A Computational Logic Handbook
Authorized Excerpts from a Proposed Second Edition

Preface to the Second Edition

Five years have elapsed since the publication of the first edition of *A Computational Logic Handbook* and the theorem proving system it described, Nqthm. During that period, several significant new features were added to the Nqthm system. These features were implemented in ‘‘patch’’ files and combinations of these files were in use in verification projects at Computational Logic, Inc. (CLI) and the University of Texas (UT). Some of these projects have produced significant new Nqthm proofs. Unfortunately, because of the use of the patch files, it was not always easy to communicate to outsiders what had been proved or how. We therefore decided to rationalize the situation by releasing an extended Nqthm. We also took the opportunity to correct the reported ‘‘bugs’’ in Nqthm (though none involved soundness). The result is Nqthm-1992. In trying to document the changes we came to the conclusion that the most coherent approach we could take would be to produce a second edition of this book. Those portions of the text added to document Nqthm-1992 are indicated by ‘‘change bars’’ printed in the margins. We have also corrected typographical errors in the first edition.

Nqthm-1992 includes new events for introducing constrained function symbols, inferring theorems via functional instantiation, and defining, enabling and disabling theories. The new system supports **COND**, **CASE**, **LET**, **LIST*** and ‘‘backquote’’ syntax. Performance is improved markedly when large sets of names are disabled and when large case structures arise. More support is offered for compiling and loading library files and for processing event files to produce ‘‘endorsed’’ data bases or for ‘‘skimming’’ event files to construct a fraudulent data base for experimental purposes. Some less noticeable changes were also made. The new release includes over ten megabytes of example

Nqthm event files.

It would be nice to say that any sequence of events acceptable to the old Nqthm is acceptable to Nqthm-1992, but this is not true. There is one syntactic incompatibility: Nqthm-1992 does not allow duplicate occurrences of hints while Nqthm does. However, because of some heuristic changes it is possible for the new system to fail to find proofs that the old system finds. In our many megabytes of example files we found a few examples of this. All of the failures in those filles were caused by the fact that the new ‘‘fast clausifier’’ sometimes produces clauses in which the literals occur in a different order than in Nqthm. This can effect the subsequent instantiation of free variables or expose loops in rewrite rules that were present but not exposed before. Matt Kaufmann found two more instances in which heuristic ‘‘improvements’’ caused old scripts not to replay. One failure was caused by a change in Nqthm’s preferred ordering for terms, which is used to decide whether a permutative rewrite rule should be applied; in the new ordering, explicit values come before non-explicit values, regardless of their lexicographic size; because the ordering has always been sensitive to the number of variables in the two terms, this change is very rarely felt because explicit values contain no variables and were thus ‘‘naturally’’ given priority over all non-explicit values except (the rare) irreducible ground expressions such as terms involving undefined constants. The second failure noted by Kaufmann was due to the fact that Nqthm-1992 can apply metafunctions to ‘‘non-tame’’ terms and hence metatheoretic simplifiers may see more use under Nqthm-1992 than under the older version. These exceptions notwithstanding, the most common experience with Nqthm-1992 is that old scripts replay without any trouble. However, users should be alert to the possibility that a script will have to be changed to make it replay.

Library files produced by the old Nqthm are not compatible with Nqthm-1992. We recommend that the new system be used to rebuild old libraries from the event lists.

Installation instructions for Nqthm-1992 are almost unchanged from those for Nqthm.

In many cases, our colleagues at CLI first implemented the features we are now adding. We are indebted to Bill Bevier for his invention of the notion of ‘‘theories’’ and their implementation, Matt Wilding for his implementation of ‘‘fast disabling,’’ and Bishop Brock for his ‘‘fast clausifier.’’ Matt Kaufmann, David Goldschlag and Bill Bevier collaborated with us in the development of the logical basis for and the design of the ‘‘constrain’’ and ‘‘functionally instantiate’’ events. Bill Schelter has rendered invaluable service to us by maintaining and improving the performance of Austin Kyoto Common Lisp.

The contributions of Matt Kaufmann deserve a special note. Matt has written an interactive enhancement to Nqthm, called ‘‘Pc-Nqthm’’ [37, 31]. It is an interactive *p*roof *c*hecker for the Nqthm logic (hence the name, *Pc*-Nqthm). As

such it provides low-level commands such as **`induct`**, **`expand`**, **`=`** (substitute), etc. It also provides high-level commands that employ the Nqthm theorem prover, such as **`prove`** (a subgoal). In addition, *macro commands* provide a capability similar to that provided by tactics in HOL [26] and Pc-Nqthm provides support for an extension of the logic that includes first-order quantification. Pc-Nqthm is integrated with Nqthm so that one can freely mix standard Nqthm-style proof development with interactive proof checking. Some theorems proved using Pc-Nqthm include the correctness of a high-level language compiler [59], several Unity programs [24, 25], and correctness proofs for parametrized hardware modules [57]. Enquiries about Pc-Nqthm may be sent to **`nqthm-request@cli.com`** or to Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, Texas 78703 U.S.A.

The ability to use the first-order quantifiers ''for all'' and ''there exists'' can be beneficial for Nqthm users, especially for the purpose of writing clear specifications. Included in the Pc-Nqthm system is a facility called **`DEFN-SK`** [36] that provides this capability. (The ''**`SK`**'' in ''**`DEFN-SK`**'' stands for ''Skolem,'' since **`DEFN-SK`** uses the technique of *Skolemization* to define functions whose definition bodies contain quantifiers.) In [36] Matt has used this enhancement in verifying some relatively deep theorems: Koenig's Tree Lemma, the infinite exponent-2 Ramsey's Theorem, and the Schroeder-Bernstein Theorem. Others have made use of **`DEFN-SK`** in their work on proving properties of computing systems. For example, David Goldschlag [25] has found **`DEFN-SK`** useful in formalizing the UNITY logic inside the Nqthm logic, and Yuan Yu [13, 62] has used **`DEFN-SK`** in his verification of MC68020 machine code for the Berkeley C String library. Although **`DEFN-SK`** is part of Pc-Nqthm, it can be loaded directly on top of Nqthm as a standalone enhancement.

Matt's creation of Pc-Nqthm has contributed in many ways to the success of Nqthm. For example, it has made Nqthm available to people who preferred a proof-checking environment. It has even helped users who prefer Nqthm-style proof development: using Pc-Nqthm to test a certain rewriting strategy often precedes the ''automation'' of that strategy via **`REWRITE`** rules. But perhaps the biggest contribution of Pc-Nqthm has been to make Matt Kaufmann intimately familiar with our code for Nqthm. His knowledge of Nqthm, combined with his apparently boundless energy and enthusiasm, has made him point man in the mission to improve Nqthm: our colleagues at CLI generally talk to Matt before they talk to us about proposed changes. Matt has taken on the responsibility of maintaining the myriad ''Nqthm patch files'' that float around the CLI Nqthm community. Matt made valuable contributions to the design and implementation of all of the features noted above, always keeping soundness uppermost in his mind. His contributions to and scrutiny of proposed changes has made it much easier for us to accept the changes. Without Matt's efforts and enthusiasm, it is unlikely Nqthm-1992 would have contained any of the new features.

After obtaining permission from the implementors above, we integrated the new ideas into our source code, often changing the implementation to bring it wholly within our style. Thus, despite our gratitude to our colleagues for their ideas and code, we remain entirely responsible for the correctness of Nqthm.

This release of Nqthm includes over ten megabytes of Nqthm script files. These files, when processed by Nqthm-1992, produce approximately 200 megabytes of proof descriptions and endorsed data bases. These files illustrate how Nqthm's logic can be used to formalize a wide variety of problems and how Nqthm can be led to deep proofs. A wide variety of topics are covered: number theory, logic, metamathematics, algorithms, hardware design, machine code programs, linkers, assemblers, compilers, multiprocessing, concurrency, asynchronous communications, and game playing. The magnitude of our debt to our user community is illustrated by the fact that the scripts assembled here are an order of magnitude larger than the implementation of Nqthm. The following people have contributed scripts to this release: Bill Bevier, Bishop Brock, Alex Bronstein, John Cowles, Mike Green, Warren Hunt, Matt Kaufmann, Ken Kunen, Misao Nagayama, David Russinoff, Natarajan Shankar, Carolyn Talcott, Matt Wilding, Bill Young, and Yuan Yu.

This release also contains an ''infix printer,'' an optional utility one may use to reformat Nqthm event scripts into more conventional mathematical notation. The infix printer translates an Nqthm event file into a LaTex or Scribe source file, which may then be processed by the appropriate document preparation system. For example, the event

```
(prove-lemma boolean-sample (rewrite)
  (implies (and (lessp a b)
                (lessp b (length c)))
           (zerop (difference a (length c)))))
```

may be printed as

THEOREM: boolean-sample
$a < b \wedge b < \mathrm{length}\,(c) \rightarrow a - \mathrm{length}\,(c) \cong \mathbf{0}$

The infix printer was initially written by Bob Boyer to produce LaTex output and was then extended by Mike Smith to produce Scribe output. See the installation guide for more details.

We would like to thank Alan Boxer, Warren Hunt, Matt Kaufmann, Donovan Kolbly, Bill Legato, Sylvan Pinsky, Larry Smith, and Morgan Witthoft for pointing out errors in the first edition of this book and earlier drafts of this edition. For proof reading diligence above and beyond the call of duty, we would like to thank Simon Read. This edition would be a less suitable manual

had it not been for his many suggestions; and it would have been more suitable still had we had the patience to implement more of his suggestions.

We also thank our sponsors for their continued support: Bill Scherlis at the Defense Advanced Research Projects Agency and Ralph Wachter at the Office of Naval Research.

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406 and the Office of Naval Research, Contract Number N00014-91-C-0130. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the sponsoring organizations, or the U.S. Government.

3 Formalization Within the Logic

Given the descriptions of the primitive functions in the logic it is easy to see that the value of

```
(APPEND NIL '(4 5 6))
```

is **'(4 5 6)**, and the value of

```
(APPEND '(1 2 3) '(4 5 6))
```

is

```
'(1 2 3 4 5 6).
```

Another way to phrase the last example is that the value of

```
(EQUAL (APPEND '(1 2 3) '(4 5 6))
       '(1 2 3 4 5 6))
```

is **T**.

Now what is the value of **(APPEND NIL Y)**? In general, we cannot answer such a question unless we know the values of the variables involved, in this case, **Y**. But whatever the value of **Y**, the value of **(APPEND NIL Y)** is the same as that of **Y**. That is,

```
(EQUAL (APPEND NIL Y) Y)
```

has the value **T**, regardless of the value of **Y**.

Similarly, the value of

```
(EQUAL (APPEND (APPEND X Y) Z)
       (APPEND X (APPEND Y Z)))
```

is **`T`**, regardless of the values of **`X`**, **`Y`**, and **`Z`**.

The obvious question is ‘‘Can you prove it?’’ It is easy to *test* these assertions by executing the given expressions under sample assignments to the variables. But is it possible to demonstrate that the value must always be **`T`**? If all we have to work with is the informal description of the logic as a programming language, as given in the previous chapter, the answer to these questions is ‘‘no, we can’t prove it, but would you like to see another test?’’

In the next chapter we present the logic again but do so formally. We define the *language* precisely so that you know exactly what is legal to write down. We give *axioms* that completely describe each function, and we give *rules of inference* with which the axioms can be manipulated. A *theorem* is any formula that is either an axiom or can be derived from theorems using the rules of inference. It happens that every theorem is always true, regardless of the values of the variables. More precisely, if **`t`** is some theorem containing the variables v_1, ..., v_n, then regardless of what values we assign to v_1, ..., v_n, the value of **`t`** in any model of our axioms is non-**`F`**. This is no accident; the rules of inference are all validity-preserving so any formula constructed from the axioms by the rules of inference is necessarily always true.

So, if for some reason you wish to determine whether some formula has the value **`T`** in all models, you can try proving it. Technically speaking, this is not always possible. That is, there are formulas that are true in all models but that cannot be proved. That is what Gödel’s incompleteness theorem tells us. But you are extremely unlikely to happen upon a truth that is not a theorem and for all practical purposes you should proceed as though every truth in the logic has a proof.

Why might you want to know whether some formula is always true? Suppose you are writing a ‘‘program’’ (i.e., function) in the logic and you need to compute **`(APPEND A (APPEND B C))`**. Suppose you have already computed **`(APPEND A B)`**, and it is the value of the variable **`AB`**. Then you might use **`(APPEND AB C)`**, i.e., **`(APPEND (APPEND A B) C)`**, for **`(APPEND A (APPEND B C))`**. This is a legitimate thing to do, *if* you know that **`APPEND`** is associative. That is, if you know that

```
(EQUAL (APPEND (APPEND X Y) Z)
       (APPEND X (APPEND Y Z)))
```

always evaluates to **`T`**.

The situation just described is a simple example of a very deep problem: *formalization*. When confronted with a practical computing question of the form ''Have I done this correctly?'' it is often possible to convert it into a formal question of the form ''Is this formula a theorem?'' We offer no mechanical assistance in this process; indeed, there is a sense in which no mechanical assistance is possible: the question bridges the gap between the informal and the formal and, by necessity, any mechanical help must confine itself to the formal side of the chasm.[1]

Almost invariably, when confronted with the challenge to formalize a problem, new users of our logic are baffled—especially if they come from a mathematical background. How can anything as complicated or as abstract as your problem be couched in the simple, nearly constructive system described here? It is hard enough to formalize concepts in a truly powerful mathematical system such as set theory. To do it in a logic with no quantification, no mutual recursion, no infinite objects, no sets, no abstraction, no higher order variables, and a very ''constructive'' sense of partial functions is, it would seem, impossible. For the authors of the system to then inveigh against the addition of axioms as being too dangerous simply seems to doom the system to total uselessness. These impressions notwithstanding, many complicated and abstract problems have been formalized within the logic. This chapter explains how it is done.

It is not necessary to be able to prove theorems in order to formalize problems. It is necessary to have a clear understanding of what formulas in the logic ''mean''—a topic that is often illuminated by the axioms and rules of inference. But we believe the primer gives a description of the language adequate for the discussion of formalization. Furthermore, the discussion of formalization clarifies some of the issues and may motivate some readers to undertake the necessary study of Chapter 4. Readers who disagree may wish to read Chapter 4 first and then return to this chapter.

This chapter is organized as follows. First, we show a few recursive definitions, in the spirit of those offered in the primer, to illustrate the use of the logic as a programming language. However, we basically assume in this book that you are sufficiently familiar with programming to use recursion and inductive data structures to implement solutions to elementary programming problems. Of more concern in this chapter is the expression of relationships and the definitions of concepts that most readers think of as being ''mathematical'' rather than ''computational.'' Thus, the second section of the chapter illustrates the formal

---

[1]We take the view that anything that is codified in a programming language—be it a low level machine code or a high level language like Prolog—is formal. Researchers trying to bring the informal world within the grasp of computing machines are, in our view, merely formalizing a fragment of that world. Regardless of the extent of their success, there will necessarily be a gap between that which has been formally modeled and that which we personally want or desire or intend of our programs.

statement of some familiar mathematical relationship such as associativity, transitivity, etc. We describe how we deal with some of the omissions from our logic: mutual recursion, unbounded quantification, sets, and infinities. We then deal with two problems uniquely associated with features of our logic (rather than omissions from it), namely the use of **`V&C$`** to discuss termination questions and the statement of general purpose ''schematic'' theorems about **`FOR`**. Finally, we discuss two commonly occurring formalization problems: nondeterminism and language semantics. The latter problem is discussed in the more general framework of the formalization of other formal systems within ours.

3.1. Elementary Programming Examples

In the primer, we showed how the primitive shells are used to represent sequences, words, tables, and trees and gave definitions for list concatenation, **`APPEND`**; table lookup, **`ASSOC`**; tree flattening, **`FLATTEN`**; and term substitution, **`SUBST`**. These are typical examples of uses of the logic as a programming language. They are also examples of formalization. Is ''list concatenation'' **`APPEND`**? We cannot answer the question, nor can we tell you how to invent the formal definition of **`APPEND`** in response to the challenge ''Formalize list concatenation.'' In lieu of definite answers, we offer more examples. In this section we'll implement in the logic push down stacks, finite queues, and an insertion sort function.

3.1.1. Push Down Stacks

Problem. Formalize push down stacks on which only numbers are pushed. Since we are dealing with an applicative programming language a stack is an object and the operations on it are functions that return ''new'' objects with certain functional relations to the input. Informally we want the following five functions:

`PUSH`	Construct the stack obtained by pushing a given number onto a given stack of numbers.
`EMPTY`	Return the empty stack of numbers.
`TOP`	Return the topmost element of a given stack of numbers.
`POP`	Return the stack obtained by popping the topmost element off a given stack of numbers.
`STACKP`	Return **`T`** or **`F`** according to whether the given argument is a stack of numbers.

This informal specification can be formalized by the following invocation of the shell principle:

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**.

We leave to the reader the difficult question ‘‘Is that really what is meant by the informal specification?’’

Several observations are in order however. Let **STK** be the push down stack constructed by **(PUSH 3 (PUSH 2 (PUSH 1 (EMPTY))))**, that is, the stack obtained by pushing **1**, then **2**, and then **3** onto the empty stack. Then **(TOP STK)** is **3**, the top of the stack. **(POP STK)** is **(PUSH 2 (PUSH 1 (EMPTY)))**, the result of popping the **3**. **(STACKP STK)** is **T**.

Several ‘‘pathological’’ cases are included in the formalization, for better or for worse. **(PUSH T STK)** produces the same stack as **(PUSH 0 STK)** because **T** is not a number and so **PUSH** ‘‘coerces’’ it to the first default object **0**. **(TOP (EMPTY))** is **0** because **TOP** returns the first default object when called on a non-stack or on the empty stack. **(POP F)** is **(EMPTY)** because **POP** returns the second default object when called on a non-stack or the empty stack.

3.1.2. Finite Queues

Problem. Formalize finite queues. Informally, we want the following functions:

MAKE-QUEUE	Return an empty finite queue of a given maximum size.
ENQUEUE	Add a new element to the front of a given finite queue.
DEQUEUE	Given a finite queue, return the finite queue with the last element removed.
ELE	Return the last element of a given finite queue.
EMPTY-QUEUEP	Return **T** or **F** according to whether a given finite queue is empty.
FULL-QUEUEP	Return **T** or **F** according to whether a given finite queue is full.

QUEUEP Return **T** or **F** according to whether the argument is a finite queue.

The basic idea behind our formalization is that a finite queue is a pair, <**list**, **max**> where **list** is a list of the elements currently in the queue, ordered from oldest to newest, and **max** is the maximum size of the queue. We cannot use the shell principle alone to formalize this approach because we have no way, within the shell principle, to enforce the constraint that the length of **list** is less than or equal to **max**. Therefore, we will use the shell principle to get a ''new'' kind of ordered pair, called a **PROTO-QUEUE**, and we will formalize queues in terms of it. By ordering **list** as indicated we can formalize **ENQUEUE** by **APPEND**ing the new element to the **list**, **DEQUEUE** by taking the **CDR**, and **ELE** by taking the **CAR**.

Here is the formalization of **PROTO-QUEUE**s:

Shell Definition.
Add the shell **PROTO-QUEUE** of 2 arguments
with recognizer function **PROTO-QUEUEP**,
accessor functions **QLIST** and **QMAX**,
type restrictions **(NONE-OF)** and **(ONE-OF NUMBERP)**, and
default functions **ZERO**, and **ZERO**.

MAKE-QUEUE is then defined as

Definition.
```
(MAKE-QUEUE MAX)
   =
(PROTO-QUEUE NIL MAX).
```

We can define **QUEUEP** to be

Definition.
```
(QUEUEP X)
   =
(AND (PROTO-QUEUEP X)
     (LEQ (LENGTH (QLIST X))
          (QMAX X))),
```

where

Definition.
```
(LENGTH X)
   =
(IF (NLISTP X)
    0
    (ADD1 (LENGTH (CDR X)))).
```

We can define **EMPTY-QUEUEP** and **FULL-QUEUEP** as follows:

Definition.
```
(EMPTY-QUEUEP Q)
    =
(AND (QUEUEP Q)
     (NLISTP (QLIST Q)))
```

Definition.
```
(FULL-QUEUEP Q)
    =
(AND (QUEUEP Q)
     (EQUAL (LENGTH (QLIST Q))
            (QMAX Q))).
```

What should **ENQUEUE**, **DEQUEUE**, and **ELE** do on ‘‘illegal’’ arguments such as non-queues or full queues or empty queues? Do we implement them with ‘‘run-time’’ checks and make them return special answers in ‘‘illegal’’ situations? For example, we could define **ENQUEUE** so that if its argument were not a **QUEUEP** or were full, it returned **F**. If we do not include run-time checks, the functions may return ‘‘garbage’’ in illegal situations. Our own personal bias is to avoid run-time checks, use the functions in a way that is consistent with their intended specifications, and prove that every use is appropriate. Here then are the definitions:

Definition.
```
(ENQUEUE E Q)
    =
(PROTO-QUEUE (APPEND (QLIST Q)
                     (LIST E))
             (QMAX Q)).
```

Observe that if **Q** is not a **QUEUEP** or is full, we return a non-**QUEUEP** in which the list component is longer than the maximum size.

