A Precise Description of a Computational Logic

Notes for CS 389R

Robert S. Boyer and J Strother Moore

1. Apologia

Traditionally, logicians keep their formal systems as simple as possible. This is desirable because logicians rarely use the formal systems themselves. Instead, they stay in the metatheory and (informally) prove results about their systems.

The system described here is intended to be used to prove interesting theorems. Furthermore, we want to offer substantial mechanical aid in constructing proofs, as a means of eliminating errors. These pragmatic considerations have greatly influenced the design of the logic itself.

The set of variable and function symbols is influenced by conventions of half a dozen different Lisp systems. The set of axioms is unnecessarily large from the purely logical viewpoint. For example, it includes axioms for both the natural numbers and ordered pairs as distinct classes of objects. We found certain formalization problems cumbersome when one of these classes is embedded in the other, as is common in less pragmatically motivated logics. Furthermore, the distinction between the two classes makes it easier for us to provide mechanical assistance appropriate to the particular domain. Similarly, a general purpose quantifier over finite domains is provided, even though recursively defined functions suffice. We provide an interpreter for the logic in the logic—at the cost of complicating both the notation for constants and the axioms to set up the necessary correspondence between objects in the logic and the term structure of the language— so that certain useful forms of metatheoretic reasoning can be carried out in the logic. Our induction principle is very complicated in comparison to those found in many other logics; but it is designed to be directly applicable to many problems and to produce simple proofs.

To achieve our goal of providing assistance in the proof of interesting theorems it must be possible, indeed, convenient, to *state* interesting theorems. This requires that we allow the user to extend the logic to introduce new objects and concepts. Logically speaking, the main theorem and all the intermediate lemmas are proved in the single final theory. But practically speaking, the theory ‘‘evolves’’ over time as the user repeatedly extends it and derives intermediate results. We provide three ‘‘extension’’ principles—the ‘‘shell principle’’ for introducing new, inductively-defined data types, the ‘‘definitional

principle'' for defining new recursive functions, and the ''constraint principle'' for introducing functions that are undefined but constrained to have certain properties. Our extension principles, while complicated, are designed to be sound, easy to use and to mechanize, and helpful in guiding the discovery of proofs.

While the logic is complicated compared to most mathematical logics, it is simple compared to most programming languages and many specification languages. If our presentation of it makes it seem ''too complicated'' it is perhaps merely that we are presenting all of the details.

2. Outline of the Presentation

In presenting our logic we follow the well-established tradition of incremental extension. We begin by defining a very simple syntax, called the *formal syntax*, of the language. A much richer syntax, called the *extended syntax*, which contains succinct abbreviations for constants such as numbers, lists, and trees, is defined only after we have axiomatized the primitive data types of the logic.

Using the formal syntax we present the axioms and rules of inference for propositional calculus with equality, the foundation of our theory. Next we embed propositional calculus and equality in the term structure of the logic by defining functional analogues of the propositional operators. We then present the shell principle and use it to add the axioms for natural numbers, ordered pairs, literal atoms, and negative integers.

At this point we have enough formal machinery to explain and illustrate the extended formal syntax.

We then present our formalization of the ordinals up to ε_0. The ''less-than'' relation on these ordinals plays a crucial role in our principles of mathematical induction and recursive definition.

Next we add axioms defining many useful functions.

Then we embed the semantics of the theory in the theory by axiomatizing an interpreter for the logic as a function. In order to do this it is necessary to set up a correspondence between the terms in the formal syntax and certain constants in the logic, called the ''quotations'' of those terms. Roughly speaking, the quotation of a term is a constant in the logic whose value under the interpreter is equal to the term.

We complete the set of axioms by defining our general purpose quantifier function, which, much like the ''mapping'' functions of Lisp, includes among its arguments objects denoting terms which are evaluated with the interpreter.

Finally, we state the principles of inductive proof and recursive definition.

We frequently pause during the presentation to illustrate the concepts discussed. However, we do not attempt to motivate the development or explain

‘‘how’’ certain functions ‘‘work’’ or the role they play in the subsequent development.

We classify our remarks into eight categories:

- **Terminology**: In paragraphs with this label we define syntactic notions that let us state our axioms or syntactic conventions precisely. The concept defined is *italicized*.
- **Abbreviation**: In paragraphs with this label we extend the previously agreed upon syntax by explaining how some string of characters is, henceforth, to be taken as shorthand for another.
- **Example**: We illustrate most of the terminology and abbreviations in paragraphs with this label. Technically, these paragraphs contain no new information, but they serve as a way for the reader to check his understanding.
- **Axiom** or **Defining Axiom**: A formula so labelled is an axiom of our system. Axioms of the latter sort are distinguished because they uniquely define a function.
- **Shell Definition**: A paragraph so labelled schematically specifies a set of axioms of our system.
- **Extension Principle**: A paragraph so labelled describes a principle which permits the sound introduction of new function symbols and axioms.
- **Rule of Inference**: A paragraph so labelled describes a rule of inference of our system.
- **Note**: Assorted remarks, such as alternative views, are collected in paragraphs with this label.

3. Formal Syntax

Terminology. A finite sequence of characters, s, is a *symbol* if and only if s is nonempty, each character in s is a member of the set

```
{A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z
 0 1 2 3 4 5 6 7 8 9
 $ ^ & * _ - + = ~ { } ? < >},
```

and the first character of s is a letter, i.e., in the set

```
{A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z}.
```

Examples. **PLUS**, **ADD1**, and **PRIME-FACTORS** are symbols. ***1*TRUE**, **123**, **A/B**, and **#.FOO** are not symbols.

Terminology. The *variable symbols* and *function symbols* of our language are the symbols other than **T**, **F**, and **NIL**.

Terminology. Associated with every function symbol is a nonnegative 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 function symbol in the Ground Zero logic is given in the table below. We also include brief descriptive comments in the hopes that they will make subsequent examples more meaningful.

Table 4.1

symbol	arity	comment
ADD1	**1**	successor function for natural numbers
ADD-TO-SET	**2**	adds an element to a list if not present
AND	**2**	logical and
APPEND	**2**	list concatenation
APPLY-SUBR	**2**	application of primitive fn to arguments
APPLY$	**2**	application of fn to arguments
ASSOC	**2**	association list lookup
BODY	**1**	body of a fn definition
CAR	**1**	first component of ordered pair
CDR	**1**	second component of ordered pair
CONS	**2**	constructs ordered pairs
COUNT	**1**	size of an object
DIFFERENCE	**2**	natural difference of two natural numbers
EQUAL	**2**	equality predicate
EVAL$	**3**	interpreter for the logic
FALSE	**0**	false object
FALSEP	**1**	predicate for recognizing **FALSE**
FIX	**1**	coerces argument to **0** if not numeric
FIX-COST	**2**	increments cost if argument is non-**F**
FOR	**6**	general purpose quantifier
FORMALS	**1**	list of formal arguments of a function
GEQ	**2**	greater than or equal on natural numbers
GREATERP	**2**	greater than on natural numbers
IDENTITY	**1**	identity function
IF	**3**	if-then-else
IFF	**2**	if and only if
IMPLIES	**2**	logical implication

LEQ	**2**	less than or equal on natural numbers
LESSP	**2**	less than on natural numbers
LISTP	**1**	recognizes ordered pairs
LITATOM	**1**	recognizes literal atoms
MAX	**2**	maximum of two natural numbers
MEMBER	**2**	membership predicate
MINUS	**1**	constructs negative of a natural number
NEGATIVEP	**1**	recognizes negatives
NEGATIVE-GUTS	**1**	absolute value of a negative
NLISTP	**1**	negation of **LISTP**
NOT	**1**	logical negation
NUMBERP	**1**	recognizes natural numbers
OR	**2**	logical or
ORDINALP	**1**	recognizes ordinals
ORD-LESSP	**2**	less than on ordinals up to ε_0
PACK	**1**	constructs a **LITATOM** from ASCII codes
PAIRLIST	**2**	pairs corresponding elements
PLUS	**2**	sum of two natural numbers
QUANTIFIER-INITIAL-VALUE	**1**	initial value of a quantifier
QUANTIFIER-OPERATION	**3**	operation performed by quantifier
QUOTIENT	**2**	natural quotient of two natural numbers
REMAINDER	**2**	mod
STRIP-CARS	**1**	list of **CAR**s of argument list
SUB1	**1**	predecessor function on natural numbers
SUBRP	**1**	recognizes primitive function symbols
SUM-CDRS	**1**	sum of **CDR**s of elements of argument list
TIMES	**2**	product of two natural numbers
TRUE	**0**	true object
TRUEP	**1**	recognizes **TRUE**
UNION	**2**	union of two lists
UNPACK	**1**	explodes a **LITATOM** into ASCII codes
V&C$	**3**	determines value and cost of an expr
V&C-APPLY$	**2**	determines value and cost of fn application
ZERO	**0**	0
ZEROP	**1**	recognizes 0 and nonnatural numbers

The arity of each user-introduced function symbol is declared when the symbol is first used as a function symbol.

Terminology. A *term* is either a variable symbol or else is a sequence consisting of a function symbol of arity n followed by n terms.

Note. Observe that we have defined a term as a tree structure rather than a character sequence. Of interest is how we display such trees.

Terminology. To *display* a symbol, we merely write down the characters in it. To *display* a term that is a variable symbol, we display the symbol. To *display* a non-variable term with function symbol **fn** and argument terms $\mathbf{t_1}$, ..., $\mathbf{t_n}$, we write down an open parenthesis, a display of **fn**, a nonempty string of spaces and/or carriage returns, a display of $\mathbf{t_1}$, a nonempty string of spaces and/or carriage returns, ..., a display of $\mathbf{t_n}$ and a close parenthesis.

Examples. The following are (displays of) terms:

```
(ZERO)

(ADD1 X)

(PLUS (ADD1 X) (ZERO))

(IF B
    (ZERO)
    (ADD1 X))

(IF B (ZERO) (ADD1 X))
```

Terminology. Our axioms are presented as formulas in propositional calculus with equality. The formulas are constructed from terms, as defined above, using the equality symbol and the symbols for ''or'' and ''not''. More precisely, an *atomic formula* is any string of the form **t1=t2**, where **t1** and **t2** are terms. A *formula* is either an atomic formula, or else of the form $\neg$**(form1)**, where **form1** is a formula, or else of the form **(form1** $\vee$ **form2)**, where **form1** and **form2** are both formulas. Parentheses are omitted when no ambiguity arises.

Abbreviations. We abbreviate $\neg$**(t1 = t2)** by **(t1**$\neq$**t2)**. If **form1** and **form2** are formulas then **(form1** $\rightarrow$ **form2)** is an abbreviation for **(**$\neg$**(form1)** $\vee$ **form2)** and **(form1** $\wedge$ **form2)** is an abbreviation for $\neg$**(**$\neg$**(form1)** $\vee$ $\neg$**(form2))**.

3.1. Terminology about Terms

Terminology. To talk about terms, it is convenient to use so-called ‘‘metavariables’’ that are understood by the reader to stand for certain variables, function symbols, or terms. In this presentation of the logic we use lower case words to denote metavariables.

Example. If **f** denotes the function symbol **PLUS**, and **t** denotes the term **(ADD1 Y)**, then **(f t X)** denotes the term **(PLUS (ADD1 Y) X)**.

Terminology. If **i** is a nonnegative integer, then we let **Xi** denote the variable symbol whose first character is **X** and whose other characters are the decimal representation of **i**.

Example. If **i** is **4**, **Xi** is the variable symbol **X4**.

Terminology. A term **t** is a *call* of **fn** with *arguments* $\mathbf{a_1}$, ..., $\mathbf{a_n}$ iff **t** has the form $\mathbf{(fn\ a_1\ \ldots\ a_n)}$.

Terminology. If a term **t** is a call of **fn** we say **fn** is the *top function symbol* of **t**. A function symbol **fn** *is called in* a term **t** iff either **t** is a call of **fn** or **t** is a nonvariable term and **fn** is called in an argument of **t**. The set of *subterms* of a term **t** is {**t**} if **t** is a variable symbol 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 **(PLUS X Y)** is a call of **PLUS** with arguments **X** and **Y**. **PLUS** is called in **(IF A (PLUS X Y) B)**. The set of subterms of **(PLUS X Y)** is {**(PLUS X Y) X Y**}. The set of variables of **(PLUS X Y)** is {**X Y**}.

3.2. Terminology about Theories

Notes. Theories evolve over time by the repeated application of extension principles. For example, to construct our logic we start with propositional calculus with equality and extend it by adding the axioms for the natural numbers. Then we extend it again to get ordered pairs and again to get symbols... We eventually start adding axioms defining functions such as Peano sum, product, etc. When we stop, the user of the theorem prover starts by invoking the extension principles to add his own data types and concepts.

Each extension principle preserves the consistency of the original logic, provided certain ''admissibility'' requirements are met. In order to describe these requirements it is necessary that we be able to talk clearly about the sequence of steps used to create the ''current'' extension.

Terminology. Formula **t** *can be proved directly from* a set of axioms A if and only if **t** may be derived from the axioms in A by applying the rules of inference of propositional calculus with equality and instantiation (see page 9) and the principle of induction (see page 59).

Terminology. There are four kinds of *axiomatic acts*: (a) an application of the shell principle (page 13), (b) an application of the principle of definition (page 60), (c) an application of the constraint principle, and (d) the addition of an arbitrary formula as an axiom.

Terminology. Each such act *adds* a set of axioms. The axioms added by an application of the first three acts are described in the relevant subsections. The axioms added by the addition of an arbitrary formula is the singleton set consisting of the formula.

Terminology. A *history* h is a finite sequence of axiomatic acts such that either (a) h is empty or (b) h is obtained by concatenating to the end of a history h**'** an axiomatic act that is ''admissible'' under h**'**. An arbitrary axiom is admissible under any h**'**. The specification of the shell and definitional principles define ''admissibility'' in those instances.

Terminology. The *axioms* of a history h is the union of the axioms added by each act in h together with the axioms described in this presentation of the logic.

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

Terminology. A function symbol **fn** is *new* in a history h iff **fn** is called in no axiom of h (except for the propositional, reflexivity, and equality axioms of page 10), **fn** is not a **CAR/CDR** symbol (see below), and **fn** is not in the set {**CASE COND F LET LIST LIST* NIL QUOTE T**}.

Terminology. We say a symbol **fn** is a *CAR/CDR symbol* if there are at least three characters in **fn**, the first character in **fn** is **C**, the last character is **R**, and each other character is either an **A** or a **D**.

Examples. The symbol **CADDR** is a **CAR/CDR** symbol. We will eventually

introduce an abbreviation that ''defines'' such symbols to stand for nests of **CAR**s and **CDR**s. Because **CADDR** is a **CAR/CDR** symbol it is not new. The definitional principle requires that the function symbol defined be ''new.'' Hence, it is impossible to define **CADDR**. Similarly, it is impossible to define nine other perfectly acceptable symbols, **CASE**, **COND**, **F**, **LET**, **LIST**, **LIST***, **NIL**, **QUOTE**, and **T**. All of these prohibited symbols will be involved in our abbreviation conventions.

