A Precise Description of the ACL2 Logic

Matt Kaufmann*and J Strother Moore
Computational Logic, Inc.†
1717 West Sixth Street, Suite 290
Austin, TX 78703-4776, USA

February 11, 1997

Abstract

The ACL2 logic is a first-order, essentially quantifier-free logic of total recursive functions providing mathematical induction and several extension principles, including symbol package definition, recursive function definition, and encapsulation. In this document we describe the logic more precisely. **THIS IS AN INCOMPLETE WORKING DRAFT.**

1 Background

Naively speaking, a mathematical logic is given by a formal language, some axioms in that language, and some rules of inference that permit one to derive new formulas, called "theorems," from those axioms. To "prove" a theorem one shows how to derive it from the axioms using the rules of inference. This game is very challenging. Even for very simple sets of axioms and rules, the resulting theorems are often non-obvious.

What prevents logic from being merely an academic game is that, like most of mathematics, it can be related to our ordinary experience. In particular, it is often possible to give *meaning* to the formulas in such a way that the axioms are all accepted as *truths* and the rules of inference are *truth preserving*. Consequently, the theorems are also truths. More precisely, the theorems are truths about what is modeled by the axioms and rules of inference.

*Matt Kaufmann's address is now Motorola @ Lakewood, P.O. Box 6000, MD F52, Austin, TX 78762

†The theorem prover used in this work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406, and the Office of Naval Research, Contract N00014-94-C-0193. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency, the Office of Naval Research, or the U.S. Government.

It is difficult — perhaps impossible — to use formal logic to model the physical world with adequate precision and completeness. Unobtrusive imperfections in the correspondence between the formal system and the objects modeled can be magnified into gross distortions of reality by the utterly precise lens of formal logic. Thus, logic is better suited to study the perfect objects of mathematics — i.e., numbers, sets, algorithms, etc. — than the imperfect objects of our physical world. Put another way, logic is best at modeling *formal systems*. Very few modern scientists have tried to use formal logic to study physical systems and thus, in contrast to many other branches of mathematics, the study and use of logic has been largely confined to formal logicians.

However, the widespread use of computing machines is changing this. Computing machines are imperfect physical artifacts. But they implement formal systems. That is to say, when such a machine is working as physically intended by its designer, it is supposed to carry out some algorithm following a fixed set of precisely specified rules. The "utterly precise lens" of formal logic is an excellent tool with which to investigate the behavior of such an abstract machine. There is often no other way to assure, for example, that a divider actually divides, that a protocol reliably enables communication, or that a calculation involving millions of steps produces the "right" answer.

So we are interested here in a "working logic." We wish to *use* a formal mathematical logic to model other formal systems — processor architectures, microcode programs and programming languages — and then to study the properties of those models. But this presupposes some formal logic suitable for our purposes. Turning to the logic textbooks for a description of a suitable logic is a frustrating experience. Logicians, the people who might have been thought most experienced in designing and using formal logics, have spent most of their time studying logics rather than using them. Compared to the logic we seek, most classical logics are like Turing machines compared to modern processors: while they are, in some technical sense, sufficiently powerful, they are impractical to use. Classical logics were designed to be simple enough to study thoroughly, not convenient to use.

Therefore, in this document we "roll our own" working logic. It is called *ACL2*, which stands for "*A* *C*omputational *L*ogic for *A*pplicative *C*ommon *L*isp" and might have been abbreviated ACL^2. ACL2 is designed to be used to model computing systems and to prove properties of those models. It is likely to be much more elaborate than the classical formal logics you have encountered.

1.1 Connection with Common Lisp

One force pushing ACL2 toward complexity is the fact that we want it to be efficiently executable on a wide range of host processors. That is, the logic can be used to *calculate*: most variable-free terms can be reduced to constants by the routine application of the axioms. To achieve this we decided to make ACL2 an extension of a useful subset of the widely used and efficiently implemented applicative programming language Common Lisp [6] and [7].

In [4] we define the notions of *gold* functions and theorems. To establish that a function or theorem is gold, certain additional theorems must be proved. These theorems show that the Common Lisp primitive functions are never applied to arguments outside their "intended domains." We make the following promise: a gold theorem in ACL2 evaluates to non-`NIL` in all compliant implementations of Common Lisp, unless the implementation encounters a "resource error" such as memory exhaustion. In this document we do not discuss further the notion of gold theorems or Common Lisp compliance. We focus simply on ACL2 as a logic, thereby defining what the theorems are.

Although we do not assume familiarity with Common Lisp, readers familiar with it will notice that we have adopted much of its syntax and many of its built-in constants and functions.

1.2 Connection with the Nqthm Logic

The logic described here is closely connected to the Nqthm ("Boyer-Moore") logic [1]. The Nqthm logic models a "home-grown" Pure Lisp, while ACL2 models Common Lisp. Some of the differences between the two are summarized below.

- Nqthm's arithmetic was essentially just that for the natural numbers, axiomatized in a Peano-like fashion. Common Lisp and ACL2 provide the rationals and the complex rationals, with the naturals being an inductively identified subset of the rationals.

- Nqthm's symbols, the `LITATOM`s, are very simple compared to Common Lisp's. In particular, Nqthm does not support multiple "packages."

- Nqthm provides a "false object," `(FALSE)`, abbreviated `F`, which is not `NIL` or any other Nqthm symbol, while Common Lisp and ACL2 "overload" `NIL` as both the false object and the end-of-list marker.

- Nqthm's functions `CAR` and `CDR` return `0` on non-`CONS` arguments, but ACL2's return `NIL`.

- Nqthm provided the "Shell Principle" for adding new data types; ACL2 provides no such facility. But in addition to the richer set of numbers, ACL2 provides character objects and strings among its primitives.

The enumeration above should not obscure the fact that the two logics in fact only differ in "minor" details. The two logics "feel" the same. Both are first-order, essentially quantifier-free logics based on total, recursively defined functions over inductively constructed data objects. Both use untyped, Lisp-like syntax and encourage the use of "terms" where other logics would use "formulas." Both allow (most) variable-free terms to be evaluated to constants via a "call-by-value" interpreter.

The similarities to Nqthm were the result of deliberate design decisions based on the success of using Nthm to model computing machines and systems. ACL2 — the logic and its implementation in a theorem-proving system — is best thought of as a successor to Nqthm intended to make the logic more efficiently executable and able to support the construction and analysis of larger system models. The paper [3] discusses the original design of ACL2.

Bob Boyer was an active participant in the formative years of the design of ACL2. Without his help the project would never have gotten off the ground.

2 Theories and Logical Events

The view that a logic is given by a language, some axioms, and some rules fails to accommodate our intention to *use* it to model other systems. Except in the unusual case that the given logic already describes what we care about, we must be able to fashion it somehow. That generally means we must be able to extend the language to include new syntactic concepts and to add new axioms about those concepts. Sometimes new rules of inference are added. Thus, in addition to a language, some axioms, and some rules of inference, we provide some *extension principles* and we view the logic as "evolving" under the control of the user who invokes these principles.

We say a formula t *can be proved directly from* a set of axioms $\mathcal{A}$ if and only if t may be derived from the axioms in $\mathcal{A}$ by applying the rules of inference of propositional calculus with equality and instantiation (see page 20) and the principle of induction (see page 47).

There are six extension principles and thus six kinds of *events*:

- an application of the constant definition principle (page 29),
- an application of the package definition principle (page 48),
- the designation of the current package (page 49),
- an application of the function definition principle (page 49),
- an application of the constraint principle (page 51), and
- the addition of an arbitrary formula as an axiom.

Each such event extends the syntax and/or the set of axioms of the logic as noted below.

Note. ACL2 provides for "Skolem axioms" but these have not yet been documented. Also, the implemented definitional principle allows for mutually recursive definitions but the one described here does not. This is just a draft.

A *history* h is a finite sequence of events such that either (a) h is empty or (b) h is obtained by concatenating to the end of a history h' an event that is "admissible" under h'. An arbitrary axiom is admissible under any h'. The specifications of the other kinds of events define "admissibility" for each such event. We refer to the order of events in a history as though they were added chronologically, i.e., the first event is the "oldest" and the last event is the "most recent."

Associated with every history are the following:

- an "arity table," specifying how many arguments certain function symbols take;
- a "package system," affecting how symbols are written;
- a "current package," affecting how symbols are written;
- the "formal constants" of the language;
- a set of "abbreviations," whereby symbolic expressions are transformed into "formal terms" and "formulas;" and
- a set of "axioms."

All but the last item, the axioms, are concerned with the syntax. Each of these concepts is defined below, with respect to a given history h. However, in general in this document the operative history is left implicit.

The *arity table* of a history is the initial arity table (Table 3, page 18) extended by an entry specifying the arity (number of arguments) of each function symbol introduced by each event in the history. Our discussion of each event makes clear the function symbols and arities introduced.

The *package system* of a history is a sequence of pairs, each pairing a "package name" with an "imports list." We discuss packages on page 11. The package system of a history is the initial package system, as described in Appendix A, extended successively by an entry for each `DEFPKG` event in the history, in chronological order. We describe the appropriate entry when we discuss `DEFPKG` (page 48).

The *current package* of a history is the package name selected by the most recent `IN-PACKAGE` event (see page 49), if any. If there is no `IN-PACKAGE` event in the history, the current package is named `"ACL2"`. In this document, unless otherwise stated, the current package is `"ACL2"`.

The *formal constants* of a history are the primitive formal constants (page 16) together with the formal constants introduced by each event. The addition of an arbitrary axiom adds no new constants. When we discuss each other kind of event we specify the constants, if any, introduced.

The *abbreviations* of a history are the abbreviation rules introduced in this document together with the abbreviations introduced by each event. The addition of an arbitrary axiom adds no abbreviations. When we discuss each other kind of event we specify the abbreviations, if any, introduced.

Finally, the *axioms* of a history h are the axioms introduced in this document (including the appropriate instances of the Propositional and Reflexivity axiom schemas for the formulas of h and the appropriate instances of the Equality Axiom for Functions for every function symbol in the arity table of h, see Section 5) together with the axioms introduced by each event. The addition of an arbitrary axiom introduces the given formula as an axiom. When we discuss each other kind of event we specify the axioms, if any, introduced.

The *syntax* of a history is the set of "well-formed formulas" for that history. A large part of this document is devoted to a careful description of this notion. In our development, a formula is a tree structure composed of formulas and other tree structures called "terms." We start by describing how we write down a certain class of tree structures, called "s-expressions." Then we identify a subset of these s-expressions as the "formal terms" in a given history. The main idea is that such a term is a variable symbol, one of a very few constants, or the application of a function symbol of the history to an appropriate number of argument terms. A "well-formed formula" of a history is then a class of s-expressions built from formal terms of the history by certain constructions we describe.

Finally, we introduce a large number of abbreviations. These abbreviations are rules for transforming s-expressions that are not formal terms or formulas into s-expressions that are formal terms or formulas. Of special importance are a collection of abbreviations that let us write a large class of constants in terms of the primitive constants and function symbols.

The *well-formed formulas* of a history h are the s-expressions that are either formulas of h as we define them here or that can be transformed into formulas of h using the abbreviations of h.

We say a formula t in the syntax of history h is a *theorem* of history h iff t can be proved directly from the axioms of h.

3 A Preamble on Notation

The utterances of a formal language are traditionally regarded as strings of characters. Such character strings are often described with a formal grammar. We do not take that approach.

The utterances of our formal language are finite tree structures composed of familiar mathematical objects. In this section we talk about those objects and how we write them down. It is not our intention here to *define* these notions, since we think formal definitions would be less clear than what the reader is likely already to understand. So we offer the following observations as a way of settling on some terminology and notation.

A binary tree is either an atom or an ordered pair of two binary trees. The atoms we most commonly use are numbers, characters, strings of characters, and symbols. We will very occasionally include other atoms in our trees, namely "pseudo-symbols."

We consider the numbers, characters, strings, symbols and ordered pairs to be five different types of objects, i.e., disjoint sets of objects.[1] The integer one is different from the character that prints as "1" and is also different from the character string containing that one character. We hope that is obvious. Many readers may never have thought about the symbol whose name consists of the single character "1". But it exists (in the universe we are imagining) and is distinct from the number, character, and string just mentioned. Finally, the ordered pair whose first component is the integer 1 and whose second component is, say, the number 0 is different from the other four objects.

We obviously need a way to write down these five objects. Here they are in the notation we use:

- the integer one: `1`
- the character "1": `#\1`
- the string containing one "1": `"1"`
- the symbol whose name is the above string: `|1|`
- the ordered pair containing `1` and `0`: `(1 . 0)`

The above display probably raises more questions than it answers! But it should bring home three points. First, five distinct mathematical objects are shown. Second, they are all examples of binary trees (the first four are atomic, the last is not). Third, we need to agree on a notation for binary trees.

Here is a display of more typical examples of these five different kinds of objects:

- numbers:
  - integers: `123`, `-17`
  - rationals: `22/7`, `-127/128`
  - complex rationals: `#c(3 1/2)` (i.e., $3 + \frac{1}{2}i$)
- characters: `#\A`, `#\a`, `#\Space`, `#\,`
- strings: `"I am."`, `"She said \"Hi!\" once."` (the character after the word "said" is `#\Space` and the character after that is "string quote mark," `#\"`)

[1] And we consider the pseudo-symbols to be a sixth type, not actually available in the ACL2 implementation.

- symbols: `X`, `NIL`, `A1`, `|a1|` (the first character in this symbol's name is a lower-case "a", not a vertical bar), `LISP::A1` (the first character in this symbol's name is an upper-case "A"; the "package name" of the symbol is `"LISP"`)
- lists and pairs: `(1 2 3)`, `((ABC 1) (DEF 2))`, `(0 . 1)`

We now discuss the notation for each of these types.

3.1 Numbers

Integers are written as sequences of digits. Base 10 is the most common one used here. So `123` is the integer one hundred twenty three.

But we might also write numbers in binary (`#b1111011`), octal (`#o173`) or hexadecimal (`#x7B`). In numeric notation, case is unimportant. So `#B1111011` and `#X7b` are also integers. In fact, the same integer is shown in each of the examples so far, namely `123`.

The (optional) sign of a number is written immediately before the digit sequence. Thus, `-6` is `#b-110` and also `#o-6`.

We use `typewriter` font when we write integers and the other formal mathematical objects. In such expressions we are always referring to the object denoted, not the particular string of glyphs chosen. For example, we might say "`#b1111100` is one larger than `123`." More likely we would say "`124` is one larger than `123`" or "`#b1111100` is one larger than `#b1111011`" but the point is that we are not talking about the notation used but the integers denoted. We might say "when we write `123` in binary as '`#b1111011`' nine characters are written." When we talk about notation we generally enclose the notation in quotation marks, as done above. But generally, except in this preamble, we do not talk about notation, just the things denoted.