Definition.
```
(DEQUEUE Q)
    =
(PROTO-QUEUE (CDR (QLIST Q))
             (QMAX Q)).
```

We intend for **DEQUEUE** to be used only on nonempty **QUEUEP**s. If **Q** is not even a **PROTO-QUEUEP**, **DEQUEUE** actually returns the empty **QUEUE** of size **0**. If **Q** is a **PROTO-QUEUE** but not a **QUEUEP**—meaning its list component is too long—**DEQUEUE** shortens the list by 1, and the answer may or may not be a **QUEUEP**. If **Q** is a **QUEUEP** but is empty, **DEQUEUE** returns an empty **QUEUEP** of the same size.

Definition.
```
(ELE Q)
    =
(CAR (QLIST Q)).
```

Observe that if **Q** is not a **PROTO-QUEUEP** or is the empty **QUEUEP**, **ELE** returns **0**. If **Q** is a **PROTO-QUEUEP** that fails to be a **QUEUEP**, **ELE** returns the first element of its list component.

Once again the hard question is left to the reader: ‘‘Is that what we mean by finite queues?’’

3.1.3. Insertion Sort

Problem. Define a function that sorts lists of numbers into ascending order. Our approach will be to define an insertion sort function.

The function **INSERT** takes a number **N** and an ordered list of numbers **L** and returns the list obtained by inserting **N** into **L** ‘‘at the right place,’’ i.e., immediately before the first element of **L** that is greater than or equal to **N**:

Definition.
```
(INSERT N L)
    =
(IF (NLISTP L)
    (LIST N)
    (IF (LEQ N (CAR L))
        (CONS N L)
        (CONS (CAR L) (INSERT N (CDR L))))).
```

Thus, **(INSERT 5 ’(2 4 5 6))** is **’(2 4 5 5 6)**.

The main function, **SORT**, takes a list **L** and recursively sorts its **CDR**, when nonempty, and then **INSERT**s the **CAR** of **L** into the result

Definition.
```
(SORT L)
    =
(IF (NLISTP L)
    NIL
    (INSERT (CAR L) (SORT (CDR L)))).
```

Thus, **(SORT ’(4 2 5 1 5 6))** is **’(1 2 4 5 5 6)**.

3.2. Elementary Mathematical Relationships

We now turn to the problem of expressing elementary relationships between functions as theorems in the logic. Below are formulas expressing familiar facts of arithmetic:

```
(EQUAL (PLUS I J)                    ;Commutativity of PLUS
       (PLUS J I))

(EQUAL (PLUS (PLUS I J) K)           ;Associativity of PLUS
       (PLUS I (PLUS J K)))

(EQUAL (PLUS I 0)                    ;Right Identity of PLUS
       I)

(EQUAL (TIMES X (PLUS I J))          ;Distributivity of TIMES
       (PLUS (TIMES X I)             ;over PLUS
             (TIMES X J)))

(IMPLIES (AND (LESSP I J)            ;Transitivity of LESSP
              (LESSP J K))
         (LESSP I K))
```

Consider the first formula. The formula is a theorem and thus it is ‘‘always true.’’ That is, no matter what values we choose for **I** and **J**, the sum **I** plus **J** is the same as that of **J** plus **I**. In other words, **PLUS** is a commutative function. In more traditional notation this might be written: $\forall$ i $\forall$ j i+j = j+i. Such elementary facts of arithmetic are so often expressed formally (i.e., as formulas) that it is not difficult to write them in our notation, once you get used to our prefix syntax and the absence of quantifiers. One way to think of the quantifier situation is that all formulas are (implicitly) universally quantified over all variables on the far outside of the formula.

However, one must be especially careful of the formalization of familiar concepts. One of the five formulas above is not a theorem. Do you know which one?

It is the third. Normally, when we say ‘‘**0** is the right identity of plus’’ we implicitly restrict our attention to numeric arguments. What is the sum of **T** and **0**? In a strictly typed language such questions don’t arise. But in our language they do and they have answers: the sum of **T** and **0** is **0**, not **T**. An accurate statement of our version of the right identity law is

Theorem.
(IMPLIES (NUMBERP I) (EQUAL (PLUS I 0) I))

or, perhaps more conveniently,

Theorem.
(EQUAL (PLUS I 0) (FIX I)).

Often it is necessary to define new concepts simply to state the desired properties of given ones. For example, suppose we wished to formalize the notion that **SORT** produces ordered output. To express this as a formula we must define the ''predicate'' that checks whether a list is ordered. One suitable definition is

Definition.
```
(ORDEREDP L)
  =
(IF (NLISTP L)                    ;An empty list is ordered.
    T

    (IF (NLISTP (CDR L))          ;A list of length 1 is too.
        T
                                  ;Otherwise, check that
                                  ;the first two elements
                                  ;are in the right relation
        (AND (LEQ (CAR L)         ;and that the rest of L
                  (CADR L))       ;is ordered.

             (ORDEREDP (CDR L))))))
```

Thus, **(ORDEREDP '(1 2 2 5 27 99))** is **T**, but **(ORDEREDP '(1 2 2 5 27 17 99))** is **F**.

The formalization of ''**SORT** produces ordered output'' is then

Theorem.
(ORDEREDP (SORT L)).

Of course, is **ORDEREDP** really what we mean by ''ordered''? That is the hard problem. Furthermore, even if we are convinced that **ORDEREDP** is correctly defined, there is the additional problem: ''If a function *does* produce ordered output, do I know it is a correct sort function?'' The answer is ''no.'' For example, the function

Definition.
(FAST-SORT L) = NIL.

has the property that it produces ordered output.

Generally, when we say a function is a ''sort function'' we mean that it permutes its input into order.

This raises the question ''How do we formalize the notion that one list is a permutation of another?'' We define **PERM** so that it scans the first list and attempts to find each element as a member of the second list. If some element is not found, the two lists are not permutations of one another. When an element is found in the second list, that occurrence of the element is deleted from the second list so that it is not used to account for duplications in the first list.

Definition.
```
(PERM X Y)
   =
(IF (NLISTP X)
    (NLISTP Y)
    (AND (MEMBER (CAR X) Y)
         (PERM (CDR X)
               (DELETE (CAR X) Y)))).
```

(DELETE E L) deletes the first occurrence of **E** in **L**:

Definition.
```
(DELETE E L)
   =
(IF (NLISTP L)
    L
    (IF (EQUAL E (CAR L))
        (CDR L)
        (CONS (CAR L) (DELETE E (CDR L))))).
```

Again, is **(PERM X Y)** what we mean by the informal remark that **X** is a permutation of **Y**? We can test **PERM** of course and see that **(PERM '(1 2 3 4) '(3 1 2 4))** is **T**, and **(PERM '(1 2 3 4) '(3 1 1 4))** is **F**. In addition, we can prove theorems about **PERM**, for instance that it is reflexive and transitive:

Theorem.
```
(PERM X X)
```

Theorem.
```
(IMPLIES (AND (PERM X Y)
              (PERM Y Z))
         (PERM X Z))
```

But in the end, we are left with the fundamental formalization question.

If we accept both **ORDEREDP** and **PERM**, then a more complete correctness conjecture for **SORT** is

Theorem.
```
(AND (PERM L (SORT L))
     (ORDEREDP (SORT L))).
```

3.3. Dealing with Omissions

3.3.1. Mutual Recursion

The logic formally prohibits mutual recursion. However, as discussed in the primer, the prohibition can be effectively skirted by defining a single function whose body is a syntactic combination of the mutually recursive definitions. See the primer for the details.

3.3.2. Quantification

Most users have been exposed to predicate calculus and have some familiarity with the unbounded universal quantifier, ∀ (pronounced ‘‘for all’’), and the unbounded existential quantifier, ∃ (pronounced ‘‘for some’’). In the attempt to formalize an intuitive statement many users therefore go through a stage in which they write down or at least think of quantified versions of the terms in our logic. They then convert these quantified sentences into terms which are interprovable in the two systems. This is not an entirely precise remark because we haven't formalized a version of our logic containing ∀ and ∃, but the idea should be clear.

The most commonly used observation is that if you want to prove ∀x p(x) in predicate calculus, you should prove **`(p x)`** in our system, where **`(p x)`** is the translation into our language of the predicate p(x). That is, the axioms, definitions, and theorems of our logic can be thought of as having implicit universal quantification on the far outside. For example

Problem. State that **`TIMES`** is commutative.

In the imagined extension of the logic to include ∀ we wish to prove

```
∀I ∀J (EQUAL (TIMES I J) (TIMES J I)).
```

But this is provable in the imagined extension of the logic precisely if

```
(EQUAL (TIMES I J) (TIMES J I))
```

is provable in the logic provided. As we have seen before, if this term is a theorem, then *for all* values of **`I`** and **`J`** the two **`TIMES`** expressions are equivalent.

This example is deceptively simple because it suggests that universal quantification, at least, can simply be ignored. This is not the case and great care is called for when eliminating universal quantifiers from any position other than the outside of a formula. We return to this in a moment, but first we discuss the similarly simple case of an outermost existential quantifier.

Problem. State that there exists a right zero for **TIMES**.

By ‘‘a right zero for **TIMES**’’ we mean a number, **z**, such that **(TIMES I z)** = **0**, for all **I**. Thus, the existence conjecture is

$$\exists \mathbf{Z}\ \forall \mathbf{I}\ \mathbf{(EQUAL\ (TIMES\ I\ Z)\ 0).} \qquad [1]$$

Since we do not have quantification, we are forced to exhibit a suitable **Z**. That is, we drop the ∃**Z** and replace all free occurrences of **Z** by a term that has the alleged property. A suitable term is **0**:

$$\forall \mathbf{I}\ \mathbf{(EQUAL\ (TIMES\ I\ 0)\ 0).}$$

If we proved this, the law of ∃-introduction in predicate calculus would immediately let us prove [1]. By the previous observations about universal quantifiers on the outside, we can drop the ∀**I** above and are left with

Theorem.
(EQUAL (TIMES I 0) 0)

as a formalization of the statement that there exists a right zero for **TIMES**.

Problem. State that **0** is the unique right zero for **TIMES**.

If by this remark we mean ‘‘If z is a right zero for **TIMES** then z is **0**,’’ the remark is not valid. **0** is not the only right zero for **TIMES**. **T** is also a right zero:

Theorem.
(EQUAL (TIMES I T) 0).

So by ‘‘**0** is unique’’ here we must mean ‘‘**0** is the only number’’ or ‘‘**0** is unique up to coercion by **FIX**.’’

That said, a suitable rendering in the imagined extension of the logic is

```
(IMPLIES [∀I (EQUAL (TIMES I Z) 0)]     [2]
         (EQUAL (FIX Z) 0)).
```

The hypothesis says ‘‘**Z** is a right zero,’’ and the conclusion says ‘‘**Z** is (coerced to) **0**.’’

Very often, new users make the mistake of simply dropping the universal quantifier in the hypothesis, producing, in this case,

```
(IMPLIES (EQUAL (TIMES I Z) 0)          [3]
         (EQUAL (FIX Z) 0)).
```

But formula [3], while in our language, is not at all equivalent to [2] and, indeed, is not a theorem! To see that [3] is not a theorem, evaluate it with **`I`** equal to **`0`** and **`Z`** equal to **`47`**. The hypothesis, **`(EQUAL (TIMES 0 47) 0)`**, is true but the conclusion, **`(EQUAL (FIX 47) 0)`**, is false. Hence, the implication is false.

What is wrong? The problem is that you simply cannot drop a universal quantifier in the hypothesis. The hypothesis in [2] is very strong. Given a **`Z`**, the hypothesis

```
[∀I (EQUAL (TIMES I Z) 0)]
```

says that **`(TIMES I Z)`** is 0 for all **`I`**. The hypothesis **`(EQUAL (TIMES I Z) 0)`** in [3] is much weaker. Given an **`I`** and a **`Z`** it says **`(TIMES I Z)`** is 0—it doesn't say that **`Z`** is a right zero for **`TIMES`**.

More generally, in predicate calculus, $((\forall x\ p) \rightarrow q)$ is not equivalent to $\forall x(p \rightarrow q)$. The tendency simply to ignore universal quantifiers is encouraged by the fact that $(p \rightarrow (\forall x\ q))$ is equivalent to $\forall x\ (p \rightarrow q)$, provided x is not free in p, and the fact that it is conventional to have implicit universal quantification of all free variables on the far outside of any theorem.

So how do we formalize [2] within the logic? In the case where the universal quantification is over a finite domain, for example, had the hypothesis read ''for all **`I`** between **`N`** and **`M`**, ...,'' the standard approach is to replace it with a **`FOR`** statement or with a call of a recursively defined function that checks the alleged property of **`I`** for each value in the domain. This approach is not applicable here because we are dealing with an unbounded universal quantifier.

Two common techniques are used, both of which rely upon an understanding of predicate calculus and Skolemization. The first is to use the fact that if x is not free in q, then $((\forall x\ p) \rightarrow q)$ is equivalent to $\exists x\ (p \rightarrow q)$. Thus, we can render [2] as

```
∀Z ∃I (IMPLIES (EQUAL (TIMES I Z) 0)
               (EQUAL (FIX Z) 0)).
```

What does this formula say? It says that for any **`Z`** you can find an **`I`** such that if **`(TIMES I Z)`** is **`0`**, then **`Z`** is (coerced to) **`0`**. To render that into our logic we must exhibit a suitable **`I`** given any **`Z`**. In particular, we define a ''witness function,'' **`I-WITNESS`**, drop the existential quantifier, ∃**`I`**, and replace all free **`I`**s by **`(I-WITNESS Z)`**.

```
∀Z (IMPLIES (EQUAL (TIMES (I-WITNESS Z) Z) 0)
            (EQUAL (FIX Z) 0)).
```

What is a suitable definition of **`(I-WITNESS Z)`**? It must be a number such that if **`Z`** is a right zero for it, then **`Z`** is a right zero for everything. Clearly, we could define **`I-WITNESS`** to be the constant function that returns **`1`**—in this case the witness need not take account of which particular **`Z`** is being considered.

Eliminating **`I-WITNESS`** altogether, replacing the call by **`1`**, and dropping the universal quantifier as previously explained, we get the following formula as a formalization of [2]:

Theorem.
```
(IMPLIES (EQUAL (TIMES 1 Z) 0)
         (EQUAL (FIX Z) 0)).
```

To many readers the following statement, which is stronger than [2], is a more convincing formalization of the uniqueness of the right zero for **`TIMES`**:

Theorem.
```
(IFF (EQUAL (TIMES I J) 0)
     (OR (EQUAL (FIX I) 0)
         (EQUAL (FIX J) 0))).
```

This formula says that the product of two numbers is **`0`** iff one of the two arguments is **`0`**.

A second approach to formalizing [2] is to Skolemize the formula and appeal to the deduction law of predicate calculus. In particular, [2] is provable in predicate calculus iff

Theorem.
```
(EQUAL (FIX (Z)) 0)
```

can be proved assuming the new

Axiom.
```
(EQUAL (TIMES I (Z)) 0)
```

where **`Z`** is a new, undefined function symbol of no arguments. This is an unusual way to formalize the problem because instead of producing a formula to prove, we produce a sequence of logical acts that if admissible in our logic assures us that the desired formula is a theorem of another logic. This approach is commonly used in program verification. See for example our work on the verification of Fortran [6]. For practical reasons we do not recommend this approach to handling universally quantified hypotheses because our theorem prover is not designed to deduce consequences from arbitrary axioms.

Finally, let us consider one more quantification problem.

Problem. State the unique prime factorization theorem. Traditionally, a natural number p is *prime* iff p>1 and the only divisors of p are 1 and p. A *prime factorization* of a number n is a finite set of pairs $\{<p_1,i_1> \ldots <p_k,i_k>\}$ with the property that each p_j is prime, each i_j is greater than 0, p_i is p_j iff i is j, and the product $p_1^{i_1} * \ldots * p_k^{i_k}$ is n. The theorem that every natural number has a unique prime factorization is then: for every natural number n there exists a prime factorization, and if both s_1 and s_2 are prime factorizations of n then $s_1 = s_2$.

Note that the notion of prime, as traditionally defined, involves an unbounded universal quantifier: no number besides 1 and p divides p. Rather than attempt to capture that unbounded quantifier, we adopt the definition of prime that says ‘‘no number less than p divides p, except 1.’’ We can define this concept recursively since the quantification is now over the finite domain of numbers less than p.

Definitions.
```
(PRIME P)
   =
(AND (NOT (ZEROP P))
     (NOT (EQUAL P 1))
     (PRIME1 P (SUB1 P)))
```

where

```
(PRIME1 X Y)
   =
(IF (ZEROP Y)
    F
    (IF (EQUAL Y 1)
        T
        (AND (NOT (DIVIDES Y X))
             (PRIME1 X (SUB1 Y))))).
```

This is not, *a priori*, the same concept as *prime* above, for there may be some **PRIME**s that fail to be *prime* because they are divisible by any of the unbounded number of naturals greater than themselves. We accept that **PRIME** is *prime* only after we have proved that no number is divisible by a number greater than itself.

Rather than use sets of pairs to define prime factorizations we use lists of primes.

We say **S** is a **PRIME-FACTORIZATION** of **N** iff **S** is a list of **PRIME**s and the product of the elements of **S** is **N**:

Definition.
```
(PRIME-FACTORIZATION S N)
   =
(AND (FOR X IN S ALWAYS (PRIME X))
     (EQUAL N (FOR X IN S MULTIPLY X)))
```

Note that here, too, the quantification is bounded so our **FOR** function is adequate.

Use of the **FOR** quantifier is unnecessary. An alternative approach is to define the functions **PRODUCT** and **PRIME-LIST** so that they recursively determine what the product is and whether **L** is a list of primes.

Definitions.

```
(PRIME-LIST L)
   =
(IF (NLISTP L)
    T
    (AND (PRIME (CAR L))
         (PRIME-LIST (CDR L))))

(PRODUCT L)
   =
(IF (NLISTP L)
    1
    (TIMES (CAR L)
           (PRODUCT (CDR L)))).
```

By defining factorizations to be lists of primes we avoid the use of sets, but this returns to haunt us when we must state uniqueness. Our statement of uniqueness is that two lists of primes with the same product are equal up to order, which is to say, they are permutations of each other. We state the uniqueness part of the main theorem as follows:

Theorem.

```
(IMPLIES (AND (PRIME-FACTORIZATION S1 N)
              (PRIME-FACTORIZATION S2 N))
         (PERM S1 S2)).
```

Finally, we must state the existence part of the theorem without using an existential quantifier. This we do by defining a ‘‘witness function,’’ **PRIME-FACTORS**, that takes any given positive number **N** and returns an alleged factorization. The existence theorem is then

Theorem.

```
(IMPLIES (AND (NUMBERP N)
         (NOT (EQUAL N 0)))
    (PRIME-FACTORIZATION (PRIME-FACTORS N) N)).
```

That is, we not only prove there exists a factorization, we exhibit a constructive function for obtaining one.

The definition we use for **PRIME-FACTORS** is irrelevant to the intended understanding of the existence theorem. In fact, the definition is relatively complicated: we define the notion of the greatest factor of a number and then we define the prime factors as follows. The prime factors of **0** and **1** is the empty list; the prime factors of a prime is the singleton list containing the prime;

and the prime factors of any other number **n** is obtained by appending the prime factors of the greatest factor, **m**, of **n**, to the prime factors of the quotient of **n** by **m**. That this is an admissible definition follows from the observations that both **m** and the quotient of **n** by **m** are smaller than **n**. The reason we defined **PRIME-FACTORS** this way was to make the proof of the existence theorem easier. It is trivial to show that **(PRIME-FACTORS N)** yields a list of primes. It is also relatively straightforward to show that the product of the prime factors is **N**, given that the greatest factor of **N** divides **N** if **N** is not prime.

In summary, we use recursion to capture the standard quantifiers, proving bounds on the universal quantification when necessary. Of course, **FOR** can be used in straightforward cases. We use lists where sets are traditionally used. Within this setting we have proved such deep results as Fermat's theorem [8], Wilson's theorem [47], and Gauss's law of quadratic reciprocity [48]. Such standard mathematical techniques as modulo arithmetic, the pigeon hole principle, and one-to-one mappings have been formalized, proved, and successfully used in other proofs.

3.3.3. Infinities

Problem. State the theorem that there are an infinite number of primes.