4. Embedded Propositional Calculus and Equality

Notes. Our logic is a quantifier-free first order extension of propositional calculus with equality, obtained by adding axioms and rules of inference. Any classical formalization of propositional calculus and equality will suit our purposes. So that this presentation of the logic is self-contained we have included as the first subsection below, one such formalization, namely that of Shoenfield [2].

We then add axioms to define the functional analogues of the propositional operators and the equality relation. This effectively embeds propositional calculus and equality into the term structure of the logic. That is, we can write down and reason about terms that contain propositional connectives, equality, and case analysis. For example, we can write

```
(IF (EQUAL N 0)
    1
    (TIMES N (FACT (SUB1 N))))
```

which is a term equal to **1** if **N** is **0** and equal to **(TIMES N (FACT (SUB1 N)))** otherwise. The ability to write such terms is very convenient later when we begin defining recursive functions.

4.1. Propositional Calculus with Equality

Shoenfield's system consists of one axiom schema and four inference rules. A *Propositional Axiom* is any formula of the form

Axiom Schema.
(¬(**a**) ∨ **a**)**.**

The four rules of inference are

Rules of Inference.

Expansion: derive (**a** ∨ **b**) from **b**;

Contraction: derive **a** from (**a** ∨ **a**);

Associativity: derive ((**a** ∨ **b**) ∨ **c**) from (**a** ∨ (**b** ∨ **c**)); and

Cut: derive (**b** ∨ **c**) from (**a** ∨ **b**) and (¬(**a**) ∨ **c**).

To formalize equality we also use Shoenfield's approach, which involves three axiom schemas. A *Reflexivity Axiom* is any formula of the form

Axiom Schema.
(a = a).

For every function symbol **fn** of arity **n** we add an *Equality Axiom for* **fn**.

Axiom Schema.
```
((X1=Y1) →
  (...      →
    ((Xn=Yn) →

      (fn X1 ... Xn) = (fn Y1 ... Yn))...))
```

Finally, we add

Axiom.
((X1=Y1) → ((X2=Y2) → ((X1=X2) → (Y1=Y2)))).

This axiom is the only instance we need of Shoenfield's ''equality axiom (schema) for predicates.''

Note. Finally, we add the rule of inference that any instance of a theorem is a theorem. To make this precise we first define substitution.

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

Example. If s is {<**X**, **(ADD1 Y)**> <**Y**, **Z**> <**G**, **FOO**>} then s is a substitution. If **p** is the term

```
(PLUS     X    (G Y    X))
```

then **p**/s is the term

```
(PLUS (ADD1 Y) (G Z (ADD1 Y))).
```

Note that even though the substitution contains the pair <**G**, **FOO**> the occurrence of **G** in **p** was not replaced by **FOO** since **G** does not occur as a variable in **p**.

Rule of Inference. *Instantiation*:
Derive **a**/s from **a**.

4.2. The Axioms for TRUE, FALSE, IF, and EQUAL

Abbreviation. We will abbreviate the term **(TRUE)** with the symbol **T** and the term **(FALSE)** with the symbol **F**.

Axiom 1.
T ≠ **F**

Axiom 2.
X = **Y** → **(EQUAL X Y)** = **T**

Axiom 3.
X ≠ **Y** → **(EQUAL X Y)** = **F**

Axiom 4.
X = **F** → **(IF X Y Z)** = **Z**

Axiom 5.
X ≠ **F** → **(IF X Y Z)** = **Y.**

4.3. The Propositional Functions

Defining Axiom 6.
(TRUEP X) = **(EQUAL X T)**

Defining Axiom 7.
(FALSEP X) = **(EQUAL X F)**

Defining Axiom 8.
(NOT P)
=
(IF P F T)

Defining Axiom 9.
```
(AND P Q)
   =
(IF P (IF Q T F) F)
```

Defining Axiom 10.
```
(OR P Q)
   =
(IF P T (IF Q T F))
```

Defining Axiom 11.
```
(IMPLIES P Q)
   =
(IF P (IF Q T F) T)
```

Abbreviation. When we refer to a term `t` as a formula, one should read in place of `t` the formula `t≠F`.

Example. The term

```
(IMPLIES (AND (P X) (Q Y)) (R X Y)),
```

if used where a formula is expected (e.g., in the allegation that it is a theorem), is to be read as

```
(IMPLIES (AND (P X) (Q Y)) (R X Y)) ≠ F.
```

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

```
((P X)≠F ∧ (Q Y)≠F)  →  (R X Y)≠F
```

which we could abbreviate

```
((P X) ∧ (Q Y))  →  (R X Y).
```

Note. The definitional principle, to be discussed later, permits the user of the logic to add new defining axioms under admissibility requirements that ensure the unique satisfiability of the defining equation. The reader may wonder why we did not invoke the definitional principle to add the defining axioms above—explicitly eliminating the risk that they render the system inconsistent. In fact, we completely avoid use of the definitional principle in this presentation of the logic. There are two reasons. First, the definitional principle also adds an axiom (the non-**SUBRP** axiom) that connects the defined symbol to the interpreter for the logic—an axiom we do not wish to have for the primitives. Second, the admissibility requirements of the definitional principle are not al-

ways met in the development of the logic.

5. The Shell Principle and the Primitive Data Types

Note. The shell principle permits the extension of the logic by the addition of a set of axioms that define a new data type. Under the conditions of admissibility described, the axioms added are guaranteed to preserve the consistency of the logic. The axioms are obtained by instantiating a set of axiom schemas described here. In order to describe the axiom schemas it is first necessary to establish several elaborate notational conventions. We then define the shell principle precisely. Then we invoke the shell principle to obtain the axioms for the natural numbers, the ordered pairs, the literal atoms, and the negative integers.

5.1. Conventions

Terminology. We say **t** is the **fn** *nest around* **b** *for* **s** iff **t** and **b** are terms, **fn** is a function symbol of arity 2, **s** is a finite sequence of terms, and either (a) **s** is empty and **t** is **b** or (b) **s** is not empty and **t** is **(fn t_1 t_2)** where t_1 is the first element of **s** and t_2 is the **fn** nest around **b** for the remaining elements of **s**. When we write **(fn t_1 ... t_n)⊗b** where a term is expected, it is an abbreviation for the **fn** nest around **b** for t_1, ..., t_n.

Note. In the first edition used ‘‘**(fn t_1 ... t_n)@b**’’ to denote what we now denote with ‘‘**(fn t_1 ... t_n)⊗b**.’’ We changed from ‘‘@’’ to ‘‘⊗’’ in the second edition because the ‘‘@’’ character is now permitted to occur in the extended syntax, in conjunction with backquote notation.

Examples. The **OR** nest around **F** for **A**, **B**, and **C** is the term **(OR A (OR B (OR C F)))**, which may also be written **(OR A B C)⊗F**.

Terminology. Each application of the shell principle introduces several ‘‘new’’ function symbols. The invocation explicitly names one symbol as the *constructor* and another as the *recognizer*. Zero or more other symbols are named as *accessors*, and one may be named as the *base* function symbol for that shell.

Terminology. The *constructor function symbols of* a history h consists exactly of the constructor function symbols of applications of the shell principle in h.

The *recognizer function symbols* of a history h is the union of {**TRUEP FALSEP**} with the set consisting exactly of the recognizer function symbols of the applications of the shell principle in h. The *base function symbols* of a history h is the union of {**TRUE FALSE**} with the set consisting exactly of the base function symbols of the applications of the shell principle in h for which a base function symbol was supplied.

Terminology. We say **r** is the *type* of **fn** iff either (a) **r** is given as the type of **fn** in Table 2 or (b) **fn** is a constructor or base function symbol introduced in the same axiomatic act in which **r** was the recognizer function symbol.

Table 2

fn	**type of fn**
TRUE	**TRUEP**
FALSE	**FALSEP**

Terminology. A *type restriction over* a set of function symbols s is a nonempty finite sequence of symbols where the first symbol is either the word **ONE-OF** or **NONE-OF** and each of the remaining is an element of s.

Terminology. A function symbol **fn** *satisfies* a type restriction **(flg s_1 ... s_n)** iff either **flg** is **ONE-OF** and **fn** is among the s_i or **flg** is **NONE-OF** and **fn** is not among the s_i.

Terminology. We say **t** is the *type restriction term for* a type restriction **(flg r_1 ... r_n)** *and* a variable symbol **v** iff **flg** is **ONE-OF** and **t** is **(OR (r_1 v) ... (r_n v))⊗F** or **flg** is **NONE-OF** and **t** is **(NOT (OR (r_1 v) ... (r_n v))⊗F)**.

Examples. Let tr_1 be **(ONE-OF LISTP LITATOM)**. Then tr_1 is a type restriction over the set {**NUMBERP LISTP LITATOM**}. The function symbol **LISTP** satisfies tr_1 but the function symbol **NUMBERP** does not. The type restriction term for tr_1 and **X1** is **(OR (LISTP X1) (OR (LITATOM X1) F))**. Let tr_2 be **(NONE-OF NUMBERP)**. Then tr_2 is a type restriction over the set {**NUMBERP LISTP LITATOM**}. The function symbol **LISTP** satisfies tr_2 but the function symbol **NUMBERP** does not. The type restriction term for tr_2 and **X2** is **(NOT (OR (NUMBERP X2) F))**.

5.2. The Shell Principle

Extension Principle . Shell Principle

The axiomatic act

Shell Definition .
Add the shell **const** *of* **n** *arguments*
with (optionally, base function **base***,)*
recognizer function **r***,*
accessor functions $\mathbf{ac_1}$, ..., $\mathbf{ac_n}$,
type restrictions $\mathbf{tr_1}$, ..., $\mathbf{tr_n}$, *and*
default functions $\mathbf{dv_1}$, ..., $\mathbf{dv_n}$,

is admissible under the history h provided

(a) **const** is a new function symbol of **n** arguments, (**base** is a new function symbol of no arguments, if a base function is supplied), **r**, $\mathbf{ac_1}$, ..., $\mathbf{ac_n}$ are new function symbols of one argument, and all the above function symbols are distinct;

(b) each $\mathbf{tr_i}$ is a type restriction over the recognizers of h together with the symbol **r**;

(c) for each **i**, $\mathbf{dv_i}$ is either **base** or one of the base functions of h; and

(d) for each **i**, if $\mathbf{dv_i}$ is **base** then **r** satisfies $\mathbf{tr_i}$ and otherwise the type of $\mathbf{dv_i}$ satisfies $\mathbf{tr_i}$.

If the $\mathbf{tr_i}$ are not specified, they should each be assumed to be **(NONE-OF)**.

If admissible, the act adds the axioms shown below. In the special case that no **base** is supplied, **T** should be used for all occurrences of **(r (base))** below, and **F** should be used for all terms of the form **(EQUAL x (base))** below.

(1)
```
(OR (EQUAL (r X) T)
    (EQUAL (r X) F))
```

(2)
```
(r (const X1 ... Xn))
```

(3)
```
(r (base))
```

(4) **(NOT (EQUAL (const X1 ... Xn) (base)))**

(5)
```
(IMPLIES (AND (r X)
              (NOT (EQUAL X (base))))
         (EQUAL (const (ac_1 X) ... (ac_n X))
                X))
```

For each **i** from 1 to **n**, the following formula

(6)
```
(IMPLIES trt_i
         (EQUAL (ac_i (const X1 ... Xn))
                Xi))
```

where $\mathbf{trt_i}$ is the type restriction term for $\mathbf{tr_i}$ and **Xi.**

For each **i** from 1 to **n**, the following formula

(7)
```
(IMPLIES (OR (NOT (r X))
             (OR (EQUAL X (base))
                 (AND (NOT trt_i)
                      (EQUAL X
                             (const X1 ... Xn)))))
         (EQUAL (ac_i X) (dv_i)))
```

where $\mathbf{trt_i}$ is the type restriction term for $\mathbf{tr_i}$ and **Xi.**

For each recognizer, **r'**, in the recognizer functions of h the formula

(8) **(IMPLIES (r X) (NOT (r' X)))**

(9)
```
(IMPLIES (r X)
         (EQUAL (COUNT X)
                (IF (EQUAL X (base))
                    (ZERO)
                    (ADD1
                     (PLUS
                      (COUNT (ac_1 X))
                      ...
                      (COUNT (ac_n X)))⊗(ZERO))))))
```

(10) The **SUBRP** axiom for each of the symbols **const**, **base** (if supplied), **r**, $\mathbf{ac_1}$, ..., $\mathbf{ac_n}$. We define the ''**SUBRP** axiom'' on page 54.

Note. In the first edition, the shell principle included two additional axioms, there labeled (8) and (9), which were in fact derivable from axiom (10) of the

first edition. Their deletion from the second edition has caused renumbering of the subsequent axioms.

5.3. Natural Numbers—Axioms 12.n

Shell Definition.
Add the shell **ADD1** of one argument
with base function **ZERO**,
recognizer function **NUMBERP**,
accessor function **SUB1**,
type restriction **(ONE-OF NUMBERP)**, and
default function **ZERO**.

Axiom 13.
```
(NUMBERP (COUNT X))
```

Axiom 14.
```
(EQUAL (COUNT T) (ZERO))
```

Axiom 15.
```
(EQUAL (COUNT F) (ZERO))
```

Defining Axiom 16.
```
(ZEROP X)
   =
(OR (EQUAL X (ZERO)) (NOT (NUMBERP X)))
```

Defining Axiom 17.
```
(FIX X) = (IF (NUMBERP X) X (ZERO))
```

Defining Axiom 18.
```
(PLUS X Y)
   =
(IF (ZEROP X)
    (FIX Y)
    (ADD1 (PLUS (SUB1 X) Y)))
```

5.4. Ordered Pairs—Axioms 19.n

Shell Definition.
Add the shell **CONS** of two arguments
with recognizer function **LISTP**,
accessor functions **CAR** and **CDR**, and
default functions **ZERO** and **ZERO**.

Notes. This invocation of the shell principle is, strictly speaking, inadmissible because there are axioms about **CONS**, **CAR**, and **CDR** in the **SUBRP** axioms added on behalf of the preceding shell. We ignore this inadmissibility and add the corresponding axioms anyway.

5.5. Literal Atoms—Axioms 20.n

Shell Definition.
Add the shell **PACK** of one argument
with recognizer function **LITATOM**,
accessor function **UNPACK**, and
default function **ZERO**.

Notes. This invocation of the shell principle is, strictly speaking, inadmissible because there are axioms about **PACK** in the **SUBRP** axioms added on behalf of the preceding shells. We ignore this inadmissibility and add the corresponding axioms anyway.

5.6. Negative Integers—Axioms 21.n

Shell Definition.
Add the shell **MINUS** of one argument
with recognizer function **NEGATIVEP**,
accessor function **NEGATIVE-GUTS**,
type restriction **(ONE-OF NUMBERP)**, and
default function **ZERO**.

6. Explicit Value Terms

Note. This section is technically an aside in the development of the logic. We define a particularly important class of terms in the logic, called the ‘‘explicit value terms.’’ Intuitively, the explicit value terms are the ‘‘canonical constants’’ in the logic. It is almost the case that every constant term—every variable-free term—can be mechanically reduced to a unique, equivalent explicit value. The only terms not so reducible are those involving (at some level in the definitional hierarchy) undefined functions, constrained functions, or calls of metafunctions such as **V&C$**. Thus, the explicit value terms are the terms upon which we can ‘‘compute’’ in the logic. They are the basis for our encoding of the terms as objects in the logic, and elaborate syntactic conventions are

adopted in the extended syntax to permit their succinct expression.

