollary^` formulas from which `:meta` rules might be built are:

```
Examples:
(equal (ev x a)                  ; Modify the rewriter to use fn to
       (ev (fn x) a))            ; transform terms.  The :trigger-fns
                                 ; of the :meta rule-class specify
                                 ; the top-most function symbols of
                                 ; those x that are candidates for
                                 ; this transformation.

(implies (and (pseudo-termp x)  ; As above, but this illustrates
              (alistp a))        ; that without loss of generality we
         (equal (ev x a)         ; may restrict x to be shaped like a
                (ev (fn x) a))) ; term and a to be an alist.

(implies (and (pseudo-termp x)  ; As above (with or without the
              (alistp a)         ; hypotheses on x and a) with the
              (ev (hyp-fn x) a)); additional restriction that the
         (equal (ev x a)         ; meaning of (hyp-fn x) is true in
                (ev (fn x) a))) ; the current context.  That is, the
                                 ; applicability of the transforma-
                                 ; tion may be dependent upon some
                                 ; computed hypotheses.
```

A non-`nil` list of function symbols must be supplied as the value of the `:trigger-fns` field in a `:meta` rule class object.

```
General Forms:
(implies (and (pseudo-termp x)    ; this hyp is optional
              (alistp a)          ; this hyp is optional
              (ev (hyp-fn x) a))  ; this hyp is optional
         (equiv (ev x a)
                (ev (fn x) a)))
```

where `equiv` is a known equivalence^ relation, `x` and `a` are distinct variable names, and `ev` is an evaluator function (see below), and `fn` is a function symbol, as is `hyp-fn` when provided. Both `fn` and `hyp-fn` should take a single argument. If `fn` and/or `hyp-fn` are guard^ed, their guard^s should be (implied by) `pseudo-termp^`. See Section 8.72 [PSEUDO-TERMP], page 335. We say the theorem above is a "metatheorem" or "metalemma" and `fn` is a "metafunction", and `hyp-fn` is a "hypothesis metafunction".

We defer discussion of the case in which there is a hypothesis metafunction and for now address the case in which the other two hypotheses are present.

In the discussion below, we refer to the argument, `x`, of `fn` and `hyp-fn` as a "term." When these metafunctions are executed by the simplifier, they will be applied to (the quotations of) terms. But during the proof of the metatheorem itself, `x` may not be the quotation of a term. If the `pseudo-termp^` hypothesis is omitted, `x` may be any object. Even with the `pseudo-termp^` hypothesis, `x` may merely "look like a term" but use non-function symbols or function symbols of incorrect arity. In any case, the metatheorem is stronger than necessary to allow us to apply the metafunctions to terms, as we do in the discussion below. We return later to the question of proving the metatheorem.

Suppose the general form of the metatheorem above is proved with the `pseudo-termp^` and `alistp^` hypotheses. Then when the simplifier encounters a term, `(h t1 ... tn)`, that begins with a function symbol, `h`, listed in `:trigger-fns`, it applies the metafunction, `fn`, to the quotation of the term, i.e., it evaluates `(fn '(h t1 ... tn))` to obtain some result, which can be written as `'val`. If `'val` is different from `'(h t1 ... tn)` and `val` is a term, then `(h t1 ... tn)` is replaced by `val`, which is then passed along for further rewriting. Because the metatheorem establishes the correctness of `fn` for all terms (even non-terms!), there is no restriction on which function symbols are listed in the `:trigger-fns`. Generally, of course, they should be the symbols that head up the terms simplified by the metafunction `fn`. See Section 6.36 [TERM-TABLE], page 197 for how one obtains some assistance towards guaranteeing that `val` is indeed a term.

The "evaluator" function, `ev`, is a function that can evaluate a certain class of expressions, namely, all of those composed of variables, constants, and applications of a fixed, finite set of function symbols, `g1`, ..., `gk`. Generally speaking, the set of function symbols handled by `ev` is chosen to be exactly the function symbols recognized and manipulated by the metafunctions being introduced. For example, if `fn` manipulates expressions in which `'equal^` and `'binary-append^` occur as function symbols, then `ev` is generally specified to handle `equal^` and `binary-append^`. The actual requirements on `ev` become clear when the metatheorem is proved. The standard way to introduce an evaluator is to use the ACL2 macro `defevaluator^`, though this is not strictly necessary. See Section 6.8 [DEFEVALUATOR], page 160 for details.

Why are we justified in using metafunctions this way? Suppose `(fn 'term1)` is `'term2`. What justifies replacing `term1` by `term2`? The first step is to assert that `term1` is `(ev 'term1 a)`, where `a` is an alist that maps `'var` to `var`, for each variable `var` in `term1`. This step is incorrect, because `'term1` may contain function symbols other than the ones, `g1`, ..., `gk`, that `ev` knows how to handle. But we can grow `ev` to a "larger" evaluator, `ev*`, an evaluator for all of the symbols that occur in `term1` or `term2`. We can prove that `ev*` satisfies the constraint^s on `ev`. Hence, the metatheorem holds for `ev*` in place of `ev`, by functional instantiation. We can then carry out the proof of the equivalence^ of `term1` and `term2` as follows: Fix `a` to be an alist that maps the quotations of the variables of `term1` and `term2` to themselves. Then,

```
term1 = (ev* 'term1 a)      ; (1) by construction of ev* and a
      = (ev* (fn 'term1) a) ; (2) by the metatheorem for ev*
      = (ev* 'term2 a)      ; (3) by evaluation of fn
      = term2               ; (4) by construction of ev* and a
```

Note that in line (2) above, where we appeal to the (functional instantiation of the) metatheorem, we can relieve its (optional) `pseudo-termp^` and `alistp^` hypotheses by appealing to the facts that `term1` is a term and `a` is an alist by construction.