One formalization of this idea is that for every prime **P** there exists a prime bigger than **P**—and of course, there *is* at least one prime, witness **7**. To eliminate the quantifier define the function **BIGGER-PRIME** which takes a number **N** and returns a prime bigger than **N**. Then prove

Theorem.
```
(IMPLIES (PRIME P)
         (AND (PRIME (BIGGER-PRIME P))
              (LESSP P (BIGGER-PRIME P))))
```

An appropriate **BIGGER-PRIME** can be defined by searching through the numbers from **P+1** to **P!-1**. That at least one of those is prime follows from the fact that no prime less than or equal to **P** divides **P!-1**. Thus, if **P!-1** is itself not prime, its greatest prime factor is greater than **P**.

If the above theorem is not enough, it can be used as the main stepping stone to another: To each natural number there corresponds a unique prime. Again, the existential quantifier is eliminated by a constructive function, **NTH-PRIME**, with the following two properties:

Theorem.
```
(IMPLIES (NUMBERP N)
         (PRIME (NTH-PRIME N)))
```

and

Theorem.
```
(IMPLIES (AND (NUMBERP N)
              (NUMBERP M)
              (EQUAL (NTH-PRIME N)
                     (NTH-PRIME M)))
         (EQUAL N M))
```

Another kind of infinity with which we frequently deal has to do with processes that ‘‘run forever.’’

Problem. Formalize a ‘‘message flow modulator,’’ a device that sits on a communications line and filters the incoming messages into two streams according to some predicate.

Some formalists in computer science are very fond of using infinite sequences to represent the three message streams. However, for very many purposes finite sequences suffice. For example, we might formalize the modulator as a function **MFM** that takes a single message **msg** as its argument and returns a pair <**tag**, **msg**> as its result, where **tag** indicates to which output stream the outgoing message is assigned. The system is then formalized as

Definition.
```
(SYSTEM S)
   =
(IF (NLISTP S)
    NIL
    (CONS (MFM (CAR S))
          (SYSTEM (CDR S)))).
```

Here we formalize the system as a function which takes an incoming stream of messages, **S**, and produces a list of pairs, each of which is of the form <**tag**, **msg**> indicating a destination and the associated input message.[2]

We might then prove the theorem that the messages sent to tag **I** are a subset of the incoming messages:

```
(SUBSETP (FOR PAIR IN (SYSTEM S)
              WHEN (EQUAL I (CAR PAIR))
              COLLECT (CDR PAIR))
         S)
```

[2]The function **SYSTEM** could have been defined equivalently as **(FOR MSG IN S COLLECT (MFM MSG))**. However, the recursive definition more easily accommodates the complexities of more realistic models.

Many people object to this formalization of the system because it suggests that only a finite number of messages come down the communications line. We have two views on the matter, depending on how ornery we're feeling. The first is that, indeed, no communications line yet has had an infinite number of messages on it and no communications line is ever going to. The second view is that the variable **S** in the model is not to be thought of as the entire history of the communications line but merely some initial segment of it, and the answer returned by **(SYSTEM S)** is not the entire output of the system but only the output in response to that initial segment. The theorem means that the messages output to tag **I** *so far* are a subset of the incoming messages *so far*.

3.3.4. Higher Order Logic

Our logic is first order. However, the ability to constrain and to use functional instantiation often allows apparently higher order acts. In this section we illustrate the use of these features of the logic to overcome the omission of higher order functions and operations. Some of the same omissions can be addressed with our **V&C$** and **FOR** functions, discussed later in this chapter.

Problem. Call a function symbol, **fn**, of two arguments ''doubly commutative'' iff **fn** is commutative and **(fn x (fn y z))** is **(fn y (fn x z))**. Prove that every doubly commutative function symbol is associative.

In a suitable higher order logic with full first order quantification, one might make definitions such as:

Definition.
```
(DOUBLY-COMMUTATIVE fn)
    =
(AND ∀X ∀Y (EQUAL (fn X Y) (fn Y X))
     ∀X ∀Y ∀Z (EQUAL (fn X (fn Y Z))
                     (fn Y (fn X Z)))),
```

and

Definition.
```
(ASSOCIATIVE fn)
    =
∀X ∀Y ∀Z (EQUAL (fn (fn X Y) Z)
                (fn X (fn Y Z)))
```

and then prove

Theorem
```
(IMPLIES (DOUBLY-COMMUTATIVE fn)
```

```
        (ASSOCIATIVE fn)).
```

We cannot make such definitions because they take functions as arguments (not to mention that they involve unbounded quantification over the possible arguments to the functions). However, with introduction by constraint and functional instantiation we can deal with some aspects of the problem.

Let us introduce the function symbol **FN** of two arguments as follows.

Constraint. **DOUBLY-COMMUTATIVE**
Constrain **FN** to have the property

```
(AND (EQUAL (FN X Y) (FN Y X))
     (EQUAL (FN X (FN Y Z))
            (FN Y (FN X Z)))).
```

A suitable witness for this constraint is the function **PLUS**. Once introduced, we may think of **FN** as any doubly commutative function symbol. We can then undertake to prove that **FN** is associative:

Theorem. **FN-IS-ASSOCIATIVE**

```
(EQUAL (FN (FN X Y) Z) (FN X (FN Y Z))).
```

Indeed, if the constraining axiom for **FN** shown above is of rule class **REWRITE**, the proof of associativity is fully automatic. In particular, the sequence of rewrites performed is shown below, where each replacement is justified by one of the two equalities in the **DOUBLY-COMMUTATIVE** constraint:

```
(FN (FN X Y) Z) = (FN Z (FN X Y))
                = (FN X (FN Z Y))
                = (FN X (FN Y Z)).
```

It is clear that this exercise captures one part of the problem, namely, showing why ‘‘double commutativity’’ implies associativity. However, is this result useful?

Problem. Suppose **CONVOLVE** is a function that has been proved to be doubly commutative. Prove that it is associative.

In the imagined higher-order logic used above, the supposition is just that **(DOUBLY-COMMUTATIVE ’CONVOLVE)** is a theorem. The desired conclusion is that **(ASSOCIATIVE ’CONVOLVE)** is a theorem. The conclusion can be obtained from the supposition by *modus ponens*.

In our case, functional instantiation is used instead. In particular, the associativity of **CONVOLVE** can be deduced from **FN-IS-ASSOCIATIVE** by functional instantiation, replacing **FN** by **CONVOLVE**. Functional instantiation requires that **CONVOLVE** satisfy the axioms on **FN**. In this case, there is just one, the constraint **DOUBLY-COMMUTATIVE**. Hence, at the cost of proving (or noting the previous proof of) the double commutativity of **CONVOLVE** we conclude that it is associative.

Problem. Show that if a function is doubly commutative then mapping it over a list produces the same result as mapping it over the reverse of the list.

To formalize this problem we must first agree on the sense of ''mapping'' a function over a list. Suppose **`x`** is a list whose elements are $\mathtt{x}_1$, ..., $\mathtt{x}_n$. Then **`(map`** *fn* **`x b)`** is **`(`***fn* $\mathtt{x}_1$ **`(`***fn* $\mathtt{x}_2$ **`... (`***fn* $\mathtt{x}_n$ **`b)...))`**. This operation is called ''foldr'' in the higher-order functional language SASL [56]. Because the desired **`map`** function takes a function as an argument, we cannot define it.

We can define the notion of mapping a particular function over a list. For example, the following function maps **`PLUS`** over a list.

Definition.
```
(MAP-PLUS X B)
   =
(IF (NLISTP X)
    B
    (PLUS (CAR X)
          (MAP-PLUS (CDR X) B))).
```

Thus, **`(MAP-PLUS X 0)`** is the sum of the elements of **`X`**.

Since we are interested in the results of mapping with doubly commutative functions, and we have constrained **`FN`** to be any such function, we might therefore consider

Definition.
```
(MAP-FN X B)
   =
(IF (NLISTP X)
    B
    (FN (CAR X)
        (MAP-FN (CDR X) B))).
```

We can think of **`MAP-FN`** as the most general mapping function for doubly commutative functions in the sense that if **`MAP-fn`** is a function that maps with some particular doubly commutative function, then **`MAP-fn`** enjoys any property **`MAP-FN`** does. We illustrate this in a moment.

Having defined **`MAP-FN`** we might formalize the challenge problem above as

Theorem. **`MAP-FN-REVERSE`**
```
(EQUAL (MAP-FN (REVERSE X) B)
       (MAP-FN X B)).
```

This formula says that **`MAP-FN`** returns the same thing on **`(REVERSE X)`** as on **`X`**. We here are imagining that **`REVERSE`** is defined to be the function that reverses the order of the elements in a list. The above theorem is easy to prove, at least for the obvious definition of **`REVERSE`**.

Again, the utility question arises: can we use **`MAP-FN-REVERSE`** to derive the analogous property about, say, **MAP-PLUS**? The answer, again, is ‘‘yes, with functional instantiation.’’ In particular,

Theorem. **MAP-PLUS-REVERSE**
```
(EQUAL (MAP-PLUS (REVERSE X) B)
       (MAP-PLUS X B))
```

can be derived from **MAP-FN-REVERSE** by functional instantiation, replacing **MAP-FN** by **MAP-PLUS** and replacing **FN** by **PLUS**. Functional instantiation requires that the new functions satisfy the axioms about the old functions. In particular, every axiom about the old functions must yield a theorem when the old function symbols are replaced by the new ones.

The only axiom about **MAP-FN** is its definition. When the old symbols are replaced by the new ones, we obtain

Theorem?
```
(MAP-PLUS X B)
   =
(IF (NLISTP X)
    B
    (PLUS (CAR X)
          (MAP-PLUS (CDR X) B))).
```

This formula is a theorem because of the definition of **MAP-PLUS**.

There are now two axioms about **FN**. The first is the **DOUBLY-COMMUTATIVE** constraint. When the old symbols are replaced by the new ones we obtain

Theorem?
```
(AND (EQUAL (PLUS X Y) (PLUS Y X))
     (EQUAL (PLUS X (PLUS Y Z))
            (PLUS Y (PLUS X Z)))).
```

That is, we are obliged to prove that **PLUS** has the double commutativity property. The second axiom about **FN** is the definition of **MAP-FN**, which, when functionally instantiated, yields the **MAP-PLUS** theorem (definition) just shown.

To summarize, we can use our general result about **MAP-FN** to derive the analogous result about some particular mapping function, provided the function mapped is doubly commutative.

3.4. Dealing with Features

In this section we discuss two rather peculiar features of the logic: the interpreter, **V&C$**, and the bounded quantifier, **FOR**. The reader should see [11] for a deeper discussion of these functions. Much of this section is extracted or paraphrased from that paper.

3.4.1. Termination Theorems with V&C$

Problem. Prove the termination properties of the partial function described by

```
(APP X Y)
   ⇐
(IF (EQUAL X (QUOTE NIL))
    Y
    (CONS (CAR X) (APP (CDR X) Y))).
```

The recurrence described here is similar to **APPEND**'s but the termination condition is that **X** is **NIL** rather than that **X** is not a **LISTP**. Note that if the first argument of **APP** is **0**, the computation of **APP** runs forever:

```
(APP 0 Y)
   ⇐
(CONS 0 (APP 0 Y))
   ⇐
(CONS 0 (CONS 0 (APP 0 Y)))
   ⇐
(CONS 0 (CONS 0 (CONS 0 (APP 0 Y))))
   ⇐
...
```

And in general, since the **CDR** of a non-**LISTP** is **0**, if the first argument to **APP** does not end in **NIL** the computation runs forever. That is the first ‘‘termination property’’ of **APP**. The second is that when it terminates, **APP** returns the same thing as **APPEND**. We wish to formalize these two statements.

An easy subproblem is to formalize what we mean by ‘‘ends in **NIL**.’’ We define **PROPERP** to recognize those lists that end in **NIL**:

Definition.

```
(PROPERP L)
   =
(IF (NLISTP L)
```

```
    (EQUAL L NIL)
    (PROPERP (CDR L))).
```

That is, a non-**LISTP** is **PROPERP** only if it is **NIL**, and a **LISTP** is **PROPERP** precisely when its **CDR** is.

Now how do we get a handle on termination?

Consider for a moment the proposed definition similar to the description of **APP** above:

Definition?

```
(APP X Y)
   =
(IF (EQUAL X NIL)
    Y
    (CONS (CAR X)
          (APP (CDR X) Y))).
```

This equation is inadmissible under the principle of definition because there is no measure of the arguments that decreases in a well-founded sense. Were the above equation an axiom we could derive

(APP 0 1) = **(CONS 0 (APP 0 1))**,

contradicting the theorem **Y** $\neq$ **(CONS X Y)**.

To investigate the termination properties of **APP** we cannot add the equation ‘‘defining’’ **APP** as an axiom! We use **V&C$** instead.

First, we define **APP** using **EVAL$** as follows:

Definition.

```
(APP X Y)
   =
(EVAL$ T '(IF (EQUAL X (QUOTE NIL))
              Y
              (CONS (CAR X)
                    (APP (CDR X) Y)))
       (LIST (CONS 'X X)
             (CONS 'Y Y))).
```

As explained in detail in Chapter 4 (page 107), one effect of this definition is to add the following axioms:

Axioms.

(SUBRP 'APP) = **F**

(FORMALS 'APP) = **'(X Y)**

(BODY 'APP)

```
    =
'(IF (EQUAL X (QUOTE NIL))
     Y
     (CONS (CAR X)
           (APP (CDR X) Y)))
```

Now recall that **V&C$** takes (the quotation of) a term and an association list assigning values to (the quotations of) variable symbols. **V&C$** returns either a pair, <**v**, **c**>, indicating the value and cost of the evaluation of the term, or else returns **F** if no cost is sufficient. Thus, the intuitive idea that an expression ‘‘runs forever’’ is here formalized by the assertion that the **V&C$** of the expression is **F**.

Here then is the statement of the first termination property of **APP** that **(APP X Y)** terminates if and only if **X** is proper:

Theorem.
```
(IFF (V&C$ T '(APP X Y)
           (LIST (CONS 'X X)
                 (CONS 'Y Y)))
     (PROPERP X))
```

Note that we had to quote the expression, **(APP X Y)** and supply the map between the quoted variables **'X** and **'Y** and the variables **X** and **Y**.

The second termination property of **APP** is that if **(APP X Y)** terminates its value is **(APPEND X Y)**. The hypothesis that the expression terminates is just that the **V&C$** of it is non-**F**. The delivered value is the **CAR** of the pair returned by **V&C$**. Thus, here is an appropriate formalization:

Theorem.
```
(IMPLIES (V&C$ T '(APP X Y)
               (LIST (CONS 'X X)
                     (CONS 'Y Y)))
         (EQUAL (CAR (V&C$ T '(APP X Y)
                           (LIST (CONS 'X X)
                                 (CONS 'Y Y))))
                (APPEND X Y))).
```

Problem. State that **(RUSSELL)** does not terminate, where

(RUSSELL) $\Leftarrow$ **(NOT (RUSSELL)).**

The solution is exactly analogous to that above. Define **RUSSELL** as

Definition.
```
(RUSSELL)
```

```
    =
(EVAL$ T '(NOT (RUSSELL)) NIL)
```

so as to add the appropriate axioms for **`FORMALS`** and **`BODY`**. Then the statement that **`(RUSSELL)`** doesn't terminate is

Theorem.

```
(EQUAL (V&C$ T '(RUSSELL) NIL)
       F).
```

Problem. Formalize the termination properties of the 91-function, where

```
     (F91 X)
        ⇐
     (IF (LESSP 100 X)
         (DIFFERENCE X 10)
         (F91 (F91 (PLUS X 11)))).
```

It turns out that this function is total and computes **`(G91 X)`**, where

Definition.

```
(G91 X)
    =
(IF (LESSP 100 X)
    (DIFFERENCE X 10)
    91).
```

To formalize this, we define **`F91`** as

Definition.

```
(F91 X)
    =
(EVAL$ T '(IF (LESSP '100 X)
              (DIFFERENCE X '10)
              (F91 (F91 (PLUS X '11))))
       (LIST (CONS 'X X))).
```

Then

Theorem.

```
(AND (V&C$ T '(F91 X) (LIST (CONS 'X X)))
     (EQUAL
      (CAR (V&C$ T '(F91 X) (LIST (CONS 'X X))))
      (G91 X))).
```

The first conjunct states that **`F91`** is total. The second states that its value is that of **G91**.

In [11] we show how such theorems can be proved within the logic. Leading the theorem prover to such proofs is often difficult. We have done it for the theorems mentioned here but do not yet have enough experience to pass on reliable advice. The interested reader should see example file 15.

3.4.2. Schematic Theorems about FOR

Suppose the user of the logic desires to discuss the list obtained by doubling the elements of **L**. One way to proceed is to define **(DOUBLE-LIST L)** as follows:

Definition.
```
(DOUBLE-LIST L)
   =
(IF (NLISTP L)
    NIL
    (CONS (TIMES 2 (CAR L))
          (DOUBLE-LIST (CDR L)))).
```

For example, **(DOUBLE-LIST '(1 2 3 4))** = **'(2 4 6 8)**.

A useful theorem about **DOUBLE-LIST** is that it ''distributes'' over **APPEND**:

Theorem.
```
(DOUBLE-LIST (APPEND A B))
   =
(APPEND (DOUBLE-LIST A)
        (DOUBLE-LIST B)).
```

However, suppose that in addition, the user wished to refer to the list obtained by adding 1 to every element of **L**. In the spirit that led to the definition of **DOUBLE-LIST** the user would have to define the function **ADD1-LIST**, so that **(ADD1-LIST '(1 2 3 4))** = **'(2 3 4 5)**. Should it be necessary to use the fact that **ADD1-LIST** distributes over **APPEND**, that fact would have to be proved. One is tempted to say ''proved again.'' But since **DOUBLE-LIST** and **ADD1-LIST** are different function symbols, the first lemma cannot be used to shorten the proof of the second.

An alternative is to use the quantifier **FOR** and write such expressions as

```
     (FOR X IN L COLLECT (TIMES 2 X))
```

and

```
     (FOR X IN L COLLECT (ADD1 X)).
```

The first expression is equivalent to **(DOUBLE-LIST L)** and the second is equivalent to **(ADD1-LIST L)**.

It is possible to state very general lemmas about quantifiers. Consider for example the schematic form

```
(FOR V IN (APPEND A B) COLLECT body(V))
   =
(APPEND (FOR V IN A COLLECT body(V))
        (FOR V IN B COLLECT body(V))),
```

where *body* is understood here to be a second order variable. (Our formalization does not introduce such variables or other new syntactic classes. We adopt this notation now only for expository purposes in the same spirit in which we earlier used ∀ and ∃ in our formulas.) This lemma is easy to prove because no time is wasted considering special properties of the particular *body* used. This lemma is more useful than the fact that **DOUBLE-LIST** distributes over **APPEND** since it also handles the analogous property for **ADD1-LIST** and other such functions.

The introduction of quantifiers has three major attractions: First, quantifiers are conceptually clarifying. Logical relationships that would otherwise be buried inside of recursive definitions are lifted out into the open. The users and readers of the logic need no longer invent and remember names for many ''artificial'' recursive functions. Second, it is possible to state general purpose, schematic theorems about quantifiers, thus reducing the number of theorems needed. Third, these schematic theorems are generally easier to prove than the corresponding theorems about recursively defined instances of the quantifiers, because irrelevant concepts have been abstracted out of the theorems. Thus, quantifiers are a boon to the reader, to the writer, and to the theorem prover. However, you have to know how to use them and that is what this section is about.

Problem. State that summation distributes over **APPEND**. That is, formalize the following schematic theorem:

```
(FOR V IN (APPEND A B) SUM body(V))
   =
(PLUS (FOR V IN A SUM body(V))
      (FOR V IN B SUM body(V))),
```

Recall the abbreviation conventions for **FOR**. In particular,

```
(FOR X IN (APPEND U V)
       WHEN (MEMBER X B)
       SUM (TIMES X C))
```

is formally represented in the logic and in the implementation by

```
(FOR 'X (APPEND U V)
        '(MEMBER X B)
        'SUM '(TIMES X C)
        (LIST (CONS 'B B) (CONS 'C C))).
```

Note that in the abbreviated form of **FOR** the bound variable symbol **X**, the conditional expression **(MEMBER X B)**, the operator **sum**, and the body **(TIMES X C)**, are written as terms but become quoted constants in the formal representation.

To state a general purpose ''schematic'' theorem it is necessary to write down a **FOR** expression in which the bound variable, condition, operation, and/or body are all arbitrary terms satisfying given conditions, rather than constants. Thus, schematic theorems about **FOR** use the unabbreviated, formal form of **FOR**, namely **(FOR var range cond op body alist)**.