Rationals are written as optionally signed, possibly improper, fractions, with a slash separating the "top" of the fraction from the "bottom." Thus, `-1/2` and `7/2` are rationals. We do not use "mixed notation" to write non-integer rationals; that is, we will not again write $3\frac{1}{2}$ to mean `7/2`.

We used the terms "top" and "bottom" above in reference to the parts of the fractional notation. We use the terms "numerator" and "denominator" exclusively in reference to rationals. The numerator and denominator of a rational r are, respectively, the relatively prime integers i and j ($j > 0$) such that $r = i/j$.

"`5/10`" is just another way to write `1/2`. More bluntly, `5/10` *is* `1/2`. The numerator of `5/10` is `1` and the denominator is `2`.

The integers are a subset of the rationals. "`12/4`" is another way to write `3`. More bluntly, `12/4` *is* `3`. The denominator of `12/4` is `1`.

Rationals can be written in binary, octal, or hexadecimal notation. `-123/20` is `#b-1111011/10100` and `#o-173/24` and also `#x-7B/14`.

We also need a notation for certain complex numbers, namely the ones whose real and imaginary parts are rationals. "`#c(`x y`)`", where x and y are rationals, is the way we write the complex number more commonly written $x + yi$. Of course, if y is 0, the number denoted is the rational x. Thus, `#c(123 0)` is `123`.

Following traditional mathematical usage, a "complex number" is any number of the form $x + yi$, for real x and y. Thus, the complex numbers include the rationals and integers. But when we say a complex number is a "complex rational" we mean its imaginary part is non-0. Thus, the rationals and the complex rationals are disjoint. Together they constitute what we call the "ACL2 numbers" or simply the "numbers," when the simpler term is not confusing. The ACL2 numbers are a subset of the complex numbers.

Different bases may be used to write the two parts of an ACL2 complex number. For example, `#c(6 -17/10)` is `#c(#b110 #x-11/A)`.

3.2 Characters

Each character object has a "name" and a unique integer "character code." Character objects are written by writing a number sign, a backslash and then the name of the character. The character names and their codes are shown in Table 1. As indicated by our examples, some characters, like `#\Space`, have names that are different from the glyph. In addition to the code for each character, Table 1 gives several other characteristics which are explained when we discuss symbols.

Thus, for example, `#\A` is the character object whose code is `65`, `#\a` is the character with code `97`, and `#\Newline` is the character with code `10`. `#\A` is a different object than `#\a`. Other than the fact that distinct characters have distinct codes and the fact that characters are distinct from the other kinds of objects, characters have no interesting properties.

The correspondence between characters and their codes is an extension of the ASCII convention. Each character corresponds to a single (but perhaps chorded) keystroke on a standard keyboard. While this document does not in general try to deal with the practical issues of using ACL2 at your terminal, note that the "control characters," that is, those where $0 \leq code \leq 31$ and *name* is a caret followed by a letter, are generally typed by chording the "Control" key and the key of the indicated letter.

We have informal names for some of the characters because the notation is sometimes jarring.

- `#\"` (code `34`): "string quote mark" or "double gritch,"
- `#\#` (code `35`): "hash mark" or "number sign"
- `#\'` (code `39`): "quote mark" or "single gritch,"
- `#\\` (code `92`): "backslash"
- `#\|` (code `124`): "vertical bar"

code	*name*	*n*	*s*	*code*	*name*	*n*	*s*	*code*	*name*	*n*	*s*
0	`^@`			43	`+`	n		86	`V`		
1	`^A`			44	`,`		s	87	`W`		
2	`^B`			45	`-`	n		88	`X`		
3	`^C`			46	`.`	n	s	89	`Y`		
4	`^D`			47	`/`			90	`Z`		
5	`^E`			48	`0`	n		91	`[`		
6	`^F`			49	`1`	n		92	`\`		s
7	`^G`			50	`2`	n		93	`]`		
8	`Backspace`			51	`3`	n		94	`^`	n	
9	`Tab`			52	`4`	n		95	`_`	n	
10	`Newline`		s	53	`5`	n		96	`` ` ``		s
11	`^K`			54	`6`	n		97	`a`		s
12	`Page`			55	`7`	n		98	`b`		s
13	`Return`			56	`8`	n		99	`c`		s
14	`^N`			57	`9`	n		100	`d`		s
15	`^O`			58	`:`		s	101	`e`		s
16	`^P`			59	`;`		s	102	`f`		s
17	`^Q`			60	`<`			103	`g`		s
18	`^R`			61	`=`			104	`h`		s
19	`^S`			62	`>`			105	`i`		s
20	`^T`			63	`?`			106	`j`		s
21	`^U`			64	`@`			107	`k`		s
22	`^V`			65	`A`			108	`l`		s
23	`^W`			66	`B`			109	`m`		s
24	`^X`			67	`C`			110	`n`		s
25	`^Y`			68	`D`			111	`o`		s
26	`^Z`			69	`E`			112	`p`		s
27	`^[`			70	`F`			113	`q`		s
28	`^\`			71	`G`			114	`r`		s
29	`^]`			72	`H`			115	`s`		s
30	`^^`			73	`I`			116	`t`		s
31	`^_`			74	`J`			117	`u`		s
32	`Space`		s	75	`K`			118	`v`		s
33	`!`			76	`L`			119	`w`		s
34	`"`		s	77	`M`			120	`x`		s
35	`#`		s	78	`N`			121	`y`		s
36	`$`			79	`O`			122	`z`		s
37	`%`			80	`P`			123	`{`		
38	`&`			81	`Q`			124	`|`		s
39	`'`		s	82	`R`			125	`}`		
40	`(`		s	83	`S`			126	`~`		
41	`)`		s	84	`T`			127	`Rubout`		
42	`*`			85	`U`						

Table 1: The ACL2 Character Set

So we might say "a string quote should be preceded by a backslash" instead of saying "`#\"` should be preceded by `#\\`."

3.3 Strings

Character strings are delimited with string quote marks (`#\"`) at each end. To indicate that the string contains a string quote mark, you must precede each such occurrence of string quote mark by a backslash. Similarly, to indicate that the string contains a backslash you must precede each such occurrence of backslash by a backslash. The number of characters in the string is called its *length*.

Thus, `"ABC"` is a string of length three; the successive characters in it are `#\A`, `#\B`, and `#\C`.

`"A\"B"` is also a string of length three; the successive characters in it are `#\A`, `#\"`, and `#\B`.

`"A\\B"` is also a string of length three; the successive characters in it are `#\A`, `#\\`, and `#\B`.

3.4 Symbols

Symbols are the most complicated of our atoms, notationally. Technically, every symbol is composed of two strings, the first called the *package name* and the second called simply the *name* of the symbol. These two are generally separated by two colons when we write symbols. For example, `LISP::ABC` is a symbol. Its package name is the string `"LISP"` and its name is the string `"ABC"`.

The notation for symbols is complicated by three factors.

- The package name and colons can sometimes be omitted.
- The names of symbols (and indeed of packages) can consist of arbitrary characters and thus can look like numbers, strings, etc.; a new "escape" convention is required.
- Some symbols are abbreviations for others, depending on the "package system."

Before discussing symbols further it is convenient to be precise about packages. A *package* is a pair consisting of a string, called the *package name* of the package, and a finite sequence of symbols, called the *imports list* of the package. No two symbols in the imports list may have the same name.

A *package system* is a finite sequence of packages, each of which has a unique package name. Furthermore, the imports list of each package in the system may only contain symbols whose package names are those of packages occurring earlier in the package system sequence. Thus, the first package in the sequence must have an empty imports list, the second package may only import symbols from the first package, the third may import symbols from both of the first two, etc.

The package system of a history, recall, is the initial package system in Appendix A as extended chronologically by each of the `DEFPKG` events in the history. The admissibility requirements on `DEFPKG` insure the invariants above on the package names and imports lists. Among the packages in the initial package system are three especially important ones, named `"KEYWORD"`, `"LISP"`, and `"ACL2"`.

The *witness symbol* for the package named p in a package system is a distinguished symbol whose package name is p. We explain how to construct the witness symbol in Appendix A.

Recall that every history also has a designated "current package." Here, that package is the one named `"ACL2"`.

When we write symbols, the package name and the colons may be omitted as follows.

- If the package name of a symbol is `"KEYWORD"`, it suffices to write just one colon preceding the symbol's name, e.g., `:ABC` is `KEYWORD::ABC`, or less bluntly, "`:ABC`" and "`KEYWORD::ABC`" are two different notations for the same symbol. The symbol in question has package name `"KEYWORD"` and name `"ABC"`.

- If the package name of a symbol is that of the current package, the package name and both colons can be omitted, e.g., if the current package name is `"ACL2"`, then `ABC` is `ACL2::ABC`. That is, (given the current package name) the package name of `ABC` is `"ACL2"` and the name is `"ABC"`.

In a way, the current package is to the notation for symbols what the "current base" would be in the notation for numbers. One can imagine a math book saying, "the numbers in this section are written in octal." Such a remark would affect one's reading of all the numbers in the section. The designation of the current package analogously affects how symbols are read and written.

Symbol names can be arbitrary strings of characters. The name of the symbol `ABC` is `"ABC"`. Since symbol names are arbitrary strings, there is a symbol with name `"123"`. Clearly, that symbol is not `123`, since `123` is an integer, and it is not `"123"`, because that is a string. The symbol whose name is `"123"` is `|123|`. The symbol whose name is `"|123|"` is `|\|123\||`. That is, to indicate a symbol whose name string contains a vertical bar, you must precede the occurrence of the vertical bar by a backslash.

Every symbol name and package name could be delimited by vertical bars (with the explicit vertical bars in the names being "escaped" with backslash as above). That is, `ABC` is `|ABC|`, which is also `|ACL2|::|ABC|`. However, by convention, we only write the vertical bars when, without them, the symbol would be mistaken for a number or some other binary tree, or when the symbol's name has lower-case characters in it. More precisely, if the first character in the symbol's name string is one of those marked with n (for "numeric") in Table 1 or the name string contains any of the characters marked with s (for "signs and lower-case"), then the vertical bar notation is mandatory. Otherwise it may be dropped; when the vertical bar notation is not used for a symbol, all the alphabetic characters in its name are understood to be in upper-case.

The convention concerning case allows us to write a symbol in lower-case even though all the alphabetic characters in its name are upper-case. For example, `abc` is `ABC`. The symbol with name `"abc"` is `|abc|`.

Technically, every symbol can be written down in the form |*p*|::|*name*|, where "*p*" is the name of a package and "*name*" is the name of the symbol. (This statement must be understood in the context of the "backslash escape" notation. For example, if "*p*" is `"|||"` then by "|*p*|" we mean "`|\|\|\||`".)

Finally, some symbols are abbreviations for others. This is akin to the convention that "`5/10`" is just another way to write `1/2`, or as we have said before, `5/10` is `1/2`. Similarly, `ACL2::NIL` is `LISP::NIL`. Why? What are the rules for determining when two different notations denote the same symbol?

The fundamental idea involved in resolving the denotation of "x/y" is the notion of relatively prime integers. The fundamental idea involved in resolving the denotation of "|*p*|::|*name*|" is the notion of importation, as specified by the current package system.

`ACL2::NIL` is `LISP::NIL` because the latter symbol is imported into the `"ACL2"` package in the current package system. The list of symbols imported into an ACL2 package is finite and fixed forever at the time the package is admitted. The imports list is recorded with the package name in the package system in the history. No two symbols with the same name may be imported into a package and the package name of every symbol imported must have been admitted earlier in the history. These invariants give us a simple algorithm for resolving symbol notation.

To read fractional notation one must be able to answer the question "what is the numerator and denominator of the denoted rational?" To read symbol notation one must be able to answer the question "what is the package name and name of the denoted symbol?"

The name of the symbol denoted by "|*p*|::|*name*|" is the string "*name*". Here is how you determine the package name of the denoted symbol. Look at the list of symbols declared imported when package "*p*" was admitted. If none of them have the name "*name*", the package name of |*p*|::|*name*| is the string "*p*". If, on the other hand, one of them, x, has the name "*name*", then recursively determine the package name of x. This algorithm is finite and deterministic because of the ACL2 importation rules.

What symbol is meant when we write "`QUOTE`" in this document? Obviously, some symbol with name `"QUOTE"`, but which one? What is the package name of `QUOTE`? The current package is understood here to be `"ACL2"`, so here `QUOTE` is `ACL2::QUOTE`. Inspection of Appendix A and Appendix B reveals that `LISP::QUOTE` is imported into the `"ACL2"` package. So the package name of

`ACL2::QUOTE` is the package name of `LISP::QUOTE`. Inspection of Appendix A again reveals that no symbols are imported into the `"LISP"` package. Thus, the package name of `LISP::QUOTE`, and hence of `ACL2::QUOTE` and of `QUOTE`, is `"LISP"`.

Similarly, the package name of `T` is `"LISP"`. The package name of `NIL` is `"LISP"`. But the package name of `ABC`, i.e., of `ACL2::ABC`, is `"ACL2"`, because no symbol with name `"ABC"` is imported into the `"ACL2"` package.

For typographic reasons, sometimes when we are speaking informally we may break long symbols on hyphens or colons, e.g.,

> The symbol `*COMMON-LISP-SYMBOLS-FROM-MAIN--`
> `LISP-PACKAGE*` is defined in an Appendix.

provided the context permits an unambiguous interpretation. An extra hyphen is added at the break.

3.5 Ordered Pairs

So far we have just talked about the "atoms" in our binary trees. How do we write pairs? We seldom use the traditional notation for pairs, $\langle x, y\rangle$. Instead, we use a more elaborate notation that is ultimately more succinct.

Consider an arbitrary ordered pair p containing x and y. One way to write it is `(`x `.` y`)`. But if y can be written in such parenthetical notation, e.g., `(`...`)`, then a more succinct way to write p is `(`x ...`)`. Finally, one way to write `NIL` is `()`, i.e., as an empty pair of parentheses.

For example, let p_1 be the pair whose first component is the symbol `X` and whose second component is `NIL`. Then we could write p_1 as `(X . NIL)` or `(X . ())` but we prefer the more succinct `(X)`.

Now let p_2 be the pair whose first component is the integer `123` and whose second component is p_1. We can write p_2 as `(123 . (X))` but usually prefer the more succinct `(123 X)`.

Finally, let p_3 be the pair whose first component is p_2 and whose second component is also p_2. Then p_3 can be written `((123 X) 123 X)`. It could also be written `((123 X) . (123 X))`, which might sometimes be preferred.