Terminology. We say **tr** is the i[th] *type restriction for a constructor function symbol* **fn** of arity n iff $1 \le i \le n$, and **tr** is the i[th] type restriction specified in the axiomatic act in which **fn** was introduced.

Examples. The first type restriction for **ADD1** is **(ONE-OF NUMBERP)**. The second type restriction for **CONS** is **(NONE-OF)**.

Terminology. We say **t** is an *explicit value term* in a history h iff **t** is a term and either (a) **t** is a call of a base function symbol in h, or (b) **t** is a call of a constructor function symbol **fn** in h on arguments $\mathbf{a_1}$, ..., $\mathbf{a_n}$ and for each $1 \le i \le \mathbf{n}$, $\mathbf{a_i}$ is an explicit value term in h and the type of the top function symbol of $\mathbf{a_i}$ satisfies the i[th] type restriction for the constructor function **fn**. We frequently omit reference to the history h when it is obvious by context.

Examples. The following are explicit value terms:

```
(ADD1 (ADD1 (ZERO)))

(CONS (PACK (ZERO)) (CONS (TRUE) (ADD1 (ZERO))))
```

The term **(ADD1 X)** is not an explicit value, since **X** is neither a call of a base function symbol nor a call of a constructor. The term **(ADD1 (TRUE))** is not an explicit value, because the top function symbol of **(TRUE)** does not satisfy the type restriction, **(ONE-OF NUMBERP)**, for the first argument of **ADD1**.

7. The Extended Syntax

Note on the Second Edition. The presentation of the logic given here extends the syntax of an earlier presentation by adding

- the Common Lisp convention for writing comments both with semi-colon and with balanced occurrences of **#|** and **|#**;
- more of Common Lisp's notation for writing integers, including binary, octal and hexadecimal notation, e.g., **#B1000** as an abbreviation of 8;
- Common Lisp's ''backquote'' notation, so that **'(A ,@X B)** is an abbreviation of **(CONS 'A (APPEND X (CONS 'B NIL)))**;
- **COND** and **CASE**, which permit the succinct abbreviations of commonly used nests of **IF**-expressions;

- **LIST***, which permits the succinct abbreviation of commonly used nests of **CONS**-expressions; and
- **LET**, which permits the ''binding'' of ''local variables'' to values, permitting the succinct abbreviation of terms involving multiple occurrences of identical subterms.

To explain the new integer notation and the backquote notation in a way that is (we believe) perfectly accurate and permits their use ''inside **QUOTE**s'' it was necessary to redevelop the foundation of the syntax as it was presented in the first edition. In particular, in the second edition we start with a class of structures larger than the first edition's ''s-expressions''—structures which include such utterances as **'(,X)**. Because the foundation changed, virtually all of the first edition's Section 4.7 (pages 112-124), has changed. But the new syntax is strictly an extension of the old; every well-formed term in the old ''extended syntax'' is well-formed in the new and abbreviates the same formal term. So despite the relatively massive change to the description, the impact of the second edition is only to add and fully document the features noted above.

Notes. The extended syntax differs from the formal syntax only in that it permits certain abbreviations. That is, every term in the formal syntax is also a term in the extended syntax, but the extended syntax admits additional well-formed utterances that are understood to stand for certain formal terms. These abbreviations primarily concern notation for shell constants such as numbers, literal atoms, and lists. In addition, the extended syntax provides some abbreviations for commonly used function nests and for the general purpose bounded quantifier function **FOR**. We delay the presentation of the quantifier abbreviations until after we have axiomatized **FOR** (see section 11, page 56) but discuss all other aspects of the extended syntax in this section.

We define the extended syntax in six steps.

1. We define a set of tree structures called ''token trees'' that includes the formal terms and some other structures as well.
2. We explain how token trees are displayed.
3. We identify three nested subsets of the token trees: the ''readable'' token trees, the ''s-expressions,'' and the ''well-formed'' s-expressions.
4. We define a mapping, called ''readmacro expansion,'' from the readable token trees to s-expressions.
5. We define a mapping, called ''translation,'' from the well-formed s-expressions to formal terms.
6. Finally, we make the convention that a readable token tree may be

used as a term in the extended syntax provided its readmacro expansion is well-formed. Such a token tree abbreviates the translation of its readmacro expansion.

For example, the token tree we display as **(LIST* X #B010 '(0 . 1))** is readable. Its readmacro expansion is **(LIST* X 2 (QUOTE (0 . 1)))**. This s-expression is well-formed. Its translation is the formal term:

```
(CONS X
      (CONS (ADD1 (ADD1 (ZERO)))
            (CONS (ZERO)
                  (ADD1 (ZERO))))).
```

Thus, **(LIST* X #B010 '(0 . 1))** may be used as a term in the extended syntax and abbreviates the **CONS**-nest above.

The token tree **(LIST* X ,A)** is not readable, because it violates rules on the use of the ''the comma escape from backquote.'' The token tree **(EQUAL '(,X . ,Y))** is readable. However, its readmacro expansion is **(EQUAL (CONS X Y))**, which is not well-formed, because the function symbol **EQUAL** requires two arguments.

We apologize to readers expecting a definition of our syntax presented in a formal grammar (e.g., BNF). We have three reasons for proceeding in this fashion. First, despite the apparent simplicity of our syntax, it has extremely powerful and complicated provisions for describing structures. These provisions allow a natural embedding of the language into its constants, which facilitates the definition of a formal metatheory in the logic, as carried out in the final sections of this presentation of the logic. Thus, much of the verbiage here devoted to syntax can be thought of as devoted to the formal development of the metatheory.

Second, we not only wish to specify the legal expressions in the extended syntax but also to map them to terms in the formal syntax. We think it unlikely that an accurate formal presentation of our syntax and its meaning would be any more clear than the informal but precise one offered here; furthermore, it would be much less accessible to most readers.

Finally, this presentation is closely related to the actual implementation of the syntax in the Nqthm system. In our implementation the user types ''displayed token trees.'' These character strings are read by the Common Lisp **read** routine. The **read** routine causes an error if the token tree presented is not ''readable'' (e.g., uses a comma outside a backquote). Otherwise, the **read** routine ''expands'' the ''**read** macros'' (single quote, backquote, and **#**) and returns the s-expression as a Lisp object. It is only then that our theorem prover gets to inspect the object to determine if it is ''well-formed'' and, if so, what term it abbreviates.

7.1. Token Trees and Their Display

Note. Our token trees are essentially the parse trees of Common Lisp s-expressions, except for the treatment of **`read`** macro characters (such as **`#`** and **`'`**) and certain atoms. For example, **`#B010`**, **`+2.`** and **`2`** are three *distinct* token trees. We define ''readmacro expansion'' so that these three trees expand into the same ''s-expression.'' We start the development with the definition of the **`#`** convention for writing down integers in various bases. Then we introduce the tokens involved in the single quote and backquote conventions. Finally, we define token trees and how to display them.

Terminology. A character c is a *base* n *digit character* if and only if n≤16 and c is among the first n characters in the sequence **`0123456789ABCDEF`**. The position of c (using zero-based indexing) in the sequence is called its *digit value*.

Note. When we define how we display ''token trees'' and the digit characters in them we will make it clear that case is irrelevant. Thus, while this definition makes it appear that only the upper case characters **`A`**-**`F`** are digit characters, we will effectively treat **`a`**-**`f`** as digit characters also.

Example. The base 2 digits are **`0`** and **`1`**. Their digit values are 0 and 1, respectively. Among the base 16 digits are **`A`** and **`F`**. The digit value of **`A`** is 10 and the digit value of **`F`** is 15.

Terminology. A sequence of characters, s, is a *base* n *digit sequence* if and only if s is nonempty and every element of s is a base n digit character. The *base* n *value* of a base n digit sequence $\mathbf{c}_k...\mathbf{c}_1\mathbf{c}_0$ is the integer $c_k n^k + ... + c_1 n^1 + c_0 n^0$, where c_i is the value of the digit $\mathbf{c}_i$.

Example. **`1011`** is a base 2 digit sequence. Its base 2 value is the integer eleven. **`1011`** is also a base 8 digit sequence. Its base 8 value is the integer five hundred and twenty one.

Terminology. A sequence of characters, s, is an *optionally signed base* n *digit sequence* if and only if s is either a base n digit sequence or s is nonempty, the first character of s is either a **`+`** or **`-`**, and the remaining characters constitute a base n digit sequence. If the first character of s is **`-`**, the *base* n *signed value* of s is the negative of the base n value of the constituent base n digit sequence. Otherwise, the *base* n *signed value* of s is the base n value of the constitutent base n digit sequence.

Example. **`A2`** and **`+A2`** are both optionally signed base 16 digit sequences

whose signed values (in decimal notation) are both 162. **-A2** is also such a sequence; its signed value is **-**162.

Terminology. A sequence of characters, s, is a *#-sequence* if and only if the length of s is at least three, the first character in s is **#**, the second character in s is in the set {**B O X**}, and the remaining characters in s constitute an optionally signed base n digit sequence, where n is 2, 8, or 16 according to whether the second character of s is **B**, **O**, or **X**, respectively.

Note. Our convention of disregarding case will allow **b**, **o**, and **x** in the second character position of **#**-sequences.

Terminology. Finally, a sequence of characters, s, is a *numeric sequence* if and only if one of the following is true:

- s is an optionally signed base 10 digit sequence (in which case its *numeric value* is the base 10 signed value of s);
- s is an optionally signed base 10 digit sequence with a dot character appended on the right (in which case its *numeric value* is the base 10 signed value of s with the dot removed);
- s is a **#**-sequence (in which case its *numeric value* is the base n signed value of the sequence obtained from s by deleting the first two characters, where n is 2, 8, or 16 according to whether the second character of s is **B**, **O**, or **X**).

Example. Table 3 shows some numeric sequences and their numeric values written in decimal notation.

Table 3

sequence	value
123	123
-5.	-5
#B1011	11
#O-123	-83
#Xfab	4011

Terminology. The character sequence containing just the single quote (sometimes called ‘‘single gritch’’) character (**'**) is called the *single quote token*. The character sequence containing just the backquote character (**`**) is called the *backquote token*. The character sequence containing just the dot character (**.**) is called the *dot token*. The character sequence containing just the comma character (**,**) is called the *comma token*. The character sequence containing just

the comma character followed by the at-sign character (**,@**) is called the *comma at-sign token*. The character sequence containing just the comma character followed by the dot character (**,.**) is called the *comma dot token*.

Terminology. A sequence of characters, s, is a *word* if and only if s is a numeric sequence or s is nonempty and each character in s is a member of the set

{**A B C D E F G H I J K L M**
N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
$ ^ & * _ - + = ~ { } ? < >}**.**

Note. All of our symbols are words, but the set of words is larger because it includes such non-symbols as ***1*TRUE**, **123**, **1AF**, and **#B1011** that begin with non-alphabetic characters.

Terminology. A *token tree* is one of the following:

- an integer;
- a word;
- a nonempty sequence of token trees (a token tree of this kind is said to be an *undotted* token tree);
- a sequence of length at least three whose second-to-last element is the dot token and all of whose other elements are token trees (a token tree of this kind is said to be a *dotted* token tree); or
- a sequence of length two whose first element is one of the tokens specified below and whose second element is a token tree (called the *constitutent token tree*):
    - the single quote token (such a token tree is said to be a *single quote* token tree),
    - the backquote token (a *backquote* token tree),
    - the comma token (a *comma escape* token tree), or
    - the comma at-sign or comma dot tokens (such token trees are said to be *splice escape* token trees).

Note. In order to explain how token trees are displayed we must first introduce the notion of ‘‘comments.’’ Following Common Lisp, we permit both ‘‘semi-colon comments’’ and ‘‘number sign comments.’’ The latter are, roughly speaking, text delimited by *balanced* occurrences of **#|** and **|#**. To define this

notion precisely, we must introduce the terminology to let us talk about **#|** and **|#** clearly.

Terminology. The integer i is a *#|-occurrence* in the character string **s** of length n iff $0 \le i \le n-2$, the i^{th} (0-based) character of **s** is the number sign character (**#**) and the $i+1^{st}$ character of **s** is the vertical bar character, **|**. We define what it is for i to be a *|#-occurrence* in strict analogy. A **|#**-occurrence, j, is *strictly to the right* of a **#|**-occurrence, i, iff $j > i+1$. Finally, if i is a **#|**-occurrence in **s** and j is a **|#**-occurrence in **s** that is strictly to the right, then the *substring of* **s** *delimited by* i *and* j is the substring of **s** that begins with the $i+2^{nd}$ character and ends with the $j-1^{st}$ character.

Examples. Consider the string **#|Comment|#**. The string has eleven characters in it. The only **#|**-occurrence is 0. The only **|#**-occurrence is 9. The substring delimited by these occurrences is **Comment**. In the string **#||#** the only **#|**-occurrence is 0 and the only **|#**-occurrence is 2, which is strictly to the right of the former. The substring delimited by these two occurrences is the empty string.

Terminology. A string of characters, **s**, has *balanced number signs* iff all the **#|**- and **|#**-occurrences can be paired (i.e., put into 1:1 correspondence) so that every **#|** has its corresponding **|#** strictly to its right and the text delimited by the paired occurrences has balanced number signs.

Examples. The string **Comment** has balanced number signs because there are no **#|**- or **|#**-occurrences. The string

code #|Comment|# and code

also has balanced number signs. The following string does not have balanced number signs:

#|code #|Comment|# and code.

Terminology. A string of characters, **s**, is a *#-comment* iff **s** is obtained from a string, **s′**, that has balanced number signs, by adding a number sign character immediately followed by a vertical bar (**#|**) to the left-hand end and a vertical bar immediately followed by a number sign (**|#**) to the right-hand end.

Notes. Generally speaking, any string of characters not including **#|** or **|#** can be made into a **#**-comment by surrounding the string with **#|** and **|#**. Such comments will be allowed in certain positions in the display of terms. Roughly speaking, the recognition that a string is a comment is license to ignore the string; that is the whole point of comments. Comments can be added to the

display of a term without changing the term displayed. We are more specific in a moment. The display of such a commented term can itself be turned into a **#**-comment by again surrounding it with **#|** and **|#**. If we did not insist on ‘‘balancing number signs’’ the attempt to ‘‘comment out’’ a term could produce ill-formed results because the ‘‘opening’’ **#|** would be ‘‘closed’’ by the first **|#** encountered in the term’s comments and the rest of the term would not be part of the comment.

The question of where **#**-comments are allowed is difficult to answer, but it is answered below. The problem is that in Common Lisp (whose **read** routine we actually use to parse terms) the number sign character does not automatically terminate the parsing of a lexeme. Thus, **(H #|text|# X)** reads as a list containing the lexeme **H** followed by the lexeme **X**, but **(H#|text|# X)** reads as the list containing the lexeme **H#text#** followed by the lexeme **X**. So some ‘‘white space’’ must appear after **H** and before the number sign in order for the number sign to be read as the beginning of a comment. As for the end of the comment, the **|#** closes the comment no matter what follows. Thus, **(H #|text|#X)** is also read as **H** followed by **X**, even though there is no space after the second number sign. The naive reader might conclude that **#|** must always be preceded by white space and **|#** may optionally be followed by white space. Unfortunately, the rule is a little more complicated. The display **(FN (H X)#|text|# X)** is read as **(FN (H X) X)**. The reason is that the close parenthesis in **(H X)** is a ‘‘lexeme terminating character’’ that terminates the lexeme before it, i.e., **X**, and constitutes a lexeme in itself. Thus, the **#|** is recognized as beginning a comment. We now explain this precisely.