The schematic theorem expressing the idea that summation distributes over **APPEND** is as follows:

Theorem.

```
(EQUAL (FOR V (APPEND A B) COND 'SUM BODY ALIST)
       (PLUS (FOR V A COND 'SUM BODY ALIST)
             (FOR V B COND 'SUM BODY ALIST))).
```

Note that the equation holds no matter what the bound variable symbol is and no matter what the conditional and body expressions are. For example, the following concrete use of **FOR**, written in the abbreviated form

```
(FOR I IN (APPEND L1 L2) SUM (TIMES 2 I))
```

is just an abbreviation for[3]

```
(FOR 'I (APPEND L1 L2) '(TRUE)
     'SUM '(TIMES 2 I) NIL).
```

We can instantiate the ''schematic'' theorem above, replacing **V**—the bound variable—by **'I**, **A** and **B** by **L1** and **L2** respectively, **COND** by **'(TRUE)**, **BODY** by **'(TIMES 2 I)**, and **ALIST** by **NIL**, and derive immediately the equivalence of the concrete **FOR** with

```
(PLUS (FOR 'I L1 '(TRUE) 'SUM '(TIMES 2 I) NIL)
      (FOR 'I L2 '(TRUE) 'SUM '(TIMES 2 I) NIL)),
```

which, in abbreviated form is

---

[3]The use of **'(TRUE)** in the abbreviated term is actually incorrect. The term abbreviated contains the expression **'(QUOTE *1*TRUE)** instead of **'(TRUE)**. These two have the same value under **EVAL$**. See page 90.

```
(PLUS (FOR I IN L1 SUM (TIMES 2 I))
      (FOR I IN L2 SUM (TIMES 2 I))),
```

as one would have expected.

Problem. Formalize the statement that summation of a **PLUS** expression is the **PLUS** of the summations. Or, in everyday mathematics,

$$\sum_{i=0}^{n} g(i)+h(i) \;=\; \sum_{i=0}^{n} g(i) \;+\; \sum_{i=0}^{n} h(i);$$

If we had higher order variables *g* and *h* we could write

```
(FOR I IN L SUM (PLUS g(I) h(I)))
   =
(PLUS (FOR I IN L SUM g(I))
      (FOR I IN L SUM h(I))).
```

However, in our logic we don't have such variables, but we don't need them here. It is sufficient simply to use the unabbreviated form of **FOR** and use ordinary variable symbols **G** and **H** to describe (the quotation of) an arbitrary **PLUS** expression:

Theorem.

```
(EQUAL (FOR V L COND 'SUM (LIST 'PLUS G H) ALIST)
       (PLUS (FOR V L COND 'SUM G ALIST)
             (FOR V L COND 'SUM H ALIST))).
```

Again, note that this theorem can be instantiated so as to let us distribute the summation over the two branches of the body below:

```
(FOR I IN L SUM (PLUS I (SQ I)))
```

The important thing to consider is that the body of the above **FOR** is actually **'(PLUS I (SQ I))** or, put another way, **(LIST 'PLUS 'I '(SQ I))**.[4]

In stating some classic quantifier manipulation theorems it is necessary to put elaborate syntactic constraints on the form of the body expression. Here is a simple example:

---

[4]One extremely useful application of backquote notation is in connection with schematic theorems about **FOR**. The left-hand side of the theorem above can be written **(FOR V L COND 'SUM `(PLUS ,G ,H) ALIST)** provided the **NQTHM** setting of **BACKQUOTE-SETTING** is in use (see page 125).

Problem. Formalize the theorem that the summation of an expression not involving the bound variable is a constant. In particular, we would like the formal version of the theorem that permits us to reduce $\sum_{i \varepsilon s} n$ to simply |s|*n.

To do this we must define the notion that the body does not ''involve'' the bound variable. Suppose we have defined **(FREE-VARS T X)** to be the variables in the (quotation of a) term **X**. Then the first cut at a formalization of the goal is

```
(IMPLIES (NOT (MEMBER V (FREE-VARS T N)))
         (EQUAL (FOR V S '(TRUE) 'SUM N ALIST)
                (TIMES (LENGTH S) N)))
```

However, this is incorrect because what is being summed by the **FOR** is the **EVAL$** of **N** under successive extensions to **ALIST** produced by binding **V**. Thus, instead of **N** in **(TIMES (LENGTH S) N)** we should write **(EVAL$ T N ALIST)**. An accurate statement of the rule is then

Theorem.
```
(IMPLIES (NOT (MEMBER V (FREE-VARS T N)))
         (EQUAL (FOR V S '(TRUE) 'SUM N ALIST)
                (TIMES (LENGTH S)
                       (EVAL$ T N ALIST))))
```

Note that this theorem can be used to simplify a concrete **FOR** provided the conditional expression in the **FOR** is **'(TRUE)** and the bound variable does not occur in the body. The restriction on the conditional expression can be relaxed by replacing **LENGTH** with the quantifier statement that counts the number of times the conditional is true in **S**.

We conclude our discussion of **FOR** by exhibiting the formalization of the concept of **FREE-VARS**. This is particularly appropriate because it forces us to consider the form of the quotations of terms, a topic with which the user must be familiar if accurate schematic theorems are to be written.

The definition of **FREE-VARS** recursively explores terms in the same way that **EVAL$** does. When **X** is treated as a term there are four cases: **X** is a variable symbol (i.e., **(LITATOM X)**), **X** is a **QUOTE**d constant, (i.e., **(LISTP X)** and the **CAR** of **X** is **'QUOTE**), **X** is a function application (i.e., **(LISTP X)** but the **CAR** is not **'QUOTE**), and **X** is of any other form (e.g., a **NUMBERP**). In the case that **X** is a function application we must recursively collect the variables in all of the arguments, and so we are dealing with another case of mutual recursion. Here is a suitable definition:

Definition.
```
(FREE-VARS FLG X)
  =
(IF (EQUAL FLG 'LIST)
```

```
    (IF (NLISTP X)
        NIL
        (UNION (FREE-VARS T (CAR X))
               (FREE-VARS 'LIST (CDR X))))

(IF (LITATOM X)
    (LIST X)
(IF (NLISTP X)
    NIL
(IF (EQUAL (CAR X) 'QUOTE)
    NIL
    (FREE-VARS 'LIST (CDR X))))))
```

Perhaps the most interesting thing about this definition is that it is not necessary for **FREE-VARS** to ‘‘know about’’ **FOR**. The reason is that **FOR** is just like every other function in our logic. It does not really introduce a ‘‘bound’’ variable or have terms as its arguments. Is **I** among the free variables of **(FOR I IN L SUM (PLUS I X))**? That is to say, is **'I** a member of the list returned by **FREE-VARS** on the quotation of the **FOR** term? The answer is no. The formal term abbreviated by **(FOR I IN L SUM (PLUS I X))** is **(FOR 'I L '(TRUE) 'SUM '(PLUS I X) (LIST (CONS 'X X)))**. The quotation of this term is **'(FOR 'I L '(TRUE) 'SUM '(PLUS I X) (LIST (CONS 'X X)))**. The result of **FREE-VARS** on this is **'(L X)**.

In [11] we show many schematic theorems about **FOR**. In example file 15 we show the events leading to the proofs of those theorems. The reader interested in using **FOR** is urged to look at those events.

3.5. Nondeterminism

Problem. Define a nondeterministic merge function which merges two input streams (finite lists) into a single output stream.

Below is an alleged definition of the concept

Definition.

```
(MERGE A B)
   =
(IF (NLISTP A)
    B
(IF (NLISTP B)
    A
(IF (RANDOM-BOOL)
    (CONS (CAR A)
```

```
          (MERGE (CDR A) B))
    (CONS (CAR B)
          (MERGE A (CDR B))))))
```

where **RANDOM-BOOL** is an undefined function.

The trouble with this definition is that **(RANDOM-BOOL)** is a constant. It is either **F** or non-**F**. Indeed,

Theorem.
```
(IMPLIES (AND (LISTP A)
              (LISTP B))
         (OR (EQUAL (MERGE A B) (APPEND A B))
             (EQUAL (MERGE A B) (APPEND B A)))).
```

Is this a property of a nondeterministic merge function? Probably not.

A typical patch is to make **RANDOM-BOOL** a function of **A** and **B**. Thus, at each stage in the recursion, an element of **A** or an element of **B** is chosen as an undefined function of the current values of **A** and **B**. This function can produce more varied output than the previous one, depending on the properties of **RANDOM-BOOL**. But the mere fact that the element chosen at each stage is a function of the current inputs permits some deductions about the function's behavior. For example, there is no definition of **RANDOM-BOOL** that makes **(MERGE '(A) '(B))** = **'(A B)** and **(MERGE '(C A) '(B))** = **'(C B A)**, even though both desired outputs are legal merges of the inputs. The reason is that once **(MERGE '(C A) '(B))** chooses to output **C** and recurses to **(MERGE '(A) '(B))**, the recursive call must yield **'(A B)**.

Clearly, **MERGE** can be patched again so that more ''state'' is given to **RANDOM-BOOL** to prevent this too. Indeed, the following definition of **MERGE** has the desired property:

Definition.
```
(MERGE A B)
   =
(MERGE1 A B (ORACLE A B))
```

where

Definition.
```
(MERGE1 A B ORACLE)
   =
(IF (NLISTP A)
    B
(IF (NLISTP B)
    A
(IF (CAR ORACLE)
```

```
    (CONS (CAR A)
          (MERGE1 (CDR A) B (CDR ORACLE))
    (CONS (CAR B)
          (MERGE1 A (CDR B) (CDR ORACLE)))))))
```

and **ORACLE** is an undefined function.

The basic idea is to provide the nondeterminacy by providing an additional input, the variable **ORACLE** in **MERGE1**, that is entirely responsible for the ‘‘nondeterministic choices’’ and to leave the value of that input unconstrained.

It is the case that for any set of triples <**a**, **b**, **x**> with the property that **x** is an interleaving of **a** and **b**, there exists a definition of the function **ORACLE** sufficient to make **(MERGE a b)** = **x** for every triple in the set. This can be proved within the logic for all finite sets of triples. (Quick! How do you capture the notion ‘‘there exists a definition of **ORACLE**?’’)

For a more elaborate example of the use of oracles, see the hardware verification work of [29]. There, it is necessary to model the response of memory to read/write requests. Oracles are used to permit arbitrary delays.

An alternative approach to the merge problem is not to *define* **MERGE** but to introduce it by constraint. In particular,

Constraint.
Constrain the function **MERGE** to have the property
(PERM (MERGE X Y) (APPEND X Y)).

This introduces **MERGE** as a function symbol of two arguments and adds the axiom that it produces a permutation of the concatenation of its arguments. For this event to be admissible the user must supply a ‘‘witness’’ for **MERGE** that satisfies the formula to be added as an axiom. A suitable witness is easy: **MERGE** could be **APPEND**.

Perhaps the biggest philosophical obstacle in the attempt to introduce **MERGE** as a function is that it must be a function. In particular, **(MERGE A B)** is always equal to **(MERGE A B)**, whether **MERGE** is defined or constrained.[5] A related remark can be made that some people may find reassuring: **(MERGE1 A B ORACLE1)** need not be the same as **(MERGE1 A B ORACLE2)**. Despite the obviousness of these observations we have seen informed users of the logic fail to make similar ones.

For example, suppose you are writing an interpreter within the logic for a programming language like Lisp (see the next section). Suppose your interpreter has the notion of an ‘‘environment’’ by which some program variables are assigned values. Let **(LOOKUP SYM ENV)** be the expression you use to denote the value of the program variable **SYM** in the environment. How do you

[5] Isn’t that a silly thing to have to say?

define **LOOKUP** to cope with the possibility that a program variable is ‘‘unbound,’’ i.e., not assigned a value in the environment? One possibility is to declare an undefined function symbol, **UNDEF**, and to return **(UNDEF)** as the value of such a **LOOKUP**. That captures some of the desired features: it will be impossible to prove much about the type or value of unbound variables. But one consequence of this approach is that all unbound program variables are thus specified to have the same (otherwise undetermined) value. That is, **(UNDEF)** equals **(UNDEF)**.

There are three conventional ways out: (a) change the problem so that nondeterministic behavior is eliminated, e.g., in the programming language example, adopt the specification that all variables have the initial value **0**; (b) model the nondeterminism in a trivial, deterministic way but avoid asking questions that depend upon your trivial convention, e.g., define the notion of a well-formed program (in which unbound variables never arise) and include among the hypotheses of all your theorems the constraint that the programs are well-formed; or (c) model the nondeterminism carefully and accurately, e.g., include an oracle in the environment.

3.6. Computing Machines and Programming Languages

Problem. Formalize a simple programming language and prove some program correct.

We discuss this formalization problem at length here because computing machines and programming languages are among the most commonly formalized systems in our logic. An operational approach is most often taken, but we discuss some other alternatives later. In this example we merely sketch a formalization that is done completely in example file 17.

3.6.1. An Operational Semantics

The programming language we will formalize is a simple, execute-only assembly-level language. It is a simplification of the Piton language formalized in [42]. A typical program in this language is shown below, in an imagined ‘‘external’’ syntax. The program multiplies the contents of register 0 by the contents of register 1, leaving the answer in register 2.

```
SUBROUTINE TIMES
0  MOVI 2,0  ; Move (immediate) a 0 into reg 2
```

```
1  JUMPZ 0,5 ; Jump to instruction 5 if reg 0 contains 0
2  ADD 2,1   ; Add contents of reg 1 into that of reg 2
3  SUBI 0,1  ; Subtract (immediate) a 1 from contents of reg 0
4  JUMP 1    ; Jump to instruction 1
5  RET       ; Return from subroutine call
```

Such programs are represented formally by their parse trees. The above program is represented by the constant

```
'(TIMES
   (MOVI 2 0)
   (JUMPZ 0 5)
   (ADD 2 1)
   (SUBI 0 1)
   (JUMP 1)
   (RET)).
```

Programs will be given meaning operationally. In particular we will define a notion of ‘‘machine state’’ and a ‘‘state transition’’ function and then consider the states produced by running programs a given number of instructions.

The state of our machine will be represented by instances of the shell class **ST**.

Shell Definition.
Add the shell **ST** of 5 arguments
with recognizer function **STP**,
accessor functions **PC**, **STK**, **MEM**, **HALTEDP**, and **DEFS**, and
default functions **ZERO**, **ZERO**, **ZERO**, **ZERO**, and **ZERO**.

Intuitively, **DEFS** is a list of programs as above; the ‘‘program counter,’’ **PC**, is a pair, e.g., **(TIMES . 0)**, pointing to some instruction in some program of **DEFS**; **STK** is a ‘‘subroutine call stack’’ containing the suspended program counters of the active subroutine calls; **MEM** is a list of the contents of an arbitrary number of consecutive memory locations starting at ‘‘address’’ 0; and **HALTEDP** indicates the status of the computation. If **HALTEDP** is **F** the computation is ‘‘proceeding normally;’’ otherwise we say it has ‘‘halted.’’ We adopt the convention of referring to the first few memory locations as ‘‘registers.’’

The ‘‘state transition’’ function is

Definition.
```
(STEP S)
    =
(IF (HALTEDP S)
    S
    (EXECUTE (FETCH (PC S) (DEFS S)) S)).
```

Thus, **STEP** is a no-op if the state has already halted. Otherwise, we **FETCH** the instruction indicated by the program counter and **EXECUTE** that instruction on the current state. **EXECUTE** is just a big case split, on the ‘‘op-code’’ of the instruction:

Definition.

```
(EXECUTE INS S)
   =
(LET ((OP (CAR INS))   ; INS = '(op a b ...)
      (A  (CADR INS))
      (B  (CADDR INS)))
 (CASE OP
  (MOVE (MOVE A B S))
  (MOVI (MOVI A B S))
  (ADD  (ADD  A B S))
  ...
  (CALL (CALL A S))
  (RET  (RET S))
  (OTHERWISE S))).
```

For each instruction in the programming language we define a function in the logic that takes as its arguments the operands of the instruction and the current state. The function returns the appropriate successor state. The functional version of the **ADD** instruction is shown below.

Definition.

```
(ADD ADDR1 ADDR2 S)
   =
(ST (ADD1-PC (PC S))
    (STK S)
    (PUT ADDR1
         (PLUS (GET ADDR1 (MEM S))
               (GET ADDR2 (MEM S)))
         (MEM S))
    F
    (DEFS S))
```

The program counter of the resulting state is one greater than that of the given state. The stack is unchanged. The memory is modified by storing at **ADDR1** the sum of the contents of the two operands, **ADDR1** and **ADDR2**. The halt flag is **F**. The program space is unmodified.

In the interests of brevity we will not give the definitions of all other instructions. They can often be inferred from their names and the definition of **ADD**. **MOVE**, for example, is like **ADD** but stores the contents of **ADDR2** into **ADDR1**. **MOVI** (move immediate) is like **MOVE** except stores **ADDR2** (rather than its contents) into **ADDR1**. **CALL** replaces the program counter by a pair that points to the first instruction of the called subroutine, **'(addr1 . 0)**, and pushes onto the stack the result of incrementing the current program counter. **RET** sets the **HALTEDP** flag to **T** if the stack is empty and otherwise restores the program counter from the top of the stack and pops the stack. See example file 17 for details.

Finally, we can define our ''small machine'' with

Definition.
```
(SM S N)
   =
(IF (ZEROP N)
    S
    (SM (STEP S) (SUB1 N))).
```

This function applies **STEP** to the initial state, **S**, **N** times. Since **STEP** is a no-op on halted states, we might have said ''This function runs **S** a maximum of **N** steps, or until termination.''

In more realistic languages instructions can cause ''errors.'' This can be accomodated by permitting the **HALTEDP** flag to take on non-**F** ''error codes'' as values. Such codes would effectively halt the machine defined above. Then, for example, we might redefine **EXECUTE** to set the halt field to **'ILLEGAL-OPCODE** in its ''otherwise clause'' above, signalling an error when an unrecognized instruction is encountered. This error would then be signalled if the program counter were ever incremented ''too far'' in some program, because the **FETCH** in **STEP** would return the illegal instruction 0. Similarly, we might redefine **ADD** to set the halt field to **'ARITHMETIC-OVERFLOW** if the required sum is ''too big,'' and **CALL** to signal **'UNDEFINED-LABEL** if the called program were not among the definitions of the state. This approach allows each instruction to signal errors without complicating (or even changing) the machine as defined above.

Many practical languages include instructions that cause unspecified effects on some parts of the state. This happens frequently when one is formalizing an informally described machine or language: the informal text simply may not enumerate the effect of every instruction on every part of the state. For example, suppose the machine in question has a Boolean ''condition code,'' intended to indicate the arithmetic sign of the result of the most recently completed operation. The informal specification of the arithmetic instructions will probably include a description of how this condition code is set. But the informal specification of, say, the **CALL** instruction, may not. In the absence of more information, a faithful formalization of such a text must somehow leave this field unspecified after **CALL**. At first sight the operational approach here fails, since there is in fact no functional relation between successive states; indeed, one might imagine a whole family of possible successor states any one of which may be chosen. The user of such a machine designs programs that do not depend on which of the allowed successors is actually taken. It is possible to formalize such a language while staying within the operational framework by introducing unspecified functions for ''determining'' the unspecified parts of the successor state. In the case of the post-**CALL** condition code, we wish to require that the new condition code be Boolean valued but otherwise we do not know anything about it. We might therefore introduce by constraint a Boolean valued function, **POST-CALL-CC**, that takes one argument, a machine state.

Constraint.
Constrain the function **POST-CALL-CC** to have the property

```
(OR (TRUEP (POST-CALL-CC S))
    (FALSEP (POST-CALL-CC S))).
```

A suitable witness is the function that always returns **T**. We could then use **POST-CALL-CC** to specify the value of the condition code after the **CALL** instruction. In this way we actually define a whole family of languages in an operational way. This is an example of the use of constrained functions to avoid ''overspecification.'' If a formalization ''feels too confining'' the judicious use of constrained functions may help.

3.6.2. Proving Programs Correct

Having defined the machine we can now turn to correctness theorems for programs. One theorem we can prove is that if **SM** is used to execute a call of the **TIMES** program shown above, on a state containing **I** in register 0 and **J** in register 1, and **SM** is allowed to run 4+4**I** instructions, then register 2 of the resulting state contains **I*****J**. This form of ''correctness theorem'' for **TIMES** appears inadquate in two respects. The first is that it seems to address only the situation in which the call of **TIMES** is given exactly the required amount of time to run. The second is that it does not tell us anything about the resulting state except the contents of register 2. It turns out that the first objection is not serious but the second is. We return to this point in a moment.

We can prove theorems about **SM** that have nothing to do with any particular program. One extremely useful theorem is

Theorem. SM-PLUS

```
(EQUAL (SM S (PLUS I J))
       (SM (SM S I) J)).
```

That is, an execution of length **I**+**J** can be decomposed into one of length **I** followed by one of length **J**. An equivalent result is that if **N** is a natural number and **N**≥**K** then **(SM S N)** is the same as running **S K** steps and then running **N**–**K** more steps. This result overcomes the first objection to the correctness theorem for **TIMES**. If **S** is a state poised to execute a call of **TIMES** and **N**≥ 4+4**I**, where **I** is the contents of register 0 in **S**, then **(SM S N)** is the same as **(SM (SM S** 4+4**I) N**–4+4**I)**. That is, we can break the run into two parts, the first of which is exactly the ''right'' length for the **CALL** of **TIMES**.