A binary tree whose rightmost branch is n long and concludes with the atom atm could be written `(`x_1 `. (`x_2 `. (... . (`x_n `.` atm`)...)))` but, using the more succinct notation above is most commonly written as `(`x_1 x_2 ... x_n `.` atm`)`. Such a tree is often called a "sequence" or "list" of "length" n. Its "elements" are enumerated in the order shown, i.e., the first element is x_1 and the last is x_n. In the special case that atm is `NIL`, we call the list a *true list* and, of course, it can be written `(`x_1 x_2 ... x_n`)`.

When we write ordered pairs we allow an arbitrary (non-empty) amount of "whitespace" where we show spaces above. By whitespace we mean spaces, new lines, and comments. A comment is text delimited on the left by a semi-colon and on the right by the end of line.

Below we show the true list of length four containing, successively, the symbol `ABC`, the list p_3 above, the string `"I am."` and the symbol `|(I am)|`.

```
(ABC       ; The first element of the sequence is a symbol.
           ; This is a comment.
 ((123 X)  ; The second element is a list.
  123
  X)
 "I am."   ; The third is a string.
 |(I am)|  ; The fourth is a symbol.
           ; And now we close.
 )
```

Finally, a binary tree of the form `(QUOTE` x`)`, that is, a true list of length two whose first element is the symbol `QUOTE` and whose second is some tree x, may be abbreviated `'`x.

Thus, for example, `(H 'ABC (G X Y))` is `(H (QUOTE ABC) (G X Y))` and `(H ''ABC)` is `(H (QUOTE (QUOTE ABC)))`.

What ordered pair is `''2`? The answer is `(QUOTE (QUOTE 2))`, i.e., the true list of length two whose first element is the symbol `QUOTE` and whose second element is the true list of length two whose first element is the symbol `QUOTE` and whose second element is the integer `2`.

One might try to parse "`''2`" as two binary trees, the first being "`''`" and the second being "2". To proceed along these lines, "`''`" would have to be parsed as "`(QUOTE ')`". But "`'`" denotes no binary tree in our notation; in particular, it is not how we write any number, character object, string, symbol or list. If one is tempted to write "`'`" as a binary tree one is probably thinking of the character `#\'`, the string `"'"`, the symbol `|'|` or perhaps the symbol `QUOTE`. But "`'`" doesn't denote a binary tree so "`''`" cannot be seen as one either.

3.6 S-Expressions

The notation we have just described for binary trees is that supported by Common Lisp. In Lisp parlance, a binary tree is called an *s-expression* (or *symbolic expression*). Henceforth, we use the term "s-expression" rather than "binary tree." But the reader should keep in mind that by s-expression we just mean a very familiar and simple mathematical object: a binary tree of several kinds of atoms.

We assume the reader is able to parse the notation so that when we exhibit an s-expression the appropriate binary tree comes to mind. We assume no special properties of the components, e.g., the characters or symbols, other than those sketched above, namely how to read and write the different objects.

To talk about s-expressions it is convenient to use so-called *metavariables* that are understood by the reader to stand for other s-expressions or their components. We use `typewriter` font when we exhibit particular s-expressions or components and *italics* when we exhibit metavariables.

For example, if f is understood to stand for the symbol `CONS`, and t to stand for the s-expression `(CAR X)`, then by `(`f t `X)` we mean the s-expression `(CONS (CAR X) X)`.

If n is the integer `123` and we use `X`n as a symbol, we mean the symbol `X123`. This meta-convention arbitrarily adopts decimal notation. (A careful reader once asked "If n is the integer `#b1111011`, doesn't `X`n mean `|X#b1111011|`?")

This concludes our review of notation. We will now identify certain s-expressions as "terms" in our formal logic and proceed to give the axioms and rules of inference.

4 Formal Syntax

Note. In this section we describe the set of s-expressions that represent the "terms" of our formal language. For those readers unfamiliar with the taxonomy of traditional formal logical syntax: formulas are built out of terms, axioms are formulas, and rules of inference allow us to manipulate formulas (sometimes by manipulating the terms in them, as by replacing the variables by other terms). The rules of inference are most clearly stated if the structure of formulas is very simple. Thus we follow logical tradition when we describe a very simple term language and then, in Section 6, extend it with a host of abbreviations so that it is convenient to use.

Terminology. A symbol v is a *variable symbol* of our language unless prohibited as below:

- Symbols marked with (*) in Appendix B are not variable symbols. This prohibition means that `T` and `NIL` (among many others) are not variable symbols.
- Symbols with package name `"KEYWORD"` are not variable symbols.
- Symbols whose names start and end with the character `#\*` are not variable symbols.
- Symbols whose names start with `#\&` are not variable symbols.
- Symbols with package name `"LISP"` that are not listed in Appendix B are not variable symbols.

Terminology. A *primitive formal constant* is an s-expression of length two whose first element is the symbol `QUOTE` and whose second element is one of the integers or symbols listed in Table 2.

Examples. `'0` and `'ACL2::WITNESS` are primitive formal constants. Perhaps surprisingly, `'3` and `'ABC` are not formal constants in our language. We have

constant	*comment*
`0`	zero
`1`	one
`NIL`	false; empty list
`STRING`	symbolic token
`ACL2::WITNESS`	`ACL2` package witness symbol
`LISP::WITNESS`	`LISP` package witness symbol
`KEYWORD::WITNESS`	`KEYWORD` package witness symbol
`ACL2-OUTPUT-CHANNEL::WITNESS`	`ACL2-OUTPUT-CHANNEL` package witness symbol
`ACL2-INPUT-CHANNEL::WITNESS`	`ACL2-INPUT-CHANNEL` package witness symbol
`ACL2-PC::WITNESS`	`ACL2-PC` package witness symbol
`ACL2-USER::WITNESS`	`ACL2-USER` package witness symbol

Table 2: The ACL2 Primitive Constants

chosen to keep the term structure exceedingly simple; in any given history, the set of formal constants is finite. We introduce abbreviation conventions allowing us to use other s-expressions, such as `'3` and `'ABC`, as though they were constants; in fact these abbreviations are "expanded" into formal terms involving the formal constants above.

Terminology. A symbol f is a *function symbol* of our language unless prohibited as below:

- Symbols with package name `"KEYWORD"` are not function symbols.
- Symbols with package name `"LISP"` are not function symbols unless they are so defined in this document.

Terminology. Associated with every function symbol in a history h is a non-negative integer called the *arity* of the symbol. The arity indicates how many argument terms must follow each application of the function symbol. The arity of each primitive function symbol is given in Table 3, along with a brief descriptive comment. Certain of our axioms, namely those labeled "(Def)," introduce additional function symbols with equations that relate the new symbols to old ones. Table 3 should be understood to be extended by appropriate entries for these "defined" symbols.

Terminology. An s-expression t is a *formal term* of a history h if and only if

- t is variable symbol;
- t is a primitive formal constant of h ; or
- t is of the form `(`f $t_1 ... t_n$`)`, where f is a function symbol with arity n in the arity table of h and the t_i are formal terms of h.

function symbol	*arity*	*comment*
`BINARY-*`	2	multiplies two numbers
`BINARY-+`	2	adds two numbers
`UNARY--`	1	negates a number
`UNARY-/`	1	inverts a number
`<`	2	less than on the rationals
`BOOLEANP`	1	recognizes `’T` and `‘NIL`
`CAR`	1	first element of a list
`CDR`	1	all but first element of a list
`CHAR-CODE`	1	maps characters to integers
`CHARACTERP`	1	recognizes characters
`CODE-CHAR`	1	maps integers to characters
`COMPLEX`	2	builds a complex from two rationals
`COMPLEX-RATIONALP`	1	recognizes a complex number
`COERCE`	2	maps between character lists and strings
`CONS`	2	builds a list
`CONSP`	1	recognizes a non-empty list
`DENOMINATOR`	1	denominator of a rational
`EQUAL`	2	equality predicate
`IF`	3	if-then-else
`IMAGPART`	1	imaginary part of a complex
`INTEGERP`	1	recognizes integers
`INTERN-IN-PACKAGE-OF-SYMBOL`	2	maps a string to a symbol
`NUMERATOR`	1	numerator of a rational
`RATIONALP`	1	recognizes rationals
`REALPART`	1	real part of a complex
`STRINGP`	1	recognizes strings of characters
`SYMBOL-NAME`	1	name of a symbol
`SYMBOL-PACKAGE-NAME`	1	package name of a symbol
`SYMBOLP`	1	recognizes symbols

Table 3: The ACL2 Primitive Function Symbols

Examples. The following are formal terms (in every history h):

```
(CONS X (CONS Y 'NIL))

(BINARY-+ '1 (BINARY-* U V))

(IF B
    (BINARY-+ '1
              (BINARY-+ '1 '1))
    (CAR (CDR X)))
```

Terminology. A term t is a *call* of f with *arguments* a_1, ..., a_n iff t has the form (f a_1 ... a_n).

Terminology. If a term t is a call of f we say f is the *top function symbol* of t. A function symbol f *is called in* a term t iff either t is a call of f or t is a nonvariable, non-constant term and f is called in an argument of t. The set of *subterms* of a term t is $\{t\}$ if t is a variable symbol or constant and otherwise is the union of $\{t\}$ together with the union of the subterms of the arguments of t. The *variables* of a term t is the set of variable subterms of t.

Examples. The term `(CONS X Y)` is a call of `CONS` with arguments `X` and `Y`. `CONS` is called in `(IF A (CONS X Y) B)`. The set of subterms of `(CONS X Y)` is {`(CONS X Y)`, `X`, `Y`}. The set of variables of `(CONS X Y)` is {`X`, `Y`}.

Terminology. A function symbol f is *new* in a history h iff f is called in no axiom of h (except for the Propositional, Reflexivity, and Equality Axioms of Section 5).

Terminology. A finite set σ of ordered pairs is said to be a *substitution* provided that for each ordered pair $\langle v, t\rangle$ in σ, v is a variable, t is a term, and no other member of σ has v as its first component. The *result of substituting* a substitution σ *into* a term or formula x, denoted x/σ, is the term or formula obtained by simultaneously replacing, for each $\langle v, t\rangle$ in σ, each occurrence of v as a variable in x with t. We sometimes say x/σ is the *result of instantiating* x with σ. We say that x' is an *instance* of x if there is a substitution σ such that x' is x/σ.

Example. If σ is {⟨ `X`, `(CAR L)` ⟩, ⟨ `Y`, `Z` ⟩, ⟨ `G`, `FOO` ⟩} then σ is a substitution. If `G` is a function symbol of two arguments, then

```
(CONS     X    (G Y    X))
```

is a term, which we shall here call p. Then p/σ is the term

```
(CONS (CAR L) (G Z (CAR L))).
```

Note that even though the substitution contains the pair ⟨ `G`, `FOO` ⟩ the occurrence of `G` in p was not replaced by `FOO` since the occurrence of `G` is not as a variable.

5 Propositional Calculus with Equality

Note. Our logic is built on top of traditional propositional calculus with equality. Any classical formalization of propositional calculus and equality will suit our purposes. So that this document is self-contained we have included one such formalization, namely that of Shoenfield [5]. Shoenfield formalizes propositional calculus with one axiom schema and four rules of inference. He introduces equality with three axiom schemas. We then add the rule of instantiation in place of the parts of [5] that refer to quantification, since our logic is quantifier-free.

Terminology. The *pseudo-symbols* are $=$, $\neq$, $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$.

Note. The pseudo-symbols are not symbols as we have defined symbols. (The careful reader will notice the difference between the pseudo-symbol $=$ and the symbol `=`.) We will use pseudo-symbols as atoms in a certain class of s-expressions described below, called formulas.

Terminology. An *atomic formula* is any s-expression of the form $(t_1 = t_2)$, where t_1 and t_2 are terms. A *formula* is either an atomic formula, or else of the form $(\neg\phi)$, where ϕ is a formula, or else of the form $(\phi_1 \vee \phi_2)$, where ϕ_1 and ϕ_2 are both formulas. Parentheses are omitted from formulas (but not from terms) when no ambiguity arises.

Terminology. Generally, we use Greek letters as metavariables standing for formulas. Greek letters, in particular "σ", are also used to stand for substitutions.

Terminology. We extend the notion of "instance" in the natural way so that an instance ϕ/σ of a formula ϕ under a substitution σ is obtained by instantiating every term in ϕ with σ.

Abbreviation. When $(t_1 \neq t_2)$ is used as a formula it is an abbreviation for the formula $(\neg(t_1 = t_2))$. When $(\phi_1 \rightarrow \phi_2)$ is used as a formula, it is an abbreviation for $(\neg\phi_1 \vee \phi_2)$. When $(\phi_1 \wedge \phi_2)$ is used as a formula, it is an abbreviation for the formula $\neg(\neg\phi_1 \vee \neg\phi_2)$. When $(\phi_1 \leftrightarrow \phi_2)$ is used as a formula, it is an abbreviation for the formula abbreviated by $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$.

Axiom Schema (the *Propositional Axiom*).

$(\neg\phi \ \vee \ \phi)$

Note. By this we mean to add such an axiom for every formula ϕ.

Rules of Inference.

- *Expansion*: derive $(\phi_1 \vee \phi_2)$ from ϕ_2;
- *Contraction*: derive ϕ from $(\phi \vee \phi)$;

- *Associativity*: derive $((\phi_1 \vee \phi_2) \vee \phi_3)$ from $(\phi_1 \vee (\phi_2 \vee \phi_3))$; and
- *Cut*: derive $(\phi_2 \vee \phi_3)$ from $(\phi_1 \vee \phi_2)$ and $(\neg\phi_1 \vee \phi_3)$.

Axiom Schema (*Reflexivity*).

$(x \;=\; x)$

Axiom Schema (*Equality Axioms for Functions*).
For every function symbol f of arity n we add:

((X1 = Y1) $\rightarrow$
  ...
    ((Xn = Yn) $\rightarrow$

    (f X1 ... Xn) = (f Y1 ... Yn))...)

Axiom. (*Equality Axiom for* =)

((X1=Y1) $\rightarrow$ ((X2=Y2) $\rightarrow$ ((X1=X2) $\rightarrow$ (Y1=Y2)))).

Rule of Inference. *Instantiation*:
Derive ϕ/σ from ϕ.

6 Primitive Macros

Terminology. When we write "*pat* $\Longrightarrow$*term*" or say that "*pat* is an abbreviation for *term*" we mean that when an s-expression x matching *pat* is used where a formal term is expected and the corresponding interpretation of *term* is (or abbreviates) a formal term, t, then x should be read as t.

The formally inclined reader may prefer to think of this section as defining a map, $\Longrightarrow$, from a certain set of s-expressions (containing all terms and all abbreviations of terms) to a subset of it (the set of terms). More precisely, $\Longrightarrow$is the transitive closure of the relation given below. Although $\Longrightarrow$is a function, it is not one-to-one; for example, we'll see that both `2` and `'2` abbreviate the same term, namely `(BINARY-+ '1 '1)`.