Finally, we turn to the second case, in which there is a hypothesis metafunction. In that case, consider as before what happens when the simplifier encounters a term, `(h t1 ... tn)`, where `h` is listed in `:trigger-fns`. This time, after it applies `fn` to `'(h t1 ... tn)` to obtain the quotation of some new term, `'val`, it then applies the hypothesis metafunction, `hyp-fn`. That is, it evaluates `(hyp-fn '(h t1 ... tn))` to obtain some result, which can be written as `'hyp-val`. If `hyp-val` is not in fact a term, the metafunction is not used. Provided `hyp-val` is a term, the simplifier attempts to establish (by conventional backchaining) that this term is non-`nil` in the current context. If this attempt fails, then the meta rule is not applied. Otherwise, `(h t1...tn)` is replaced by `val` as in the previous case (where there was no hypothesis metafunction).

Why is it justified to make this extension to the case of hypothesis metafunctions? First, note that the rule

```
(implies (and (pseudo-termp x)
              (alistp a)
              (ev (hyp-fn x) a))
         (equal (ev x a)
                (ev (fn x) a)))
```

is logically equivalent to the rule

```
(implies (and (pseudo-termp x)
```

```
            (alistp a))
       (equal (ev x a)
              (ev (new-fn x) a)))
```

where (new-fn x) is defined to be (list 'if (hyp-fn x) (fn x) x). (If we're careful, we realize that this argument depends on making an extension of ev to an evaluator ev* that handles if^ and the functions manipulated by hyp-fn.) If we write 'term for the quotation of the present term, and if (hyp-fn 'term) and (fn 'term) are both terms, say hyp1 and term1, then by the previous argument we know it is sound to rewrite term to (if hyp1 term1 term). But since we have established in the current context that hyp1 is non-nil, we may simplify (if hyp1 term1 term) to term1, as desired.

We now discuss the role of the pseudo-termp^ hypothesis. (Pseudo-termp x) checks that x has the shape of a term. Roughly speaking, it insures that x is a symbol, a quoted constant, or a true list consisting of a lambda expression or symbol followed by some pseudo-terms. Among the properties of terms not checked by pseudo-termp^ are that variable symbols never begin with ampersand, lambda expressions are closed, and function symbols are applied to the correct number of arguments. See Section 8.72 [PSEUDO-TERMP], page 335.

There are two possible roles for pseudo-termp^ in the development of a metatheorem: it may be used as the guard^ of the metafunction and/or hypothesis metafunction and it may be used as a hypothesis of the metatheorem. Generally speaking, the pseudo-termp^ hypothesis is included in a metatheorem only if it makes it easier to prove. The choice is yours. (An extreme example of this is when the metatheorem is invalid without the hypothesis!) We therefore address ourselves the question: should a metafunction have a pseudo-termp^ guard^? A pseudo-termp^ guard^ for a metafunction, in connection with other considerations described below, improves the efficiency with which the metafunction is used by the simplifier.

To make a metafunction maximally efficient you should (a) provide it with a pseudo-termp^ guard^ and exploit the guard^ when possible in coding the body of the function (see Section 8.39.4 [GUARDS-AND-EVALUATION], page 290, especially the section on efficiency issues), (b) verify the guard^s of the metafunction (see Section 6.38 [VERIFY-GUARDS], page 199), and (c) compile the metafunction (see Section 9.5 [COMP], page 379). When these three steps have been taken the simplifier can evaluate (fn 'term1) by running the compiled "primary code" (see Section 8.39.4 [GUARDS-AND-EVALUATION], page 290) for fn directly in Common Lisp.

Before discussing efficiency issues further, let us review the for a moment the general case in which we wish to evaluate (fn 'obj) for some :logic^ function. We must first ask whether the guard^s of fn have been verified. If not, we must evaluate fn by executing its logic definition. This effectively checks the guard^s of every subroutine and so can be slow. If, on the other hand, the guard^s of fn have been verified, then we can run the primary code for fn, provided 'obj satisfies the guard^ of fn. So we must next evaluate the guard^ of fn on 'obj. If the guard^ is met, then we run the primary code for fn, otherwise we run the logic code.

Now in the case of a metafunction for which the three steps above have been followed, we know the guard^ is (implied by) pseudo-termp^ and that it has been verified. Furthermore, we know without checking that the guard^ is met (because term1 is a term and hence 'term1 is a pseudo-termp^). Hence, we can use the compiled primary code directly.

We strongly recommend that you compile your metafunctions, as well as all their subroutines. Guard verification is also recommended.

14.13 REFINEMENT

REFINEMENT record that one equivalence relation refines another

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example :corollary^ formula from which a :refinement rule might be built is:

```
Example:
(implies (bag-equal x y) (set-equal y x)).
```

Also see Section 6.12 [DEFREFINEMENT], page 166.

```
General Form:
(implies (equiv1 x y) (equiv2 x y))
```

Equiv1 and equiv2 must be known equivalence relations. The effect of such a rule is to record that equiv1 is a refinement of equiv2. This means that equiv1 :rewrite^ rules may be used while trying to maintain equiv2. See Section 14.6 [EQUIVALENCE], page 632 for a general discussion of the issues.

The macro form (defrefinement equiv1 equiv2) is an abbreviation for a defthm^ of rule-class :refinement that establishes that equiv1 is a refinement of equiv2. See Section 6.12 [DEFREFINEMENT], page 166.

Suppose we have the :rewrite^ rule

```
(bag-equal (append a b) (append b a))
```

which states that `append^` is commutative modulo bag-equality. Suppose further we have established that bag-equality refines set-equality. Then when we are simplifying `append^` expressions while maintaining set-equality we use `append^`'s commutativity property, even though it was proved for bag-equality.

Equality is known to be a refinement of all equivalence relations. The transitive closure of the refinement relation is maintained, so if `set-equality`, say, is shown to be a refinement of some third sense of equivalence, then `bag-equality` will automatially be known as a refinement of that third equivalence.

`:refinement` lemmas cannot be disabled. That is, once one equivalence relation has been shown to be a refinement of another, there is no way to prevent the system from using that information. Of course, individual `:rewrite^` rules can be disabled.

More will be written about this as we develop the techniques.