Terminology. The *white space characters* are defined to be space, tab, and newline (i.e., carriage return). The *lexeme terminating characters* are defined to be semicolon, open parenthesis, close parenthesis, single quote mark, backquote mark, and comma. (Note that **#** is *not* a lexeme terminator!) The union of the white space characters and the lexeme terminating characters is called the *break characters*.

Terminology. A *comment* is any of the following:

- a (possibly empty) sequence of white space characters,
- a **#**-comment,
- a sequence of characters that starts with a semicolon, ends with a newline, and contains no other newlines, or
- the concatenation of two comments.

A comment is said to be a *separating comment* if it is nonempty and its first character is either a white space character or a semicolon.

Example. Thus,

```
;This is a comment.
```

is a separating comment. The string **#|And this is a comment.|#** is a comment but not a separating comment. It can be made into a separating comment by adding a space before the first **#**. The following is a display of a separating comment (since it starts with a semicolon) that illustrates that comments may be strung together:

```
; This is a long comment.
; It spans several lines and
#|Contains both semicolon and
number sign comments.  In addition,
immediately after the following
number sign is a white space
comment.|#   ; And here's another
;semicolon comment.
```

Terminology. Suppose $\mathbf{s_1}$ and $\mathbf{s_2}$ are non-empty character strings. Then *suitable separation* between $\mathbf{s_1}$ and $\mathbf{s_2}$ is any comment with the following properties. Let $\mathbf{c_1}$ be the last character in $\mathbf{s_1}$, and let $\mathbf{c_2}$ be the first character in $\mathbf{s_2}$.

- If $\mathbf{c_1}$ is a break character, suitable separation is any comment.
- If $\mathbf{c_1}$ is not a break character but $\mathbf{c_2}$ is a break character, suitable separation is either the empty comment or any separating comment.
- If neither $\mathbf{c_1}$ nor $\mathbf{c_2}$ is a break character, suitable separation is any separating comment.

When we use the phrase ''suitable separation'' it is always in a context in which $\mathbf{s_1}$ and $\mathbf{s_2}$ are implicit.

Terminology. A *suitable trailer* after a string **s** is any suitable separation between **s** and a string beginning with a break character. That is, a suitable trailer may be any comment if the last character of **s** is a break character and otherwise may be either an empty comment or a separating comment.

Examples. We use suitable separations and trailers to separate the lexemes in our displays of terms in the extended syntax. For example, in **(FN X)** the lexeme **FN** is separated from the lexeme **X** by the separating comment consisting of a single space. Any separating comment is allowed in the extended syntax. Thus,

```
(FN;Comment
X)
```

is an acceptable display. The empty string is not a separating comment. Thus, **(FNX)** is an unacceptable display. Similarly, because **#|Comment|#** is not a separating comment, **(FN#|Comment|#X)** is unacceptable. On the other hand, by adding a space to the front of the **#**-comment above we obtain a separating comment and thus **(FN #|Comment|#X)** is an acceptable display. Now consider **(FN (H X) Y)**. Because the last character of **FN** is not a break character but open parenthesis is a break character, suitable separation between displays of **FN** and **(H X)** is either the empty string or a separating comment. Thus, the first four of the following five displays will be acceptable. The last will not, because the comment used is not a separating comment or the empty comment.

```
(FN(H X) Y)            ; the empty comment

(FN
 (H X) Y)              ; a white space comment

(FN;text               ; a semicolon comment
  (H X) Y)

(FN #|text|# (H X) Y) ; a separating #-comment

(FN#|text|# (H X) Y)  ; unacceptable!
```

Any comment may be used to separate **(H X)** and **Y**, because the close parenthesis character is a break character. Thus, **(FN(H X)Y)** will be an allowed display, as will **(FN(H X)#|text|#Y)**.

Terminology. To *display* a token tree that is an integer or word, we write down a comment, a decimal representation of the integer or the characters in the word, followed by a suitable trailer. Upper case characters in a word may be displayed in lower case. To *display* a dotted or undotted token tree consisting of the elements t_1, ..., t_n, where t_{n-1} may or may not be the dot token, we write down a comment, an open parenthesis, suitable separation, a display of t_1, suitable separation, a display of t_2, suitable separation, ..., a display of t_n, suitable separation, a close parenthesis, and a comment, where we display the dot token as though it were a word. All other token trees consist of a token and a constituent token tree. To display them we write down a comment, the characters in the token, a comment, a display of the constituent token tree, and a suitable trailer.

Note. What token tree displays as **123**? Unfortunately, there are two: the tree consisting of the integer 123 and the tree consisting of the numeric sequence **123**. These two trees readmacro expand to the same tree and since readmacro expansion is involved in our abbreviation convention, it is irrelevant which of the two trees was actually displayed.

Example. Below we show some (displays of) token trees:

```
(EQUAL X                    ;Comments may
       (QUOTE (A B . C))) ;be included

(EQUAL X (QUOTE (A B . C)))

(equal x (quote (a b . c)))

(QUOTE (A . B))

'(A . B)

`(A ,X ,@(STRIP-CARS Y) B)

((1AB B**3 C) 12R45 (ADD1))
```

The first, second and third are different displays of the same token tree—the only difference between them is the comments used and the case of the characters. The fourth and fifth are displays of two different token trees, both of length two with the same second element, **(A . B)**, but one having the word **QUOTE** as its first element and the other having the single quote token as its first element. Note the difference in the way they are displayed. It will turn out that these two different token trees readmacro expand to the same s-expression. The sixth display shows how a backquote token tree containing two escapes may be displayed. Once we have defined readmacro expansion and translation it will be clear that all of the token trees except the last abbreviate terms.

Notes. Because of our convention of using lower case typewriter font words as metavariables, we will refrain from using lower case when displaying token trees. Thus, if **fn** is understood to be any function symbol of two arguments, then **(PLUS X Y)** is to be understood to be ''of the form'' of the token tree we display as **(fn X Y)** while it is not ''of the form'' of the tree we display as **(FN X Y)**. The reader may wonder why we permit token trees to be displayed in lower case if we never so display them. The point is that we never so display them in this presentation of the logic. But in other papers in which formulas are displayed, users frequently find lowercase characters more attractive. Furthermore, our implementation of Nqthm inherits from the host Lisp system the

default action of raising lower case characters to upper case characters when reading from the user's terminal or files. (Special action has to be taken to prevent this.) Thus, many users type formulas in lower case. Since metavariables are not part of the system, this presents no ambiguity.

7.2. Readable Token Trees, S-Expressions and Readmacro Expansion

Note. The token tree displayed as **`'(,,X)`** is problematic because it contains a doubly-nested backquote escape in a singly nested backquote tree. The token trees displayed as **`',@X`** and **`'(1 . ,@X)`** are problematic because they use the **`,@`** notation in ''non-element positions.'' We make precise these requirements by defining the notion of ''readable'' token trees for depth n and then restricting our attention later to the readable token trees for depth 0.

Terminology. A token tree **`s`** is *readable* from depth n iff one of the following holds:

- **`s`** is an integer or word;
- **`s`** is an undotted token tree and each element of **`s`** is readable from n;
- **`s`** is a dotted token tree and each element of **`s`** (other than the dot token) is readable from n and the last element of **`s`** is not a splice escape tree;
- **`s`** is a single quote token tree and the constituent token tree is readable from n;
- **`s`** is a backquote token tree and the constituent token tree is readable from n+1 and is not a splice escape tree; or
- **`s`** is a comma escape or splice escape token tree, n>0, and the constituent token tree is readable from n-1.

Example. The token tree **`(A ,X B)`** is not readable from depth 0 but is readable from depth 1. Thus, **`'(A ,X B)`** is readable from depth 0. The expressions **`',@X`** and **`(A . ,@X)`** are not readable from any depth.

Terminology. An *s-expression* is a token tree containing no numeric sequences or tokens other than the dot token.

Terminology. If **`s`** is a token tree readable from depth 0, then its *readmacro*

expansion is the s-expression obtained by the following four successive transformations.

- Replace every numeric sequence by its numeric value.
- Replace every single quote token by the word **QUOTE**.
- Replace every backquote token tree, **'x**, by the ''backquote expansion'' of **x** (see below), replacing innermost trees first.
- At this point the transformed tree contains no tokens other than, possibly, the dot token. Replace every subtree of **s** of the form, **(x_1 x_2 ... x_k . (y_1 ... y_n))**, by **(x_1 x_2 ... x_k y_1 ... y_n)** and every subtree of the form **(x_1 x_2 ... x_k . NIL)** by **(x_1 x_2 ... x_k)**. That is, a subtree should be replaced if it is a dotted token tree and its last element is either a dotted or undotted token tree or the word **NIL**. Let **x** be such a tree and let **y** be its last element. If **y** is a dotted or undotted token tree, then **x** is replaced by the concatenation of **y** onto the right-hand end of the sequence obtained from **x** by deleting the dot and the last element. If **y** is the word **NIL** then **x** is replaced by the sequence obtained from **x** by deleting the dot and the last element.

Examples. The readmacro expansion of **(A B . '(#B100 . C))** proceeds in the following steps. First, the numeric sequence is replaced by its integer value, producing **(A B . '(4 . C))**. Second, the single quote token tree is replaced by a **QUOTE** tree, **(A B . (QUOTE (4 . C)))**. Since there are no backquote tokens in the tree, that step of readmacro expansion is irrelevant here. Finally we consider the dotted token trees. The innermost one is not of the form that is replaced. The outermost one is replaceable. The result is **(A B QUOTE (4 . C))**.

The readmacro expansion of **(PLUS #B100 (SUM '(1 2 3)))** is **(PLUS 4 (SUM (QUOTE (1 2 3))))**. Note that the ambiguity concerning the display of integers and numeric sequences makes it impossible to uniquely determine the token tree initially displayed. For example, we do not know if it contained the integer two or the numeric sequence **2**. However, the second tree displayed is the readmacro expansion of the first. As such, it is an s-expression and contains no numeric sequences. Thus, the second token tree is uniquely determined by the display and the fact that it is known to be an s-expression.

Note. We now define ''backquote expansion.'' It is via this process that **'(,X ,Y . ,Z)** is readmacro expanded into **(CONS X (CONS Y Z))**, for example. Our interpretation of backquote is consistent with the Common Lisp interpretation [3, 4]. However, Common Lisp does not specify what s-

expression is built by backquote; instead it specifies what the value of the s-expression must be. Different implementations of Common Lisp actually produce different readmacro expansions. For example, in one Common Lisp, **`'(,X)`** might be **`(CONS X NIL)`** and in another it might be **`(LIST X)`**. The values of these expressions are the same. But consider what happens when a backquoted form is quoted, e.g., **`''(,X)`**. In one of the two hypothetical Lisps mentioned above, this string reads as the constant **`'(CONS X NIL)`**, while in the other it reads as **`'(LIST X)`**. This is intolerable in our setting since it would mean that the token tree **`(EQUAL (CAR ''(,X)) 'CONS)`** might read as a theorem in one Common Lisp and read as a non-theorem in another. We therefore have to choose a particular readmacro expansion of backquote (consistent with Common Lisp). We document it here. It is implemented by suitable changes to the Lisp readtable in Nqthm.

Terminology. If **`s`** is a token tree readable from some depth n and **`s`** contains no numeric sequences, single quote tokens or backquote tokens, then its *backquote expansion* is the token tree defined as follows.

- If **`s`** is an integer or a word, then **`(QUOTE s)`** is its backquote expansion.
- If **`s`** is a comma escape token tree, **`,x`**, or a splice escape token tree, **`,@x`** or **`,.x`**, then **`x`** is its backquote expansion.
- If **`s`** is an undotted token tree or a dotted token tree, then its backquote expansion is the ‘‘backquote-list expansion’’ of **`s`** (see below).

Terminology. We now define the *backquote-list expansion* for a dotted or undotted token tree, **`s`**, that is readable from some depth n and contains no numeric sequences, single quote tokens, or backquote tokens. Let **`x`** be the backquote expansion of the first element of **`s`**. Let **`y`** be as defined by cases below. Then the backquote-list expansion of **`s`** is either **`(APPEND x y)`** or **`(CONS x y)`**, according to whether the first element of **`s`** is a splice escape token tree or not. The definition of **`y`** is by cases:

- If **`s`** is a token tree of length 1, then **`y`** is **`(QUOTE NIL)`**.
- If **`s`** is a dotted token tree of length 3, then **`y`** is the backquote expansion of the last element of **`s`**.
- If **`s`** is any other dotted or undotted token tree, then **`y`** is the backquote-list expansion of the token tree obtained by removing the first element from **`s`**.

Examples. To illustrate the first case above, where **`s`** is an undotted token tree

of length 1, consider the backquote-list expansion of **(A)**. The result is **(CONS (QUOTE A) (QUOTE NIL))** because the backquote expansion of **A** is **(QUOTE A)**. The second case is illustrated by **(B . ,Z)**. Its backquote-list expansion is **(CONS (QUOTE B) Z)**. We combine these to illustrate the third case. The backquote-list expansion of **((A) B . ,Z)** is **(CONS (CONS (QUOTE A) (QUOTE NIL)) (CONS (QUOTE B) Z))**. The backquote-list expansion of **(A ,@Z B)** is **(CONS (QUOTE A) (APPEND Z (CONS (QUOTE B) (QUOTE NIL))))**.

Since the readmacro expansion of **'(A)** is just the backquote expansion of **(A)**, which, in turn, is the backquote-list expansion of **(A)**, we see that the readmacro expansion of **'(A)** is **(CONS (QUOTE A) (QUOTE NIL))**. In Table 4 we show some other examples of the readmacro expansion of backquote trees.

Table 4

token tree	readmacro expansion

`'(X ,X ,@X)`	`(CONS (QUOTE X)`
	`  (CONS X`
	`  (APPEND X`
	`  (QUOTE NIL))))`

`'((A . ,X)`
`  (B . ,Y)`
`  . ,REST)`
	`(CONS (CONS (QUOTE A) X)`
	`  (CONS (CONS (QUOTE B) Y)`
	`  REST))`

`''(,,X)`	`(CONS (QUOTE CONS)`
	`  (CONS X`
	`  (CONS (CONS (QUOTE QUOTE)`
	`  (CONS (QUOTE NIL)`
	`  (QUOTE NIL)))`
	`  (QUOTE NIL))))`

We can explain the last example of Table 4. The readmacro expansion of **''(,,X)** proceeds by first replacing the innermost backquote tree by its backquote expansion. So let us consider the backquote expansion of **'(,,X)**. We might first observe that while this tree is not readable from depth 0, its backquote expansion is nevertheless well defined. Indeed, its backquote expansion is the token tree **(CONS ,X (QUOTE NIL))** because the backquote expansion of **,,X** is **,X**. So replacing the constituent token tree, **'(,,X)**, of **''(,,X)** by its backquote expansion produces **'(CONS ,X (QUOTE NIL))**. It is easy to check that the readmacro expansion of this tree is as shown in the table.