Note. ACL2 provides a "macro" facility (derived from the one in Common Lisp [7]) whereby the user can add abbreviations. In our implementation of ACL2 we add most of the abbreviations below as macros that expand as shown here.

Abbreviation. We use `(AND` $p_1 ... p_n$`)` to abbreviate certain formal terms as indicated by the sequence of examples below.

- `(AND)` $\Longrightarrow$`'T`.
- `(AND` p_1`)` $\Longrightarrow p_1$.

- `(AND` p_1 p_2`)` $\Longrightarrow$`(IF` p_1 p_2 `'NIL)`.
- `(AND` p_1 p_2 p_3`)` $\Longrightarrow$`(IF` p_1 `(IF` p_2 p_3 `'NIL) 'NIL)`.
- `(AND` p_1 p_2 p_3 p_4`)` $\Longrightarrow$`(IF` p_1 `(IF` p_2 `(IF` p_3 p_4 `'NIL) 'NIL) 'NIL)`.
- etc.

Abbreviation. We use `(OR` $p_1 ... p_n$`)` to abbreviate certain formal terms as indicated by the sequence of examples below.

- `(OR)` $\Longrightarrow$`'NIL`.
- `(OR` p_1`)` $\Longrightarrow$$p_1$.
- `(OR` p_1 p_2`)` $\Longrightarrow$`(IF` p_1 p_1 p_2`)`.
- `(OR` p_1 p_2 p_3`)` $\Longrightarrow$`(IF` p_1 p_1 `(IF` p_2 p_2 p_3`))`.
- `(OR` p_1 p_2 p_3 p_4`)` $\Longrightarrow$`(IF` p_1 p_1 `(IF` p_2 p_2 `(IF` p_3 p_3 p_4`)))`.
- etc.

Abbreviation. We use `(+` $x_1 ... x_n$`)` to abbreviate certain formal terms as indicated by the sequence of examples below.

- `(+)` $\Longrightarrow$`'0`.
- `(+` x_1`)` $\Longrightarrow$`(BINARY-+ '0` x_1`)`.
- `(+` x_1 x_2`)` $\Longrightarrow$`(BINARY-+` x_1 x_2`)`.
- `(+` x_1 x_2 x_3`)` $\Longrightarrow$`(BINARY-+` x_1 `(BINARY-+` x_2 x_3`))`.
- `(+` x_1 x_2 x_3 x_4`)` $\Longrightarrow$

  `(BINARY-+` x_1 `(BINARY-+` x_2 `(BINARY-+` x_3 x_4`)))`.
- etc.

Abbreviation. We use `(*` $x_1 ... x_n$`)` to abbreviate certain formal terms as indicated by the sequence of examples below.

- `(*)` $\Longrightarrow$`'1`.
- `(*` x_1`)` $\Longrightarrow$`(BINARY-* '1` x_1`)`.
- `(*` x_1 x_2`)` $\Longrightarrow$`(BINARY-*` x_1 x_2`)`.
- `(*` x_1 x_2 x_3`)` $\Longrightarrow$`(BINARY-*` x_1 `(BINARY-*` x_2 x_3`))`.

- `(*` x_1 x_2 x_3 x_4`)` $\Longrightarrow$

  `(BINARY-*` x_1 `(BINARY-*` x_2 `(BINARY-*` x_3 x_4`)))`.

- etc.

Abbreviation.

- `(-` x_1`)` $\Longrightarrow$`(UNARY--` x_1`)`.
- `(-` x_1 x_2`)` $\Longrightarrow$`(BINARY-+` x_1 `(UNARY--` x_2`))`.

Abbreviation.

- `(/` x_1`)` $\Longrightarrow$`(UNARY-/` x_1`)`.
- `(/` x_1 x_2`)` $\Longrightarrow$`(BINARY-*` x_1 `(UNARY-/` x_2`))`.

Abbreviation. `(LET ((`v_1 a_1`)` ... `(`v_n a_n`))` $term$`)` $\Longrightarrow$ $term_{\{\langle v_1,a_1\rangle,\ldots\langle v_n,a_n\rangle\}}$, where the v_i are distinct variable symbols.

Abbreviation. `((LAMBDA (`v_1 ... v_n`)` $term$`)` a_1 ... a_n`)` $\Longrightarrow$ `(LET ((`v_1 a_1`)` ... `(`v_n a_n`))` $term$`)`.

Abbreviation.

- `(LET* ()` $term$`)` $\Longrightarrow$$term$.
- `(LET* ((`v_1 a_1`)` ...`)` $term$`)` $\Longrightarrow$
  `(LET ((`v_1 a_1`))` `(LET* (`...`)` $term$`))`.

Abbreviation.

- `(COND)` $\Longrightarrow$`'NIL`.
- `(COND (T` x`))` $\Longrightarrow$$x$.
- `(COND (`p x`)` ...`)` $\Longrightarrow$`(IF` p x `(COND` ...`))`, when p is not the s-expression `T`.

Abbreviation.

- `(LIST)` $\Longrightarrow$`'NIL`.
- `(LIST` x_1 ...`)` $\Longrightarrow$`(CONS` x_1 `(LIST` ...`))`.

Abbreviation.

- (LIST* x_1) $\Longrightarrow x_1$.
- (LIST* x_1 x_2 ...) $\Longrightarrow$(CONS x_1 (LIST* x_2 ...)).

Examples. Thus, (LIST X Y Z) $\Longrightarrow$(CONS X (CONS Y (CONS Z 'NIL))) while (LIST* X Y Z) $\Longrightarrow$(CONS X (CONS Y Z)).

Abbreviation.

- (<= x y) $\Longrightarrow$(NOT (< y x)).
- (> x y) $\Longrightarrow$(< y x).
- (>= x y) $\Longrightarrow$(NOT (< x y)).

Abbreviation.

- (CAAR x) $\Longrightarrow$(CAR (CAR x)).
- (CADR x) $\Longrightarrow$(CAR (CDR x)).
- (CDAR x) $\Longrightarrow$(CDR (CAR x)).
- (CDDR x) $\Longrightarrow$(CDR (CDR x)).
- (CAAAR x) $\Longrightarrow$(CAR (CAAR x)).
- (CAADR x) $\Longrightarrow$(CAR (CADR x)).
- (CADAR x) $\Longrightarrow$(CAR (CDAR x)).
- (CADDR x) $\Longrightarrow$(CAR (CDDR x)).
- (CDAAR x) $\Longrightarrow$(CDR (CAAR x)).
- (CDADR x) $\Longrightarrow$(CDR (CADR x)).
- (CDDAR x) $\Longrightarrow$(CDR (CDAR x)).
- (CDDDR x) $\Longrightarrow$(CDR (CDDR x)).
- (CAAAAR x) $\Longrightarrow$(CAR (CAAAR x)).
- (CAAADR x) $\Longrightarrow$(CAR (CAADR x)).
- (CAADAR x) $\Longrightarrow$(CAR (CADAR x)).
- (CAADDR x) $\Longrightarrow$(CAR (CADDR x)).
- (CADAAR x) $\Longrightarrow$(CAR (CDAAR x)).

- `(CADADR` x`)` $\Longrightarrow$`(CAR (CDADR` x`))`.
- `(CADDAR` x`)` $\Longrightarrow$`(CAR (CDDAR` x`))`.
- `(CADDDR` x`)` $\Longrightarrow$`(CAR (CDDDR` x`))`.
- `(CDAAAR` x`)` $\Longrightarrow$`(CDR (CAAAR` x`))`.
- `(CDAADR` x`)` $\Longrightarrow$`(CDR (CAADR` x`))`.
- `(CDADAR` x`)` $\Longrightarrow$`(CDR (CADAR` x`))`.
- `(CDADDR` x`)` $\Longrightarrow$`(CDR (CADDR` x`))`.
- `(CDDAAR` x`)` $\Longrightarrow$`(CDR (CDAAR` x`))`.
- `(CDDADR` x`)` $\Longrightarrow$`(CDR (CDADR` x`))`.
- `(CDDDAR` x`)` $\Longrightarrow$`(CDR (CDDAR` x`))`.
- `(CDDDDR` x`)` $\Longrightarrow$`(CDR (CDDDR` x`))`.

7 Abbreviations for Quoted Constants

Note. It would be nice to be able to say that "`((+ 2 2) = 4)` is a theorem." But `2` and `4` are not formal terms. It is convenient to have a notation for representing "constants," that is, variable-free terms constructed from the primitive constants and function symbols. We now introduce abbreviation conventions that codify the construction of all of the s-expressions built from numbers, characters, strings and/or symbols. That is, if x is such an s-expression then the rules below are sufficient to make `'`x an abbreviation for a formal term.

Terminology.

- `'2` $\Longrightarrow$`(+ '1 '1)`.
- `'3` $\Longrightarrow$`(+ '1 '2)`.
- `'4` $\Longrightarrow$`(+ '1 '3)`.
- More generally, `'`n, where n is an integer greater than 1, and $n = m + 1$, abbreviates `(+ '1 '`m`)`.
- `'-1` $\Longrightarrow$`(- '1)`.
- `'-2` $\Longrightarrow$`(- '2)`.
- `'-3` $\Longrightarrow$`(- '3)`.

- More generally, `'-`n, where n is a positive integer, abbreviates `(- '`n`)`.
- `'`r, where r is a non-integer rational with numerator i and denominator j (i.e., $r = i/j$ where i and j are relatively-prime integers and $j > 1$), abbreviates `(* '`i `(/ '`j`))`.
- `'`c, where c is a complex rational with real part x and imaginary part y, abbreviates `(COMPLEX '`x `'`y`)`.

Together with `'0` and `'1`, the above terms are the *numeric constants*.

Example. `'4` abbreviates the formal term

```
(BINARY-+ '1 (BINARY-+ '1 (BINARY-+ '1 '1))).
```

The s-expression `'4/3` abbreviates (the same term as abbreviated by) `(* '4 (/ '3))`. The s-expression `'#c(4 3)` abbreviates (the same term as abbreviated by) `(COMPLEX '4 '3)`.

Abbreviation. `'`*char* $\Longrightarrow$`(CODE-CHAR '`*code*`)`, when *char* is a character object with character code *code*. (See Table 1.) Such terms are the *character constants*.

Examples. `'#\A` is an abbreviation for `(CODE-CHAR '65)`. `'#\Newline` is an abbreviation for `(CODE-CHAR '10)`.

Abbreviation. `'`*string* $\Longrightarrow$

`(COERCE (LIST '`$char_1$ ... `'`$char_n$`) 'STRING)`

when *string* is a string of length n containing, successively, the character objects $char_1, ..., char_n$. Such terms are the *string constants*.

Example. `'"I am"` is an abbreviation for

```
 (COERCE (LIST '#\I '#\Space '#\a '#\m) 'STRING).
```

The term `'"Say \"Hi!\" Jo"` is an abbreviation for

```
(COERCE (LIST '#\S '#\a '#\y '#\Space
              '#\" '#\H '#\i '#\! '#\" '#\Space
              '#\J '#\o)
        'STRING)
```

Abbreviation. `'`*symbol* $\Longrightarrow$
`(INTERN-IN-PACKAGE-OF-SYMBOL '`*name* `'`*witness*`)`, when *symbol* is a symbol not listed in Table 2, *name* is the name of *symbol* and '*witness* is the witness symbol (see below) for the package name of *symbol*. (Note that `'`*witness* is thus a primitive constant.) Together with the primitive constants other than `'0` and `'1`, such terms are the *symbol constants*.

Example. The symbol constant `'ABC` is an abbreviation for

```
(INTERN-IN-PACKAGE-OF-SYMBOL '"ABC" 'ACL2::WITNESS).
```

That is, in turn, an abbreviation for

```
(INTERN-IN-PACKAGE-OF-SYMBOL
  (COERCE (LIST '#\A '#\B '#\C)
          'STRING)
  'ACL2::WITNESS)
```

which is an abbreviation for

```
(INTERN-IN-PACKAGE-OF-SYMBOL
  (COERCE (LIST (CODE-CHAR '65)
                (CODE-CHAR '66)
                (CODE-CHAR '67))
          'STRING)
  'ACL2::WITNESS).
```

We could, of course, further expand the integers above into the form `(BINARY-+ '1 (BINARY-+ '1 ...))`. Furthermore, since the current package is `ACL2` we could write `'ACL2::WITNESS` as simply `'WITNESS` but we chose to make the package explicit since that is the role of the second argument of `INTERN-IN--PACKAGE-OF-SYMBOL`.

Example. Since the `ACL2` package is the current package, `'T` is `'ACL2::T`. But the symbol `ACL2::T` is `LISP::T`, since the symbol `LISP::T` is imported into the `ACL2` package. Hence, the formal term abbreviated by `'T` is

```
(INTERN-IN-PACKAGE-OF-SYMBOL '"T" 'LISP::WITNESS).
```

Abbreviation. `'(`s_1 s_2 ... s_n`)` $\Longrightarrow$`(LIST '`s_1 `'`s_2 ... `'`s_n`)`.

Abbreviation. `'(`s_1 s_2 ... . s_n`)` $\Longrightarrow$`(LIST* '`s_1 `'`s_2 ... `'`s_{n-1} `'`s_n`)`.

Note. We appear to have given two conflicting abbreviation rules above. For example the first can be applied directly to `'(1 2)` to produce `(LIST '1 '2)`. But the second can be applied to `'(1 2)` also if we first write that s-expression equivalently as `'(1 2 .  NIL)`. This produces the term `(LIST* '1 '2 'NIL)`. Which is meant? The answer is that it doesn't matter: the two "terms" are abbreviations for the same term.

Note. The foregoing abbreviations, together with the primitive constants of Table 2, suffice to allow us to write `'`x as a term, for all s-expressions x composed of the numeric, character, string or symbol atoms. It is convenient to allow the single quote mark to be dropped in certain cases where ambiguity does not arise.

Abbreviation.

- `T` $\Longrightarrow$`'T`.

- `NIL` $\Longrightarrow$`'NIL`.
- *keyword* $\Longrightarrow$`'`*keyword*, when *keyword* is a symbol whose package name is `"KEYWORD"`.
- $n \Longrightarrow$`'`n, when n is a number.
- *char* $\Longrightarrow$`'`*char*, when *char* is a character.
- *string* $\Longrightarrow$`'`*string*, when *string* is a string.

Note. If we were to drop the single quote mark on an arbitrary symbol, allowing `ABC` to be an abbreviation for `'ABC`, then ambiguity results because `ABC` is a formal term, namely a variable symbol. Similarly, if we were to drop the single quote mark on lists, allowing `(CAR X)` to be an abbreviation for `'(CAR X)`, then ambiguity might result since some lists are formal terms.

Example. It is clarifying of our conventions to consider what is meant when `''2` is used as a term. To use `''2` as a term is to imply it is an s-expression. Recall that the s-expression `''2` is `(QUOTE (QUOTE 2))`.