14.14 REWRITE

REWRITE make some `:rewrite` rules (possibly conditional ones)

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. Example `:corollary^` formulas from which `:rewrite` rules might be built are:

```
Example:
(equal (+ x y) (+ y x))            replace (+ a b) by (+ b a) provided
                                   certain heuristics approve the
                                   permutation.

(implies (true-listp x)            replace (append a nil) by a, if
         (equal (append x nil) x)) (true-listp a) rewrites to t

(implies                           replace (member a (append b c)) by
    (and (eqlablep e)              (member a (append c b)) in contexts
         (true-listp x)            in which propositional equivalence
         (true-listp y))           is sufficient, provided (eqlablep a)
    (iff (member e (append x y))   (true-listp b) and (true-listp c)
         (member e (append y x)))) rewrite to t and the permutative
```

```
                                        heuristics approve

General Form:
(and ...
     (implies (and ...hi...)
              (implies (and ...hk...)
                       (and ...
                            (equiv lhs rhs)
                            ...)))
     ...)
```

Note: One `:rewrite` rule class object might create many rewrite rules from the `:corollary^` formula. To create the rules, we first translate the formula (expanding all macros; also see Section 9.16 [TRANS], page 390). Next, we eliminate all `lambda`s; one may think of this step as simply substituting away every `let^`, `let*^`, and `mv-let^` in the formula. We then flatten the `and^` and `implies^` structure of the formula, transforming it into a conjunction of formulas, each of the form

```
(implies (and h1 ... hn) concl)
```

where no hypothesis is a conjunction and `concl` is neither a conjunction nor an implication. If necessary, the hypothesis of such a conjunct may be vacuous. We then further coerce each `concl` into the form `(equiv lhs rhs)`, where `equiv` is a known equivalence^ relation, by replacing any `concl` not of that form by `(iff concl t)`. A `concl` of the form `(not term)` is considered to be of the form `(iff term nil)`. By these steps we reduce the given `:corollary^` to a sequence of conjuncts, each of which is of the form

```
(implies (and h1 ... hn)
         (equiv lhs rhs))
```

where `equiv` is a known equivalence^ relation. See Section 14.6 [EQUIVALENCE], page 632 for a general discussion of the introduction of new equivalence^ relations.

We create a `:rewrite` rule for each such conjunct, if possible, and otherwise cause an error. It is possible to create a rewrite rule from such a conjunct provided lhs is not a variable, a quoted constant, a `let^`-expression, a `lambda` application, or an `if^`-expression.

A `:rewrite` rule is used when any instance of the `lhs` occurs in a context in which the equivalence^ relation is operative. First, we find a substitution that makes `lhs` equal to the target term. Then we attempt to relieve the instantiated hypotheses of the rule. Hypotheses that are fully instantiated are relieved by recursive rewriting. Hypotheses that contain “free variables” (variables not assigned by the unifying substitution) are relieved by attempting to guess a suitable instance so as to make the hypothesis equal to some known assumption in the context of the target. If the hypotheses are relieved, and certain restrictions that prevent some forms of infinite regress are met (see Section 8.62 [LOOP-STOPPER], page 324), the target is replaced by the instantiated **rhs**, which is then recursively rewritten.

At the moment, the best description of how ACL2 `:rewrite` rules are used may be found in the discussion of "Replacement Rules" pp 234 of A Computational Logic Handbook.

14.15 TYPE-PRESCRIPTION

TYPE-PRESCRIPTION   make a rule that specifies the type of a term

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. Some example `:corollary^` formulas from which `:type-prescription` rules might be built are:

```
Examples:
(implies                              (nth n lst) is of type characterp
 (and (character-listp lst)           provided the two hypotheses can
      (< n (length lst)))             be established by type reasoning
 (characterp (nth n lst))).

(implies                              (demodulize a lst 'value ans) is
 (and (atom a)                        either a nonnegative integer or
      (true-listp lst)                of the same type as ans, provided
      (member-equal a lst))           the hyps can be established by type
 (or (and                             reasoning
       (integerp
        (demodulize a lst 'value ans))
       (>= (demodulize a lst 'value ans) 0))
     (equal (demodulize a lst 'value ans) ans))).
```

To specify the term whose type (see Section 8.87 [TYPE-SET], page 358) is described by the rule, provide that term as the value of the `:typed-term` field of the rule class object.

```
General Form:
(implies hyps
         (or type-restriction1-on-pat
             ...
             type-restrictionk-on-pat
             (equal pat var1)
             ...
```

```
                  (equal pat varj)))
```

where `pat` is the application of some function symbol to some arguments, each `type-restrictioni-on-pat` is a term involving `pat` and containing no variables outside of the occurrences of `pat`, and each `vari` is one of the variables of `pat`. Generally speaking, the `type-restriction` terms ought to be terms that inform us as to the type of `pat`. Ideally, they should be "primitive recognizing expressions" about `pat`; see Section 14.2 [COMPOUND-RECOGNIZER], page 624.

If the `:typed-term` is not provided in the rule class object, it is computed heuristically by looking for a term in the conclusion whose type is being restricted. An error is caused if no such term is found.

Roughly speaking, the effect of adding such a rule is to inform the ACL2 typing mechanism that `pat` has the type described by the conclusion, when the hypotheses are true. In particular, the type of `pat` is within the union of the types described by the several disjuncts. The "type described by" `(equal pat vari)` is the type of `vari`.

More operationally, when asked to determine the type of a term that is an instance of `pat`, ACL2 will first attempt to establish the hypotheses. **This is done by type reasoning alone, not rewriting!** Of course, if some hypothesis is to be forced, it is so treated; see Section 8.35 [FORCE], page 277. Provided the hypotheses are established by type reasoning, ACL2 then unions the types described by the `type-restrictioni-on-pat` terms together with the types of those subexpressions of `pat` identified by the `vari`. The final type computed for a term is the intersection of the types implied by each applicable rule. Type prescription rules may be disabled.

Because only type reasoning is used to establish the hypotheses of `:type-prescription` rules, some care must be taken with the hypotheses. Suppose, for example, that the non-recursive function `my-statep` is defined as

```
(defun my-statep (x)
  (and (true-listp x)
       (equal (length x) 2)))
```

and suppose `(my-statep s)` occurs as a hypothesis of a `:type-prescription` rule that is being considered for use in the proof attempt for a conjecture with the hypothesis `(my-statep s)`. Since the hypothesis in the conjecture is rewritten, it will become the conjunction of `(true-listp s)` and `(equal (length s) 2)`. Those two terms will be assumed to have type `t` in the context in which the `:type-prescription` rule is tried. But type reasoning will be unable to deduce that `(my-statep s)` has type `t` in this context. Thus, either `my-statep` should be disabled (see Section 15.2

[DISABLE], page 670) during the proof attempt or else the occurrence of `(my-statep s)` in the `:type-prescription` rule should be replaced by the conjunction into which it rewrites.