Observe that backquote expansion does not always yield an s-expression: the

backquote expansion of **,,X** is **,X**. However, the backquote expansion of a token tree readable from depth 0 produces an s-expression because each escape token tree is embedded in enough backquotes to ensure that all the escape and backquote tokens are eliminated.

7.3. Some Preliminary Terminology

Note. We have described how readmacro expansion transforms readable token trees into s-expressions. We now turn to the identification of a subset of the s-expressions, called ''well-formed'' s-expressions and the formal terms they abbreviate. We need a few preliminary concepts first.

Terminology. The *NUMBERP corresponding* to a nonnegative integer n is the term **(ZERO)** if n is 0, and otherwise is the term **(ADD1 t)**, where **t** is the **NUMBERP** corresponding to n-1. The *NEGATIVEP corresponding* to a negative integer n is the term **(MINUS t)**, where **t** is the **NUMBERP** corresponding to -n.

Examples. The **NUMBERP** corresponding to 2 is **(ADD1 (ADD1 (ZERO)))**. The **NEGATIVEP** corresponding to **-**1 is **(MINUS (ADD1 (ZERO)))**.

Terminology. If **fn** is a **CAR/CDR** symbol, we call the sequence of characters in **fn** starting with the second and concluding with the next to last the *A/D sequence* of **fn**.

Terminology. If **s** is a character sequence of **A**'s and **D**'s, the *CAR/CDR nest* for **s** around a term **b** is the term **t** defined as follows. If **s** is empty, **t** is **b**. Otherwise, **s** consists of either an **A** or **D** followed by a sequence **s'**. Let **t'** be the **CAR/CDR** nest for **s'** around **b**. Then **t** is **(CAR t')** or **(CDR t')**, according to whether the first character of **s** is **A** or **D**.

Example. The symbol **CADDAAR** is a **CAR/CDR** symbol. Its **A/D** sequence is the sequence **ADDAA**. The **CAR/CDR** nest for **ADDAA** around **L** is **(CAR (CDR (CDR (CAR (CAR L)))))**.

Terminology. We say a term **e** is the *explosion of* a sequence of ASCII characters, **s**, iff either (a) **s** is empty and **e** is **(ZERO)** or (b) **s** is a character **c** followed by some sequence **s'** and **e** is **(CONS i e')** where **i** is the **NUMBERP** corresponding to the ASCII code for **c** and **e'** is the explosion of **s'**.

Example. The ASCII codes for the characters **A**, **B**, and **C** are 65, 66, and 67

respectively. Let **t$_{65}$**, **t$_{66}$**, and **t$_{67}$** denote, respectively, the **NUMBERP**s corresponding to 65, 66, and 67. For example, **t$_{65}$** here denotes a nest of **ADD1**s 65 deep with a **(ZERO)** at the bottom. Then the explosion of **ABC** is the formal term

(CONS t$_{65}$ (CONS t$_{66}$ (CONS t$_{67}$ (ZERO)))).

Terminology. We say the term **e** is the *LITATOM corresponding to* a symbol **s** iff **e** is the term **(PACK e')** where **e'** is the explosion of **s**.

Example. The **LITATOM** corresponding to the symbol **ABC** is

(PACK (CONS t$_{65}$ (CONS t$_{66}$ (CONS t$_{67}$ (ZERO))))),

where **t$_{65}$**, **t$_{66}$**, and **t$_{67}$** are as in the last example.

7.4. Well-Formed S-Expressions and Their Translations

Notes. The terminology we have developed is sufficient for defining what it means for an s-expression to be ''well-formed'' and what its ''translation'' is, for all s-expressions except those employing **QUOTE** or abbreviated **FOR**s. Rather than define the concepts necessary to pin down these conventions, we now jump ahead in our development of the syntax and define ''well-formed'' and ''translation.'' Such a presentation here necessarily involves undefined concepts—the notions of well-formedness and translation of both **QUOTE** and **FOR** expressions. However, by providing the definition at this point in the development we can use some s-expressions to illustrate and motivate the discussion of the more elaborate notations.

Terminology. Below we define two concepts: what it means for an s-expression **x** to be a *well-formed term in the extended syntax* and, for well-formed s-expressions, what is the *translation* into a term in the formal syntax. These definitions are made implicitly with respect to a history because **QUOTE** notation permits the abbreviation of explicit values, a concept which, recall, is sensitive to the history. Our style of definition is to consider any s-expression **x** and announce whether it is well-formed or not and if so, what its translation is.

- If **x** is an integer, it is well-formed and its translation is the explicit value term denoted by **x** (see page 39).
- If **x** is a symbol then
  - If **x** is **T**, it is well-formed and its translation is the formal term **(TRUE)**.

- If **x** is **F**, it is well-formed and its translation is the formal term **(FALSE)**.
- If **x** is **NIL**, it is well-formed and its translation is the explicit value term denoted by **NIL** (see page 39).
- If **x** is any other symbol, it is well-formed and its translation is the formal term **x**.

- If **x** is a dotted s-expression, it is not well-formed.
- If the first element of **x** is the symbol **QUOTE**, then **x** is well-formed iff it is of the form **(QUOTE e)** where **e** is an explicit value descriptor (see page 39). If well-formed, the translation of **x** is the explicit value term denoted by **e** (see page 39).
- If the first element of **x** is the symbol **COND**, then the well-formedness of **x** and its translation are defined by cases as follows.
  - If **x** is of the form **(COND (T v))**, then it is well-formed iff **v** is well-formed. If **x** is well-formed, the translation of **x** is the translation of **v**.
  - If **x** is of the form **(COND (w v)** x_1 **...** x_n**)**, where **n**>0, then **x** is well-formed iff **w** is not **T** and **w**, **v** and **(COND** x_1 **...** x_n**)** are well-formed. If **x** is well-formed, let **test**, **val**, and **rest** be, respectively, the translations of **w**, **v**, and **(COND** x_1 **...** x_n**)**. Then the translation of **x** is **(IF test val rest)**.
  - Otherwise, **x** is not well-formed.
- If the first element of **x** is the symbol **CASE**, then the well-formedness of **x** and its translation are defined by cases as follows.
  - If **x** is of the form **(CASE w (OTHERWISE v))**, then **x** is well-formed iff **w** and **v** are well-formed. If **x** is well-formed, the translation of **x** is the translation of **v**.
  - If **x** is of the form **(CASE w (e v)** x_1 **...** x_n**)**, where **n**>0, then **x** is well-formed iff no x_i is of the form **(e** x'_i**)** and in addition **w**, **(QUOTE e)**, **v**, and **(CASE w** x_1 **...** x_n**)** are well-formed. If **x** is well-formed, then let **key**, **obj**, **val**, and **rest** be, respectively, the translations of **w**, **(QUOTE e)**, **v**, and **(CASE w** x_1 **...** x_n**)**. Then the translation of **x** is **(IF (EQUAL key obj) val rest)**.
  - Otherwise, **x** is not well-formed.

- If the first element of **x** is the symbol **LET**, then **x** is well-formed iff **x** is of the form **(LET ((w$_1$ v$_1$) ... (w$_n$ v$_n$)) y)**, the **w$_i$**, **v$_i$** and **y** are well-formed, the translation of each **w$_i$** is a symbol and the translation of **w$_i$** is different from that of **w$_j$** when **i** is different from **j**. If **x** is well-formed, let **var$_i$** be the translation of **w$_i$**, let **t$_i$** be the translation of **v$_i$** and let **body** be the translation of **y**. Let **s** be the substitution that replaces **var$_i$** by **t$_i$**, i.e., {<**var$_1$**, **t$_1$**> ... <**var$_n$**, **t$_n$**>}. Then the translation of **x** is **body**/**s**.
- Otherwise, **x** is of the form **(fn x$_1$ ... x$_n$)**, where **fn** is a symbol other than **QUOTE**, **COND**, **CASE**, or **LET**, and each **x$_i$** is an s-expression. If some **x$_i$** is not well-formed, then **x** is not well-formed. Otherwise, let **t$_i$** be the translation of **x$_i$** below:
  - If **fn** is in the set {**NIL T F**}, **x** is not well-formed.
  - If **fn** is the symbol **LIST** then **x** is well-formed and its translation is the **CONS** nest around the translation of **NIL** for **t$_1$**, ..., **t$_n$**.
  - If **fn** is the symbol **LIST*** then **x** is well-formed iff **n**≥1. If **x** is well-formed, its translation is the **CONS** nest around **t$_n$** for **t$_1$**, ..., **t$_{n-1}$**.
  - If **fn** is a **CAR/CDR** symbol, then **x** is well-formed iff **n** is 1 and, if well-formed, its translation is the **CAR/CDR** nest around **t$_1$** for the **A/D** sequence of **fn**.
  - If **fn** is a function symbol of arity **n**, then **x** is well-formed and its translation is the formal term **(fn t$_1$ ... t$_n$)**.
  - If **fn** is the symbol **FOR** and n is 5 or 7, then **x** is well-formed iff **x** is an abbreviated **FOR** (see page 58). If well-formed, the translation of **x** is the **FOR** expression denoted by **x** (see page 58).
  - If **fn** is in the set {**AND OR PLUS TIMES**} and **n**>2, then **x** is well-formed and its translation is the **fn** nest around **t$_n$** for **t$_1$**, ..., **t$_{n-1}$**.
  - Otherwise, **x** is not well-formed.

Examples. Table 5 shows well-formed s-expressions on the left and their translations to formal terms on the right.

Certain formal terms, such as the translation of **NIL**, are exceedingly painful to write down because they contain deep nests of **ADD1**s. Table 6 also contains translations, except this time the right-hand column is, technically, not a formal

Table 5

s-expression	translation
`T`	`(TRUE)`
`2`	`(ADD1 (ADD1 (ZERO)))`

`(COND (P X)`
`(Q Y)`
`(T Z))`
	`(IF P X (IF Q Y Z))`

`(LIST* A B C)`	`(CONS A (CONS B C))`
`(CADR X)`	`(CAR (CDR X))`
`(PLUS I J K)`	`(PLUS I (PLUS J K))`

term but rather another well-formed s-expression with the same translation. In particular, in Table 6 we use decimal notation in the right-hand column, but otherwise confine ourselves to formal syntax.

Table 6

s-expression	s-expression with same translation

`NIL`	`(PACK`
	`(CONS 78`
	`(CONS 73`
	`(CONS 76 0))))`

`(LIST 1 2 3)`	`(CONS 1` ;first element
	`(CONS 2` ;second element
	`(CONS 3` ;third element
	`(PACK` ;NIL
	`(CONS 78`
	`(CONS 73`
	`(CONS 76 0)))))))`

7.5. QUOTE Notation

Notes and Example. In this subsection we define what we mean by an ‘‘explicit value descriptor’’ and the ‘‘explicit value denoted’’ by such a descriptor. These are the concepts used to define the well-formedness and meaning of s-expressions of the form **(QUOTE e)**.

Each explicit value term can be written in **QUOTE** notation. That is, for each explicit value term **t** there is exactly one s-expression **e** such that the s-expression **(QUOTE e)** is well-formed and translates to **t**. We call **e** the ‘‘explicit value descriptor’’ of **t**. For example, consider the s-expression

```
(CONS 1
      (CONS (PACK
             (CONS 65 (CONS 66 (CONS 67 0))))
            (CONS 2 3))).
```

This s-expression is well-formed and translates to an explicit value— indeed, except for the use of decimal notation, this s-expression is an explicit value. Call that explicit value term **t**. The explicit value descriptor for **t** is the s-expression **(1 ABC 2 . 3)**. Thus, the translation of **(QUOTE (1 ABC 2 . 3))** is **t**.

Our **QUOTE** notation is derived from the Lisp notation for data structures composed of numbers, symbols, and ordered pairs, but is complicated by the need to denote structures containing user-defined shell constants. That is, after the theory has been extended by the addition of a new shell, it is possible to build constants containing both primitive shells and user-defined ones, e.g., lists of stacks. Unlike Lisp’s **QUOTE** notation, the notation described here permits such constants to be written down, via an ‘‘escape’’ mechanism.

Following the precedent set for well-formedness and translation, we proceed in a top-down fashion to define what we mean by an ‘‘explicit value descriptor’’ and its ‘‘denoted explicit value’’ without first defining the terminology to discuss the ‘‘escape’’ mechanism. Immediately following the definition below we illustrate the use of **QUOTE** notation on primitive shell constants, e.g., lists, numbers, and literal atoms. We define the escape mechanism for user-declared shells in the next subsection.

Terminology. Below we define two concepts: what it is for an s-expression **e** to be an *explicit value descriptor* and, for explicit value descriptors, what is the *denoted explicit value term*. These definitions are made with respect to a history which is used implicitly below. Our style of definition is to consider any s-expression **e** and announce whether it is an explicit value descriptor or not and if so, what its denoted explicit value term is.

- If **e** is an integer, it is an explicit value descriptor and the explicit value term it denotes is the **NEGATIVEP** or **NUMBERP** corresponding to **e**, according to whether **e** is negative or not.
- If **e** is a word, then
  - If **e** is the word ***1*TRUE**, **e** is an explicit value descriptor and denotes the explicit value **(TRUE)**.
  - If **e** is the word ***1*FALSE**, **e** is an explicit value descriptor and denotes the explicit value **(FALSE)**.
  - If **e** is a symbol, **e** is an explicit value descriptor and denotes the **LITATOM** corresponding to **e**.
  - Otherwise, **e** is not an explicit value descriptor.
- If the first element of **e** is the word ***1*QUOTE**, **e** is an explicit value descriptor iff it is an explicit value escape descriptor (see the next subsection). If so, it has the form **(*1*QUOTE fn e_1 ... e_n)**, where **fn** is a constructor or base function symbol of arity **n** and each e_i is an explicit value descriptor denoting an explicit value t_i. The explicit value denoted by **e** is then **(fn t_1 ... t_n)**. (That this is, indeed, an explicit value is assured by the definition of ''explicit value escape descriptor.'')
- If **e** is a dotted s-expression of length 3, i.e., **(e_1 . e_2)**, then **e** is an explicit value descriptor iff each e_i is an explicit value descriptor. If so, let t_i be the explicit value denoted by e_i. Then the explicit value denoted by **e** is **(CONS t_1 t_2)**.
- If **e** is an s-expression of length 1, i.e., **(e_1)**, **e** is an explicit value descriptor iff e_1 is an explicit value descriptor. If so, let t_1 be the explicit value denoted by e_1 and let **nil** be the explicit value denoted by **NIL**. Then the explicit value denoted by **e** is **(CONS t_1 nil)**.
- Otherwise, either **e** is a dotted s-expression of length greater than 3 or **e** is a non-dotted s-expression of length greater than 1. Let e_1 be the first element of **e** and let e_2 be the sequence consisting of the remaining elements of **e**. Observe that e_1 and e_2 are both s-expressions. **e** is an explicit value descriptor iff each e_i is an explicit value descriptor. If so, let t_i be the explicit value denoted by e_i. Then the explicit value denoted by **e** is **(CONS t_1 t_2)**.