So what term is abbreviated by `''2`? The answer is `(CONS 'QUOTE (CONS '2 'NIL))`:[2]

`''2` *is the same as*
`'(QUOTE 2)` *which abbreviates*
`(LIST 'QUOTE '2)` *which abbreviates*
`(CONS 'QUOTE (CONS '2 'NIL))`

where `'QUOTE` and `'2` could be further expanded, though we stop here.

These abbreviation conventions are compatible with those of Common Lisp [7]. Some newcomers to Lisp mistake `''2` as just another way to write `2`, arguing "you can drop the quote marks before numbers." This reasoning is incorrect. An accurate reading of "$n \Longrightarrow$`'`n, when n is a number" is that you can drop the quote marks before a number *when the number is used as a term.* But in `''2` the `2` is not being used as a term, it is just a component of the s-expression `''2`. It is the s-expression `''2` that is being used as a term.

Another mistake is to read `''2` as `'(BINARY-+ '1 '1)`. The specious reasoning here is to first expand `'2` into "the term it abbreviates." But again, `'2` abbreviates `(BINARY-+ '1 '1)` *only when* `'2` *is used as a term* and `'2` is not used as a term in `''2`.

[2] Technically, since neither of these s-expressions is a formal term, we should say that they both abbreviate the same formal term.

8 Abbreviations for Constant Symbols

Note. The conventions above allow us to write terms such as `(CONS X '(A B))` in which quoted constants appear. Because quoted constants can be quite large it is convenient to be able to give them names. For example, if the symbol `*LST*` were understood to be `'(A B)` then the above term could be written as `(CONS X *LST*)`. This could be accomplished by adding the new abbreviation convention `*LST*` $\Longrightarrow$`'(A B)`. This would make the symbol `*LST*` look like a "global" variable symbol with a fixed value. Such a convention is unambiguous because the symbol `*LST*` is not a variable symbol (its name begins and ends with `#\*`) and so that symbol does not already have a meaning as a formal term. We now make a convention by which the user can add such abbreviations. In particular, the event `(DEFCONST *LST* '(A B))` would add the abbreviation "`*LST*` $\Longrightarrow$`'(A B)`."

Terminology. A symbol v is a *constant symbol* of our language if the first and last character in its name is `#\*` and the package name of the symbol is neither `"KEYWORD"` nor `"LISP"`.

Terminology. A term t is *evaluable* (in an implicit history h) iff there is an s-expression v such that (t = `'`v) is a theorem (in h). We call any such v a *value* of t. Unless the history is inconsistent, there is at most one value for any term.

Note. We can define a subset of the evaluable terms syntactically, namely those terms containing no variables such that every function symbol called is syntactically evaluable in h. A function is syntactically evaluable in a history if it is one of the symbols axiomatized here or was introduced into the history with the definitional principle (cf. Section 14) and every function symbol called in its body is evaluable.

A call-by-value interpreter will compute the value of any syntactically evaluable term (given sufficient stack and memory resources). A defined function fails to be evaluable when it is defined in terms of a constrained (i.e., undefined) function. The implementation of ACL2 can, in addition, determine the value of certain other evaluable terms, namely those for which the call-by-value interpreter never encounters an undefined function.

To prove that there is at least one value for the syntactically evaluable terms, one shows how the axioms of the logic can be used to "compute," i.e., to reduce every evaluable term into a formal term that can be abbreviated by `'`v, for some s-expression v. To prove that there is at most one such constant one must prove that if two quoted s-expressions can be proved equal then the two s-expressions are identical. We do not give the proofs here. However, we illustrate the idea.

When we use an s-expression as an evaluable term we are of course using it as a term, and hence the abbreviation rules for terms apply. So for example, if we say "(LIST 1 2) is an evaluable term" we mean that the formal term it abbreviates is an evaluable term.

Examples. (+ 2 2) is an evaluable term. Note that it abbreviates the formal term (BINARY-+ (BINARY-+ '1 '1) (BINARY-+ '1 '1)). Using the axiom of the associativity of BINARY-+ we can prove this term equal to (BINARY-+ '1 (BINARY-+ '1 (BINARY-+ '1 '1))), which may be written as '4. Thus, the value of the evaluable term (+ 2 2) is 4. Here is another example. (CONS (+ 2 2) 3) is an evaluable term. Its value is '(4 . 3).

Event.

```
(DEFCONST name
          term)
```

Admissibility Requirements.
For this event to be admissible in a history *h*, *name* must be a constant symbol that has not already been assigned a value as an abbreviation in *h* and *term* must be evaluable in *h*.

Syntactic Extension.
If admissible, then add to the abbreviations of *h* the abbreviation *name* $\Longrightarrow$'*v*, where *v* is the value of *term* in *h*.

Axiomatic Extension.
No new axioms are added by this event.

In Appendices B and C we give the definitions of three new constant symbols,

- *COMMON-LISP-SYMBOLS-FROM-MAIN-LISP-PACKAGE*,
- *COMMON-LISP-SPECIALS-AND-CONSTANTS*, and
- *ACL2-EXPORTS*.

Each is a true list of symbols.

9 Axioms

Note. In this draft of the document, there are four changes to the axioms presented in the draft dated "January, 1997." Axiom 10 has been modified so that it additionally declares that < is a Boolean function. Second, Axiom 60 has been modified so that it additionally declares that SYMBOL-NAME returns a STRINGP. Axioms 79.1 and 79.2 have been added, defining the function symbols ATOM and MAKE-CHARACTER-LIST. The latter function symbol is used in Axiom 80 but was left undefined in the previous draft. Finally, Axiom 95 has been modified to say that ACL2-COUNT returns a nonnegative INTEGERP.

Note. We now present the axioms of ACL2. The axioms essentially specify the value of every evaluable term. These axioms are alleged to be consistent with Common Lisp, with two major caveats. First, certain "predicates" in ACL2 are assumed to be Boolean valued while Common Lisp does not require that. We discuss this briefly in Subsection 9.2. Second, Common Lisp functions are only partially defined, i.e., defined on a subset of the possible arguments. ACL2 "completes" the definitions by providing "default" values for arguments of "unexpected type." See Subsection 9.8.

9.1 Basics

Axiom 1.

`T` $\neq$ `NIL`

Axiom 2.

`X` $=$ `Y` $\rightarrow$ `(EQUAL X Y)` $=$ `T`

Axiom 3.

`X` $\neq$ `Y` $\rightarrow$ `(EQUAL X Y)` $=$ `NIL`

Axiom 4.

`X` $=$ `NIL` $\rightarrow$ `(IF X Y Z)` $=$ `Z`

Axiom 5.

`X` $\neq$ `NIL` $\rightarrow$ `(IF X Y Z)` $=$ `Y`.

Axiom 6.

`(NOT P)` $=$ `(IF P NIL T)`

Axiom 7.

`(IMPLIES P Q)` $=$ `(IF P (IF Q T NIL) T)`

Axiom 8.

`(IFF P Q)` $=$ `(AND (IMPLIES P Q) (IMPLIES Q P))`

Abbreviation. When we refer to a term t as a formula, one should read in place of t the formula $t \neq$ `NIL`.

Example. If `P`, `Q`, `F` and `G` are function symbols of the indicated arity, then

```
(IMPLIES (AND (P X) (Q Y))
         (EQUAL (F X Y) (G X Y))),
```

is a term. If that term is used where a formula is expected (e.g., in the allegation that it is an axiom or a theorem), then it is to be read as the formula

```
(IMPLIES (AND (P X) (Q Y))
         (EQUAL (F X Y) (G X Y)))
 ≠
NIL.
```

Given the foregoing axioms and the rules of inference of propositional calculus and equality, the above formula can be shown equivalent to

```
((P X)≠'NIL ∧ (Q Y)≠'NIL)→(F X Y)=(G X Y)
```

which, following the same abbreviation convention, we can write as

```
((P X) ∧ (Q Y)) →  (F X Y) = (G X Y).
```

9.2 Boolean Valued Functions

Axiom 9.

```
(BOOLEANP X)
 =
(IF (EQUAL X T)
    T
    (EQUAL X NIL))
```

Axiom 10.

```
(AND (BOOLEANP (COMPLEX-RATIONALP X))
     (BOOLEANP (RATIONALP X))
     (BOOLEANP (INTEGERP X))
     (BOOLEANP (EQUAL X Y))
     (BOOLEANP (CONSP X))
     (BOOLEANP (SYMBOLP X))
     (BOOLEANP (STRINGP X))
     (BOOLEANP (CHARACTERP X))
     (BOOLEANP (< X Y)))
```

Note. The Common Lisp definition [7] does not specify that these functions are Boolean. Instead, it says that they are "predicates." Then we learn (cf. [7], page 95), "One may think of a predicate as producing a Boolean value, where `nil` stands for *false* and anything else stands for *true*." And finally, "If no better non-`nil` value is available for the purpose of indicating success, by convention the symbol `t` is used as the 'standard' true value." We are unaware of any Common Lisp implementation that does not in fact obey the axiom above. But as the implementation of ACL2 now stands, it does *not* accurately model those (hypothetical) Common Lisp implementations that use non-standard indicators of success for these predicates.

9.3 Disjointness

Axiom 11 (Def).

```
(ACL2-NUMBERP X)
 =
(OR (COMPLEX-RATIONALP X)
    (RATIONALP X))
```

Axiom 12.

```
(AND (IMPLIES (COMPLEX-RATIONALP X) (NOT (RATIONALP X)))
     (IMPLIES (RATIONALP X)         (NOT (COMPLEX-RATIONALP X))))
```

Axiom 13.

```
(IMPLIES (INTEGERP X) (RATIONALP X))
```

Axiom 14.

```
(AND (IMPLIES (ACL2-NUMBERP X) (NOT (CHARACTERP X)))
     (IMPLIES (ACL2-NUMBERP X) (NOT (CONSP X)))
     (IMPLIES (ACL2-NUMBERP X) (NOT (STRINGP X)))
     (IMPLIES (ACL2-NUMBERP X) (NOT (SYMBOLP X)))
     (IMPLIES (CHARACTERP X)   (NOT (ACL2-NUMBERP X)))
     (IMPLIES (CHARACTERP X)   (NOT (CONSP X)))
     (IMPLIES (CHARACTERP X)   (NOT (STRINGP X)))
     (IMPLIES (CHARACTERP X)   (NOT (SYMBOLP X)))
     (IMPLIES (CONSP X)        (NOT (ACL2-NUMBERP X)))
     (IMPLIES (CONSP X)        (NOT (CHARACTERP X)))
     (IMPLIES (CONSP X)        (NOT (STRINGP X)))
     (IMPLIES (CONSP X)        (NOT (SYMBOLP X)))
     (IMPLIES (STRINGP X)      (NOT (ACL2-NUMBERP X)))
     (IMPLIES (STRINGP X)      (NOT (CHARACTERP X)))
     (IMPLIES (STRINGP X)      (NOT (CONSP X)))
     (IMPLIES (STRINGP X)      (NOT (SYMBOLP X)))
     (IMPLIES (SYMBOLP X)      (NOT (ACL2-NUMBERP X)))
     (IMPLIES (SYMBOLP X)      (NOT (CHARACTERP X)))
     (IMPLIES (SYMBOLP X)      (NOT (CONSP X)))
     (IMPLIES (SYMBOLP X)      (NOT (STRINGP X))))
```

9.4 Arithmetic

Axiom 15.

```
(AND (ACL2-NUMBERP (+ X Y))
     (ACL2-NUMBERP (* X Y))
     (ACL2-NUMBERP (- X))
```

```
    (ACL2-NUMBERP (/ X)))
```

Axiom 16.

```
(EQUAL (+ (+ X Y) Z) (+ X (+ Y Z)))
```

Axiom 17.

```
(EQUAL (+ X Y) (+ Y X))
```

Axiom 18 (Def).

```
(FIX X)
 =
(IF (ACL2-NUMBERP X) X 0)
```

Axiom 19.

```
(EQUAL (+ 0 X) (FIX X))
```

Axiom 20.

```
(EQUAL (+ X (- X)) 0)
```

Axiom 21.

```
(EQUAL (* (* X Y) Z) (* X (* Y Z)))
```

Axiom 22.

```
(EQUAL (* X Y) (* Y X))
```

Axiom 23.

```
(EQUAL (* 1 X) (FIX X))
```

Axiom 24.

```
(IMPLIES (AND (ACL2-NUMBERP X)
              (NOT (EQUAL X 0)))
         (EQUAL (* X (/ X)) 1))
```

Axiom 25.

```
(EQUAL (* X (+ Y Z))
       (+ (* X Y) (* X Z)))
```

Axiom 26.

```
(EQUAL (< X Y)
       (< (+ X (- Y)) 0))
```

Axiom 27.

```
(NOT (< 0 0))
```

Axiom 28.

```
(AND
 (IMPLIES (ACL2-NUMBERP X)
          (OR (< 0 X)
              (EQUAL X 0)
```

```
              (< 0 (- X))))
 (OR (NOT (< 0 X))
     (NOT (< 0 (- X)))))
```

Axiom 29.

```
(AND (IMPLIES (AND (< 0 X) (< 0 Y))
              (< 0 (+ X Y)))
     (IMPLIES (AND (RATIONALP X)
                   (RATIONALP Y)
                   (< 0 X)
                   (< 0 Y))
              (< 0 (* X Y))))
```

Axiom 30.

```
(IMPLIES (RATIONALP X)
         (AND (INTEGERP (DENOMINATOR X))
              (INTEGERP (NUMERATOR X))
              (< 0 (DENOMINATOR X))))
```

Axiom 31.

```
(IMPLIES (RATIONALP X)
         (EQUAL (* (NUMERATOR X) (/ (DENOMINATOR X))) X))
```

Axiom 32.

```
(AND (RATIONALP (REALPART X))
     (RATIONALP (IMAGPART X)))
```

Axiom 33.

```
(IMPLIES (AND (RATIONALP X)
              (RATIONALP Y))
         (EQUAL (COMPLEX X Y)
                (+ X (* #C(0 1) Y))))
```

Axiom 34.

```
(EQUAL (* #C(0 1) #C(0 1)) -1)
```

Axiom 35.

```
(IMPLIES (COMPLEX-RATIONALP X)
         (NOT (EQUAL 0 (IMAGPART X))))
```

Axiom 36.

```
(IMPLIES (ACL2-NUMBERP X)
         (EQUAL (COMPLEX (REALPART X) (IMAGPART X)) X))
```

Axiom 37.

```
(IMPLIES (AND (RATIONALP X)
              (RATIONALP Y))
         (EQUAL (REALPART (COMPLEX X Y))
```

```
               X))
```

Axiom 38.

```
(IMPLIES (AND (RATIONALP X)
              (RATIONALP Y))
         (EQUAL (IMAGPART (COMPLEX X Y))
               Y))
```