While this example makes it clear how non-recursive predicates can cause problems, non-recursive functions in general can cause problems. For example, if `(mitigate x)` is defined to be `(if (rationalp x) (1- x) x)` then the hypothesis `(pred (mitigate s))` in the conjecture will rewrite, opening `mitigate` and splitting the conjecture into two subgoals, one in which `(rationalp s)` and `(pred (1- x))` are assumed and the other in which `(not (rationalp s))` and `(pred x)` are assumed. But `(pred (mitigate s))` will not be typed as `t` in either of these contexts. The moral is: beware of non-recursive functions occuring in the hypotheses of `:type-prescription` rules.

Because of the freedom one has in forming the conclusion of a type-prescription, we have to use heuristics to recover the pattern, `pat`, whose type is being specified. In some cases our heuristics may not identify the intended term and the `:type-prescription` rule will be rejected as illegal because the conclusion is not of the correct form. When this happens you may wish to specify the `pat` directly. This may be done by using a suitable rule class token. In particular, when the token `:type-prescription` is used it means ACL2 is to compute pat with its heuristics; otherwise the token should be of the form `(:type-prescription :typed-term pat)`, where `pat` is the term whose type is being specified.

The defun event may generate a `:type-prescription` rule. Suppose `fn` is the name of the function concerned. Then `(:type-prescription fn)` is the rune given to the type-prescription, if any, generated for `fn` by `defun^`. (The trivial rule, saying `fn` has unknown type, is not stored, but `defun^` still allocates the rune and the corollary of this rune is known to be `t`.)

14.16 TYPE-SET-INVERTER

TYPE-SET-INVERTER exhibit a new decoding for an ACL2 type-set

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas.

```
Example Rule Class:
(:type-set-inverter
  :corollary (equal (and (counting-number x) (not (equal x 0)))
                    (and (integerp x) (< x 0)))
  :type-set 2)
```

```
General Forms of Rule Class:
:type-set-inverter, or
(:type-set-inverter :type-set n)

General Form of Theorem or Corollary:
(EQUAL new-expr old-expr)
```

where `n` is a `type-set^` (see Section 8.87 [TYPE-SET], page 358) and `old-expr` is the term containing `x` as a free variable that ACL2 currently uses to recognize `type-set^ n`. For a given `n`, the exact form of `old-expr` is generated by

```
(convert-type-set-to-term 'x n (ens state) (w state) nil)].
```

If the `:type-set^` field of the rule-class is omitted, we attempt to compute it from the right-hand side, `old-expr`, of the corollary. That computation is done by `type-set-implied-by-term` (see Section 8.87 [TYPE-SET], page 358). However, it is possible that the type-set we compute from `lhs` does not have the required property that when inverted with `convert-type-set-to-term` the result is `lhs`. If you omit `:type-set^` and an error is caused because `lhs` has the incorrect form, you should manually specify both `:type-set^` and the `lhs` generated by `convert-type-set-to-term`.

The rule generated will henceforth make `new-expr` be the term used by ACL2 to recognize type-set `n`. If this rule is created by a `defthm^` event named `name` then the rune of the rule is `(:type-set-inverter name)` and by disabling that rune you can prevent its being used to decode type-sets.

Type-sets are inverted when forced assumptions are turned into formulas to be proved. In their internal form, assumptions are essentially pairs consisting of a context and a goal term, which was forced. Abstractly a context is just a list of hypotheses which may be assumed while proving the goal term. But actually contexts are alists which pair terms with type-sets, encoding the current hypotheses. For example, if the original conjecture contained the hypothesis `(integerp x)` then the context used while working on that conjecture will include the assignment to `x` of the type-set `*ts-integer*`.

14.17 WELL-FOUNDED-RELATION

WELL-FOUNDED-RELATION show that a relation is well-founded on a set

See Chapter 14 [RULE-CLASSES], page 617 for a general discussion of rule classes and how they are used to build rules from formulas. An example `:corollary^` formula from which a `:well-`

founded-relation rule might be built is as follows. (Of course, the functions `pairp`, `lex2p`, and `ordinate` would have to be defined first.)

```
Example:
(and (implies (pairp x) (e0-ordinalp (ordinate x)))
     (implies (and (pairp x)
                   (pairp y)
                   (lex2p x y))
              (e0-ord-< (ordinate x) (ordinate y))))
```

The above example establishes that `lex2p` is a well-founded relation on `pairp`s. We explain and give details below.

Exactly two general forms are recognized:

```
General Forms
(AND (IMPLIES (mp x) (E0-ORDINALP (fn x)))      ; Property 1
     (IMPLIES (AND (mp x)                       ; Property 2
                   (mp y)
                   (rel x y))
              (E0-ORD-< (fn x) (fn y)))),
```

or

```
(AND (E0-ORDINALP (fn x))                       ; Property 1
     (IMPLIES (rel x y)                         ; Property 2
              (E0-ORD-< (fn x) (fn y))))
```

where `mp`, `fn`, and `rel` are function symbols, `x` and `y` are distinct variable symbols, and no other `:well-founded-relation` theorem about `fn` has been proved. When the second general form is used, we act as though the first form were used with `mp` being the function that ignores its argument and returns `t`. The discussion below therefore considers only the first general form.

Note: We are very rigid when checking that the submitted formula is of one of these forms. For example, in the first form, we insist that all the conjuncts appear in the order shown above. Thus, interchanging the two properties in the top-level conjunct or rearranging the hyptheses in the second conjunct both produce unrecognized forms. The requirement that each `fn` be proved well-founded at most once insures that for each well-founded relation, `fn`, there is a unique `mp` that recognizes the domain on which `rel` is well-founded. We impose this requirement simply so that `rel` can be used as a short-hand when specifying the well-founded relations to be used in definitions; otherwise the specification would have to indicate which `mp` was to be used.