However, this highlights the second inadequacy in the correctness result for **TIMES**. To use it to prove the correctness of programs that **CALL TIMES**, it is necessary (at least in many cases) to characterize the entire state produced by the **CALL**, not just the ''answer.''

We now give a useable correctness theorem for our **TIMES** program. A formal statement of the theorem is given below. We paraphrase and explain it here. First, suppose **S** is a state poised to execute a **CALL** of **TIMES**. By that we mean that the program counter of **S** points to a **'(CALL TIMES)** instruction and that the definition of **TIMES** in **S** is in fact the program constant we exhibited above. Characterizing the state this way permits the theorem to be applied to any call of **TIMES** in any program and permits the list of definitions to be arbitrary as long as it includes our **TIMES** somewhere. Next, suppose that the memory of **S** contains at least three registers. This is necessary since the program uses register 2. Let us call the contents of register 0 and 1 **I** and **J** respectively. Suppose **I** is a natural number. We comment on this condition in a moment. Finally, suppose the halt field of **S** is **F**, i.e., the state has not been halted.

Given these conditions, the theorem tells us explicitly what state is produced by running **S** 4+4**I** steps. In particular, it is the state obtained from **S** by

- incrementing the program counter by one,
- leaving the stack is unmodified,
- storing a 0 into register 0 and **I*****J** into register 2,
- leaving the halt flag **F**, and
- leaving the definitions unmodified.

The condition that **I** be numeric comes from our decision to characterize the new memory as having a 0 in register 0. If **I** is not a natural number the program terminates without changing register 0.

A formal statement of this correctness result is shown below. We have defined **(TIMES-PROGRAM)** to be the **TIMES** subroutine constant above and **(TIMES-CLOCK I)** to be 4+4**I**. We adopt the discipline of abstracting the runtime of each program into a corresponding ''clock'' function so composition is easier. The beautiful thing about the following result is that it is easy to prove and, in combination with the **SM-PLUS** theorem, permits us in the future to treat

(CALL TIMES) exactly as we would a primitive instruction, except that the instruction count is decremented by **(TIMES-CLOCK I)** instead of by one.

Theorem. TIMES-CORRECT
```
(IMPLIES
 (AND (EQUAL (FETCH (PC S) (DEFS S))
             '(CALL TIMES))
      (EQUAL (ASSOC 'TIMES (DEFS S))
             (TIMES-PROGRAM))
      (LESSP 2 (LENGTH (MEM S)))
      (EQUAL I (GET 0 (MEM S)))
      (EQUAL J (GET 1 (MEM S)))
      (NUMBERP I)
      (NOT (HALTEDP S)))
 (EQUAL (SM S (TIMES-CLOCK I))
        (ST (ADD1-PC (PC S))
            (STK S)
            (PUT 0 0
              (PUT 2 (TIMES I J)
                (MEM S)))
            F
            (DEFS S))))
```

We call theorems such as that above the ‘‘standard correctness’’ theorem for the subroutine concerned, in this case **TIMES**. The reader interested in how Nqthm can be led to the proofs of such theorems should see example file 17. That file also illustrates the use of this theorem in the proof of the standard correctness theorem for a program that calls **TIMES**. That is, the file illustrates how standard correctness theorems can be combined.

3.6.3. Termination and Non-Termination

Observe that the standard correctness theorem implicitly establishes that the subroutine terminates: control eventually returns to the instruction after the **CALL** and the stack is unchanged. One might even define ‘‘termination’’ with this condition on the program counter and stack. However, there is an attractively direct statement of termination, that we call the ‘‘standard termination’’ theorem.

The standard termination theorem for some subroutine, **fn**, exhibits an expression, **clock**, which is generally a function of the ‘‘input’’ to **fn**, with the property that a ‘‘top-level’’ execution of the definition of **fn** sets the halt flag in **clock** steps. By a ‘‘top-level’’ execution we mean one starting in a state in which **fn** is defined as desired, the program counter points to the first instruction in the program, **`'(fn . 0)`**, and this is the ''main program,'' i.e., the stack is **`NIL`**. The standard termination theorem is attractive because it establishes that in a certain context, the execution of **`fn`** actually sets the processor's halt flag within an explicitly given number of steps. We return to this idea in the next section.

It is also possible to state and (sometimes, at least) prove that some programs do not terminate, or do not terminate under certain conditions on their inputs. In particular, one might prove that, under certain conditions on the state **`S`**, **`(NOT (HALTEDP (SM S N)))`**. Since such a theorem is true for all values of **`N`**, we feel justified in claiming that **`S`** is a ''non-terminating'' state.

3.6.4. Proving Theorems about the Language

Because we have formalized the semantics of the programming language it is possible to prove theorems about the language itself, rather than just about particular programs. We have already seen one such theorem,

Theorem. **`SM-PLUS`**

```
(EQUAL (SM S (PLUS I J))
       (SM (SM S I) J)).
```

This theorem puts no restrictions on the program being run.

The following theorem is a non-termination result. It says that if the current program counter in a running state points to a **`JUMP`** to itself, the state is non-terminating. Again, no particular program is involved.

Theorem.

```
(IMPLIES (AND (NOT (HALTEDP S))
              (EQUAL (FETCH (PC S) (DEFS S))
                     (LIST 'JUMP I))
              (NUMBERP I)
              (EQUAL (CDR (PC S)) I))
         (NOT (HALTEDP (SM S N)))).
```

A final illustration of a general theorem about the **`SM`** language relates the standard correctness theorem to the standard termination theorem. The theorem may be paraphrased as follows. If the current instruction in a running state is a **`CALL`** of some program **`prog`**, and in some non-zero number of steps, **`n`**, the stack is the same as it was at the **`CALL`**, then a top-level execution of **`prog`** sets the halt flag within **`n`** steps.

Theorem.

```
(IMPLIES (AND (NOT (HALTEDP S))
              (EQUAL (FETCH (PC S) (DEFS S))
```

```
                    (LIST 'CALL PROG))
             (NOT (ZEROP N))
             (EQUAL (STK (SM S N)) (STK S)))
        (HALTEDP
         (SM (ST (CONS PROG 0)
                 NIL
                 (MEM S)
                 F
                 (DEFS S))
             N))).
```

This theorem is difficult to prove because it involves two different executions of the program: the ‘‘continuing execution’’ in state **S** and the ‘‘top-level execution’’ in the state mentioned in the conclusion above. A sketch of the argument is as follows. From the hypotheses we can conclude that the continuing computation executes a return instruction ‘‘balancing’’ the **CALL** within **N** steps. Say this happens at step **k**. (We are not told that **N** is the minimum number of steps necessary to get back to the original stack; indeed, the hypotheses allow for the called program to return immediately and for the computation to proceed, possibly even exiting from the caller before eventually reentering!) Then, with the help of various general theorems about **SM**, which establish that the two computations proceed isomorphically through the program, we can conclude that in the top-level execution, at the **k**–1st step, a return is executed while the stack is **NIL**. Such a return sets the halt flag and the flag stays set for the remaining **n**–(**k**–1) steps of the top-level execution. The proof of this theorem is in example file 17.

3.6.5. Other Approaches

The idea of using constrained functions to avoid overspecification can be generalized. Suppose, for example, we introduce by constraint two functions,

Constraint.
Constrain the functions **STATEP** and **STEP** to
have the property
(IMPLIES (STATEP S) (STATEP (STEP S))).

We could then define a ‘‘generic interpreter’’

Definition.

```
(INTERP S N)
   =
(IF (ZEROP N)
    S
```

```
    (INTERP (STEP S) (SUB1 N)))
```

and prove some interesting theorems about it, such as

Theorem.
```
(IMPLIES (STATEP S) (STATEP (INTERP S N)))
```

Theorem.
```
(EQUAL (INTERP S (PLUS I J))
       (INTERP (INTERP S I) J)).
```

Any particular interpreter whose definition is analogous to that of **INTERP**, i.e., is just the iterated application of some ‘‘step’’ function, enjoys the properties of the generic interpreter. Indeed, the particular properties can be derived from the generic ones by functional instantiation.

Similarly, we might use a combination of constrained and defined functions to introduce states that have a control stack and halt flag but are otherwise unspecified. We might introduce a step function that implements **CALL** and **RET** but leaves unspecified all other state components—with the important exception that none of the other instructions change the control stack or halt flag. We would have thus defined a language with subroutine call and return, without specifying any of the ‘‘primitive’’ operations in the language. We could then prove as generic theorems the theorems relating, say, call and return to top-level termination or otherwise develop the theory of subroutine call, without having to worry about the particular operations carried out by the language. We could then define some particular language, providing many elaborate primitive operations, and immediately take advantage of the ‘‘theory of call and return’’ without having to rederive it amid the complexities of the actual state and instruction set.

It is possible to relate the operational semantics to other approaches towards formalizing program semantics. In the example file we show how one can prove that particular programs ‘‘correspond’’ to recursively defined functions that carry out the same steps. This is sometimes called the ‘‘functional approach’’ to modeling semantics and was suggested was McCarthy in [40]. For complicated **SM** programs it is often most convenient to first prove that the operational interpretation of the program corresponds to some particular, and usually peculiarly defined, function and then prove that the function has the desired properties. The first step is straightforward because the function in question exactly mimics the execution of the program. But by taking the first step one eliminates **SM** from the problem in the second step.

Also in the example file we illustrate how it is possible to ‘‘trade in’’ the verification conditions generated by a straightforward application of the so-called ‘‘inductive assertion’’ method of Hoare [27] and Floyd [21], for a theorem about the operational interpretation of a given program.

It should be noted that the logic can be used to address program correctness problems without necessarily formalizing the language semantics in the logic. For example, in [6] we present a verification condition generator (*vcg*) for a subset of ANSI Fortran 66 and 77. A verification condition generator is a program that takes two inputs, a program and a specification, and generates formulas, called ‘‘verification conditions,’’ that, if theorems, establish that the program satisfies its specification. The language semantics are embedded in the vcg program, and the logic is used simply to express specifications and the verification conditions. We present a vcg for a simple language in [5].

3.7. Embedded Formal Systems

An increasingly common use of our system is in the definition of what we will call ‘‘interpreters’’ for other systems. Such systems might be programming languages or computing machines, as discussed above. However, the operational approach is extremely flexible and can be used to deal with a variety of other formal (or formalized) systems. We give six brief examples ranging from an interpreter for a simple physical system, through more obviously formal systems such as hardware design to purely formal systems such as lambda-calculus and set theory.

3.7.1. Real Time Control

The simple real time control problem discussed in [4] is to show that a certain control program keeps a vehicle on a straight-line course in a varying crosswind. We formalized the problem with an interpreter that is essentially a simulator for (a drastic simplification of) the physical system. The interpreter takes two arguments. The first is the ‘‘state’’ of the vehicle and the world—the vehicle position and velocity and the wind velocity. The second is a ‘‘wind history’’—a list of wind velocity increments describing how the wind changes at discrete time units over some finite time interval. The interpreter determines the final state of the vehicle by ‘‘simulating’’ each state change, that is, by calculating for each sampling unit the new wind velocity, the resulting position of the vehicle, the associated sensor reading, the response of the control program, and the effect on the vehicle’s velocity. Two theorems are proved about the simulated vehicle: the vehicle does not wander outside of a certain corridor if the wind changes ‘‘smoothly,’’ and the vehicle homes to the proper course if the wind stays steady for a certain amount of time. For the events necessary to lead the theorem prover to these proofs, see example file 5.

3.7.2. Games

Game playing programs raise many of the same issues raised in control problems. Such programs attempt to ‘‘control’’ the board in the face of hostile interaction with a human opponent. Games may be formalized with interpreters that can be thought of as ‘‘referees.’’ Such an interpreter takes a series of boards as its input and announces whether a legal game was played and, if so, what the outcome was. A game playing program is a function which generates the board sequence as a function of the moves of the opponent. Proving that a certain strategy wins (or never loses) is very similar to proving some control loops: an invariant is identified and shown to be maintained by the strategy and to result in a win in the terminal state. This proof method is applied to the game of Nim, where the two players alternately remove sticks from various piles, in example file 53. In file 18 we prove that a certain strategy for playing tic-tac-toe never loses. That proof is interesting because it uses a brute force method, forcing Nqthm to consider all possible games. Specifying game playing programs is surprisingly subtle. Intuitively, one merely wishes to say that the game playing program generates a board sequence that is approved by the referee and ends in a win, say, for the program. But this ignores many issues. For example, the game playing program must respect the opponent’s moves: the resulting sequence of boards must accurately reflect the moves by the opponent. Imagine a game playing program that refused to put your piece on the specified square because it upset the program’s plan! But this respect for the opponent’s moves means that not all games generated are legal. A more accurate specification must say that if an illegal game is generated then it is the fault of the opponent. Another problem is that the game playing program cannot ‘‘look ahead’’ into the opponent’s moves, i.e., it cannot prompt the opponent for two moves before replying to the first. We deal with these problems for tic-tac-toe in example file 18. These kind of problems also arise when one tries to formalize the interaction between a control program and the environment. Such problems are easier to study in the setting of games because the rules of engagement are well understood.

3.7.3. Towers of Hanoi

The Towers of Hanoi is a child’s puzzle with which you are probably familiar. It is played on a board with three pegs and a stack of disks each of which has a peg-sized hole in the center and a diameter different than that of all other disks. Initially the disks are arranged in a stack on a single peg so that no disk sits atop one that is smaller. The object of the game is to move the disks one at a time so as to reassemble the initial stack on one of the two initially empty pegs without ever placing a disk atop one that is smaller. Writing a computer program to ‘‘solve’’ this problem is a typical exercise in programming language courses when introducing recursion. We wished to write a function that solves the problem, and we wished to prove that it is correct.

In our formalization, the function called **HANOI** takes as input three pegs, **A**, **B**, and **C**, and a number **n** and generates a list of ‘‘moves’’ that supposedly is sufficient to move a tower of **n** disks from peg **A** to peg **C** using peg **B** for temporary storage. A ‘‘move’’ is just a pair <**i**, **j**> whose interpretation is that the top disk on peg **i** is to be moved to peg **j**. To state the correctness of **HANOI** we defined an ‘‘interpreter’’ for a list of moves. The interpreter takes as arguments a ‘‘state’’ and a ‘‘move list’’ and returns either the final state or **F** indicating that an illegal move was attempted. A state is simply a triple of lists, each list representing the configuration of a given peg by showing the diameters of the disks on that peg. The interpreter is defined to carry out each move on the state successively, checking each time that the designated source peg is non-empty and that its top disk is smaller than the top disk, if any, on the designated destination peg. The theorem proved was that if the initial state was as expected—i.e., there is a well-formed tower on peg **A** and the other pegs are empty—then the final state, after playing out the moves delivered by **(HANOI A B C N)**, is non-**F** and configured as desired—i.e., the tower is on peg **C** and the other pegs are empty.[6] The assurance that **HANOI** never put a disk on top of a smaller one is not stated explicitly in the theorem but rather follows from the definition of the interpreter.

3.7.4. Hardware Verification

Combinational logic can be modeled by propositional expressions. That is, **(AND S1 S2)** can be thought of as a circuit that takes two input bits, **S1** and **S2**, and delivers their logical **AND**. However, consider modeling a device, such as a cpu, that contains state-holding components that are changed over time by combinational logic. Such a model of a microcoded cpu is presented in [29]. Essentially a register-transfer model is used, except the model is formalized as a recursively defined function. All but one of the arguments of the function represent state-holding components in the device, such as the registers of the processor, the microaddress register, the memory, etc. One argument is an oracle that represents the values found on various input lines at each clock cycle

[6]Actually, the statement of the theorem is rather delicate because one must know that **A**, **B**, and **C** are distinct peg numbers. The proof was carried out by Matt Kaufmann.

over some time interval, as well as ‘‘nondeterministic’’ parameters determining memory response time. The definition of the function is such that if the oracle is exhausted, the entire state is returned. Otherwise, the function recurses, using combinational logic expressions (and, where appropriate, input from the oracle) to determine the new value of every state holding component. The theorem proved about this machine is that, under a certain abstraction, it is equivalent to a machine-code interpreter. In [30], the formalization of a fragment of a commercial hardware description language is described. The formalization is an interpreter which takes as one of its arguments a ‘‘netlist’’ describing the interconnection of a collection of ‘‘modules.’’ Some of the modules are primitive and others are defined. Another argument to the interpreter specifies initial values on input lines. The interpreter delivers both the values on the output lines and the values of the ‘‘state holding devices’’ in the netlist. Functions are also provided for determining the fanout, gate-delay, and other aspects of the design. An extensive library of lemmas was developed to allow hierarchically composed proofs about designs. The hardware description language has been used to describe the design of a microprocessor, the FM9001, and to prove that the design implemented a machine language interpreter. The FM9001 was later fabricated from the design.

3.7.5. Lambda Calculus

The Church-Rosser theorem for lambda calculus says that if a lambda-expression x can be reduced to two expressions y and z then there is a w such that both y and z reduce to w. Briefly, the formalization of the lambda calculus used in [51] proceeds along the following lines. Represent lambda-expressions as list or shell constants, e.g., **`'(LAMBDA (X) (X X))`** is such a constant. Define what is meant by α- and β-*steps* by defining predicates that determine whether one lambda-expression is obtained from another by the appropriate transformation. Define the concept that one lambda-expression *reduces* to another via a given finite sequence of lambda-expressions, each obtained from the previous by an α- or β-step. The theorem is then: if **`X`**, **`Y`**, and **`Z`** are lambda-expressions and **`X`** reduces to **`Y`** via the sequence **`S1`** and **`X`** reduces to **`Z`** via the sequence **`S2`** then there exist a lambda-expression **`W`** and sequences **`S3`** and **`S4`** such that **`Y`** reduces to **`W`** via **`S3`**, and **`Z`** reduces to **`W`** via **`S4`**. The existential quantifiers are then eliminated by defining functions for constructing the alleged **`W`**, **`S3`**, and **`S4`**. For the complete details see example file 54.

Once again we see a trade-off between difficulty of statement and difficulty of proof: key steps in the proof, namely, the construction of **`W`**, **`S3`**, and **`S4`**, are taken during the formalization of the problem.

3.7.6. Gödel's Incompleteness Theorem

To state Gödel's incompleteness theorem it was necessary to formalize within our logic another logic. The object logic was Shoenfield's first order logic with Cohen's axioms for hereditarily finite set theory, Z2. The formalization was done by representing *formulas* in the object logic by list- and shell-constants and then defining within our logic a proof checker for the object logic. That is, the function defined the concept that **`P`** is a *proof* of **`X`**. The theorem was then formalized by saying that there exists a closed formula, **`X`**, of the object logic such that if **`P1`** is a proof of **`X`** or **`P2`** is a proof of its negation, then there exist proofs **`P3`** and **`P4`** of both **`X`** and its negation. The existential quantifiers are again removed by defining the appropriate functions. The details are given in [50] and in example file 55.

4 A Precise Description of the Logic

4.1. 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.

4.2. 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 assume that the reader interested in such asides will have first read the primer, Chapter 2. Familiarity with the primer is completely unnecessary to follow the precise development of the theory.

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.

4.3. 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.

Note. In the first edition of this handbook, we permitted comments in the display of formal terms. In the second edition, we have disallowed comments in formal terms. Comments are permitted in the extended syntax, which is the syntax used throughout the book (once that syntax is defined) and in the implementation. Thus, the only effect of this change is to delay the precise description of the comment convention until we present the extended syntax.

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))**.

4.3.1. 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 book 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 **(fn** $\mathbf{a_1}$ **...** $\mathbf{a_n}$**)**.

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`**}.

4.3.2. 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 59) and the principle of induction (see page 111).

Terminology. There are four kinds of *axiomatic acts*: (a) an application of the shell principle (page 63), (b) an application of the principle of definition (page 112), (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 chapter.

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 60), **`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.

4.4. 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 book is self-contained we have included as the first subsection below, one such formalization, namely that of Shoenfield [52].

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.

4.4.1. 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`.

4.4.2. 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.`

4.4.3. 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**$\neq$**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)) $\neq$ **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)$\neq$**F** $\wedge$ **(Q Y)**$\neq$**F)** $\rightarrow$ **(R X Y)**$\neq$**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 always met in the development of the logic.

4.5. 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.

4.5.1. 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`** $\mathtt{t_1}$ $\mathtt{t_2}$**`)`** where $\mathtt{t_1}$ is the first element of **`s`** and $\mathtt{t_2}$ is the **`fn`** nest around **`b`** for the remaining elements of **`s`**. When we write **`(fn`** $\mathtt{t_1}$ **`...`** $\mathtt{t_n}$**`)`**⊗**`b`** where a term is expected, it is an abbreviation for the **`fn`** nest around **`b`** for $\mathtt{t_1}$, ..., $\mathtt{t_n}$.