Axiom 39.

```
(IMPLIES (RATIONALP X)
         (<= 0 (* X X)))
```

Axiom 40.

```
(INTEGERP 0)
```

Axiom 41.

```
(INTEGERP 1)
```

Axiom 42.

```
(< 0 1)
```

Axiom 43.

```
(IMPLIES (INTEGERP X)
         (AND (INTEGERP (+ X 1))
              (INTEGERP (+ X -1))))
```

Axiom 44.

```
(IMPLIES (AND (INTEGERP N)
              (RATIONALP X)
              (INTEGERP R)
              (INTEGERP Q)
              (< 0 N)
              (EQUAL (NUMERATOR X) (* N R))
              (EQUAL (DENOMINATOR X) (* N Q)))
         (EQUAL N 1))
```

9.5 Lists

Axiom 45.

```
(CONSP (CONS X Y))
```

Axiom 46.

```
(IMPLIES (CONSP X)
         (EQUAL (CONS (CAR X) (CDR X)) X))
```

Axiom 47.

```
(EQUAL (CAR (CONS X Y)) X))
```

Axiom 48.

```
(EQUAL (CDR (CONS X Y)) Y)
```

9.6 Characters and Strings

Axiom 49.

```
(AND (INTEGERP (CHAR-CODE X))
     (<= 0 (CHAR-CODE X))
     (< (CHAR-CODE X) 256))
```

Axiom 50.

```
(CHARACTERP (CODE-CHAR N))
```

Axiom 51.

```
(IMPLIES (CHARACTERP C)
         (EQUAL (CODE-CHAR (CHAR-CODE C)) C))
```

Axiom 52.

```
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (< N 256))
         (EQUAL (CHAR-CODE (CODE-CHAR N)) N))
```

Axiom 53 (Def).

```
(CHARACTER-LISTP l)
  =
(IF (CONSP L)
    (AND (CHARACTERP (CAR L))
         (CHARACTER-LISTP (CDR L)))
    (EQUAL L NIL))
```

Axiom 54.

```
(IMPLIES (CHARACTER-LISTP X)
         (EQUAL (COERCE (COERCE X 'STRING) 'LIST) X))
```

Axiom 55.

```
(IMPLIES (STRINGP X)
         (EQUAL (COERCE (COERCE X 'LIST) 'STRING) X))
```

Axiom 56.

```
(STRINGP (COERCE X 'STRING))
```

Axiom 57.

```
(CHARACTER-LISTP (COERCE X 'LIST))
```

9.7 Symbols

Axiom 58.

```
(AND
 (EQUAL 'KEYWORD::WITNESS
        (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESS"
                                     'KEYWORD::WITNESS))
 (EQUAL (SYMBOL-PACKAGE-NAME 'KEYWORD::WITNESS)
        "KEYWORD")
 (EQUAL 'LISP::WITNESS
        (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESS"
                                     'LISP::WITNESS))
 (EQUAL (SYMBOL-PACKAGE-NAME 'LISP::WITNESS)
        "LISP")
 (EQUAL 'ACL2::WITNESS
        (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESS"
                                     'ACL2::WITNESS))
 (EQUAL (SYMBOL-PACKAGE-NAME 'ACL2::WITNESS)
        "ACL2")
 (EQUAL 'ACL2-OUTPUT-CHANNEL::WITNESS
        (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESS"
                                     'ACL2-OUTPUT-CHANNEL::WITNESS))
 (EQUAL (SYMBOL-PACKAGE-NAME 'ACL2-OUTPUT-CHANNEL::WITNESS)
        "ACL2-OUTPUT-CHANNEL")
 (EQUAL 'ACL2-INPUT-CHANNEL::WITNESS
        (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESS"
                                     'ACL2-INPUT-CHANNEL::WITNESS))
 (EQUAL (SYMBOL-PACKAGE-NAME 'ACL2-INPUT-CHANNEL::WITNESS)
        "ACL2-INPUT-CHANNEL")
 (EQUAL 'ACL2-PC::WITNESS
        (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESS"
                                     'ACL2-PC::WITNESS))
 (EQUAL (SYMBOL-PACKAGE-NAME 'ACL2-PC::WITNESS)
        "ACL2-PC")
 (EQUAL 'ACL2-USER::WITNESS
        (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESS"
                                     'ACL2-USER::WITNESS))
 (EQUAL (SYMBOL-PACKAGE-NAME 'ACL2-USER::WITNESS)
        "ACL2-USER"))
```

Axiom 59.

```
(AND
 (EQUAL 'LISP::NIL
        (INTERN-IN-PACKAGE-OF-SYMBOL "NIL"
                                     'LISP::WITNESS))
 (EQUAL 'LISP::STRING
        (INTERN-IN-PACKAGE-OF-SYMBOL "STRING"
```

```
                                     'LISP::WITNESS)))
```

Axiom 60.

```
(AND (STRINGP (SYMBOL-NAME X))
     (STRINGP (SYMBOL-PACKAGE-NAME X)))
```

Axiom 61.

```
(SYMBOLP (INTERN-IN-PACKAGE-OF-SYMBOL X Y))
```

Axiom 62.

```
(IMPLIES (AND (SYMBOLP X)
              (EQUAL (SYMBOL-PACKAGE-NAME X) (SYMBOL-PACKAGE-NAME Y)))
         (EQUAL (INTERN-IN-PACKAGE-OF-SYMBOL (SYMBOL-NAME X) Y) X))
```

Axiom 63.

```
(IMPLIES (AND (STRINGP X)
              (SYMBOLP Y))
         (EQUAL (SYMBOL-NAME (INTERN-IN-PACKAGE-OF-SYMBOL X Y)) X))
```

Axiom 64.

```
(IMPLIES (AND (STRINGP X)
              (SYMBOLP Y)
              (EQUAL (SYMBOL-PACKAGE-NAME Y)
                     "ACL2-INPUT-CHANNEL"))
         (EQUAL (SYMBOL-PACKAGE-NAME (INTERN-IN-PACKAGE-OF-SYMBOL X Y))
                "ACL2-INPUT-CHANNEL"))
```

Axiom 65.

```
(IMPLIES (AND (STRINGP X)
              (SYMBOLP Y)
              (EQUAL (SYMBOL-PACKAGE-NAME Y)
                     "ACL2-OUTPUT-CHANNEL"))
         (EQUAL (SYMBOL-PACKAGE-NAME (INTERN-IN-PACKAGE-OF-SYMBOL X Y))
                "ACL2-OUTPUT-CHANNEL"))
```

Axiom 66 (Def).

```
(MEMBER-SYMBOL-NAME STR L)
  =
(COND ((NOT (CONSP L)) NIL)
      ((EQUAL STR (SYMBOL-NAME (CAR L))) L)
      (T (MEMBER-SYMBOL-NAME STR (CDR L))))
```

Axiom 67.

```
(IMPLIES (AND (STRINGP X)
              (NOT (MEMBER-SYMBOL-NAME
                    X
                    *COMMON-LISP-SYMBOLS-FROM-MAIN-LISP-PACKAGE*))
              (SYMBOLP Y)
```

```
          (EQUAL (SYMBOL-PACKAGE-NAME Y) "ACL2"))
     (EQUAL (SYMBOL-PACKAGE-NAME (INTERN-IN-PACKAGE-OF-SYMBOL X Y))
            "ACL2"))
```

See Appendix B for the definition of the constant symbol *COMMON-LISP--SYMBOLS-FROM-MAIN-LISP-PACKAGE*.

Axiom 68.

```
(IMPLIES (AND (MEMBER-SYMBOL-NAME
               X
               *COMMON-LISP-SYMBOLS-FROM-MAIN-LISP-PACKAGE*)
              (SYMBOLP Y)
              (EQUAL (SYMBOL-PACKAGE-NAME Y) "ACL2"))
         (EQUAL (INTERN-IN-PACKAGE-OF-SYMBOL X Y)
                (CAR (MEMBER-SYMBOL-NAME
                      X
                      *COMMON-LISP-SYMBOLS-FROM-MAIN-LISP-PACKAGE*))))
```

Axiom 69.

```
(IMPLIES (AND (STRINGP X)
              (SYMBOLP Y)
              (EQUAL (SYMBOL-PACKAGE-NAME Y)
                     "KEYWORD"))
         (EQUAL (SYMBOL-PACKAGE-NAME (INTERN-IN-PACKAGE-OF-SYMBOL X Y))
                "KEYWORD"))
```

Axiom 70.

```
(IMPLIES (AND (STRINGP X)
              (SYMBOLP Y)
              (EQUAL (SYMBOL-PACKAGE-NAME Y) "LISP"))
         (EQUAL (SYMBOL-PACKAGE-NAME (INTERN-IN-PACKAGE-OF-SYMBOL X Y))
                "LISP"))
```

9.8 Completions

Note. What is the value of `(+ T 5)`? This is (an abbreviation of) a well-formed term involving only the primitive functions and no variables. Common Lisp does not specify a value for this term, but ACL2 provides one. The first axiom below makes this term proveably equal to `(+ 0 5)` and can be viewed as "coercing" the arguments of `+` to ACL2 numbers by defaulting non-numbers to `0`.

Axiom 71.

```
(EQUAL (+ X Y)
       (IF (ACL2-NUMBERP X)
           (IF (ACL2-NUMBERP Y)
               (+ X Y)
               X)
           (IF (ACL2-NUMBERP Y)
               Y
```

```
              0)))
```

Axiom 72.

```
(EQUAL (* X Y)
       (IF (ACL2-NUMBERP X)
           (IF (ACL2-NUMBERP Y)
               (* X Y)
               0)
           0))
```

Axiom 73.

```
(EQUAL (- X)
       (IF (ACL2-NUMBERP X)
           (- X)
           0))
```

Axiom 74.

```
(EQUAL (/ X)
       (IF (AND (ACL2-NUMBERP X)
                (NOT (EQUAL X 0)))
           (/ X)
           0))
```

Axiom 75.

```
(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)))))))
```

Axiom 76.

```
(EQUAL (CAR X)
       (IF (CONSP X) (CAR X) NIL))
```

Axiom 77.

```
(EQUAL (CDR X)
       (IF (CONSP X) (CDR X) NIL))
```

Axiom 78.

```
(EQUAL (CHAR-CODE X)
       (IF (CHARACTERP X)
           (CHAR-CODE X)
           0))
```

Axiom 79.

```
(EQUAL (CODE-CHAR X)
```

```
       (IF (AND (INTEGERP X)
                (>= X 0)
                (< X 256))
           (CODE-CHAR X)
           (CODE-CHAR 0)))
```

Axiom 80.

```
(EQUAL (COMPLEX X Y)
       (COMPLEX (IF (RATIONALP X) X 0)
                (IF (RATIONALP Y) Y 0)))
```

Axiom 80.1 (Def).

```
(ATOM X)
 =
(NOT (CONSP X))
```

Axiom 80.2 (Def).

```
(MAKE-CHARACTER-LIST X)
 =
(COND ((ATOM X) NIL)
      ((CHARACTERP (CAR X))
       (CONS (CAR X)
             (MAKE-CHARACTER-LIST (CDR X))))
      (T (CONS (CODE-CHAR 0)
               (MAKE-CHARACTER-LIST (CDR X)))))
```

Axiom 81.

```
(EQUAL (COERCE X Y)
       (IF (EQUAL Y 'LIST)
           (IF (STRINGP X)
               (COERCE X 'LIST)
               NIL)
           (COERCE (MAKE-CHARACTER-LIST X) 'STRING)))
```

Axiom 82.

```
(EQUAL (DENOMINATOR X)
       (IF (RATIONALP X)
           (DENOMINATOR X)
           1))
```

Axiom 83.

```
(EQUAL (IMAGPART X)
       (IF (ACL2-NUMBERP X)
           (IMAGPART X)
           0))
```

Axiom 84.

```
(EQUAL (INTERN-IN-PACKAGE-OF-SYMBOL X Y)
```

```
       (IF (AND (STRINGP X)
                (SYMBOLP Y))
           (INTERN-IN-PACKAGE-OF-SYMBOL X Y)
           NIL))
```

Axiom 85.

```
(EQUAL (NUMERATOR X)
       (IF (RATIONALP X)
           (NUMERATOR X)
           0))
```

Axiom 86.

```
(EQUAL (REALPART X)
       (IF (ACL2-NUMBERP X)
           (REALPART X)
           0))
```

Axiom 87.

```
(EQUAL (SYMBOL-NAME X)
       (IF (SYMBOLP X)
           (SYMBOL-NAME X)
           ""))
```

Axiom 88.

```
(EQUAL (SYMBOL-PACKAGE-NAME X)
       (IF (SYMBOLP X)
           (SYMBOL-PACKAGE-NAME X)
           ""))
```

10 The Ordinals

Note. Using the nonnegative integers and lists we can represent the ordinals up to ϵ_0. (For readers familiar with ordinals: ϵ_0 is the first infinite ordinal that is closed under ordinal exponentiation.) The ACL2 notion of ordinal is the same as that found in [1] and both are very similar to the development given in [2]. The following notes are only intended to provide some intuition about ordinals. We ultimately axiomatize two functions below, `E0-ORDINALP` and `E0-ORD-<`, which formalize the concepts.

Very intuitively, think of each non-zero natural number as by being denoted by a series of the appropriate number of strokes, i.e.,

ordinal	
0	
1	|
2	||

3	$\mid\mid\mid$
4	$\mid\mid\mid\mid$
...	...

Then ω is the ordinal that might be written as $\mid\mid\mid$..., i.e., an infinite number of strokes.

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

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

$\omega \times 2$	$\mid\mid\mid$... $\mid\mid\mid$...

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

ω^2	$\mid\mid\mid$... $\mid\mid\mid$... $\mid\mid\mid$... $\mid\mid\mid$... $\mid\mid\mid$... ...

or

ω^2	$\omega\ \omega\ \omega\ \ldots$

We can analogously construct ω^3 by replacing each stroke in ω by ω^2 (which, it turns out, is the same as replacing each stroke in ω^2 by ω). That is, we can construct ω^3 as ω copies of ω^2,

ω^3	$\omega^2\ \omega^2\ \omega^2\ \ldots$

Then we can construct ω^4 as ω copies of ω^3, ω^5 as ω copies of ω^4, etc., ultimately suggesting ω^ω. We can then stack ωs, i.e., ω^{ω^ω}, etc. Consider the "limit" of all of those stacks, which we might display as $\omega^{\omega^{\omega^{\cdots}}}$. That ordinal is called ϵ_0.

It is possible to construct a sequence of s-expressions in 1:1 correspondence with the ordinals up to ϵ_0. In Table 4 we list some of the ordinals up to ϵ_0; the reader can fill in the gaps at his or her leisure. (!) We show in the left column the conventional notation and in the right column the corresponding s-expression.