`Mp` is a predicate that recognizes the objects that are supposedly ordered in a well-founded way by `rel`. We call such an object an "`mp`-measure" or simply a "measure" when `mp` is understood. Property 1 tells us that every measure can be mapped into an ACL2 ordinal. (See Section 8.24 [E0-ORDINALP], page 263.) This mapping is performed by `fn`. Property 2 tells us that if the measure `x` is smaller than the measure `y` according to `rel` then the image of `x` under `fn` is a smaller than that of `y`, according to the well-founded relation `e0-ord-<^`. (See Section 8.23 [E0-ORD-<], page 262.) Thus, the general form of a `:well-founded-relation` formula establishes that there exists a `rel`-order preserving embedding (namely via `fn`) of the `mp`-measures into the ordinals. We can thus conclude that `rel` is well-founded on `mp`-measures.

Such well-founded relations are used in the admissibility test for recursive functions, in particular, to show that the recursion terminates. To illustrate how such information may be used, consider a generic function definition

```
(defun g (x) (if (test x) (g (step x)) (base x))).
```

If `rel` has been shown to be well-founded on `mp`-measures, then `g`'s termination can be insured by finding a measure, `(m x)`, with the property that `m` produces a measure:

```
(mp (m x)),                                    ; Defun-goal-1
```

and that the argument to `g` gets smaller (when measured by `m` and compared by `rel`) in the recursion,

```
(implies (test x) (rel (m (step x)) (m x))).   ; Defun-goal-2
```

If `rel` is selected as the `:well-founded-relation` to be used in the definition of `g`, the definitional principal will generate and attempt to prove `defun-goal-1` and `defun-goal-2` to justify `g`. We show later why these two goals are sufficient to establish the termination of `g`. Observe that neither the ordinals nor the embedding, `fn`, of the `mp`-measures into the ordinals is involved in the goals generated by the definitional principal.

Suppose now that a `:well-founded-relation` theorem has been proved for `mp` and `rel`. How can `rel` be "selected as the `:well-founded-relation`" by `defun^`? There are two ways. First, an `xargs^` keyword to the `defun^` event allows the specification of a `:well-founded-relation`. Thus, the definition of `g` above might be written

```
(defun g (x)
 (declare (xargs :well-founded-relation (mp . rel)))
 (if (test x) (g (step x)) (base x)))
```

Alternatively, `rel` may be specified as the `:default-well-founded-relation` in `acl2-defaults-table^` by executing the event

```
(set-default-well-founded-relation rel).
```

When a `defun^` event does not explicitly specify the relation in its `xargs^` the default relation is used. When ACL2 is initialized, the default relation is `e0-ord-<^`.

Finally, though it is probably obvious, we now show that `defun-goal-1` and `defun-goal-2` are sufficient to insure the termination of `g` provided `property-1` and `property-2` of `mp` and `rel` have been proved. To this end, assume we have proved `defun-goal-1` and `defun-goal-2` as well as `property-1` and `property-2` and we show how to admit `g` under the primitive ACL2 definitional principal (i.e., using only the ordinals). In particular, consider the definition event

```
(defun g (x)
 (declare (xargs :well-founded-relation e0-ord-<
                 :measure (fn (m x))))
 (if (test x) (g (step x)) (base x)))
```

Proof that `g` is admissible: To admit the definition of `g` above we must prove

```
(e0-ordinalp (fn (m x)))                        ; *1
```

and

```
(implies (test x)                               ; *2
         (e0-ord-< (fn (m (step x))) (fn (m x)))).
```

But *1 can be proved by instantiating `property-1` to get

```
(implies (mp (m x)) (e0-ordinalp (fn (m x)))),
```

and then relieving the hypothesis with `defun-goal-1`, `(mp (m x))`.

Similarly, *2 can be proved by instantiating `property-2` to get

```
(implies (and (mp (m (step x)))
              (mp (m x))
              (rel (m (step x)) (m x)))
         (e0-ord-< (fn (m (step x))) (fn (m x))))
```

and relieving the first two hypotheses by appealing to two instances of `defun-goal-1`, thus obtaining

```
(implies (rel (m (step x)) (m x))
         (e0-ord-< (fn (m (step x))) (fn (m x)))).
```

By chaining this together with `defun-goal-2`,

```
(implies (test x)
         (rel (m (step x)) (m x)))
```

we obtain *2. Q.E.D.

15 THEORIES

THEORIES sets of rune^s to enable/disable in concert

```
Example: '((:definition app)
           (:executable-counterpart app)
           rv
           (rv)
           assoc-of-app)
```

See:

A theory is a list of "runic designators" as described below. Each runic designator denotes a set of "runes" (see Section 15.10 [RUNE], page 675) and by unioning together the runes denoted by each member of a theory we define the set of runes corresponding to a theory. Theories are used to control which rules are "enable^d," i.e., available for automatic application by the theorem prover. There is always a "current" theory. A rule is enable^d precisely if its rune^ is an element of the set of rune^s corresponding to the current theory. At the top-level, the current theory is the theory selected by the most recent `in-theory^` event, extended with the rule names introduced since then. Inside the theorem prover, the `:in-theory^` hint (see Section 8.41 [HINTS], page 297) can be used to select a particular theory as current during the proof attempt for a particular goal.

Theories are generally constructed by "theory expressions." Formally, a theory expression is any term, containing at most the single free variable `world^`, that when evaluated with `world^` bound to the current ACL2 world (see Section 8.89 [WORLD], page 362) produces a theory. ACL2 provides various functions for the convenient construction and manipulation of theories. These are called "theory functions" (see Section 15.13 [THEORY-FUNCTIONS], page 678). For example, the theory function `union-theories^` takes two theories and produces their union. The theory function `universal-theory^` returns the theory containing all known rule names as of the introduction of a given logical name. But a theory expression can contain constants, e.g.,

```
'(assoc (assoc) (:rewrite car-cons) car-cdr-elim)
```

and user-defined functions. The only important criterion is that a theory expression mention no variable freely except `world^` and evaluate to a theory.