Note. In the first edition used ‘‘**`(fn`** $\mathtt{t_1}$ **`...`** $\mathtt{t_n}$**`)@b`**’’ to denote what we now denote with ‘‘**`(fn`** $\mathtt{t_1}$ **`...`** $\mathtt{t_n}$**`)`**⊗**`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 4.2 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 4.2

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** $\mathbf{s_1}$ **...** $\mathbf{s_n}$**)** iff either **flg** is **ONE-OF** and **fn** is among the $\mathbf{s_i}$ or **flg** is **NONE-OF** and **fn** is not among the $\mathbf{s_i}$.

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

Examples. Let $\mathtt{tr}_1$ be **`(ONE-OF LISTP LITATOM)`**. Then $\mathtt{tr}_1$ is a type restriction over the set {**`NUMBERP LISTP LITATOM`**}. The function symbol **`LISTP`** satisfies $\mathtt{tr}_1$ but the function symbol **`NUMBERP`** does not. The type restriction term for $\mathtt{tr}_1$ and **`X1`** is **`(OR (LISTP X1) (OR (LITATOM X1) F))`**. Let $\mathtt{tr}_2$ be **`(NONE-OF NUMBERP)`**. Then $\mathtt{tr}_2$ is a type restriction over the set {**`NUMBERP LISTP LITATOM`**}. The function symbol **`LISTP`** satisfies $\mathtt{tr}_2$ but the function symbol **`NUMBERP`** does not. The type restriction term for $\mathtt{tr}_2$ and **`X2`** is **`(NOT (OR (NUMBERP X2) F))`**.

4.5.2. 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 $\mathtt{ac}_1$**`, ..., `**$\mathtt{ac}_n$,
type restrictions $\mathtt{tr}_1$**`, ..., `**$\mathtt{tr}_n$, *and*
default functions $\mathtt{dv}_1$**`, ..., `**$\mathtt{dv}_n$,

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`**, $\mathtt{ac}_1$, ..., $\mathtt{ac}_n$ are new function symbols of one argument, and all the above function symbols are distinct;

(b) each $\mathtt{tr}_\mathtt{i}$ is a type restriction over the recognizers of h together with the symbol **`r`**;

(c) for each **`i`**, $\mathtt{dv}_\mathtt{i}$ is either **`base`** or one of the base functions of h; and

(d) for each **`i`**, if $\mathtt{dv}_\mathtt{i}$ is **`base`** then **`r`** satisfies $\mathtt{tr}_\mathtt{i}$ and otherwise the type of $\mathtt{dv}_\mathtt{i}$ satisfies $\mathtt{tr}_\mathtt{i}$.

If the $\mathtt{tr}_\mathtt{i}$ 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 (ac1 X) ... (acn X))
                X))
```

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

(6)
```
(IMPLIES trti
         (EQUAL (aci (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 trti)
                      (EQUAL X
                             (const X1 ... Xn)))))
         (EQUAL (aci X) (dvi)))
```

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 (ac1 X))
 ...
 (COUNT (acn 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 106.

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.

4.5.3. 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**.

Note. In Appendix II we explicitly list the axioms added by this invocation of the shell principle. Each axiom has a number of the form **12.***n*, where *n* indicates the corresponding axiom schema of the shell principle.

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)))
```

4.5.4. 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.

In Appendix II we explicitly list the axioms added by this invocation of the shell principle. Each axiom has a number of the form **19.***n*, where *n* indicates the corresponding axiom schema of the shell principle.

4.5.5. 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.

In Appendix II we explicitly list the axioms added by this invocation of the shell principle. Each axiom has a number of the form **20.***n*, where *n* indicates the corresponding axiom schema of the shell principle.

4.5.6. 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`**.

Note. In Appendix II we list explicitly the axioms added by this invocation of the shell principle. Each axiom has a number of the form **21.*n***, where *n* indicates the corresponding axiom schema of the shell principle.

4.6. 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 $1 \le i \le$ **`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`**.

4.7. The Extended Syntax

Note on the Second Edition. The second edition of this handbook extends the syntax of the first edition 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.

In the first edition, Appendix I contained a formalization of the old syntax and abbreviation conventions; however, that formalization was developed independently of the informal description of the syntax (though the two were carefully scrutinized for consistency). In this edition we have undertaken in Appendix I to formalize not just the syntax and abbreviation conventions but this particular informal description of them. Thus, virtually every concept defined informally in this section (and in Section 4.11.2) is defined formally in Appendix I. Readers who are struggling with the problem of formalizing some system—and here we do not mean to limit ourselves just to formal languages—might benefit from a comparative study of this section and Appendix I. Here we describe in the mathematician's precise English a collection of concepts that we there render formally.

Finally, to aid the new user in trying to understand our abbreviation conventions, we have included the machine readable text of Appendix I among our standard example files, as file 11. By noting this library one can execute the formal concepts. Comments in the file give some tips for making experimentation convenient.

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 4.11, page 108) 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 chapter. 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.

In Appendix I we not only formalize the concepts defined below but we provide a recursive descent parser that recognizes when a string of ASCII character codes is the display of a token tree.

4.7.1. 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 $\mathtt{c}_k ... \mathtt{c}_1\mathtt{c}_0$ is the integer $c_k n^k + ... + c_1 n^1 + c_0 n^0$, where c_i is the value of the digit $\mathtt{c}_i$.

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 4.3 shows some numeric sequences and their numeric values written in decimal notation.

Table 4.3

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

- If c_1 is a break character, suitable separation is any comment.
- If c_1 is not a break character but c_2 is a break character, suitable separation is either the empty comment or any separating comment.
- If neither c_1 nor 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 s_1 and 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 in this book 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 book. 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.

In Appendix I we define the inverse of the display process: a parser that determines whether a list of ASCII character codes is a display of a token tree and, if so, what token tree was displayed (up to the integer/numeric sequence ambiguity which is resolved by always adopting the numeric sequence interpretation). The parser is defined in two passes, one that breaks the list of character codes into lexemes and a second which attempts to construct a token tree from the lexemes.

4.7.2. 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, $(x_1\ x_2\ \ldots\ x_k\ .\ (y_1\ \ldots\ y_n))$, by $(x_1\ x_2\ \ldots\ x_k\ y_1\ \ldots\ y_n)$ and every subtree of the form $(x_1\ x_2\ \ldots\ x_k\ .\ \texttt{NIL})$ by $(x_1\ x_2\ \ldots\ x_k)$. 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 [53, 54]. 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.[7]

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).

---

[7] Actually, we do not force our implementation of backquote onto the Nqthm user typing Lisp at the top-level. See **BACKQUOTE-SETTING** in the Reference Guide, page 125.

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 4.4 we show some other examples of the readmacro expansion of backquote trees.

We can explain the last example of Table 4.4. 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

Table 4.4

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))))`

backquote tokens are eliminated.

4.7.3. 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'**.

Note. See the definition of **ASCII-TABLE** in Appendix I for the codes of the printing ASCII characters.

Example. The ASCII codes for the characters **A**, **B**, and **C** are 65, 66, and 67 respectively. Let $\mathbf{t_{65}}$, $\mathbf{t_{66}}$, and $\mathbf{t_{67}}$ denote, respectively, the **NUMBERP**s corresponding to 65, 66, and 67. For example, $\mathbf{t_{65}}$ 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 $\mathbf{t_{65}}$ (CONS $\mathbf{t_{66}}$ (CONS $\mathbf{t_{67}}$ (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 $\mathbf{t_{65}}$ (CONS $\mathbf{t_{66}}$ (CONS $\mathbf{t_{67}}$ (ZERO))))),

where $\mathbf{t_{65}}$, $\mathbf{t_{66}}$, and $\mathbf{t_{67}}$ are as in the last example.

4.7.4. 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 90).
- 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 90).
  - 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 90). If well-formed, the translation of **`x`** is the explicit value term denoted by **`e`** (see page 90).
- 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., $\{\langle var_1, t_1\rangle \ldots \langle var_n, t_n\rangle\}$. 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 $\mathbf{t_1}$ 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** $\mathbf{t_1}$ **...** $\mathbf{t_n}$**)**.
- 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 109). If well-formed, the translation of **x** is the **FOR** expression denoted by **x** (see page 109).
- 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 $\mathbf{t_n}$ for $\mathbf{t_1}$, ..., $\mathbf{t_{n-1}}$.
- Otherwise, **x** is not well-formed.

Examples. Table 4.5 shows well-formed s-expressions on the left and their translations to formal terms on the right.

Table 4.5

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))`

Certain formal terms, such as the translation of **NIL**, are exceedingly painful to write down because they contain deep nests of **ADD1**s. Table 4.6 also contains translations, except this time the right-hand column is, technically, not a formal term but rather another well-formed s-expression with the same translation. In particular, in Table 4.6 we use decimal notation in the right-hand column, but otherwise confine ourselves to formal syntax.

Table 4.6

```
                       s-expression with
s-expression           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)))))))
----------------------------------------------------------------
```

4.7.5. 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** $\mathbf{e_1}$ **...** $\mathbf{e_n}$**)**, where **fn** is a constructor or base function symbol of arity **n** and each $\mathbf{e_i}$ is an explicit value descriptor denoting an explicit value $\mathbf{t_i}$. The explicit value denoted by **e** is then **(fn** $\mathbf{t_1}$ **...** $\mathbf{t_n}$**)**. (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., **(**$\mathbf{e_1}$ **.** $\mathbf{e_2}$**)**, then **e** is

an explicit value descriptor iff each $\mathbf{e_i}$ is an explicit value descriptor. If so, let $\mathbf{t_i}$ be the explicit value denoted by $\mathbf{e_i}$. Then the explicit value denoted by **e** is **(CONS** $\mathbf{t_1}$ $\mathbf{t_2}$**)**.

- If **e** is an s-expression of length 1, i.e., **(**$\mathbf{e_1}$**)**, **e** is an explicit value descriptor iff $\mathbf{e_1}$ is an explicit value descriptor. If so, let $\mathbf{t_1}$ be the explicit value denoted by $\mathbf{e_1}$ and let **nil** be the explicit value denoted by **NIL**. Then the explicit value denoted by **e** is **(CONS** $\mathbf{t_1}$ **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 $\mathbf{e_1}$ be the first element of **e** and let $\mathbf{e_2}$ be the sequence consisting of the remaining elements of **e**. Observe that $\mathbf{e_1}$ and $\mathbf{e_2}$ are both s-expressions. **e** is an explicit value descriptor iff each $\mathbf{e_i}$ is an explicit value descriptor. If so, let $\mathbf{t_i}$ be the explicit value denoted by $\mathbf{e_i}$. Then the explicit value denoted by **e** is **(CONS** $\mathbf{t_1}$ $\mathbf{t_2}$**)**.

Examples. Table 4.7 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.

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

Table 4.7

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))))`

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**.

*4.7.6. *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** $\mathbf{e_1}$ **...** $\mathbf{e_n}$**)** where **fn** is a shell constructor or base function and the $\mathbf{e_i}$ 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** $\mathbf{e_1}$ **...** $\mathbf{e_n}$**)** 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 4.8 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 4.8

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.

4.7.7. *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.

Note. In Appendix I we exhibit a parser for the extended syntax as a function defined within the logic.

4.8. 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))
```

Examples. See the primer for a table of example ordinals.

4.9. Useful Function Definitions

4.9.1. Boolean Equivalence

Defining Axiom 26.

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

4.9.2. 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)))
```

4.9.3. 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)
```

4.10. 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.

4.10.1. 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** $\mathbf{a_1}$ **...** $\mathbf{a_n}$**)** and **e** is **(CONS efn elst)** where **efn** is the **LITATOM** corresponding to **fn** and **elst** is a ‘‘quotation list’’ (see below) of $\mathbf{a_1}$ ... $\mathbf{a_n}$. 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 4.9 we give some terms and examples of their quotations.

Table 4.9

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** $\mathbf{a_1}$ **...** $\mathbf{a_n}$**)**, **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 $\mathbf{a_1}$ ... $\mathbf{a_n}$.

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'**.

4.10.2. V&C$ and EVAL$

Note. The axiomatization of **V&C$** and **EVAL$** are rather subtle. In the primer, we explain many of the constraints and ''design decisions.'' In addition, the interested reader is urged to see [11].

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))))))
```

4.10.3. 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 4.12 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)))).
```

4.11. Quantification

Note. The reader is urged to see [11] for a motivated development of our axiomatization of **FOR**.

4.11.1. 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)))
```

4.11.2. 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 101) of **v**, **t-cond**, **op**, and **t-body** respectively, and **alist** is the standard alist (see page 107) 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*.

4.12. 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** $\mathbf{x_1}$ **...** $\mathbf{x_n}$**)** = **body**. For each of the five function symbols we add the non-**SUBRP** axiom for **fn**, **(**$\mathbf{x_1}$ **...** $\mathbf{x_n}$**)**, 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**.

4.13. Induction and Recursion

4.13.1. 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

```
(IMPLIES qi (ORD-LESSP m/si,j m)).
```

Then **p** is a theorem if

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

is a theorem and

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

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

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 $\mathbf{i}^{\text{th}}$ 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

$\mathbf{i}^{\text{th}}$ 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 $\mathbf{j}^{\text{th}}$ instance for the $\mathbf{i}^{\text{th}}$ 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.

4.13.2. 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. $\mathbf{(f\ x_1\ ...\ x_n)}$ = **body**

is admissible under the history h provided

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

(b) $\mathbf{x_1}$, ..., $\mathbf{x_n}$ are distinct variables;

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

(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 $\mathtt{t_1}$, ..., $\mathtt{t_k}$ governing it, the following formula can be proved directly in h:

```
(IMPLIES (AND t1 ... tk)⊗T
         (ORD-LESSP m/s m)),
```

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

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.`**

4.14. 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.

In the paper [3] (and in its more detailed counterpart [2]) we derive introduction by constraint and functional instantiation and we show many examples of their use, including formal studies of ‘‘mapping functions’’ such as **FOLDR** and **FOLDL**, alternatives to quantifiers, and properties of generic language interpreters. These examples are also included among the example files of Nqthm-1992. See example file 8.

Terminology. A *LAMBDA expression* is an s-expression of the form **(LAMBDA** $(\mathbf{a}_1 \ \ldots \ \mathbf{a}_n)$ **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 $(\mathbf{f}\ \mathbf{t}_1\ \ldots\ \mathbf{t}_n)$, let $\mathbf{t}_i'$ be the functional instantiation of $\mathbf{t}_i$, 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 $(\mathbf{f}'\ \mathbf{t}_1'\ \ldots\ \mathbf{t}_n')$. If a pair <**f**, **(LAMBDA** $(\mathbf{a}_1\ \ldots\ \mathbf{a}_n)$ **term)**> is in fs, the result is **term**/{... <$\mathbf{a}_i$, $\mathbf{t}_i'$> ...}. Otherwise, the result is $(\mathbf{f}\ \mathbf{t}_1'\ \ldots\ \mathbf{t}_n')$.

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 $\mathtt{f_i}$ under fs is called the *witness* for $\mathtt{f_i}$.

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 [3] 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 function 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 **`t`**$_1$ is a *variant* of term **`t`**$_2$ if **`t`**$_1$ is an instance of **`t`**$_2$ and **`t`**$_2$ is an instance of **`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 [3] 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.

12 Reference Guide

We now present an alphabetical listing of the commands to the theorem prover along with the definitions of concepts used in the command descriptions.

12.1. Aborting or Interrupting Commands

Means for aborting a computation or interrupting and resuming a computation are not specified in *Common Lisp The Language*, but there always seems to be an implementation dependent means for doing these things. Of course, even if it is possible to abort or interrupt a process, the exact point in the computation where the effect will occur is bound to be very time dependent, i.e., it's a random thing to do, in the vernacular.

In general, aborting a computation means that control is taken away from the ongoing computation and is given to the top-level read-eval-print loop of Common Lisp. To abort a computation:

- On a Symbolics, simultaneously press the **`CONTROL`**, **`META`**, and **`ABORT`** keys.
- In Lucid on a Sun, simultaneously press the **`CONTROL`** and **`C`** keys and then type **`:A`** to the resulting break prompt.
- In KCL or CMU Common Lisp on a Sun, simultaneously press the **`CONTROL`** and **`C`** keys and then type **`:Q`** to the resulting break prompt.
- In Allegro on a Sun, simultaneously press the **`CONTROL`** and **`C`** keys

and then type **`:RES`** to the resulting break prompt.

It is sometimes necessary to try several times to abort because certain Lisp programs (e.g., the garbage collector) may cause keystrokes to be ignored. In addition, if the theorem prover is printing to the terminal when you try to abort, you may get several pages of output before the requested action seems to occur. This is because terminal output lags behind the actual computation (e.g., because of buffering).

By ‘‘interrupting’’ a computation we mean that control is temporarily taken away from the ongoing computation and given to an inferior read-eval-print loop. From within this loop you can evaluate Lisp expressions, e.g., to inspect the state. See **`BREAK-REWRITE`** for example. When you are ready for the interrupted computation to resume, you communicate this to the inferior read-eval-print loop, which then terminates and returns control to the interrupted computation. To interrupt a computation

- On a Symbolics, simultaneously press the **`CONTROL`**, **`META`**, and **`SUSPEND`** keys. To resume, press the **`RESUME`** key.
- In Lucid on a Sun, simultaneously press the **`CONTROL`** and **`c`** keys. To resume, type **`:C`** to the break prompt.
- In KCL on a Sun, simultaneously press the **`CONTROL`** and **`c`** keys. To resume, type **`:R`** to the break prompt.
- In Allegro on a Sun, simultaneously press the **`CONTROL`** and **`c`** keys. To resume, type **`:CONT`** to the break prompt.
- In CMU Common Lisp on a Sun, simultaneously press the **`CONTROL`** and **`c`** keys. To resume, type **`:GO`** to the break prompt.

Warning: It is technically dangerous to abort any command that changes the data base, including the event commands such as **`ADD-SHELL`**, **`DEFN`**, and **`PROVE-LEMMA`**. Such aborts may leave the data base in an inconsistent state. The only proofs we endorse are those constructed by an uninterrupted sequence of event commands.

Having thus officially disavowed the practice, let us now make a few practical remarks about the effects of aborting event commands. All commands that change the data base adhere to the policy of first storing ‘‘undo’’ information and then making the change. The undo information allows us later to remove an event from the data base. Thus, if you do abort an event command that might have made changes to the data base, use **`UNDO-NAME`** to remove the aborted event.

What do we mean by ‘‘that might have made changes to the data base?’’ How can you tell? The answer is, technically, see if the new name has an

EVENT-FORM in the data base (see **DATA-BASE**). However, it doesn't hurt anything to do an **UNDO-NAME**; the worst that will happen is that an error is reported because the ''event'' to be undone does not exist.

However, we personally behave in a somewhat less rigid way. Whenever we abort a **DEFN** or **ADD-SHELL** we do an **UNDO-NAME**. We don't bother to do an **UNDO-NAME** when we abort a **PROVE-LEMMA** because **PROVE-LEMMA** doesn't change the data base until after the proof has successfully completed. Because the theorem prover's output lags behind its actual computation, it is possible that we will someday abort a proof—thinking that it is doomed to fail—after it has in fact succeeded. But it has not happened yet. If it happens, the inconsistency will show up if the same event name is submitted later. In any case, we never feel like we have completed a proof project until an uninterrupted run of the events is performed by **PROVE-FILE** (see page 180).

12.2. ACCUMULATED-PERSISTENCE

General and Example Form:
(ACCUMULATED-PERSISTENCE)

This routine prints a summary of the persistence totals accumulated against function and rule names for the most recently started proof attempt during which the rewrite path was maintained. See **BREAK-LEMMA** and **MAINTAIN-REWRITE-PATH** for an explanation of related features.

The rewrite path is a stack of ''frames'' maintained by the rewriter and encodes a complete description of the currently active calls to the rewrite routine. Most frames record a call of the rewriter; each such frame is associated with a non-variable term (the term being rewritten) or a rule (the rule being applied). When the rewriter has completed the processing of the term or rule, the frame is popped off the rewrite path. The number of frames built while a given frame is on the path is called the persistence of the frame and is an indication of the amount of work attributable to the term or rule associated with the frame. When we pop a frame, we compute its persistence and add it into an accumulating persistence table under the topmost function symbol of the term or the name of the rule. Thus, if a proof (successful or otherwise) has been run while the rewrite path was being maintained, the accumulated persistence table will indicate the ''expense'' of dealing with all of the function names and rules involved in the proof.

ACCUMULATED-PERSISTENCE prints to ***STANDARD-OUTPUT*** a summary of the accumulated persistence table. In particular, it says how many names have been seen by the rewriter and it lists the persistence totals of the 20 most persistent user-introduced names and the 5 most persistent primitive names.[8] If names that are irrelevant to your proof appear in this listing, you could **DISABLE** them to speed up the proof.