Each of the s-expressions in the right-hand column of Table 4, when quoted, represents a constant in ACL2 and in that sense the logic contains a representation of each ordinal up to ϵ_0. The function `E0-ORDINALP`, defined below, recognizes these ordinals. Readers familiar with ordinals will find it useful to realize that what we are really doing is mapping s-expressions to ordinals by the map f defined as the identity on atoms and extended to pairs as follows, using "+" to denote ordinal addition: $f((\mathtt{A.B})) = \omega^{f(\mathtt{A})} + f(\mathtt{B})$.

ordinal	s-expression
0	`0`
1	`1`
2	`2`
3	`3`
$\ldots$	$\ldots$
ω	`(1 . 0)`
$\omega + 1$	`(1 . 1)`
$\omega + 2$	`(1 . 2)`
$\ldots$	$\ldots$
$\omega \times 2 = \omega + \omega$	`(1 1 . 0)`
$(\omega \times 2) + 1$	`(1 1 . 1)`
$\ldots$	$\ldots$
$\omega \times 3 = \omega + \omega \times 2$	`(1 1 1 . 0)`
$(\omega \times 3) + 1$	`(1 1 1 . 1)`
$\ldots$	$\ldots$
ω^2	`(2 . 0)`
$\ldots$	$\ldots$
$\omega^2 + \omega \times 4 + 3$	`(2 1 1 1 1 . 3)`
$\ldots$	$\ldots$
ω^3	`(3 . 0)`
$\ldots$	$\ldots$
ω^ω	`((1 . 0) . 0)`
$\ldots$	$\ldots$
$\omega^\omega + \omega^{99} + \omega \times 4 + 3$	`((1 . 0) 99 1 1 1 1 . 3)`
$\ldots$	$\ldots$
ω^{ω^2}	`((2 . 0) . 0)`
$\ldots$	$\ldots$
ω^{ω^ω}	`(((1 . 0) . 0) . 0)`
$\ldots$	$\ldots$

Table 4: Some Ordinals in ACL2

Observe that the sequence of s-expressions starts with the nonnegative integers. That is, the natural numbers are ordinals. Thus, if we require that a given term be proved to produce an ordinal (as we do in the induction principle) then it suffices to prove that the term produces a natural number.

Axiom 89 (Def).

```
(E0-ORD-< X Y)
 =
(IF (CONSP X)
    (IF (CONSP Y)
        (IF (E0-ORD-< (CAR X) (CAR Y))
            T
            (IF (EQUAL (CAR X) (CAR Y))
                (E0-ORD-< (CDR X) (CDR Y))
                NIL))
        NIL)
    (IF (CONSP Y)
        T
        (< (IF (RATIONALP X) X 0)
           (IF (RATIONALP Y) Y 0))))
```

The ordinals in Table 4 are listed in ascending order. This ordering is recognized by the function `E0-ORD-<`, defined above. Fundamental to ACL2 is the fact that `E0-ORD-<` is well-founded on `E0-ORDINALP`s. That is, there is no "infinitely descending chain" of such ordinals.

Axiom 90 (Def).

```
(E0-ORDINALP X)
 =
(IF (CONSP X)
    (AND (E0-ORDINALP (CAR X))
         (NOT (EQUAL (CAR X) 0))
         (E0-ORDINALP (CDR X))
         (OR (NOT (CONSP (CDR X)))
             (NOT (E0-ORD-< (CAR X) (CADR X)))))
    (AND (INTEGERP X) (>= X 0)))
```

10.1 Measures

Axiom 91 (Def).

```
(LEN X)
 =
(IF (CONSP X)
    (+ 1 (LEN (CDR X)))
    0)
```

Axiom 92 (Def).

```
(LENGTH X)
 =
```

```
(IF (STRINGP X)
    (LEN (COERCE X 'LIST))
    (LEN X))
```

Axiom 93 (Def).

```
(INTEGER-ABS X)
 =
(IF (INTEGERP X)
    (IF (< X 0) (- X) X)
    0)
```

Axiom 94 (Def).

```
(ACL2-COUNT X)
 =
(IF (CONSP X)
    (+ 1 (ACL2-COUNT (CAR X))
         (ACL2-COUNT (CDR X)))
    (IF (RATIONALP X)
        (IF (INTEGERP X)
            (INTEGER-ABS X)
            (+ (INTEGER-ABS (NUMERATOR X))
               (DENOMINATOR X)))
        (IF (COMPLEX-RATIONALP X)
            (+ 1 (ACL2-COUNT (REALPART X))
                 (ACL2-COUNT (IMAGPART X)))
            (IF (STRINGP X) (LENGTH X) 0)))))
```

Axiom 95.

```
(AND (INTEGERP (ACL2-COUNT X))
     (<= 0 (ACL2-COUNT X)))
```

11 Induction

Rule of Inference. *Induction*:
Derive p from

- *Base Case*:

  (IMPLIES (AND (NOT q_1) ... (NOT q_k)) p), and

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

  (IMPLIES (AND q_i
  $p/\sigma_{i,1}$
  ...
  $p/\sigma_{i,h_i}$)
  p) ,

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

- *Ordinal Condition*:

  `(EO-ORDINALP` m`)` , and

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

  `(IMPLIES` q_i `(EO-ORD-<` $m/\sigma_{i,j}$ m`))` .

12 Package Definition

Event.

`(DEFPKG` *name term*`)`

Admissibility Requirements.

For this event to be admissible in a history h, *name* must be a string that is not the name of any package in the package system of h and *term* is an evaluable term whose value in h, *imports*, is a true list of symbols such that no two elements of *imports* have the same name.

Syntactic Extension.

If admissible, this event extends the package system of h by adding a package with name *name* and imports list *imports*.

In addition, an admissible `DEFPKG` event introduces the witness symbol for the package as a new formal constant symbol. The witness symbol, *witness*, is specified below.

Let x be the shortest sequence of zero or more exclamation marks (`#\!`) such that the evaluable term `(MEMBER-SYMBOL-NAME "WITNESS`x`"` `'`*imports*`)` has value `NIL` in h. Let *witness* be the symbol `|`*name*`|::WITNESS`x.

Add `'`*witness* to the formal constants of the language.

Axiomatic Extension.

If admissible, add the following three axioms to the axioms of h.

- **Axiom**

```
(AND (EQUAL 'witness
            (INTERN-IN-PACKAGE-OF-SYMBOL "WITNESSx"
                                         'witness))
     (EQUAL (SYMBOL-PACKAGE-NAME 'witness)
            name))
```

- **Axiom**

```
(IMPLIES (AND (STRINGP X)
              (NOT (MEMBER-SYMBOL-NAME X 'imports))
              (SYMBOLP Y)
              (EQUAL (SYMBOL-PACKAGE-NAME Y) name))
         (EQUAL (SYMBOL-PACKAGE-NAME
                 (INTERN-IN-PACKAGE-OF-SYMBOL X Y))
                name))
```

- **Axiom**

```
(IMPLIES (AND (MEMBER-SYMBOL-NAME X 'imports)
              (SYMBOLP Y)
              (EQUAL (SYMBOL-PACKAGE-NAME Y) name))
         (EQUAL (INTERN-IN-PACKAGE-OF-SYMBOL X Y)
                (CAR (MEMBER-SYMBOL-NAME X name))))
```

13 Current Package Selection

Event.
`(IN-PACKAGE` *name*`)`

Admissibility Requirements.
For this event to be admissible in a history h, *name* must be a string that is currently one of the package names in the package system of h.

Syntactic Extension.
By virtue of this event being in the new history, the current package of that history will (by our definition of "current package") be *name* (until another `IN-PACKAGE` event is admitted).

Axiomatic Extension.
No new axioms are added by this event.

14 Function Definition

Terminology. We say that a term t *governs* an occurrence of a term s in a term b iff either (a) b contains a subterm of the form `(IF` t p q`)` and the occurrence of s is in p or (b) b contains a subterm of the form `(IF` t' p q`)`, where t is `(NOT` t'`)` and the occurrence of s is in q.

Examples. The terms `P` and `(NOT Q)` govern the first occurrence of `S` in

```
(IF P
    (IF (IF Q A S)
        S
        B)
    C)
```

The terms `P` and `(IF Q A S)` govern the second occurrence of `S`.

Note. The mechanization of the logic is slightly more restrictive because it only inspects the "top-level" `IF`s in b. Thus, the mechanization recognizes that `P` governs `S` in `(IF P (FN (IF Q S A)) B)` but it does not recognize that `Q` governs `S` also. The implementation does this because the mechanical theorem prover's induction heuristic derives "induction schemas" from recursive definitions and then manipulates these schemas. By keeping the schemas simple (sometimes at the expense of forcing the user to rearrange definitions) we find the heuristics are more often successful at choosing an appropriate induction.

Event.

`(DEFUN` f `(`x_1 ... x_n`)`
    body`)`

Admissibility Requirements.
For this event to be admissible,

- f must be a new function symbol in h,
- the x_i must be distinct variable symbols,
- *body* must be a term in the history obtained from h by adding an entry to the arity table of h declaring f to have arity n, and *body* must mention no symbol as a variable other than the x_i; and
- there is a term m such that
  - `(E0-ORDINALP` m`)` can be proved directly in h, and
  - for each occurence of a subterm of the form `(`f y_1 ... y_n`)` in *body*, the following formula can be proved directly in h:

    `(IMPLIES (AND` t_1 ... t_k`)`
        `(E0-ORD-LESSP` m/σ m`))`

    where the terms t_1, ..., t_k govern the occurrence in question and σ is the substitution $\{\langle x_1, y_1\rangle ... \langle x_n, y_n\rangle\}$.

Syntactic Extension.
If admissible, add a new entry to the arity table of the history. The new entry gives f arity n.

Axiomatic Extension.
If admissible, this event adds the following axiom to the axioms of the history.
Axiom

$(f\ x_1\ \ldots\ x_n)\ =\ body.$

15 Constrained Functions

This section is not yet written.

A The Initial Package System

Below, for each initial package, we specify the symbols imported.

- `"KEYWORD"`: No symbols are imported into this package.
- `"LISP"`: No symbols are imported into this package.
- `"ACL2"`: The symbols listed in Appendix B are imported into this package. All of those symbols have package name `"LISP"`.
- `"ACL2-OUTPUT-CHANNEL"`: No symbols are imported into this package.
- `"ACL2-INPUT-CHANNEL"`: No symbols are imported into this package.
- `"ACL2-PC"`: No symbols are imported into this package.
- `"ACL2-USER"`: The symbols listed in Appendix B together with the symbols listed in Appendix C are imported into this package.

The *witness symbol* of a package p in a package system is the symbol whose package name is p and whose name is shortest string s of the form `WITNESS`x where x is a sequence of zero or more exclamation points such that no symbol with name s is imported into p in the given package system.

For each initial package p shown above, the witness symbol for p is p`::WITNESS`. If a package, say `"PKG"` imported both, say, `LISP::WITNESS` and `ACL2::WITNESS!` (and no others), then the witness symbol `"PKG"` would be `PKG::WITNESS!!`. In fact, we do not care what the name of the witness symbol is, as long as for every package we can write at least one symbol whose package name is that package. Since only a finite number of symbols are imported into every package, there is always such a symbol.

B Common Lisp Symbols

In this appendix we list some symbols. All of the symbols listed are to be understood to have package name `"LISP"`. These symbols are all imported into the `"ACL2"` package. That is, to determine whether a symbol we might print as `ACL2::`*name* actually stands for `LISP::`*name*, determine whether *name* occurs as one of the symbols below.

To make it easier to find a given *name* we list the symbols in alphabetical order and group them according to their first letter.

`*COMMON-LISP-SYMBOLS-FROM-MAIN-LISP-PACKAGE*` abbreviates a quoted list containing just these symbols.

Finally, some of the symbols have a parenthesized asterisk (*) after them. `*COMMON-LISP-SPECIALS-AND-CONSTANTS*` abbreviates a quoted list containing just the so marked symbols.

Symbols starting with signs

&ALLOW-OTHER-KEYS
&AUX
&BODY
&ENVIRONMENT
&KEY
&OPTIONAL
&REST
&WHOLE
* (*)
** (*)
*** (*)
BREAK-ON-SIGNALS (*)
COMPILE-FILE-PATHNAME (*)
COMPILE-FILE-TRUENAME (*)
COMPILE-PRINT (*)
COMPILE-VERBOSE (*)
DEBUG-IO (*)
DEBUGGER-HOOK (*)
DEFAULT-PATHNAME-DEFAULTS (*)
ERROR-OUTPUT (*)
FEATURES (*)
GENSYM-COUNTER (*)
LOAD-PATHNAME (*)
LOAD-PRINT (*)
LOAD-TRUENAME (*)
LOAD-VERBOSE (*)
MACROEXPAND-HOOK (*)
MODULES (*)
PACKAGE (*)
PRINT-ARRAY (*)
PRINT-BASE (*)
PRINT-CASE (*)
PRINT-CIRCLE (*)
PRINT-ESCAPE (*)
PRINT-GENSYM (*)
PRINT-LENGTH (*)
PRINT-LEVEL (*)
PRINT-LINES (*)
PRINT-MISER-WIDTH (*)
PRINT-PPRINT-DISPATCH (*)
PRINT-PRETTY (*)
PRINT-RADIX (*)
PRINT-READABLY (*)
PRINT-RIGHT-MARGIN (*)
QUERY-IO (*)
RANDOM-STATE (*)
READ-BASE (*)
READ-DEFAULT-FLOAT-FORMAT (*)
READ-EVAL (*)
READ-SUPPRESS (*)
READTABLE (*)
STANDARD-INPUT (*)
STANDARD-OUTPUT (*)
TERMINAL-IO (*)
TRACE-OUTPUT (*)
+ (*)
++ (*)
+++ (*)
- (*)
/ (*)
// (*)
/// (*)
/=
1+
1-
<
<=
=
>
>=

A

ABORT
ABS
ACONS
ACOS
ACOSH
ADD-METHOD
ADJOIN
ADJUST-ARRAY
ADJUSTABLE-ARRAY-P
ALLOCATE-INSTANCE
ALPHA-CHAR-P
ALPHANUMERICP
AND
APPEND
APPLY