More often than not, theory expressions typed by the user do not mention the variable `world^`. This is because user-typed theory expressions are generally composed of applications of ACL2's theory functions. These "functions" are actually macros that expand into terms in which `world^` is used freely and appropriately. Thus, the technical definition of "theory expression" should not mislead you into thinking that interestng theory expressions must mention `world^`; they probably do and you just didn't know it!

One aspect of this arrangement is that theory expressions cannot generally be evaluated at the top-level of ACL2, because `world^` is not bound. To see the value of a theory expression, `expr`, at the top-level, type

```
ACL2 !>(LET ((WORLD (W STATE))) expr).
```

However, because the built-in theories are quite long, you may be sorry you printed the value of a theory expression!

A theory is a true list of runic designators and to each theory there corresponds a set of rune^s, obtained by unioning together the sets of rune^s denoted by each runic designator. For example, the theory constant

```
'(assoc (assoc) (:rewrite car-cons) car-cdr-elim)
```

corresponds to the set of rune^s

```
{(:definition assoc)
 (:executable-counterpart assoc)
 (:rewrite car-cons)
 (:rewrite car-cdr-elim)
 (:elim car-cdr-elim)} .
```

Observe that the theory contains four elements but its runic correspondent contains five. That is because some designators denote sets of several rune^s. If the above theory were selected as current then the five rules named in its runic counterpart would be enable^d and all other rules would be disable^d.

We now precisely define the runic designators and the set of rune^s denoted by each.

o A rune is a runic designator and denotes the singleton set containing that rune.

o If `symb` is a function symbol introduced with a `defun^` (or `defuns^`) event, then `symb` is a runic designator and denotes the singleton set containing the rune `(:definition symb)`.

o If `symb` is a function symbol introduced with a `defun^` (or `defuns^`) event, then `(symb)` is a runic designator and denotes the singleton set containing the rune `(:executable-counterpart symb)`.

o If `symb` is the name of a `defthm^` (or `defaxiom^`) event that introduced at least one rule, then `symb` is a runic designator and denotes the set of the names of all rules introduced by the named event.

o If `str` is the string naming some `defpkg^` event and `symb` is the symbol returned by `(intern str "ACL2")`, then `symb` is a runic designator and denotes the singleton set containing `(:rewrite symb)`, which is the name of the rule stating the conditions under which the `symbol-package-name^` of `(intern x str)` is `str`.

o If `symb` is the name of a `deftheory^` event, then `symb` is a runic designator and denotes the runic theory corresponding to `symb`.

These conventions attempt to implement the Nqthm-1992 treatment of theories. For example, including a function name, e.g., `assoc^`, in the current theory enable^s that function but does not enable^ the executable counterpart. Similarly, including `(assoc)` enable^s the executable counterpart (Nqthm's `*1*assoc`) but not the symbolic definition. And including the name of a proved lemma enable^s all of the rules added by the event. These conventions are entirely consistent with Nqthm usage. Of course, in ACL2 one can include explicitly the rune^s naming the rules in question and so can avoid entirely the use of non-runic elements in theories.

Because a rune^ is a runic designator denoting the set containing that rune^, a list of rune^s is a theory and denotes itself. We call such theories "runic theories." To every theory there corresponds a runic theory obtained by unioning together the sets denoted by each designator in the theory. When a theory is selected as "current" it is actually its runic correspondent that is effectively used.

That is, a rune^ is enable^d iff it is a member of the runic correspondent of the current theory. The value of a theory defined with `deftheory^` is the runic correspondent of the theory computed by the defining theory expression. The theory manipulation functions, e.g., `union-theories^`, actually convert their theory arguments to their runic correspondents before performing the required set operation. The manipulation functions always return runic theories. Thus, it is sometimes convenient to think of (non-runic) theories as merely abbreviations for their runic correspondents, abbreviations which are "expanded" at the first opportunity by theory manipulation functions and the "theory consumer" functions such as `in-theory^` and `deftheory^`.

15.1 CURRENT-THEORY

CURRENT-THEORY currently enable^d rules as of logical name

```
Examples:
(current-theory :here)
(current-theory 'lemma3)
```

See Section 8.61 [LOGICAL-NAME], page 322.

```
General Form:
(current-theory logical-name)
```

Returns the current theory as it existed immediately after the introduction of `logical-name^`. See Chapter 15 [THEORIES], page 667 and see Section 8.61 [LOGICAL-NAME], page 322. This theory is in fact the theory selected by the `in-theory^` event most recently preceding logical name, extended by the rules introduced up through `logical-name^`.

You may experience a fencepost problem in deciding which logical name to use. `Deflabel^` can always be used to mark unambiguously for future reference a particular point in the development of your theory. The order of events^ in the vicinity of an `encapsulate^` is confusing. See Section 6.18 [ENCAPSULATE], page 176.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

15.2 DISABLE

DISABLE deletes names from current theory

```
Example:
(disable fact (fact) associativity-of-app)

General Form:
(disable name1 name2 ... namek)
```

where each `namei` is a runic designator; see Chapter 15 [THEORIES], page 667. The result is the theory that contains all the names in the current theory except those listed. Note that this is merely a function that returns a theory. The result is generally a very long list of rune^s and you will probably regret printing it.

The standard way to "disable" a fixed set of names, is:

```
(in-theory (disable name1 name2 ... namek)) ; globally
:in-theory (disable name1 name2 ... namek)  ; locally
```

Note that all the names are implicitly quoted. If you wish to disable a computed list of names, `lst`, use the theory expression `(set-difference-theories (current-theory :here) lst)`.

15.3 ENABLE

ENABLE adds names to current theory

```
Example:
(enable fact (fact) associativity-of-app)

General Form:
(enable name1 name2 ... namek)
```

where each `namei` is a runic designator; see Chapter 15 [THEORIES], page 667. The result is the theory that contains all the names in the current theory plus those listed. Note that this is merely a function that returns a theory. The result is generally a very long list of rune^s and you will probably regret printing it.