12.3. ADD-AXIOM

General Form:
(ADD-AXIOM name rule-classes term)

Example Form:
(ADD-AXIOM NUMBERP-LOC (REWRITE) (NUMBERP (LOC X M)))

ADD-AXIOM is an event command for adding a new axiom to the logic and storing it and possibly some generated rules in the data base. It does not evaluate its arguments. **name** must be a new name and will be made the name of the event in the data base, **rule-classes** must be a (possibly empty) list of rule classes and **term** must be a well-formed term (see **TRANSLATE**). **ADD-AXIOM** stores **term** in the data base as an axiom; in addition, rules of the classes specified by **rule-classes** are generated from **term** and stored. Each rule has the name **name** and is initially **ENABLED**. An error is caused if **term** is unsuitable for some class in **rule-classes** and no change is made in the data base. If successful, **ADD-AXIOM** prints a time triple for the event and returns the name of the new event, **name**.

When formulating input for **ADD-AXIOM** you must be cognizant of the rule interpretation of **term** in addition to its mathematical content. See Chapter 11 for a detailed explanation of the effect of each type of rule and how rules are generated from formulas and rule classes.

Note that if **rule-classes** is **NIL** then the term is stored as an axiom and will be available for **USE** hints but will generate no rules.

We strongly advise against the use of **ADD-AXIOM**. Moderate use of the theorem prover usually provides convincing evidence for the proposition that users frequently believe formulas to be valid when they are not. Conferring upon an unproved formula the stamp **Axiom** in no way lessens the danger that it is inconsistent with the other axioms. If a concept or relation can be defined within the logic, we urge you to so define it. If a formula is in principle provable, we urge you to prove it. If your intention is to constrain some new function symbols to have certain properties, we urge you to use **CONSTRAIN**.

[8] The numbers 20 and 5 are not fixed. **ACCUMULATED-PERSISTENCE** actually takes two optional arguments, the first of which defaults to 20 and the second of which defaults to 5.

12.4. ADD-SHELL

General Form:
`(ADD-SHELL const base r ac-descriptors)`

Example Form:
```
(ADD-SHELL PUSH EMPTY-STACK STACKP
           ((TOP (ONE-OF NUMBERP LITATOM) ZERO)
            (POP (ONE-OF STACKP) EMPTY-STACK)))
```

`ADD-SHELL` is an event command for adding a new shell to the logic. It does not evaluate its arguments. **`ac-descriptors`** must be a list of **`n`** elements. Each element must be a triple of the form **(**$\mathbf{ac_i}$ $\mathbf{tr_i}$ $\mathbf{dv_i}$**)**. If **`base`** is non-**`NIL`**, the command has the theoretical effect of the axiomatic act

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

If **`base`** is **`NIL`** no base function symbol is supplied. Note that the **`ADD-SHELL`** command causes an error if the corresponding invocation of the shell principle is inadmissible in the current history. If admissible, the command adds a new event in the data base whose name is **`const`**. In addition, the command adds a large number of rules to the data base. The names of the rules are generated from the user-supplied names above. We do not describe the rules generated or their precise names here. The command prints a time triple and returns **`const`**.

It is with **`ADD-SHELL`** that **`BOOT-STRAP`** builds in the rules for the natural numbers, lists, and the other primitive shells. The only ways in which those primitive shells are ‘‘more built-in’’ than user declared shells are (a) because they are satellites of **`GROUND-ZERO`** they cannot be undone individually and we do not keep track of references to them, (b) the **`QUOTE`** notation provides succinct abbreviations for them, and (c) the linear arithmetic procedure is built specifically for the **`NUMBERP`**s.

Note. The time **`ADD-SHELL`** takes to operate depends on the number of accessors, n, and the number of recognizers, k, listed in the **`ONE-OF`** and **`NONE-OF`** clauses of the type restrictions. Roughly speaking, the size of the normal form of one of the rewrite rules added by **`ADD-SHELL`** grows exponentially with n and quadratically with k. The rewrite rule in question, called **`const-EQUAL`**, is essentially a conjunction of n terms, each of which is a disjunction of k terms. Computing the normal form of **const-EQUAL** can take a while, for large n and k. In the pre-1992 releases of Nqthm this problem was exacerbated by the coding of the routine for storing rewrite rules. That routine processed each rule in a way that was quadratic in the size of the (normalized) rule. As a result, multi-accessor shell declarations with non-trivial type restrictions on each component could cause **ADD-SHELL** to take hours of cpu time. Nqthm-1992 contains a short cut, inspired by a suggestion of Matt Kaufmann's, that makes the storage processing linear for **const-EQUAL**. Nevertheless, the exponential growth is still inherent in the shell axioms. If you have an n-ary shell declaration that is taking a very long time, you might consider trivializing all of the type restrictions, by using **(NONE-OF)** for each accessor position. That will introduce a class of n-tuples with no restrictions on the components. You may then define a predicate that recognizes when such an n-tuple has the desired types of components. The **IF**-normal form of that predicate will suggest to you the size of the problem. If you use that predicate in your theorems, it may possibly cause combinatoric explosions. However, you can deal with them through conventional means: decompose your predicate into a nest of nonrecursively defined functions and disable those functions.

12.5. AXIOM

General Forms:
(AXIOM name rule-classes term)
and
(AXIOM name rule-classes term hints)

Example Form:
(AXIOM NUMBERP-LOC (REWRITE) (NUMBERP (LOC X M)))

The form **(AXIOM name rule-classes term)** is just an abbreviation for **(ADD-AXIOM name rule-classes term)**. The **hints** argument, if supplied, is ignored. This odd functionality is provided so that during proof development **AXIOM** commands can be used in place of selected **PROVE-LEMMA** commands— as one might want to do if an event list must be ''replayed'' to reproduce some data base state (e.g., to demonstrate a bug or rewrite-rule loop) but some **PROVE-LEMMA** commands in it take ''too long.'' See also **SKIM-FILE**.

12.6. BACKQUOTE-SETTING

General and Example Forms:
(BACKQUOTE-SETTING 'ORIG)
and
(BACKQUOTE-SETTING 'NQTHM)

If you do not use backquote, or at least, do not use backquote in the formulas you submit to Nqthm, then **BACKQUOTE-SETTING** is unimportant to you. If you do use backquote in your theorems, and especially if you use it both in Nqthm formulas and in special-purpose Lisp utilities you use in connection with Nqthm, then **BACKQUOTE-SETTING** is very important.

The Nqthm user interface is just the read-eval-print loop of the host Common Lisp. Commands are read with the standard Lisp reader. This raises a problem, illustrated below, because the backquote notation has different interpretations in different Common Lisps. Depending on which Common Lisp is used, an utterance involving backquote might read as a theorem, a non-theorem, or an ill-formed formula. To avoid this dependency on the host's interpretation of backquote, Nqthm supplies an interpretation that is consistent with the Common Lisp standard [53, 54]. If you use backquote in Nqthm terms, then you should be sure the terms are read under Nqthm's interpretation of backquote. However, the s-expressions produced by Nqthm's backquote, while consistent with the standard, are generally not as efficient, when treated as Lisp code, as those produced by most Common Lisps. Thus, if you use backquote in Lisp utility functions, you might wish to use the host interpretation when those utilities are loaded.

Common Lisp's ''current readtable'' (***READTABLE***) determines how the Common Lisp reader interprets backquote. The function **BACKQUOTE-SETTING** modifies the readtable so as to provide either of the two interpretations backquote.

(BACKQUOTE-SETTING 'ORIG) modifies the current Common Lisp readtable so that backquote is given the interpretation initially present in the host Common Lisp. That is, after invoking **(BACKQUOTE-SETTING 'ORIG)**, type-in is read according to the host's interpretation of backquote. **(BACKQUOTE-SETTING 'NQTHM)** modifies the current Common Lisp readtable so that backquote is given the interpretation defined in Section 4.7.

Warning. The initial interpretation of backquote in Nqthm is that of the host Common Lisp. If you want our formal interpretation in a session, you must invoke **(BACKQUOTE-SETTING 'NQTHM)**.

We now illustrate the problem caused by backquote. The Common Lisp standard [53, 54] does not specify the s-expression read in response to a backquote expression. Instead, it specifies what the value of that s-expression must be. Different implementations of Common Lisp produce different s-expressions for the same backquote expression. For example, the Nqthm interpretation of **`‘(,X)`** is **`(CONS X ’NIL)`**; the AKCL (Austin-Kyoto Common Lisp) interpretation of the same backquote expression is **`(LIST X)`**; the Lucid interpretation is **`(LUCID-RUNTIME-SUPPORT:BQ-LIST X)`**. Thus, **`(EQUAL (CAR ’‘(,X)) ’CONS)`** reads as a theorem under Nqthm’s interpretation of backquote, reads as a falsehood under AKCL’s, and reads as an ill-formed formula under Lucid’s because it involves the character sequence **`LUCID-RUNTIME-SUPPORT:BQ-LIST`** which is not a legal word (see page 75) because it contains a colon.

Many experienced Nqthm users have special-purpose Lisp programs and macros they use in connection with Nqthm. Such users must be especially careful about which interpretation is given backquote. When their Lisp utilities are read and compiled, the **`ORIG`** backquote setting should be in use, for efficiency. When Nqthm formulas are read, the **`NQTHM`** setting should be in use.

The Nqthm utility **`PROVE-FILE`**, which reads Nqthm events from a file and executes them, always uses the Nqthm interpretation of backquote, regardless of what setting is in force in Common Lisp’s read-eval-print loop. This is part of **`PROVE-FILE`**’s assurance that the file contains nothing but well-formed events. **`PROVE-FILE`** restores the previous interpretation upon termination.

Our decision to make the host’s backquote setting be the default setting stems from the fact that few Nqthm users are tempted to use backquote in Nqthm terms. But this decision admits the following unhappy scenario: The user fires up a Lisp and begins to develop a sequence of events leading to some interesting result, submitting each event to the read-eval-print loop as development proceeds. At some point, backquote is used in an event. The user, however, has forgotten the backquote problem and is accidentally still using the host’s interpretation. It happens that the host in question expands backquoted forms into legal Nqthm terms but different ones than Nqthm’s interpretation would produce. Thus, without knowing it, the user develops a data base of rules around this spurious interpretation of type-in. When the final event list is submitted to **`PROVE-FILE`**, which uses the correct Nqthm interpretation of backquote, it fails to replay. The moral is that if you use backquote in terms, use **`BACKQUOTE-SETTING`** to ensure you get the correct treatment of your type-in. Unfortunately, it is sometimes difficult, in the heat of the chase, to note that you’ve begun to use backquote.

12.7. BOOT-STRAP

General Form:
(BOOT-STRAP flg)

Example Form:
(BOOT-STRAP NQTHM)

BOOT-STRAP is an event command. It erases the current data base and initializes it to the Ground Zero logic. This command creates a very large number of rules corresponding to the primitive shells and the definitions of the functions in the Ground Zero logic. It prints a time triple and returns the event name **GROUND-ZERO**.[9]

The **flg** argument lets the user select which of two sets of Ground Zero axioms should be used. The argument is not evaluated by Lisp. If **flg** is **NQTHM** the theory used is that documented in this handbook. If no **flg** argument is supplied or if **NIL** is supplied or if **flg** is **THM**, the system is booted into a constructive logic very similar to the one we used before adopting the current one. We call the first logic ''**NQTHM** logic'' (for ''New Quantified THM'') and say the theorem prover is running in ''**NQTHM** mode'' when that logic is being used. We call the second logic ''**THM** logic'' and the associated mode ''**THM** mode.''

THM logic differs from **NQTHM** logic in the following respects:

- **THM** does not support the ordinals. **ORD-LESSP** and **ORDINALP** are not defined in **THM**. Two simple lexicographic relations, **LEX2** and **LEX3**, are defined. **(LEX2 (LIST** i_1 j_1**) (LIST** i_2 j_2**))** is **T** precisely if either **(LESSP** i_1 i_2**)** or **(EQUAL** i_1 i_2**)** and **(LESSP** j_1 j_2**)**. **LEX3** is defined similarly. **LESSP**, **LEX2**, and **LEX3** are the only accepted well-founded relations in **THM**.

- **THM** does not support **V&C$** but does provide a more restricted sense of **EVAL$**. In particular, **ASSOC**, **PAIRLIST**, **FIX-COST**, **STRIP-CARS**, **SUM-CDRS**, **SUBRP**, **APPLY-SUBR**, **FORMALS**, **BODY**, **V&C$**, and **V&C-APPLY$** are defined in **NQTHM** but are not defined in **THM**. **EVAL$** and **APPLY$** are axiomatized in **THM** but they are not equivalent to their counterparts in **NQTHM**. **(EVAL$ T X A)** in **THM** is equivalent to **(MEANING X A)**, as it is defined in [7]. **(EVAL$ 'LIST X A)** is equivalent to **(MEANING-LST**

---

[9] A quick and dirty sketch of the theory created by **BOOT-STRAP** is available by printing the value of the Lisp variable **BOOT-STRAP-INSTRS**.

X A) of [7]. **APPLY$** in **THM** is axiomatized as is **APPLY** in [7]. Roughly speaking this means that **EVAL$** ''does not know its own name.'' Additional restrictions are enforced in **THM**: no definition body may use **EVAL$** or **APPLY$**. The consequence of these restrictions is that mapping functions like **FOR** cannot be defined, but **EVAL$** of **'term** is always equal to **term** under the standard alist, provided **term** does not mention **EVAL$** or **APPLY$**. The only intended use for **EVAL$** in **THM** is in the statement of correctness for metafunctions.

- **THM** does not support **FOR**. Hence, the functions included in the **NQTHM** Ground Zero theory exclusively for the definition of **FOR** are not defined in **THM**: **ADD-TO-SET**, **APPEND**, **MAX**, **UNION**, **QUANTIFIER-INITIAL-VALUE**, **QUANTIFIER-OPERATION**, and **FOR**.
- **THM** contains definitions for the functions **LENGTH**, **SUBSETP**, and **LAST**, which are not defined in the Ground Zero logic of **NQTHM**.

THM logic differs somewhat from the logic supported by the previous release of our theorem prover. Let us call the previous version of the logic the ''old'' logic. The differences between **THM** logic and the old logic are as follows:

- **IFF** is included in **THM** but not in the old logic. This permits the implementation, in **THM** mode, of ''propositional replacement'' rules. If you have an old-style event list including a definition of **IFF**, you should delete it (if it is defined as is our **IFF**) or rename it and all its uses otherwise.
- **EVAL$** and **APPLY$** are included in **THM** but not in the old logic. In fact, they ''replace'' the functions **MEANING**, **MEANING-LST**, and **APPLY** of the old logic. The implementation of metalemmas in **THM** and **NQTHM** was simplified by changing the names of **MEANING** and **APPLY** to **EVAL$** and **APPLY$**. See the next note. If you have an old-style event list involving theorems about **MEANING** and **APPLY**, use **EVAL$** and **APPLY$** instead.
- The form of metalemmas in **THM** is as described here, not as described in [7]. The old style of metalemmas required proving that the metafunction returns a ''**FORMP**''. The new style does not. If you have an old-style event list containing a metalemma, reformulate it as described here. The new theorem will be simpler.
- In addition to **MEANING**, **MEANING-LST**, and **APPLY**, the following functions were defined in the old logic but are not defined in **THM**: **FORMP**, **FORM-LSTP**, **ARITY**, **SYMBOLP**, **LEGAL-CHAR-CODE-SEQ**, **ILLEGAL-FIRST-CHAR-CODES**, and **LEGAL-CHAR-CODES**. These functions all participated in the

definition of **FORMP** and hence were necessary for the old-style form of metalemma. Because they are no longer necessary for metalemmas, and because we imagine that no user relied upon them except for metalemmas, we simply omitted putting them into the **THM** logic.

12.8. BREAK-LEMMA

General Form:
(BREAK-LEMMA name)

Example Form:
(BREAK-LEMMA 'REVERSE-REVERSE)

The argument, **name**, is evaluated and should be the name of a replacement rule or linear arithmetic rule. **BREAK-LEMMA** alerts the rewriter to look out for all attempted applications of the named rule. Whenever the rule is tried, an interactive break occurs. To remove **name** from the list of monitored rules, use **UNBREAK-LEMMA**. Note that **BREAK-LEMMA** cannot be used to break on function definitions or on type-prescription, compound recognizer, meta, elimination, or generalization rules. If **name** is not the name of an **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** event with lemma type **REWRITE**, **BREAK-LEMMA** prints a warning message but alerts the rewriter anyway. It is most often the case that this warning message means you misspelled the name of the rule to be monitored, but it is sometimes useful to break on names that are satellites of other events (e.g., a rewrite rule introduced by an **ADD-SHELL**) so the simple test that **name** names an event is insufficient.

Warning. Because it is possible to execute an arbitrary Common Lisp form while in the interactive break under the theorem prover, it is possible to do arbitrary damage. For example, executing **(THROW 'PROVE T)** will immediately terminate the proof attempt successfully! We do not endorse proofs in which interactive breaks occur. We provide **BREAK-LEMMA** and **BREAK-REWRITE** simply as a means to help you diagnose what is going wrong with your rewrite rules.

Recall that to apply a replacement or linear arithmetic rule the first step is to find a substitution under which the left-hand side of the replacement rule or the key term of the linear arithmetic rule matches some target term being simplified. The substitution is computed by a simple pattern matcher that is complete: if a substitution exists to make the rule syntactically identical to the target, we find it (quickly). Subsequent steps in the application of a rule include relieving the hypotheses and rewriting the right-hand side of the conclusion.

When a matched rule's name appears on the list of monitored rules, an interactive break occurs immediately after the pattern matcher has succeeded. When the pattern matcher fails, no substitution exists and no break occurs even though the rule was actually considered briefly. Thus, if a rule is being monitored and no break for it occurs during a proof attempt, then either the rule is disabled or no instance of its left-hand side (or key term, as the case may be) was ever encountered by the rewriter. In this case it is probable that your expected target term got mangled somehow by other rules. We offer no mechanical help for tracking down the rules or identifying the mangled target. Our best advice is to think hard about what rules might have applied to your target. Some users would here use Pc-Nqthm [37] to carry out the rewrites which supposedly produce the target term, to verify that the term is as expected; Pc-Nqthm can also be used to determine all the rules applicable to a given term, which might suggest how the intended term got mangled.

If a substitution is found, then an interactive break occurs. By typing various commands you can inspect the state of the rewriter and step through the process of attempting to apply the rule. For replacement rules the steps are first relieve the hypotheses and then rewrite the right-hand side of the conclusion. For linear arithmetic rules there are more steps: relieve the hypotheses, rewrite the conclusion, convert the rewritten conclusion into polynomial form, and make heuristic checks on the resulting polynomials. Keep in mind that additional breaks may occur during the recursive processing done to relieve hypotheses and rewrite conclusions.

If at any step it is determined that the rule cannot be applied, the interactive break provides commands for displaying an explanation. However, the interactive break is not intended as a proof-checker: no facilities are provided for directing the theorem prover's strategy or affecting in any way the outcome of the attempt to apply the rule.

The interactive break is a Lisp read-eval-print loop where certain atoms are treated specially. The name of the Lisp function that implements the command interpreter is **BREAK-REWRITE** which, under certain conditions, can be invoked directly by the user during Lisp breaks to inspect the state of the rewriter (see page 136). The prompt character for the break is a colon. Your commands are read by the Lisp reader. The command **?** (i.e., the Lisp symbol obtained by reading a question mark followed by a carriage return) will cause a brief summary of the commands to be printed.

The available commands are divided into two categories: those that are specific to this particular step in the attempt to apply the rule and general commands for inspecting the state of the rewriter. The general commands are available all the time. The context-specific or ‘‘special’’ commands change with each step. For example, on entry to the break, the command **OK** means ‘‘try to relieve the hypotheses and break again when the attempt has finished.’’ But immediately after the hypotheses have been relieved, **OK** means ‘‘proceed to rewrite the conclusion and break again when that has finished.’’ In addition, the special commands at any particular time may include some that make sense only at that time. For example, one can ask to the see the **TARGET** term anytime, but one can ask for the **FAILED-HYP** only after a failed attempt to relieve the hypotheses.

Roughly speaking, the special command **OK** always means ‘‘go ahead with the next step and break when it is done’’ while the special command **GO** always means ‘‘go ahead with the next step and all subsequent steps of this application.’’ **GO** is useful when you wish to shortcut the interactive processing of a rule because you have determined (say from the **TARGET** term) that the application at hand is not interesting.

Until you are familiar with the special commands you should type ‘‘**?**’’ often to see what your options are.

The general commands for inspecting the state of the rewriter are not summarized by the ‘‘**?**’’ command, since ‘‘**?**’’ is typed fairly often. A summary of the general commands may be obtained by typing ‘‘**??**’’. Using the general commands you can inspect the ‘‘rewrite path’’ from the current target term up to the top-level goal of the simplifier. The rewrite path is a stack of frames. Roughly speaking, each frame (except the first) represents a call of the theorem prover’s rewrite routine. Commands permit you to see the entire list of frames sketchily or to focus attention on a particular frame and get more information. The frame on which attention is focused is called the ‘‘current frame’’ and is initially the frame in which the target term is being rewritten. You may use general purpose commands to move the current frame up or down the stack. Other commands display detailed information about the current frame. However, the position of the current frame in no way affects the simplifier or rewriter! That is, if you move the current frame up to a point where some other target is being rewritten, the theorem prover’s attention does not shift to that target! The notion of the ‘‘current frame’’ is merely a convenient way to let you inspect the stack.