CADADR
CADAR
CADDAR
CADDDR
CADDR
CADR
CALL-ARGUMENTS-LIMIT (*)
CALL-METHOD
CALL-NEXT-METHOD
CAR
CASE
CATCH
CCASE
CDAAAR
CDAADR
CDAAR
CDADAR
CDADDR
CDADR
CDAR
CDDAAR
CDDADR
CDDAR
CDDDAR
CDDDDR
CDDDR
CDDR
CDR
CEILING
CELL-ERROR
CELL-ERROR-NAME
CERROR
CHANGE-CLASS
CHAR
CHAR-CODE
CHAR-CODE-LIMIT (*)
CHAR-DOWNCASE
CHAR-EQUAL
CHAR-GREATERP
CHAR-INT
CHAR-LESSP
CHAR-NAME
CHAR-NOT-EQUAL
CHAR-NOT-GREATERP
CHAR-NOT-LESSP
CHAR-UPCASE
CHAR/=
CHAR<
CHAR<=
CHAR=
CHAR>
CHAR>=
CHARACTER
CHARACTERP
CHECK-TYPE
CIS
CLASS
CLASS-NAME
CLASS-OF
CLEAR-INPUT
CLEAR-OUTPUT
CLOSE
CLRHASH
CODE-CHAR
COERCE
COMPILATION-SPEED
COMPILE
COMPILE-FILE
COMPILE-FILE-PATHNAME
COMPILED-FUNCTION
COMPILED-FUNCTION-P
COMPILER-MACRO
COMPILER-MACRO-FUNCTION
COMPLEMENT
COMPLEX
COMPLEXP
COMPUTE-APPLICABLE-METHODS
COMPUTE-RESTARTS
CONCATENATE
CONCATENATED-STREAM
CONCATENATED-STREAM-STREAMS
COND
CONDITION
CONJUGATE
CONS
CONSP
CONSTANTLY
CONSTANTP

CONTINUE
CONTROL-ERROR
COPY-ALIST
COPY-LIST
COPY-PPRINT-DISPATCH
COPY-READTABLE
COPY-SEQ
COPY-STRUCTURE
COPY-SYMBOL
COPY-TREE
COS
COSH
COUNT
COUNT-IF
COUNT-IF-NOT
CTYPECASE

D

DEBUG
DECF
DECLAIM
DECLARATION
DECLARE
DECODE-FLOAT
DECODE-UNIVERSAL-TIME
DEFCLASS
DEFCONSTANT
DEFGENERIC
DEFINE-COMPILER-MACRO
DEFINE-CONDITION
DEFINE-METHOD-COMBINATION
DEFINE-MODIFY-MACRO
DEFINE-SETF-EXPANDER
DEFINE-SYMBOL-MACRO
DEFMACRO
DEFMETHOD
DEFPACKAGE
DEFPARAMETER
DEFSETF
DEFSTRUCT
DEFTYPE
DEFUN
DEFVAR
DELETE
DELETE-DUPLICATES
DELETE-FILE
DELETE-IF
DELETE-IF-NOT
DELETE-PACKAGE
DENOMINATOR
DEPOSIT-FIELD
DESCRIBE
DESCRIBE-OBJECT
DESTRUCTURING-BIND
DIGIT-CHAR
DIGIT-CHAR-P
DIRECTORY
DIRECTORY-NAMESTRING
DISASSEMBLE
DIVISION-BY-ZERO
DO
DO*
DO-ALL-SYMBOLS
DO-EXTERNAL-SYMBOLS
DO-SYMBOLS
DOCUMENTATION
DOLIST
DOTIMES
DOUBLE-FLOAT
DOUBLE-FLOAT-EPSILON (*)
DOUBLE-FLOAT-NEGATIVE-EPSILON (*)
DPB
DRIBBLE
DYNAMIC-EXTENT

E

ECASE
ECHO-STREAM
ECHO-STREAM-INPUT-STREAM
ECHO-STREAM-OUTPUT-STREAM
ED
EIGHTH
ELT
ENCODE-UNIVERSAL-TIME
END-OF-FILE
ENDP
ENOUGH-NAMESTRING
ENSURE-DIRECTORIES-EXIST

GETHASH
GO
GRAPHIC-CHAR-P

H

HANDLER-BIND
HANDLER-CASE
HASH-TABLE
HASH-TABLE-COUNT
HASH-TABLE-P
HASH-TABLE-REHASH-SIZE
HASH-TABLE-REHASH-THRESHOLD
HASH-TABLE-SIZE
HASH-TABLE-TEST
HOST-NAMESTRING

I

IDENTITY
IF
IGNORABLE
IGNORE
IGNORE-ERRORS
IMAGPART
IMPORT
IN-PACKAGE
INCF
INITIALIZE-INSTANCE
INLINE
INPUT-STREAM-P
INSPECT
INTEGER
INTEGER-DECODE-FLOAT
INTEGER-LENGTH
INTEGERP
INTERACTIVE-STREAM-P
INTERN
INTERNAL-TIME-UNITS-PER-SECOND (*)
INTERSECTION
INVALID-METHOD-ERROR
INVOKE-DEBUGGER
INVOKE-RESTART
INVOKE-RESTART-INTERACTIVELY
ISQRT

K

KEYWORD
KEYWORDP

L

LABELS
LAMBDA
LAMBDA-LIST-KEYWORDS (*)
LAMBDA-PARAMETERS-LIMIT (*)
LAST
LCM
LDB
LDB-TEST
LDIFF
LEAST-NEGATIVE--
DOUBLE-FLOAT (*)
LEAST-NEGATIVE--
LONG-FLOAT (*)
LEAST-NEGATIVE--
NORMALIZED-DOUBLE-FLOAT (*)
LEAST-NEGATIVE--
NORMALIZED-LONG-FLOAT (*)
LEAST-NEGATIVE--
NORMALIZED-SHORT-FLOAT (*)
LEAST-NEGATIVE--
NORMALIZED-SINGLE-FLOAT (*)
LEAST-NEGATIVE--
SHORT-FLOAT (*)
LEAST-NEGATIVE--
SINGLE-FLOAT (*)
LEAST-POSITIVE--
DOUBLE-FLOAT (*)
LEAST-POSITIVE--
LONG-FLOAT (*)
LEAST-POSITIVE--
NORMALIZED-DOUBLE-FLOAT (*)
LEAST-POSITIVE--
NORMALIZED-LONG-FLOAT (*)
LEAST-POSITIVE--
NORMALIZED-SHORT-FLOAT (*)
LEAST-POSITIVE--
NORMALIZED-SINGLE-FLOAT (*)
LEAST-POSITIVE--
SHORT-FLOAT (*)

LEAST-POSITIVE--
    SINGLE-FLOAT (*)
LENGTH
LET
LET*
LISP-IMPLEMENTATION-TYPE
LISP-IMPLEMENTATION-VERSION
LIST
LIST*
LIST-ALL-PACKAGES
LIST-LENGTH
LISTEN
LISTP
LOAD
LOAD-LOGICAL-PATHNAME-TRANSLATIONS
LOAD-TIME-VALUE
LOCALLY
LOG
LOGAND
LOGANDC1
LOGANDC2
LOGBITP
LOGCOUNT
LOGEQV
LOGICAL-PATHNAME
LOGICAL-PATHNAME-TRANSLATIONS
LOGIOR
LOGNAND
LOGNOR
LOGNOT
LOGORC1
LOGORC2
LOGTEST
LOGXOR
LONG-FLOAT
LONG-FLOAT-EPSILON (*)
LONG-FLOAT-NEGATIVE-EPSILON (*)
LONG-SITE-NAME
LOOP
LOOP-FINISH
LOWER-CASE-P

M

MACHINE-INSTANCE
MACHINE-TYPE
MACHINE-VERSION
MACRO-FUNCTION
MACROEXPAND
MACROEXPAND-1
MACROLET
MAKE-ARRAY
MAKE-BROADCAST-STREAM
MAKE-CONCATENATED-STREAM
MAKE-CONDITION
MAKE-DISPATCH-MACRO-CHARACTER
MAKE-ECHO-STREAM
MAKE-HASH-TABLE
MAKE-INSTANCE
MAKE-INSTANCES-OBSOLETE
MAKE-LIST
MAKE-LOAD-FORM
MAKE-LOAD-FORM-SAVING-SLOTS
MAKE-METHOD
MAKE-PACKAGE
MAKE-PATHNAME
MAKE-RANDOM-STATE
MAKE-SEQUENCE
MAKE-STRING
MAKE-STRING-INPUT-STREAM
MAKE-STRING-OUTPUT-STREAM
MAKE-SYMBOL
MAKE-SYNONYM-STREAM
MAKE-TWO-WAY-STREAM
MAKUNBOUND
MAP
MAP-INTO
MAPC
MAPCAN
MAPCAR
MAPCON
MAPHASH
MAPL
MAPLIST
MASK-FIELD
MAX
MEMBER
MEMBER-IF
MEMBER-IF-NOT

N

O

P

Q

R

REMOVE
REMOVE-DUPLICATES
REMOVE-IF
REMOVE-IF-NOT
REMOVE-METHOD
REMPROP
RENAME-FILE
RENAME-PACKAGE
REPLACE
REQUIRE
REST
RESTART
RESTART-BIND
RESTART-CASE
RESTART-NAME
RETURN
RETURN-FROM
REVAPPEND
REVERSE
ROOM
ROTATEF
ROUND
ROW-MAJOR-AREF
RPLACA
RPLACD

S

SAFETY
SATISFIES
SBIT
SCALE-FLOAT
SCHAR
SEARCH
SECOND
SEQUENCE
SERIOUS-CONDITION
SET
SET-DIFFERENCE
SET-DISPATCH-MACRO-CHARACTER
SET-EXCLUSIVE-OR
SET-MACRO-CHARACTER
SET-PPRINT-DISPATCH
SET-SYNTAX-FROM-CHAR
SETF
SETQ
SEVENTH
SHADOW
SHADOWING-IMPORT
SHARED-INITIALIZE
SHIFTF
SHORT-FLOAT
SHORT-FLOAT-EPSILON (*)
SHORT-FLOAT-NEGATIVE-EPSILON (*)
SHORT-SITE-NAME
SIGNAL
SIGNED-BYTE
SIGNUM
SIMPLE-ARRAY
SIMPLE-BASE-STRING
SIMPLE-BIT-VECTOR
SIMPLE-BIT-VECTOR-P
SIMPLE-CONDITION
SIMPLE-CONDITION-FORMAT-ARGUMENTS
SIMPLE-CONDITION-FORMAT-CONTROL
SIMPLE-ERROR
SIMPLE-STRING
SIMPLE-STRING-P
SIMPLE-TYPE-ERROR
SIMPLE-VECTOR
SIMPLE-VECTOR-P
SIMPLE-WARNING
SIN
SINGLE-FLOAT
SINGLE-FLOAT-EPSILON (*)
SINGLE-FLOAT-NEGATIVE-EPSILON (*)
SINH
SIXTH
SLEEP
SLOT-BOUNDP
SLOT-EXISTS-P
SLOT-MAKUNBOUND
SLOT-MISSING
SLOT-UNBOUND
SLOT-VALUE
SOFTWARE-TYPE
SOFTWARE-VERSION
SOME
SORT

SPACE
SPECIAL
SPECIAL-OPERATOR-P
SPEED
SQRT
STABLE-SORT
STANDARD
STANDARD-CHAR
STANDARD-CHAR-P
STANDARD-CLASS
STANDARD-GENERIC-FUNCTION
STANDARD-METHOD
STANDARD-OBJECT
STEP
STORAGE-CONDITION
STORE-VALUE
STREAM
STREAM-ELEMENT-TYPE
STREAM-ERROR
STREAM-ERROR-STREAM
STREAM-EXTERNAL-FORMAT
STREAMP
STRING
STRING-CAPITALIZE
STRING-DOWNCASE
STRING-EQUAL
STRING-GREATERP
STRING-LEFT-TRIM
STRING-LESSP
STRING-NOT-EQUAL
STRING-NOT-GREATERP
STRING-NOT-LESSP
STRING-RIGHT-TRIM
STRING-STREAM
STRING-TRIM
STRING-UPCASE
STRING/=
STRING<
STRING<=
STRING=
STRING>
STRING>=
STRINGP
STRUCTURE
STRUCTURE-CLASS
STRUCTURE-OBJECT
STYLE-WARNING
SUBLIS
SUBSEQ
SUBSETP
SUBST
SUBST-IF
SUBST-IF-NOT
SUBSTITUTE
SUBSTITUTE-IF
SUBSTITUTE-IF-NOT
SUBTYPEP
SVREF
SXHASH
SYMBOL
SYMBOL-FUNCTION
SYMBOL-MACROLET
SYMBOL-NAME
SYMBOL-PACKAGE
SYMBOL-PLIST
SYMBOL-VALUE
SYMBOLP
SYNONYM-STREAM
SYNONYM-STREAM-SYMBOL

T

T (*)
TAGBODY
TAILP
TAN
TANH
TENTH
TERPRI
THE
THIRD
THROW
TIME
TRACE
TRANSLATE-LOGICAL-PATHNAME
TRANSLATE-PATHNAME
TREE-EQUAL
TRUENAME
TRUNCATE

U

V

W

Y

Z

C ACL2 Exports

The ACL2 constant symbol `*ACL2-EXPORTS*` has as its value a list of symbols that we believe most users will find convenient to import into other packages so that when those other packages are "current" the `"ACL2"` package prefix need not be typed for common ACL2 events. The symbols listed below are those in `*ACL2-EXPORTS*`

```
@                             IMPLIES
ACL2-COUNT                    IN-THEORY
ASSIGN                        INCLUDE-BOOK
ASSUME                        INTERSECTION-THEORIES
CERTIFY-BOOK                  LD
CURRENT-THEORY                LOCAL
DECLARE                       MUTUAL-RECURSION
DEFAXIOM                      MV
DEFCONST                      MV-LET
DEFDOC                        MV-NTH
DEFINE-PC-ATOMIC-MACRO        PROVE
DEFINE-PC-MACRO               RETRIEVE
DEFLABEL                      SET-DIFFERENCE-THEORIES
DEFMACRO                      STATE
DEFPKG                        TABLE
DEFTHEORY                     THEORY
DEFTHM                        THM
DEFUN                         TOGGLE-PC-MACRO
DEFUNS                        UBT
DISABLE                       UNION-THEORIES
ENABLE                        UNIVERSAL-THEORY
ENCAPSULATE                   VERIFY
EXECUTABLE-COUNTERPART-THEORY VERIFY-GUARDS
FORCE                         VERIFY-TERMINATION
FUNCTION-THEORY               XARGS
IFF
```

References

[1] R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press: New York, 1988.

[2] G. Gentzen, "New Version of the Consistency Proof for Elementary Number Theory" in M. E. Szabo, ed., *The Collected Papers of Gerhard Gentzen*, North-Holland Publishing Company: Amsterdam, 1969, pp. 132–213.

[3] M. Kaufmann and J S. Moore, "Design Goals of ACL2," Technical Report 101, Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, TX 78703, 1994.

[4] M. Kaufmann and J S. Moore. ACL2: An Industrial Strength Version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance* (COMPASS-96), IEEE Computer Society Press, June, 1996, pp. 23–34.

[5] J. R. Shoenfield, *Mathematical Logic*, Addison-Wesley: Reading, MA, 1967.

[6] G. L. Steele, Jr. *Common LISP: The Language*, Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

[7] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.