The standard way to "enable" a fixed set of names, is

```
(in-theory (enable name1 name2 ... namek)) ; globally
```

```
:in-theory (enable name1 name2 ... namek)  ; locally
```

Note that all the names are implicitly quoted. If you wish to enable a computed list of names, `lst`, use the theory expression `(union-theories (current-theory :here) lst)`.

15.4 EXECUTABLE-COUNTERPART-THEORY

EXECUTABLE-COUNTERPART-THEORY executable counterpart rules as of logical name

```
Examples:
(executable-counterpart-theory :here)
(executable-counterpart-theory 'lemma3)
```

See Section 8.61 [LOGICAL-NAME], page 322.

```
General Form:
(executable-counterpart-theory logical-name)
```

Returns the theory containing all the `:executable-counterpart^` rune^s, whether enable^d or not, that existed immediately after `logical-name^` was introduced. See the documentation for theories^, logical-name^, executable-counterpart^ and `function-theory^`.

You may experience a fencepost problem in deciding which logical name to use. `Deflabel^` can always be used to mark unambiguously for future reference a particular point in the development of your theory. The order of events^ in the vicinity of an `encapsulate^` is confusing. See Section 6.18 [ENCAPSULATE], page 176.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

15.5 FUNCTION-THEORY

FUNCTION-THEORY function symbol rules as of logical name

```
Examples:
(function-theory :here)
```

```
(function-theory 'lemma3)
```

See Section 8.61 [LOGICAL-NAME], page 322.

```
General Form:
(function-theory logical-name)
```

Returns the theory containing all the `:definition^` rune^s, whether enable^d or not, that existed immediately after `logical-name^` was introduced. See the documentation for theories^, logical-name^ and `executable-counterpart-theory^`.

You may experience a fencepost problem in deciding which logical name to use. `Deflabel^` can always be used to mark unambiguously for future reference a particular point in the development of your theory. The order of events^ in the vicinity of an `encapsulate^` is confusing. See Section 6.18 [ENCAPSULATE], page 176.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

15.6 GROUND-ZERO

GROUND-ZERO   enable^d rules in the startup^ theory

ACL2 concludes its initialization `(boot-strapping)` procedure by defining the theory `ground-zero`; see Chapter 15 [THEORIES], page 667. In fact, this theory is just the theory defined by `(current-theory :here)` at the conclusion of initialization; see Section 15.1 [CURRENT-THEORY], page 670.

Note that by executing the event `(in-theory 'ground-zero)`, you can restore the current theory to its value at the time you started up ACL2.

15.7 INCOMPATIBLE

INCOMPATIBLE   declaring that two rules should not both be enable^d

```
Example:
(theory-invariant (incompatible (:rewrite left-to-right)
                                (:rewrite right-to-left)))

General Form:
(incompatible rune1 rune2)
```

where `rune1` and `rune2` are two specific rune^s. The arguments are not evaluated. `Invariant` is just a macro that expands into a term that checks that `theory^` does not contain both rune^s. See Section 6.37 [THEORY-INVARIANT], page 197.

15.8 INTERSECTION-THEORIES

INTERSECTION-THEORIES intersect two theories^

```
Example:
(intersection-theories (current-theory :here)
                       (theory 'arith-patch))

General Form:
(intersection-theories th1 th2)
```

where `th1` and `th2` are theories (see Chapter 15 [THEORIES], page 667). To each of the arguments there corresponds a runic theory. This function returns the intersection of those two runic theories^, represented as a list and ordered chronologically.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

15.9 RULE-NAMES

RULE-NAMES How rules are named.

```
Examples:
(:rewrite assoc-of-app)
(:linear delta-aref . 2)
(:definition length)
(:executable-counterpart length)
```

See Section 15.10 [RUNE], page 675.

15.10 RUNE

RUNE a rule name

```
Examples:
(:rewrite assoc-of-app)
(:linear delta-aref . 2)
(:definition length)
(:executable-counterpart length)
```

Background: The theorem prover is driven from a data base of rules. The most common rules are `:rewrite^` rules, which cause the simplifier to replace one term with another. Events^ introduce rules into the data base. For example, a `defun^` event may introduce as many as four rules: one for symbolically replacing a function call by its instantiated body, one for evaluating the function on constants, and two for determining the type of a call of the function (one of which is deduced before guard^ verification and the other of which is deduced after guard^ verification). `Defthm^` may introduce several rules, one for each of the rule classes specified.

Every rule in the system has a name. Each name is a structured object called a "rune," which is short for "rule name". Runes are always of the form `(:token symbol . x)`, where `:token` is some keyword symbol indicating what kind of rule is named, `symbol` is the event name that created the rule (and is called the "base symbol" of the rune), and `x` is either `nil` or a natural number that makes the rule name unique.

For example, an event of the form

```
(defthm name thm
  :rule-classes ((:REWRITE :COROLLARY term1)
                 (:REWRITE :COROLLARY term2)
                 (:ELIM    :COROLLARY term3)))
```

creates three rules, each with a unique rune. The runes are

```
(:REWRITE name . 1), (:REWRITE name . 2), and (:ELIM name).
```

The function `corollary^` will return the corollary^ term associated with a given rune in a given world^. Example:

```
(corollary '(:TYPE-PRESCRIPTION DIGIT-TO-CHAR) (w state))
```

However, the preferred way to see the corollary term associated with a rune or a name is to use `:pf`; see Section 7.9 [PF], page 210.

The `defun^` event creates as many as four rules. `(:definition fn)` is the rune given to the equality axiom defining the function, `fn`. `(:executable-counterpart fn)` is the rune given to the rule for computing `fn` on known arguments. A type prescription rule may be created under the name `(:type-prescription fn)`, and an induction^ rule may be created under the name `(:induction fn)`.

Runes may be individually enable^d and disable^d, according to whether they are included in the current theory. See Chapter 15 [THEORIES], page 667. Thus, it is permitted to disable^ `(:elim name)`, say, while enabling the other rules derived from name. Similarly, `(:definition fn)` may be disable^d while `(:executable-counterpart fn)` and the type prescriptions for `fn` are enable^d.