Examples. Table 7 illustrates the **QUOTE** notation. The two columns contain token trees rather than s-expressions simply to save space—we write **'x** in

place of the s-expression **(QUOTE x)**. Each token tree is readable and read-macro expands to a well-formed s-expression. The s-expressions of the left-hand column are all examples of **QUOTE** forms. The s-expressions of the right-hand column use **QUOTE** only to represent literal atoms. Corresponding s-expressions in the two columns have identical translations.

Table 7

token tree for s-expression in QUOTE notation	**token tree for s-expression with same translation**
`'123`	`123`
`'ABC`	`(PACK '(65 66 67 . 0))`
`'(65 66 67 . 0)`	`(CONS 65` `  (CONS 66 (CONS 67 0)))`

`'(PLUS I J)`	`(CONS 'PLUS`
	`  (CONS 'I`
	`  (CONS 'J 'NIL)))`

`'((I . 2) (J . 3))`	`(LIST (CONS 'I 2)` `  (CONS 'J 3))`

`'((A . *1*TRUE)`
`  (B . T))`
	`(LIST (CONS 'A (TRUE))`
	`  (CONS 'B`
	`  (PACK (CONS 84 0))))`

Note. Of particular note is the possible confusion of the meaning of the symbol **T** (and, symmetrically, of **F**) in s-expressions. If **T** is used ''outside a **QUOTE**'' it denotes **(TRUE)**. If **T** is used ''inside a **QUOTE**'' it denotes the literal atom whose ''print name'' is the single character **T**. To include **(TRUE)** among the elements of a **QUOTE**d list, the non-symbol ***1*TRUE** should be written.

If the s-expression **(QUOTE ABC)** is used as a term it denotes the term also denoted by **(PACK (QUOTE (65 66 67 . 0)))**. However, if the s-expression **(QUOTE ABC)** is used ''inside a **QUOTE**,'' i.e., as an explicit value descriptor as in the term **(QUOTE (QUOTE ABC))**, it denotes the term also denoted by **(LIST (QUOTE QUOTE) (QUOTE ABC))**. The translation of **(QUOTE ABC)** is a **LITATOM** constant; the translation of **(QUOTE (QUOTE ABC))** is a **LISTP** constant.

Here is another example illustrating the subtlety of the situation. The innocent reader may have, by now, adopted the convention that whenever **(CADR X)** is seen, **(CAR (CDR X))** is used in its place. This is incorrect. When **(CADR X)** is used as a term, i.e., when we are interested in its translation into a formal term, it denotes **(CAR (CDR X))**. But if **(CADR X)** is ''inside a **QUOTE**'' it is not being used as a term but rather as an explicit value descriptor. In particular, the translation of **(QUOTE (CADR X))** is a list whose first element is the **LITATOM** denoted by the term **(QUOTE CADR)**, not a list whose first element is the **LITATOM** denoted by **(QUOTE CAR)**. While the translations of **(CADR X)** and **(CAR (CDR X))** are the same, the translations of **(QUOTE (CADR X))** and **(QUOTE (CAR (CDR X)))** are different. Similarly the translation of **(QUOTE 1)** is not the same as that of **(QUOTE (ADD1 (ZERO)))**; the first is a **NUMBERP**, the second is a **LISTP**.

*7.6. *1*QUOTE Escape Mechanism for User Shells*

Notes and Example. In this section we describe how to use the ***1*QUOTE** convention to write down user-declared shell constants. In particular, we define the notion of the ''explicit value escape descriptor'' used above.

Roughly speaking, an explicit value escape descriptor is an s-expression of the form **(*1*QUOTE fn e_1 ... e_n)** where **fn** is a shell constructor or base function and the e_i are explicit value descriptors denoting its arguments. Thus, if **PUSH** is a shell constructor of two arguments and **EMPTY** is the corresponding base function then **(*1*QUOTE PUSH 3 (*1*QUOTE EMPTY))** is an explicit value escape descriptor, and hence an explicit value descriptor, that denotes the constant term also denoted by **(PUSH 3 (EMPTY))**. We restrict the legal escape descriptors so that the mechanism cannot be used to write down alternative representations of constants that can be written in the conventional **QUOTE** notation. For example, **(*1*QUOTE CONS 1 2)** is *not* an explicit value escape descriptor because if it were it would be an alternative representation of **(CONS 1 2)**. Furthermore, we must restrict the escape descriptors so that they indeed denote explicit values. Is **(PUSH 3 (EMPTY))** an explicit value? The answer depends upon the type restrictions for the **PUSH** shell. To answer this question it is necessary to know the current history.

Terminology. The s-expression **e** is an *explicit value escape descriptor* with respect to a history h iff **e** has the form **(*1*QUOTE fn e_1 ... e_n)** and each of the following is true:

- **fn** is a constructor or base function symbol of arity **n** in history h;

- **fn** is not **ADD1**, **ZERO**, or **CONS**;
- each $\mathbf{e_i}$ is an explicit value descriptor with corresponding denoted term $\mathbf{t_i}$ in h;
- if **fn** is a constructor, the top function symbol of each $\mathbf{t_i}$ satisfies the corresponding type restriction for **fn**;
- if **fn** is **PACK**, $\mathbf{t_1}$ is not the explosion of any symbol; and
- if **fn** is **MINUS**, $\mathbf{t_1}$ is **(ZERO)**.

Notes and Examples. We now illustrate the use of the ***1*QUOTE** escape mechanism. Suppose we are in a history obtained by extending the current one with the following:

Shell Definition.
Add the shell **PUSH** of 2 arguments
with base function **EMPTY**,
recognizer function **STACKP**,
accessor functions **TOP** and **POP**
type restrictions **(ONE-OF NUMBERP)** and **(ONE-OF STACKP)**, and
default functions **ZERO** and **EMPTY**.

Table 8 contains some example s-expressions employing the ***1*QUOTE** mechanism. To save space we again exhibit token trees whose readmacro expansions are the s-expressions in question.

Table 8

token tree for s-expression	token tree for s-expression with same translation
`'(A (*1*QUOTE MINUS 0))`	`(LIST 'A (MINUS 0))`
`'((*1*QUOTE PUSH 2 (*1*QUOTE PUSH 3 (*1*QUOTE EMPTY))) FOO . 45)`	`(CONS (PUSH 2 (PUSH 3 (EMPTY))) (CONS 'FOO 45))`
`'(*1*QUOTE PACK (97 98 99 . 0))`	`(PACK '(97 98 99 . 0))`

***1*QUOTE** can be used not only to denote constants of ''new'' types but also

to write down ‘‘unusual’’ constants of the primitive types, namely **(MINUS 0)** and ‘‘**LITATOM**s corresponding to non-symbols.’’

***1*QUOTE** notation is actually a direct reflection of the internal representation of shell constants in the Nqthm system. If **STACKP** constants, say, are allowed as terms, then it is desirable to have a unique representation of them that can be used in the representation of other constants as well. The representation we developed is the one suggested by ***1*QUOTE** notation. We did not have to make this implementation decision be visible to the user of the logic. We could have arranged for only the primitive data types to be abbreviated by **QUOTE** notation and all other constants built by the application of constructor and base functions. Making the convention visible is actually an expression of our opinion that the syntax should not hide too much from the user. For example, the user writing and verifying metafunctions will appreciate knowing the internal forms.

Finally, it should be noted that backquote notation can often be used where ***1*QUOTE** is otherwise needed. For example, **(QUOTE (A (*1*QUOTE EMPTY) B))** can also be written as **‘(A ,(EMPTY) B)**. The translation of the former is the same as the translation of the readmacro expansion of the latter.

7.7. The Definition of the Extended Syntax

Abbreviation. When a token tree that is readable from depth 0 and that is not an s-expression is used as an s-expression, we mean the s-expression obtained by readmacro expanding the token tree. Thus, henceforth, when we say something like ‘‘consider the s-expression **’ABC**’’ we mean ‘‘consider the s-expression **(QUOTE ABC)**.’’

Terminology. The *extended syntax* consists of the token trees readable from depth 0 whose readmacro expansions are well-formed.

Abbreviation. When an expression in the extended syntax is used as a term, it is an abbreviation for the translation of its readmacro expansion. When an expression in the extended syntax is used as a formula, it is an abbreviation for **t≠(FALSE)**, where **t** is the translation of the readmacro expansion of the expression.

]

8. Ordinals

Defining Axiom 22.

```
(LESSP X Y)
   =
(IF (ZEROP Y)
    F
    (IF (ZEROP X)
        T
        (LESSP (SUB1 X) (SUB1 Y))))
```

Axiom 23.

```
(IMPLIES (NOT (ZEROP I))
         (LESSP (SUB1 I) I))
```

Note. **Axiom 23** permits us to apply the induction principle to prove the fundamental properties of **LESSP**, **PLUS**, and **COUNT**, which in turn permit us to induct in more sophisticated ways.

Defining Axiom 24.

```
(ORD-LESSP X Y)
   =
(IF (NOT (LISTP X))
    (IF (NOT (LISTP Y))
        (LESSP X Y)
        T)
    (IF (NOT (LISTP Y))
        F
        (IF (ORD-LESSP (CAR X) (CAR Y))
            T
            (AND (EQUAL (CAR X) (CAR Y))
                 (ORD-LESSP (CDR X) (CDR Y))))))
```

Defining Axiom 25.

```
(ORDINALP X)
   =
(IF (LISTP X)
    (AND (ORDINALP (CAR X))
         (NOT (EQUAL (CAR X) 0))
         (ORDINALP (CDR X))
         (OR (NOT (LISTP (CDR X)))
             (NOT (ORD-LESSP (CAR X) (CADR X)))))
    (NUMBERP X))
```

9. Useful Function Definitions

9.1. Boolean Equivalence

Defining Axiom 26.

```
(IFF P Q)
   =
(IF P
    (IF Q T F)
    (IF Q F T))
```

9.2. Natural Number Arithmetic

Defining Axiom 27.

```
(GREATERP I J) = (LESSP J I)
```

Defining Axiom 28.

```
(LEQ I J) = (NOT (LESSP J I))
```

Defining Axiom 29.

```
(GEQ I J) = (NOT (LESSP I J))
```

Defining Axiom 30.

```
(MAX I J) = (IF (LESSP I J) J (FIX I))
```

Defining Axiom 31.

```
(DIFFERENCE I J)
  =
(IF (ZEROP I)
    0
    (IF (ZEROP J)
        I
        (DIFFERENCE (SUB1 I) (SUB1 J))))
```

Defining Axiom 32.

```
(TIMES I J)
   =
(IF (ZEROP I)
    0
    (PLUS J (TIMES (SUB1 I) J)))
```

Defining Axiom 33.

```
(QUOTIENT I J)
   =
```

```
(IF (ZEROP J)
    0
    (IF (LESSP I J)
        0
        (ADD1 (QUOTIENT (DIFFERENCE I J) J))))
```

Defining Axiom 34.

```
(REMAINDER I J)
   =
(IF (ZEROP J)
    (FIX I)
    (IF (LESSP I J)
        (FIX I)
        (REMAINDER (DIFFERENCE I J) J)))
```

9.3. List Processing

Defining Axiom 35.

```
(NLISTP X) = (NOT (LISTP X))
```

Defining Axiom 35a.

```
(IDENTITY X) = X
```

Defining Axiom 36.

```
(APPEND L1 L2)
   =
(IF (LISTP L1)
    (CONS (CAR L1) (APPEND (CDR L1) L2))
    L2)
```

Defining Axiom 37.

```
(MEMBER X L)
   =
(IF (NLISTP L)
    F
    (IF (EQUAL X (CAR L))
        T
        (MEMBER X (CDR L))))
```

Defining Axiom 38.

```
(UNION L1 L2)
   =
(IF (LISTP L1)
    (IF (MEMBER (CAR L1) L2)
        (UNION (CDR L1) L2)
```

```
        (CONS (CAR L1) (UNION (CDR L1) L2)))
    L2)
```

Defining Axiom 39.

```
(ADD-TO-SET X L)
   =
(IF (MEMBER X L)
    L
    (CONS X L))
```

Defining Axiom 40.

```
(ASSOC X ALIST)
   =
(IF (NLISTP ALIST)
    F
    (IF (EQUAL X (CAAR ALIST))
        (CAR ALIST)
        (ASSOC X (CDR ALIST))))
```

Defining Axiom 41.

```
(PAIRLIST L1 L2)
   =
(IF (LISTP L1)
    (CONS (CONS (CAR L1) (CAR L2))
          (PAIRLIST (CDR L1) (CDR L2)))
    NIL)
```

10. The Formal Metatheory

Note. In this section we describe the interpreter for the logic. We start by presenting the notion of the ‘‘quotation’’ of terms. Roughly speaking, the quotation of a term is an explicit value that has a structure isomorphic to that of the term; for example, the quotation of **(PLUS X Y)** is the explicit value **'(PLUS X Y)**. An important property of quotations is that, for most terms, the interpreted value of the quotation under a certain standard assignment is equal to the term. For example, the value of **'(PLUS X Y)** as determined by our interpreter, when **'X** has the value **X** and **'Y** has the value **Y**, is **(PLUS X Y)**. After defining quotations we define the interpreter. Finally, we describe the **SUBRP** and non-**SUBRP** axioms that tie **QUOTE**d symbols to the interpreter.

10.1. The Quotation of Terms

Note. The ‘‘quotation’’ of an explicit value term may be rendered either by nests of constructor function applications or by embedding the term in a **QUOTE** form. This makes the notion of ‘‘quotation’’ depend upon the notion of ‘‘explicit value,’’ which, recall, involves a particular history h from which the constructor and base functions are drawn. This is the only sense in which the notion of ‘‘quotation’’ depends upon a history.

Terminology. We say **e** is a *quotation* of **t** (in some history h which is implicit throughout this definition) iff **e** and **t** are terms and either (a) **t** is a variable symbol and **e** is the **LITATOM** corresponding to **t**, (b) **t** is an explicit value term and **e** is **(LIST 'QUOTE t)**, or (c) **t** has the form **(fn a_1 ... a_n)** and **e** is **(CONS efn elst)** where **efn** is the **LITATOM** corresponding to **fn** and **elst** is a ‘‘quotation list’’ (see below) of **a_1** ... **a_n**. Note that clauses (b) and (c) are not mutually exclusive.

Terminology. We say **elst** is a *quotation list of* **tlst** (in some history h which is implicit throughout this definition) iff **elst** is a term and **tlst** is a sequence of terms, and either (a) **tlst** is empty and **elst** is **NIL** or (b) **tlst** consists of a term **t** followed by a sequence **tlst'** and **elst** is **(CONS e elst')** where **e** is a quotation of **t** and **elst'** is a quotation list of **tlst'**.

Examples. In Table 9 we give some terms and examples of their quotations.

Table 9

term	quotation displayed in the extended syntax
`ABC`	`'ABC`
`(ZERO)`	`'(ZERO)`
`(ZERO)`	`'(QUOTE 0)`
`(PLUS 3 (TIMES X Y))`	`'(PLUS (QUOTE 3) (TIMES X Y))`

Note. To describe the axioms for the **BODY** function, we wish to say something like ‘‘for each defined function symbol, **fn**, **(BODY 'fn)** is the quotation of the body of the definition of **fn**.’’ But note that explicit values, e.g., **(ZERO)**

above, have multiple quotations. (Indeed, all terms containing explicit values have multiple quotations.) Consequently, we cannot speak of ‘‘the’’ quotation of a term. To get around this we define the notion of the ‘‘preferred quotation.’’ The preferred quotation of **(ZERO)** is **'(QUOTE 0)**. In general, the definitions of ‘‘preferred quotation’’ and ‘‘preferred quotation list,’’ below, are strictly analogous to the definitions of ‘‘quotation’’ and ‘‘quotation list,’’ above, except that explicit values must be encoded in **'QUOTE** form. This is done by making clauses (b) and (c) of the definition of ‘‘quotation’’ be mutually exclusive with clause (b) the superior one.