Associated with most runes is the formula justifying the rule named. This is called the "corollary^ formula" of the rune and may be obtained via the function `corollary^`, which takes as its argument a rune and a property list world^. Also see Section 7.9 [PF], page 210. The corollary^ formula for `(:rewrite name . 1)` after the `defthm^` event above is `term1`. The corollary formulas for `(:definition fn)` and `(:executable-counterpart fn)` are always identical: the defining axiom. Some runes, e.g., `(:definition car)`, do not have corollary formulas. `Corollary^` returns `nil` on such runes. In any case, the corollary formula of a rune, when it is non-`nil`, is a theorem and may be used in the `:use` and `:by` hints^.

Note: The system has many built in rules that, for regularity, ought to have names but don't because they can never be disable^d. One such rule is that implemented by the linear arithmetic package. Because many of our subroutines are required by their calling conventions to return the justifying rune, we have invented the notion of "fake runes." Fake runes always have the base symbol `nil`, use a keyword token that includes the phrase "fake-rune", and are always enable^d. For example, `(:fake-rune-for-linear nil)` is a fake rune. Occasionally the system will print a fake rune where a rune is expected. For example, when the linear arithmetic fake rune is reported among the rules used in a proof, it is an indication that the linear arithmetic package was used. However, fake runes are not allowed in theories^, they cannot be enable^d or disable^d, and they do not have associated corollary^ formulas. In short, despite the fact that the user may sometimes see fake runes printed, they should never be typed.

15.11 SET-DIFFERENCE-THEORIES

SET-DIFFERENCE-THEORIES difference of two theories^

```
Example:
(set-difference-theories (current-theory :here)
                         '(fact (fact)))

General Form:
(set-difference-theories th1 th2)
```

where `th1` and `th2` are theories^ (see Chapter 15 [THEORIES], page 667). To each of the arguments there corresponds a runic theory. This function returns the set-difference of those two runic theories^, represented as a list and ordered chronologically. That is, a rune^ is in the result iff it is in the first runic theory but not in the second.

The standard way to "disable" a theory, `lst`, is: `(in-theory (set-difference-theories (current-theory :here) lst))`.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

15.12 THEORY

THEORY retrieve named theory

```
Example:
(theory 'my-theory)

General Form:
(theory name)
```

where `name` is the name of a previously executed `deftheory^` event. Returns the named theory. See Chapter 15 [THEORIES], page 667.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

15.13 THEORY-FUNCTIONS

THEORY-FUNCTIONS functions for obtaining or producing theories^

```
Example Calls of Theory Functions:
(universal-theory :here)
(union-theories th1 th2)
(set-difference-theories th1 th2)
```

The theory functions are documented individually:

Related topics other than immediate subtopics:
* CURRENT-THEORY:: currently enable^d rules as of logical name
* DISABLE:: deletes names from current theory
* ENABLE:: adds names to current theory
* EXECUTABLE-COUNTERPART-THEORY:: executable counterpart rules as of logical name
* FUNCTION-THEORY:: function symbol rules as of logical name
* INTERSECTION-THEORIES:: intersect two theories^
* SET-DIFFERENCE-THEORIES:: difference of two theories^
* THEORY:: retrieve named theory
* UNION-THEORIES:: union two theories^
* UNIVERSAL-THEORY:: all rules as of logical name

The functions (actually, macros) mentioned above are convenient ways to produce theories^. (See Chapter 15 [THEORIES], page 667.) Some, like `universal-theory^`, take a logical name (see Section 8.61 [LOGICAL-NAME], page 322) as an argument and return the relevant theory as of the time that name was introduced. Others, like `union-theories^`, take two theories^ and produce a new one. See Section 8.76 [REDUNDANT-EVENTS], page 337 for a caution about the use of logical names in theory expressions.

Theory expressions are generally composed of applications of theory functions. Formally, theory expressions are expressions that involve, at most, the free variable `world^` and that when evaluated with `world^` bound to the current ACL2 world (see Section 8.89 [WORLD], page 362) return theories^. The "theory functions" are actually macros that expand into forms that involve the free variable `world^`. Thus, for example `(universal-theory :here)` actually expands to `(universal-theory-fn :here world)` and when that form is evaluated with `world^` bound to the current ACL2 world^, `universal-theory-fn` scans the ACL2 property lists and computes the current universal theory. Because the theory functions all implicitly use `world^`, the variable does not generally appear in anything the user types.

15.14 UNION-THEORIES

UNION-THEORIES union two theories^

```
Example:
(union-theories (current-theory 'lemma3)
                (theory 'arith-patch))

General Form:
(union-theories th1 th2)
```

where `th1` and `th2` are theories (see Chapter 15 [THEORIES], page 667). To each of the arguments there corresponds a runic theory. This function returns the union of those two runic theories^, represented as a list and ordered chronologically.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

15.15 UNIVERSAL-THEORY

UNIVERSAL-THEORY all rules as of logical name

```
Examples:
(universal-theory :here)
(universal-theory 'lemma3)
```

See Section 8.61 [LOGICAL-NAME], page 322.

```
General Form:
(universal-theory logical-name)
```

Returns the theory consisting of all the rune^s that existed immediately after `logical-name^` was introduced. See Chapter 15 [THEORIES], page 667 and see Section 8.61 [LOGICAL-NAME], page 322. The theory includes `logical-name^` itself (if there is a rule by that name). (Note that since some events^ do not introduce rules (e.g., `defmacro^`, `defconst^` or `defthm^` with `:rule-classes^ nil`), the universal-theory does not necessarily include a rune^ for every event name.) The universal-theory is very long and you will probably regret printing it.

You may experience a fencepost problem in deciding which logical-name^ to use. `Deflabel^` can always be used to mark unambiguously for future reference a particular point in the development of your theory. This is convenient because `deflabel^` does not introduce any rules and hence it doesn't matter if you count it as being in the interval or not. The order of events^ in the vicinity of an `encapsulate^` is confusing. See Section 6.18 [ENCAPSULATE], page 176.

This "function" is actually a macro that expands to a term mentioning the single free variable `world^`. When theory expressions are evaluated by `in-theory^` or the `:in-theory^` hint, `world^` is bound to the current ACL2 world^.

Index

M

N

O

P

Q

R

S

T

Table of Contents