Terminology. We say **e** is the *preferred quotation* of **t** (in some history h which is implicit throughout this definition) iff **e** and **t** are terms and either (a) **t** is a variable symbol and **e** is the **LITATOM** corresponding to **t**, (b) **t** is an explicit value term and **e** is **(LIST 'QUOTE t)**, or (c) **t** has the form **(fn a_1 ... a_n)**, **t** is not an explicit value, and **e** is **(CONS efn elst)** where **efn** is the **LITATOM** corresponding to **fn** and **elst** is the ‘‘preferred quotation list’’ (see below) of a_1 ... a_n.

Terminology. We say **elst** is the *preferred quotation list of* **tlst** (in some history h which is implicit throughout this definition) iff **elst** is a term and **tlst** is a sequence of terms, and either (a) **tlst** is empty and **elst** is **NIL** or (b) **tlst** consists of a term **t** followed by a sequence **tlst'** and **elst** is **(CONS e elst')** where **e** is the preferred quotation of **t** and **elst'** is the preferred quotation list of **tlst'**.

10.2. V&C$ and EVAL$

Note. The axiomatization of **V&C$** and **EVAL$** are rather subtle.

Defining Axiom 42.

```
(FIX-COST VC N)
   =
(IF VC
    (CONS (CAR VC) (PLUS N (CDR VC)))
    F)
```

Defining Axiom 43.

```
(STRIP-CARS L)
   =
(IF (NLISTP L)
    NIL
    (CONS (CAAR L) (STRIP-CARS (CDR L))))
```

Defining Axiom 44.

```
(SUM-CDRS L)
    =
(IF (NLISTP L)
    0
    (PLUS (CDAR L) (SUM-CDRS (CDR L))))
```

Note. We now ‘‘define’’ **V&C$**. This axiom defines a partial function and would not be admissible under the definitional principle. Because of its complexity we include comments in the axiom.

Defining Axiom 45.

```
(V&C$ FLG X VA)
    =
(IF (EQUAL FLG 'LIST)

    ;X is a list of terms.  Return a list of value-cost
    ;‘‘pairs’’—some ‘‘pairs’’ may be F.

    (IF (NLISTP X)
        NIL
        (CONS (V&C$ T (CAR X) VA)
              (V&C$ 'LIST (CDR X) VA)))


    ;Otherwise, consider the cases on the X.

(IF (LITATOM X)                        ;Variable
    (CONS (CDR (ASSOC X VA)) 0)

(IF (NLISTP X)                         ;Constant
    (CONS X 0)

(IF (EQUAL (CAR X) 'QUOTE)             ;QUOTEd
    (CONS (CADR X) 0)

(IF (EQUAL (CAR X) 'IF)                ;IF-expr

    ;If the test of the IF is defined, test the value and
    ;interpret the appropriate branch.  Then, if the branch
    ;is defined, increment its cost by that of the test plus
    ;one.  If the test is undefined, X is undefined.

    (IF (V&C$ T (CADR X) VA)
        (FIX-COST
```

```
        (IF (CAR (V&C$ T (CADR X) VA))
            (V&C$ T (CADDR X) VA)
            (V&C$ T (CADDDR X) VA))
        (ADD1 (CDR (V&C$ T (CADR X) VA))))
       F)
```

;Otherwise, **X** is the application of a **SUBRP** or
;defined function. If some argument is undefined, so is **X**.

```
(IF (MEMBER F (V&C$ 'LIST (CDR X) VA))
    F

(IF (SUBRP (CAR X))                    ;SUBRP
```

;Apply the primitive to the values of the arguments and
;let the cost be one plus the sum of the argument costs.

```
    (CONS (APPLY-SUBR (CAR X)
                      (STRIP-CARS
                         (V&C$ 'LIST (CDR X) VA)))
          (ADD1 (SUM-CDRS
                    (V&C$ 'LIST (CDR X) VA))))


                                           ;Defined fn
```

;Interpret the **BODY** on the values of the arguments
;and if that is defined increment the cost by one plus
;the sum of the argument costs.

```
    (FIX-COST
     (V&C$ T (BODY (CAR X))
             (PAIRLIST
               (FORMALS (CAR X))
               (STRIP-CARS (V&C$ 'LIST (CDR X) VA))))
     (ADD1
       (SUM-CDRS
         (V&C$ 'LIST (CDR X) VA)))))))))))
```

Note. Having defined **V&C$** we can now define the general purpose ''apply'' function in terms of it:

Defining Axiom 46.
(V&C-APPLY$ FN ARGS)
=

```
(IF (EQUAL FN 'IF)
 (IF (CAR ARGS)
     (FIX-COST (IF (CAAR ARGS)
                   (CADR ARGS)
                   (CADDR ARGS))
               (ADD1 (CDAR ARGS)))
     F)
 (IF (MEMBER F ARGS)
     F
     (IF (SUBRP FN)
         (CONS (APPLY-SUBR
                  FN
                  (STRIP-CARS ARGS))
               (ADD1 (SUM-CDRS ARGS)))
         (FIX-COST
           (V&C$ T
                 (BODY FN)
                 (PAIRLIST (FORMALS FN)
                           (STRIP-CARS ARGS)))
           (ADD1 (SUM-CDRS ARGS))))))
```

Note. A trivial consequence of the definitions of **V&C$** and **V&C-APPLY$** is that the following is a theorem:

Theorem.

```
(V&C$ FLG X VA)
   =
(IF (EQUAL FLG 'LIST)
    (IF (NLISTP X)
        NIL
        (CONS (V&C$ T (CAR X) VA)
              (V&C$ 'LIST (CDR X) VA)))
    (IF (LITATOM X) (CONS (CDR (ASSOC X VA)) 0)
        (IF (NLISTP X) (CONS X 0)
            (IF (EQUAL (CAR X) 'QUOTE)
                (CONS (CADR X) 0)
                (V&C-APPLY$
                  (CAR X)
                  (V&C$ 'LIST (CDR X) VA))))))
```

Note. We finally define the functions **APPLY$** and **EVAL$**:

Defining Axiom 47.

```
(APPLY$ FN ARGS)
   =
```

```
(CAR (V&C-APPLY$ FN (PAIRLIST ARGS 0)))
```

Defining Axiom 48.

```
(EVAL$ FLG X A)
   =
(IF (EQUAL FLG 'LIST)
 (IF (NLISTP X)
     NIL
     (CONS (EVAL$ T (CAR X) A)
           (EVAL$ 'LIST (CDR X) A)))
 (IF (LITATOM X) (CDR (ASSOC X A))
     (IF (NLISTP X) X
         (IF (EQUAL (CAR X) 'QUOTE) (CADR X)
             (APPLY$ (CAR X)
                     (EVAL$ 'LIST (CDR X) A))))))
```

10.3. The SUBRP and non-SUBRP Axioms

Notes. We now axiomatize the functions **SUBRP**, **APPLY-SUBR**, **FORMALS**, and **BODY** and define what we mean by the ''**SUBRP**'' and ''non-**SUBRP** axioms.''

The function **SUBRP** is Boolean:

Axiom 49.

```
(OR (EQUAL (SUBRP FN) T) (EQUAL (SUBRP FN) F))
```

The three functions **SUBRP**, **FORMALS**, and **BODY** ''expect'' **LITATOM**s as arguments, i.e., the quotations of function symbols. We tie down the three functions outside their ''expected'' domain with the following three axioms:

Axiom 50.

```
(IMPLIES (NOT (LITATOM FN)) (EQUAL (SUBRP FN) F))
```

Axiom 51.

```
(IMPLIES (NOT (LITATOM FN)) (EQUAL (FORMALS FN) F))
```

Axiom 52.

```
(IMPLIES (NOT (LITATOM FN)) (EQUAL (BODY FN) F))
```

Note. In a similar spirit, we define the **FORMALS** and **BODY** of **SUBRP**s to be **F**, and we define the result of applying a non-**SUBRP** with **APPLY-SUBR** to be **F**:

Axiom 53.

```
(IMPLIES (SUBRP FN) (EQUAL (FORMALS FN) F))
```

Axiom 54.
```
(IMPLIES (SUBRP FN) (EQUAL (BODY FN) F))
```

Axiom 55.
```
(IMPLIES (NOT (SUBRP FN)) (EQUAL (APPLY-SUBR FN X) F))
```

Note. In section 12 we enumerate the primitive **SUBRP**s and non-**SUBRP**s. For each we will add either the ‘‘**SUBRP** axiom’’ or the ‘‘non-**SUBRP** axiom,’’ which we now proceed to define.

Terminology. We say term **t** is the **n***th CDR nest around* the term **x** iff **n** is a natural number and either (a) **n** is 0 and **t** is **x** or (b) **n**>0 and **t** is **(CDR t’)** where **t’** is the **n**-1st **CDR** nest around **x**. When we write **(CDR^{n} x)** where a term is expected it is an abbreviation for the **n**th **CDR** nest around **x**.

Example. **(CDR^{2} A)** is **(CDR (CDR A))**.

Terminology. The *SUBRP axiom* for **fn**, where **fn** is a function symbol of arity **n**, is

```
(AND (EQUAL (SUBRP 'fn) T)
     (EQUAL (APPLY-SUBR 'fn L)
            (fn (CAR (CDR^0 L))
                ...
                (CAR (CDR^n-1 L)))))
```

where **’fn** is the **LITATOM** corresponding to **fn**.

Example. The **SUBRP** axiom for **PLUS** is

```
(AND (EQUAL (SUBRP 'PLUS) T)
     (EQUAL (APPLY-SUBR 'PLUS L)
            (PLUS (CAR L) (CAR (CDR L)))))
```

Terminology. The *standard alist* for a sequence of variable symbols **args** is **NIL** if **args** is empty and otherwise is **(CONS (CONS ’v v) alist)** where **v** is the first symbol in **args**, **’v** is the **LITATOM** corresponding to **v**, and **alist** is the standard alist for the sequence of symbols obtained by deleting **v** from **args**.

Example. The standard alist for **X**, **ANS**, and **L** is

```
(LIST (CONS 'X X)
      (CONS 'ANS ANS)
      (CONS 'L L))
```

Terminology. The *non-SUBRP axiom* for **fn**, **args**, and **body**, where **fn** is a function symbol, **args** is a sequence of variable symbols, and **body** is a term, is

```
(AND (EQUAL (SUBRP 'fn) F)
     (EQUAL (FORMALS 'fn) eargs)
     (EQUAL (BODY 'fn) ebody))
```

where **'fn** is the **LITATOM** corresponding to **fn**, **eargs** is the quotation list for **args**, and **ebody** is the preferred quotation of **body** unless **body** has the form **(EVAL$ flg ebody1 alist)** where

1. **flg** is an explicit value other than **'LIST**;
2. **ebody1** is an explicit value that is a quotation of some term **body1**;
3. **alist** is the standard alist for **args**; and
4. the set of variables in **body1** is a subset of those in **args**,

in which case **ebody** is the preferred quotation of **body1**.

Examples. The non-**SUBRP** axiom for **ADD2**, **(X Y)**, and **(PLUS 2 X Y)** is

```
(AND (EQUAL (SUBRP 'ADD2) F)
     (EQUAL (FORMALS 'ADD2) '(X Y))
     (EQUAL (BODY 'ADD2)
            '(PLUS (QUOTE 2) (PLUS X Y)))).
```

The non-**SUBRP** axiom for **RUS**, **()**, and

```
(EVAL$ T '(ADD1 (RUS)) NIL)
```

is

```
(AND (EQUAL (SUBRP 'RUS) F)
     (EQUAL (FORMALS 'RUS) NIL)
     (EQUAL (BODY 'RUS) '(ADD1 (RUS)))).
```

11. Quantification

11.1. The Definition of FOR and its Subfunctions

Defining Axiom 56.

```
(QUANTIFIER-INITIAL-VALUE OP)
   =
(CDR (ASSOC OP '((ADD-TO-SET . NIL)
                 (ALWAYS . *1*TRUE)
                 (APPEND . NIL)
                 (COLLECT . NIL)
                 (COUNT . 0)
                 (DO-RETURN . NIL)
                 (EXISTS . *1*FALSE)
                 (MAX . 0)
                 (SUM . 0)
                 (MULTIPLY . 1)
                 (UNION . NIL))))
```

Defining Axiom 57.

```
(QUANTIFIER-OPERATION OP VAL REST)
   =
(IF (EQUAL OP 'ADD-TO-SET) (ADD-TO-SET VAL REST)
(IF (EQUAL OP 'ALWAYS)     (AND VAL REST)
(IF (EQUAL OP 'APPEND)     (APPEND VAL REST)
(IF (EQUAL OP 'COLLECT)    (CONS VAL REST)
(IF (EQUAL OP 'COUNT)      (IF VAL (ADD1 REST) REST)
(IF (EQUAL OP 'DO-RETURN)  VAL
(IF (EQUAL OP 'EXISTS)     (OR VAL REST)
(IF (EQUAL OP 'MAX)        (MAX VAL REST)
(IF (EQUAL OP 'SUM)        (PLUS VAL REST)
(IF (EQUAL OP 'MULTIPLY)   (TIMES VAL REST)
(IF (EQUAL OP 'UNION)      (UNION VAL REST)
                           0))))))))))))
```

Defining Axiom 58.

```
(FOR V L COND OP BODY A)
   =
(IF (NLISTP L)
    (QUANTIFIER-INITIAL-VALUE OP)
    (IF (EVAL$ T COND (CONS (CONS V (CAR L)) A))
        (QUANTIFIER-OPERATION OP
           (EVAL$ T BODY (CONS (CONS V (CAR L)) A))
           (FOR V (CDR L) COND OP BODY A))
        (FOR V (CDR L) COND OP BODY A)))
```

11.2. The Extended Syntax for FOR—Abbreviations II

Note. This section completes the precise specification of the extended syntax by defining when an s-expression is an ''abbreviated **FOR**'' and the ''**FOR** expression denoted'' by such an s-expression.

Terminology. An s-expression **x** of the form **(FOR v IN lst WHEN cond op body)**—i.e., **x** is an s-expression of length eight whose first element is the word **FOR**, third element is the word **IN**, and fifth element is the word **WHEN**—is an *abbreviated FOR* iff each of the following is true:

- **v** is a variable symbol,
- **lst**, **cond**, and **body** are well-formed s-expressions whose translations are the terms **t-lst**, **t-cond**, and **t-body**, and
- **op** is an element of the set {**ADD-TO-SET ALWAYS APPEND COLLECT COUNT DO-RETURN EXISTS MAX SUM MULTIPLY UNION**}.

The *FOR expression denoted* by such an **x** is **(FOR 'v t-lst 't-cond 'op 't-body alist)** where **'v**, **'t-cond**, **'op**, and **'t-body** are the preferred quotations (see page 50) of **v**, **t-cond**, **op**, and **t-body** respectively, and **alist** is the standard alist (see page 55) on the sequence of variable symbols obtained by deleting **v** from the union of the variable symbols of **t-cond** with those of **t-body** and then sorting the resulting set alphabetically. An s-expression of the form **(FOR x IN lst op body)** is an *abbreviated FOR* iff **(FOR x IN lst WHEN T op body)** is an abbreviated **FOR** and, if so, denotes the same **FOR** expression as that denoted by **(FOR x IN lst WHEN T op body)**. No other form of s-expression is an *abbreviated FOR*.

12. SUBRPs and non-SUBRPs

Note. The symbol **QUOTE**, which is treated specially by **V&C$** and cannot be defined by the user, is not a **SUBRP**.

Axiom 59.
```
(NOT (SUBRP 'QUOTE)).
```

Axioms 60-64. We now add the non-**SUBRP** axiom for each of the following five function symbols: **APPLY$**, **EVAL$**, **V&C$**, **V&C-APPLY$**, and **FOR**. Each of these symbols was introduced with a defining axiom of the form **(fn** x_1 **...** x_n**)** = **body**. For each of the five function symbols we add the non-**SUBRP** axiom for **fn**, **(**x_1 **...** x_n**)**, and **body**.

Axioms 65-121. We add the **SUBRP** axiom for every other function symbol that is mentioned in an axiom of the current theory. The complete list of **SUBRP**s is: **ADD1**, **ADD-TO-SET**, **AND**, **APPEND**, **APPLY-SUBR**, **ASSOC**, **BODY**, **CAR**, **CDR**, **CONS**, **COUNT**, **DIFFERENCE**, **EQUAL**, **FALSE**, **FALSEP**, **FIX**, **FIX-COST**, **FORMALS**, **GEQ**, **GREATERP**, **IDENTITY**, **IF**, **IFF**, **IMPLIES**, **LEQ**, **LESSP**, **LISTP**, **LITATOM**, **MAX**, **MEMBER**, **MINUS**, **NEGATIVEP**, **NEGATIVE-GUTS**, **NLISTP**, **NOT**, **NUMBERP**, **OR**, **ORDINALP**, **ORD-LESSP**, **PACK**, **PAIRLIST**, **PLUS**, **QUANTIFIER-INITIAL-VALUE**, **QUANTIFIER-OPERATION**, **QUOTIENT**, **REMAINDER**, **STRIP-CARS**, **SUB1**, **SUBRP**, **SUM-CDRS**, **TIMES**, **TRUE**, **TRUEP**, **UNION**, **UNPACK**, **ZERO**, and **ZEROP**.

13. Induction and Recursion

13.1. Induction

Rule of Inference. Induction

Suppose

(a) **p** is a term;
(b) **m** is a term;
(c) $\mathbf{q_1}$, ..., $\mathbf{q_k}$ are terms;
(d) $\mathbf{h_1}$, ..., $\mathbf{h_k}$ are positive integers;
(e) it is a theorem that **(ORDINALP m)**;
and
(f) for $1 \le \mathbf{i} \le \mathbf{k}$ and $1 \le \mathbf{j} \le \mathbf{h_i}$, $\mathbf{s_{i,j}}$
is a substitution and it is a theorem that

$$\mathbf{(IMPLIES\ q_i\ (ORD\text{-}LESSP\ m/s_{i,j}\ m))}.$$

Then **p** is a theorem if

(1) $\mathbf{(IMPLIES\ (AND\ (NOT\ q_1)\ ...\ (NOT\ q_k))\otimes T\ p)}$

is a theorem and

for each $1 \le \mathbf{i} \le \mathbf{k}$,

(2) $\mathbf{(IMPLIES\ (AND\ q_i\ p/s_{i,1}\ ...\ p/s_{i,h_i})\otimes T}$

p)

is a theorem.

Notes. In informally describing an application of the induction principle to some conjecture **p** we generally say the induction is *on* the variables occurring in the term m, which is called the *measure*. An inductive proof splits the problem into **k+1** cases, a *base case*, given by formula (1) above, and **k** *induction steps*, given by the **k** formulas of form (2) above. The *cases* are given by the $\mathbf{q_i}$. The $\mathbf{i}^{th}$ induction step requires proving **p** under case $\mathbf{q_i}$. The base case requires proving **p** under the conjunction of the negations of the $\mathbf{q_i}$. In the $\mathbf{i}^{th}$ induction step one may assume an arbitrary number (namely $\mathbf{h_i}$) of instances, $\mathbf{p/s_{i,j}}$, of the conjecture being proved. The $\mathbf{j}^{th}$ instance for the $\mathbf{i}^{th}$ case is given by substitution $\mathbf{s_{i,j}}$. Each instance is called an *induction hypothesis*. To *justify* an induction one must show in the theorems of supposition (f) that some ordinal measure **m** of the induction variables decreases under each substitution in each respective case.

13.2. The Principle of Definition

Terminology. We say that a term **t** *governs* an occurrence of a term **s** in a term **b** iff (a) either **b** contains a subterm of the form **(IF t p q)** and the occurrence of **s** is in **p** or (b) if **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.

Extension Principle. Definition

The axiomatic act

Definition. **(f x_1 ... x_n)** = **body**

is admissible under the history h provided

(a) **f** is a function symbol of **n** arguments and
is new in h;

(b) x_1, ..., x_n are distinct variables;

(c) **body** is a term and mentions no symbol as a
variable other than x_1, ..., x_n;

(d) there is a term m such that (a) **(ORDINALP m)**
can be proved directly in h, and (b) for each
occurrence of a subterm of the form **(f y_1 ... y_n)**
in **body** and the terms t_1, ..., t_k governing it,
the following formula can be proved directly in h:

(IMPLIES (AND t_1 ... t_k)⊗T
(ORD-LESSP m/s m)),

where **s** is the substitution {<x_1, y_1> ... <x_n, y_n>}.

If admissible, we add the

Defining Axiom.
(f x_1 ... x_n) = **body.**

In addition, we add

Axiom.
the non-**SUBRP** axiom for **f**, **(x_1, ..., x_n)**, and **body.**

14. Contraints and Functional Instantiation

Notes. In this section we describe another extension principle and a new rule of inference: the introduction of function symbols by ''constraint'' and derivation by ''functional instantiation.'' The validity of these principles can be derived from the foregoing material, but because they change the ''flavor'' of the logic by permitting certain apparently higher-order acts we prefer to include their description here.

Functional instantiation permits one to infer new theorems from old ones by instantiating function symbols instead of variables. To be sure that such an instantiation actually produces a theorem, we first check that the formulas that result from similarly instantiating certain of the axioms about the function symbols being replaced are also theorems. Intuitively speaking, the correctness of this derived rule of inference consists of little more than the trivial observation that one may systematically change the name of a function symbol to a new name in a first-order theory without losing any theorems, modulo the renaming. However, we have found that this trivial observation has important potential practical ramifications in reducing mechanical proof efforts. We also find that this observation leads to superficially shocking results, such as the proof of the associativity of **APPEND** by instantiation rather than induction. And finally, we are intrigued by the extent to which such techniques permit one to capture the power of higher-order logic within first-order logic.

In order to make effective use of functional instantiation, we have found it necessary to augment the logic with an extension principle that permits the introduction of new function symbols without completely characterizing them. This facility permits one to add axioms about a set of new function symbols, provided one can exhibit ''witnessing'' functions that have the alleged properties. The provision for witnesses ensures that the new axioms do not render the logic inconsistent.

An example of the use of such constraints is to introduce a two-place function symbol, **FN**, constrained to be commutative. Any commutative function, e.g., a constant function of two arguments, suffices to witness the new axiom about **FN**. One can then prove theorems about **FN**. These theorems necessarily depend upon only one fact about **FN**, the fact that it is commutative. Thus, no matter how complicated these theorems are to prove, the analogous theorems about some other function symbol can be inferred via functional instantiation at the mere cost of confirming that the new symbol is commutative.

Terminology. A *LAMBDA expression* is an s-expression of the form **(LAMBDA (**$\mathbf{a_1}$ **...** $\mathbf{a_n}$**) body)**, where the $\mathbf{a_i}$ are distinct variable symbols and **body** is a term. The *arity* of such a **LAMBDA** expression is **n**, its *argument list* is the sequence $\mathbf{a_1}$, ..., $\mathbf{a_n}$, and its *body* is **body**.

Terminology. A *functional substitution* is a function on a finite set of function symbols such that for each pair <**f**, **g**> in the substitution, **g** is either a function symbol or **LAMBDA** expression and the arity of **f** is the arity of **g**.

Terminology. We recursively define the *functional instantiation of a term* **t** *under a functional substitution* fs. If **t** is a variable, the result is **t**. If **t** is the term **(f t_1 ... t_n)**, let t_i' be the functional instantiation of t_i, for i from **1** to **n** inclusive, under fs. If, for some function symbol **f'**, the pair <**f**, **f'**> is in fs, the result is **(f' t_1' ... t_n')**. If a pair <**f**, **(LAMBDA (a_1 ... a_n) term)**> is in fs, the result is **term**/{... <a_i, t_i'> ...}. Otherwise, the result is **(f t_1' ... t_n')**.

Note. Recall that ''**term**/σ'' denotes the result of applying the ordinary (variable) substitution σ to **term**. If σ is the variable substitution {<**X**, **(FN A)**> <**Y**, **B**>}, then **(PLUS X Y)**/σ is **(PLUS (FN A) B)**.

Example. The functional instantiation of the term **(PLUS (FN X) (TIMES Y Z))** under the functional substitution {<**PLUS**, **DIFFERENCE**> <**FN**, **(LAMBDA (V) (QUOTIENT V A))**>} is the term **(DIFFERENCE (QUOTIENT X A) (TIMES Y Z))**.

Terminology. We recursively define the *functional instantiation of a formula* ϕ *under a functional substitution* fs. If ϕ is $\phi_1 \vee \phi_2$, then the result is $\phi_1' \vee \phi_2'$, where ϕ_1' and ϕ_2' are the functional instantiations of ϕ_1 and ϕ_2 under fs. If ϕ is $\neg\phi_1$, then the result is $\neg\phi_1'$, where ϕ_1' is the functional instantiation of ϕ_1 under fs. If ϕ is **(x = y)**, then the result is **(x' = y')**, where **x'** and **y'** are the functional instantiations of **x** and **y** under fs.

Terminology. A variable **v** is said to be *free* in **(LAMBDA (a_1 ... a_n) term)** if and only if **v** is a variable of **term** and **v** is not among the a_i. A variable **v** is said to be *free* in a functional substitution if and only if it is free in a **LAMBDA** expression in the range of the substitution. A variable **v** is said to be *bound* in **(LAMBDA (a_1 ... a_n) term)** if and only if **v** is among the a_i.

Terminology. We denote functional instantiation with \ to distinguish it from ordinary (variable) substitution, which is denoted with /.

Example. If ρ is the functional substitution {<**PLUS**, **(LAMBDA (U V) (ADD1 U))**>} then **(PLUS X Y)**\ρ is **(ADD1 X)**.

Extension Principle. Conservatively constraining new function symbols.

The axiomatic act

Constraint.
Constrain **f**$_1$, ..., and **f**$_n$ to have property **ax**,

is admissible under the history h provided there exists a functional substitution fs with domain {**f**$_1$ ... **f**$_n$} such that

1. the **f**$_i$ are distinct new function symbols,
2. each member of the range of fs is either an old function symbol or is a **LAMBDA** expression whose body is formed of variables and old function symbols,
3. no variable is free in fs, and
4. **ax**\fs is a theorem of h.

If admissible, the act adds the axiom **ax** to the history h. The image of **f**$_i$ under fs is called the *witness* for **f**$_i$.

Note. Unlike the shell and definitional events, the constraint event does not add any **SUBRP** or non-**SUBRP** axioms about the new function symbols.

Terminology. A functional substitution fs is *tolerable* with respect to a history h provided that the domain of fs contains only function symbols introduced into h by the user via extension principles other than the shell mechanism.

Notes. We do not want to consider functionally substituting for built-in function symbols or shell function symbols because the axioms about them are so diffuse in the implementation. We especially do not want to consider substituting for such function symbols as **ORD-LESSP**, because they are used in the principle of induction.

The rule of functional instantiation requires that we prove appropriate functional instances of certain axioms. Roughly speaking, we have to prove that the ''new'' functions satisfy all the axioms about the ''old'' ones. However we must be careful to avoid ''capture''-like problems. For example, if the axiom constraining an old function symbol happens to involve a variable that is free in the functional substitution, then the functional instantiation of that axiom may be a weaker formula than the soundness argument in [1] requires. We illustrate this problem in the example below.

Example. Imagine that **ID1** is a function of two arguments that always returns its first argument. Let the defining axiom for **ID1** be the term **(EQUAL (ID1 X Y) X)**. Call this term **t**. Let fs be the functional substitution {<**ID1**, **(LAMBDA (A B) X)**>}. This substitution replaces **ID1** by the constant func-

tion that returns **X**. Observe that the fs instance of the defining axiom for **ID1** is a theorem, i.e., **t**\fs is **(EQUAL X X)**. A careless definition of the rule of functional instantiation would therefore permit the conclusion that any fact about **ID1** holds for the constant function that returns **X**. But this conclusion is specious: **(EQUAL (ID1 (ADD1 X) Y) (ADD1 X))** is a fact about **ID1** but its functional instance under fs, **(EQUAL X (ADD1 X))**, is not. What is wrong? A variable in the defining axiom, **t**, occurs freely in fs. We were just ''lucky'' that the fs instance of the defining axiom, **t**, was a theorem. Had **ID1** been defined with the equivalent axiom **(EQUAL (ID1 U V) U)**, which we will call **t'**, we would have been required to prove **t'**\fs which is **(EQUAL X U)** and is unprovable. The point is that when we attempt to prove the functional instances of the axioms, we must first rename the variables in the axioms so as to avoid the free variables in the functional substitution.

Terminology. Term $\mathbf{t_1}$ is a *variant* of term $\mathbf{t_2}$ if $\mathbf{t_1}$ is an instance of $\mathbf{t_2}$ and $\mathbf{t_2}$ is an instance of $\mathbf{t_1}$.

Example. **(EQUAL (ID1 U V) U)** is a variant of **(EQUAL (ID1 X Y) X)**.

Terminology. If fs is a functional substitution and **t** is a term, then *an* fs *renaming of* **t** is any variant of **t** containing no variable free in fs.

Derived Rule Of Inference. Functional Instantiation.

If h is a history, fs is a tolerable functional substitution, **thm** is a theorem of h, and the fs instance of an fs renaming of every axiom of h can be proved in h, then **thm**\fs is a theorem of a definitional/constrain extension of h.

Note. In [1] we show that the introduction of constrained function symbols preserves the consistency of the logic and we derive the rule of functional instantiation. In fact, we prove a more general version of it that permits us to ignore the ''irrelevant'' axioms and definitions in h.

References

1. R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore. Functional Instantiation in First Order Logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed., Academic Press, 1991, pp. 7-26.

2. J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Ma., 1967.

3. G. L. Steele, Jr. *Common Lisp The Language.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

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

Index

C

D

E

F

Q

R

S

T

U

Contents