We now present an example of the use of general commands. Suppose we have the rules generated from the following two events:

```
(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
  (IMPLIES (PROPERP X)
           (EQUAL (REVERSE (REVERSE X)) X)))


(PROVE-LEMMA PROPERP-APPEND (REWRITE)
  (IMPLIES (PROPERP B) (PROPERP (APPEND A B))))
```

Suppose also we have installed a break on **PROPERP-APPEND**:

```
(BREAK-LEMMA 'PROPERP-APPEND)
```

Finally, suppose we then execute the event

```
(PROVE-LEMMA LEMMA NIL
  (IMPLIES (AND (PROPERP B)
                (PROPERP A))
           (EQUAL
             (REVERSE (REVERSE (APPEND A B)))
             (APPEND A B))))
```

We show the resulting output below in **typewriter** font.

```
(Entering break on replacement rule PROPERP-APPEND

: ?
You are in the BREAK-REWRITE command interpreter.
The commands specific to this break are:

   cmd       effect
  OK         Attempt to relieve the hypotheses and
                then break.
  GO         Proceed without further interaction.
  NAME       Print name of the broken rule.
  HYPS       List the hypotheses of the broken rule.
  CONCL      Print the conclusion of the broken rule.
  SUBST      Print the substitution being applied to
                the broken rule.
  TARGET     Print the term to which the rule is being
                applied.
  ??         General purpose commands.


: TARGET
(PROPERP (APPEND A B))
```

Note that we are trying to use **PROPERP-APPEND** to prove that **(APPEND A B)** is **PROPERP**. We might wish to know how this subgoal has arisen. To find out, we use the general purpose commands. First we use **??** to get a summary of them.

```
: ??
```

```
You are in the BREAK-REWRITE command interpreter.
The general purpose commands are:

   cmd              effect
  PATH        Highlight the REWRITE path.
  FRAME       Print the current frame, pruning
                deep terms.
  PATH!       Print every frame in the path, pruning
                deep terms.
  FRAME!      Print the current frame, with no pruning.
  PATH!!      Print the path, with no pruning.
  ASSUMPTIONS  Print the governing assumptions.
  ANCESTORS   Print the negations of backchaining
                 hypotheses.
  ACCUMULATED-PERSISTENCE
              Print the accumulated persistence totals.
  BK          Move one frame towards the top-level
                 SIMPLIFY.
  NX          Move one frame away from the top-level
                 SIMPLIFY.
  TOP         Go to the top-level SIMPLIFY frame.
  BTM         Go to the frame farthest from the
                 top-level SIMPLIFY.
  n (a natural number)
              Go to frame number n.
  s-expr (a list expression)
              Evaluate Common Lisp s-expr
  ?           Special purpose commands.
```

The rewrite path will tell us how we got to the current target from the top-level goal of the simplifier. The sketchiest display of the rewrite path is given by the **PATH** command:

```
: PATH
 ( 90)   0. (top)
 ( 62)   1. Rewriting (EQUAL ...)
*( 61)   2. Rewriting (REVERSE ...)
 (  3)   3. Applying  REVERSE-REVERSE
*(  2)   4. Rewriting (PROPERP ...)
 (  0)   5. Applying  PROPERP-APPEND
```

There are six frames on the stack, numbered **0** through **5**. The topmost one, **0**, is the initial call of the simplifier. The bottommost one, **5**, is the frame in which we are working on the target term. From the above display we can see that we got to the target from the top by (a) rewriting an **EQUAL** expression in the topmost goal, (b) rewriting a **REVERSE** expression below that, (c) attempting to apply the rule **REVERSE-REVERSE** to that, (d) rewriting a **PROPERP** expression as part of the attempt, and (e) attempting to apply **PROPERP-APPEND** to that. However, in this sketch of the path we are not told the relation of one frame to the next.

Note also the parenthesized numbers on the left-hand edge of the display. Each number is the ‘‘persistence’’ of the corresponding frame. The persistence of a frame is the number of frames built since the given frame was built. The persistence of a frame is an indication of the amount of work expended so far on behalf of that frame. A frame is marked with a ‘‘*****’’ if the persistence of the frame is at least twice the persistence of the frame immediately below it. In some sense, the frames marked with *****’s contain terms or rewrite rules that have caused the rewriter a lot of work relative to the total amount of work so far.

From the fact that the persistence of frame **1** is **62** we know that 62 frames have been built since we began rewriting the **EQUAL** expression in frame **1**. Since only four of them are now active (frames **2-5**), we know the others have since been discarded. At first sight then, the **EQUAL** expression in frame **1** is relatively expensive to rewrite. But the persistence of frame **2** is **61**. Thus, except for the construction of frame **2** itself, all the work involved in frame **1** has been done under frame **2**. Now note the persistence of frame **3**. It is only **3**. Thus, 58 frames (61 minus 3) were built under frame **2** before we got around to trying **REVERSE-REVERSE** in frame **3**. We conclude that the **REVERSE** expression in frame **2** was relatively expensive to rewrite and frame **2** is marked with a *****. What work was done? After the frame was built we rewrote the arguments to the **REVERSE** expression before looking for rules that match the target. Thus, the 58 frames in question were used for the arguments and for any **REVERSE** rules tried before **REVERSE-REVERSE**. See also **ACCUMULATED-PERSISTENCE**.

The entire rewrite path is displayed in more detail by the following command:

```
: PATH!
---- Frame 0 -----           (persistence 90)
Goal:
(IMPLIES (AND (PROPERP B) (PROPERP A))
         (EQUAL (REVERSE #) (APPEND A B)))


---- Frame 1 -----           (persistence 62)
Rewriting the conclusion of the top-level goal:
(EQUAL (REVERSE (REVERSE #))
       (APPEND A B))
under the substitution:
NIL
```

```
---- Frame 2 -----           (persistence 61)  <-***
Rewriting the first argument of the term in frame 1:
(REVERSE (REVERSE (APPEND A B)))
under the substitution:
NIL


---- Frame 3 -----           (persistence 3)
Attempting to apply the replacement rule
REVERSE-REVERSE using the substitution:
X <- (APPEND A B)


---- Frame 4 -----           (persistence 2)  <-***
Rewriting the first hypothesis of REVERSE-REVERSE:
(PROPERP X)
under the substitution:
X <- (APPEND A B)


---- Frame 5 -----           (persistence 0)
Attempting to apply the replacement rule
PROPERP-APPEND using the substitution:
B <- B
A <- A
```

Observe that now we see more of each term being rewritten. However, we do not see each term in its entirety—note frame **1**. The symbol **#** is printed in place of deep subterms. This is what is meant by ''pruning'' in the **??** command summary. **PATH!** also gives an explanation relating the activity in each frame to the activity in the frame above. In particular, we see that the **REVERSE** expression being rewritten in frame **2** is in fact **(REVERSE (REVERSE (APPEND A B)))** and is the first argument of the **EQUAL** expression being worked on in frame **1**.

When the stack contains hundreds of frames, **PATH!** often provides too much information. That is why the notion of current frame exists. The commands **FRAME** and **FRAME!** print out the current frame. The command **n**, where **n** is a natural number, means ''select frame **n** as the current frame.'' The commands **BK** (for ''back'') and **NX** (for ''next'') let you move the current frame back and forth and are useful for exploring an area of the path.

12.9. BREAK-REWRITE

General Form:
(BREAK-REWRITE)

This Lisp routine permits the user to inspect the rewrite path (see page 131) as though a **BREAK-LEMMA** rule monitor had caused an interactive break. **BREAK-REWRITE** is a user-level entry into the command interpreter and can be invoked directly by the user after forcing a Lisp break while the rewriter is operating.

The method used to force a Lisp break while a computation is in progress is implementation dependent. For example, in KCL the ''control-C'' key is pressed while on the Symbolics Lisp Machine the ''Suspend'' key is pressed. See page 120. Once a break has been caused, executing **(BREAK-REWRITE)** will enable you to inspect the rewrite path with the general purpose commands described in **BREAK-LEMMA** above. (The special purpose commands (e.g., **TARGET** or **CONCL**) do not make sense since there is in fact no rule associated with the break.) The command **OK** exits **BREAK-REWRITE** and returns to the Lisp break from which it was invoked.

For efficiency, the theorem prover does not maintain the rewrite path all the time. If **BREAK-REWRITE** is called when the path is not being maintained, or when the rewrite path is empty (so that there is no topmost frame), it prints a message and returns immediately. See **MAINTAIN-REWRITE-PATH**.

BREAK-REWRITE is useful when you are trying to find out why the theorem prover is spending an unusually long time silently simplifying a particular formula. (Of course, what is unusual depends on what is usual for you. There are successful proofs in which the system is silent for many minutes.) If you have previously enabled the maintenance of the rewrite path you might force a Lisp break. If you have not previously enabled the maintenance of the rewrite path you should abort the now silent proof attempt, use **(MAINTAIN-REWRITE-PATH T)** to turn on maintenance, restart the proof attempt and force a break as before when the silent simplification is in progress. Once you have forced a break while the simplifier is running, call **BREAK-REWRITE** and use the path and frame commands to see what is going on. Most likely you will be surprised at the depth of the path and perhaps at the names of some of the rules or functions on it. Typically in these situations the simplifier is backchaining through rules you have long since forgotten and has set itself subgoals that have little to do with the problem at hand; another common problem is that nonrecursive functions have been expanded and have introduced many cases or concepts needlessly. In any case, it is frequently the case that by inspecting the path you will find the names of rules and functions that can be **DISABLE**d to make the proof go faster.

In addition, **BREAK-REWRITE** is often useful in tracking down circularities in your rewrite rules. Circularities in the rewrite rules lead to stack overflows in the rewriter. Often when stack overflow occurs, there is no way to investigate it, because the Lisp stack is already as big as permitted. However, the rewrite path maintained by the rewriter is not part of the stack but is a separate data structure that persists even after an aborted computation. If the rewrite path was being maintained during a proof attempt that caused a stack overflow, you may abort out of the stack overflow error—thereby gaining all that stack space. Then, from the top-level Lisp read-eval-print loop invoke **(BREAK-REWRITE)**. If it reports that the theorem prover is not in the simplifier, it means ‘‘the theorem prover was not in the simplifier when the stack overflow occurred.’’[10] Otherwise, you will be able to use the **BREAK-REWRITE** commands to investigate the rewrite path as it stood at the time of the stack overflow. This will often make the loop manifest. ▮

```
This conjecture can be simplified, using the
abbreviations ADDRP and IMPLIES, to:

Error: Bind stack overflow.
Error signalled by PROVE-LEMMA-FN.

Broken at PROVE-LEMMA-FN.  Type :H for Help.
>>:Q

Top level.

>(BREAK-REWRITE)
: PATH

 (3404)   0. (top)
 (3394)   1. Rewriting (GET ...)
 (3392)   2. Applying  GET-NORMALIZER
 (3382)   3. Rewriting (GET ...)
 (3359)   4. Applying  GET-NORMALIZER
  ...        ...
 (  92) 202. Applying  GET-NORMALIZER
 (  82) 203. Rewriting (GET ...)
 (  59) 204. Applying  GET-NORMALIZER
```

[10]Some replacement rules are considered so simple that they are used as ‘‘abbreviations’’ and applied exhaustively *before* we begin normal simplification. These rules may contain loops, and at present we offer no aids in tracking down such loops.

```
 (  49) 205. Rewriting (GET ...)
 (  26) 206. Applying  GET-NORMALIZER
 (  16) 207. Rewriting (GET ...)
 (  15) 208. Rewriting (ADD-ADDR ...)
 (  13) 209. Rewriting (CONS ...)
*(   8) 210. Rewriting (PLUS ...)
*(   2) 211. Rewriting (IF ...)
 (   0) 212. Rewriting (IF ...)
```

In using **BREAK-REWRITE** to look for circularities you must keep in mind that the path does not show everything that *has been used* to derive the current term, but what is being used right now. It might be that, in some sense, the ‘‘cause’’ of the loop is a rule not shown on the current path but which fires and completes, introducing a term which will permit it to fire again later during the rewriting.

12.10. CH

General Forms:
(CH)
and
(CH n)
and
(CH m n)

Example Forms:
(CH 10)
(CH ’PROPERP-REVERSE)
(CH 12 20)

CH prints out the names of the events that have been created so far. **CH** takes zero, one or two arguments which are used to specify an interval in the chronological sequence of events. The arguments are both evaluated and should be either event names or event numbers, where 0 is the most recent event, 1 the next most recent, etc. Events are printed in reverse chronological order, i.e., the youngest (most recently created) event is displayed first. If no arguments are supplied, the entire chronology is printed. If only one argument is supplied, it is taken as the rightmost end point, and 0 (the most recently created event) is used as the leftmost.

Here is an example. Suppose that we have just completed the proof described in Chapter 8. In particular, starting from **BOOT-STRAP** we defined **REVERSE** and **PROPERP** and then worked our way through the proofs of several lemmas about **PROPERP**, **APPEND**, and **REVERSE**, concluding with **REVERSE-REVERSE**. Then the following use of **CH** prints out a sketch of the entire session in reverse chronological order:

```
(CH 'GROUND-ZERO)

0   (PROVE-LEMMA REVERSE-REVERSE ...)
1   (PROVE-LEMMA REVERSE-APPEND ...)
2   (PROVE-LEMMA APPEND-IS-ASSOCIATIVE ...)
3   (PROVE-LEMMA PROPERP-REVERSE ...)
4   (PROVE-LEMMA PROPERP-APPEND ...)
5   (PROVE-LEMMA APPEND-RIGHT-ID ...)
6   (DEFN PROPERP ...)
7   (DEFN REVERSE ...)
8   (BOOT-STRAP NQTHM ...)
```

Equivalent **CH** commands are **(CH 8)**, **(CH 0 8)**, and **(CH 'REVERSE-REVERSE 'GROUND-ZERO)**.

CH prints a **D** after the number if the event is currently disabled. Thus, suppose we execute **(DISABLE PROPERP-REVERSE)** and **(DISABLE PROPERP-APPEND)**. If we print the events between **REVERSE-REVERSE** and **APPEND-RIGHT-ID** we get

```
2   (PROVE-LEMMA REVERSE-REVERSE ...)
3   (PROVE-LEMMA REVERSE-APPEND ...)
4   (PROVE-LEMMA APPEND-IS-ASSOCIATIVE ...)
5 D (PROVE-LEMMA PROPERP-REVERSE ...)
6 D (PROVE-LEMMA PROPERP-APPEND ...)
7   (PROVE-LEMMA APPEND-RIGHT-ID ...)
```

Observe in the example above that the event numbers of the events shown have been increased by 2 because the two **DISABLE**s (actually **TOGGLE** events) have become events 0 and 1. In general, event numbers are so volatile that they are useful only in successive calls of **CH** as one scans through the chronology looking for a given interval. The only other command in our system that uses event numbers is **UBT**.

When the theorem prover is being used with a fast terminal and text editor it is often practical to print the entire chronology, as with **(CH)** and then scan it at leisure with the editor. But with slow terminals or systems that discard text that has been scrolled off the top, the primitive interval handling of **CH** is useful.

12.11. CHRONOLOGY

General Form:
CHRONOLOGY

The Lisp variable **CHRONOLOGY** is set to the list of all event names in reverse chronological order. Thus, the first element of **CHRONOLOGY** is the name of the most recently processed event. The last element is always **GROUND-ZERO**, the name of the oldest event. Users typically print the value of **CHRONOLOGY** to refresh their memory of ‘‘where they are’’ or what names they have used. Printing **CHRONOLOGY** is a good way to see what is in a data base that has just been loaded. However, the value of **CHRONOLOGY** is often very long and the command **CH** is often more convenient.

Users should not bind or set **CHRONOLOGY**.

12.12. COMMENT

General Form:
(COMMENT x_1 ... x_k)

Example Form:
(COMMENT THE NEXT EVENT IS THE CRUCIAL LEMMA)

COMMENT is defined as a Lisp macro that takes an arbitrary number of arguments and returns **T**. **COMMENT** ignores its arguments.

COMMENT is provided for historical reasons: an early version of Nqthm was written in Interlisp [55] and the **COMMENT** macro was the Interlisp primitive supporting the insertion of comments into code. The arguments to **COMMENT** were generally symbols or strings expressing some comment in \"natural language\" (e.g., English, German, etc). Because of this Interlisp ancestory, users grew accustomed to inserting such comments in event lists, where the **COMMENT** form appeared as though it were an event command. **COMMENT** is not an event; it does not change **CHRONOLOGY** or add any names or rules. But we permit **COMMENT** forms in places where events forms are expected, e.g., **PROVE-FILE**.

Because Nqthm is now written in Common Lisp, we generally use the Common Lisp semicolon and number sign comments. Such comments have the advantage that they can be written virtually anywhere a \"white space\" can be written, e.g., before or after an event, or \"inside\" an event or formula, and they do not have to follow any syntactic rules except those regarding their delimiters.

12.13. COMPILE-UNCOMPILED-DEFNS

General Form:
(COMPILE-UNCOMPILED-DEFNS filename)

Example Form:
(COMPILE-UNCOMPILED-DEFNS "/temp/foo")

The argument, **filename**, is evaluated and should be a file name. This routine creates a file of the specified name, containing the Common Lisp definitions of the currently uncompiled executable counterparts (see page 161) of all functions in the current data base; it then compiles that file, and loads it. Thus, after calling this routine all executable counterparts are defined with compiled code. This will significantly increase the speed with which **R-LOOP** computes the values of submitted expressions and may increase the speed of the theorem prover if your proofs involve computation with explicit values.

The name of the file created has as its extension the value of the variable **FILE-EXTENSION-LISP**. The directory on which the file resides is affected by the value of the variable ***DEFAULT-NQTHM-PATH***. The extension of the file loaded after the compilation is specified by the variable **FILE-EXTENSION-BIN** (see page 163).

We do not always automatically compile each executable counterpart as it is created because in some circumstances (e.g., when using Kyoto Common Lisp) invoking the function **COMPILE** involves significant overhead and the creation of weirdly named temporary files. In addition, because files are created there is the possibilities of protection violations and resource allocation errors, problems that the theorem prover is not able to gracefully report much less recover from. However, if you wish to have each executable counterpart **COMPILE**d as it is defined, you may set the flag ***COMPILE-FUNCTIONS-FLG*** to **T**. We advise against doing this except on Lisp machines.

12.14. CONSTRAIN

General Forms:
(CONSTRAIN name rule-classes term fs)
and
(CONSTRAIN name rule-classes term fs hints)

Example Form:
```
(CONSTRAIN H-COMMUTATIVITY2 (REWRITE)
           (EQUAL (H X (H Y Z))
```

```
        (H Y (H X Z)))
((H PLUS))
;Hints
((USE (PLUS-COMMUTATIVITY2))))
```

CONSTRAIN is an event command that introduces new function symbols that are constrained to satisfy a given axiom. It does not evaluate its arguments. **name** must be a new name and will be made the name of the event in the data base and **rule-casses** must be a (possibly empty) list of rule classes. **fs** must be a functional substitution; the domain of **fs**, {$\mathbf{f_1}$ ... $\mathbf{f_n}$}, must consist entirely of new names and may not include **name**, and no **LAMBDA** expression in the range of **fs** may contain free variables. The s-expression **term** must be a term, *provided* that the symbols in the domain of **fs** are considered function symbols with the arities of their images under **fs**. Finally, **hints**, if provided, must be as required by the **PROVE-LEMMA** event (see page 188). The **CONSTRAIN** command has the same theoretical effect as the axiomatic act

Constraint.
Constrain $\mathbf{f_1}$, ..., and $\mathbf{f_n}$ to have property **term**,

with the additional restrictions that **fs** is a functional substitution satisfying the requirements of the axiomatic act and that **term****fs** is not just provable in the logic but provable by the implemented theorem prover.

Operationally, **CONSTRAIN** invokes the theorem prover on the functional instantiation of **term** under **fs**, i.e., on **term****fs**, supplying the prover with any **hints** provided. Note that **term****fs** is a term and if it is a theorem then there exist definitions of the functions in the domain of **fs** such that **term** is a theorem. Thus, if the proof succeeds, **CONSTRAIN** declares each of the symbols in the domain of **fs** to be a function symbol of the appropriate arity, adds **term** as an axiom, and in addition, generates from **term** the rules specified by **rule-classes** and stores each of them in the data base. The name of each rule is **name**. An error is caused and there is no change in the data base if **term****fs** cannot be proved or if **term** is of a form inappropriate for some class in **rule-classes**.

Because the only axiom added by **CONSTRAIN** is **term**, the values of the functions **BODY**, **FORMALS**, **SUBRP**, and **APPLY-SUBR** on the new function symbols are undefined.

When formulating input for **CONSTRAIN** you must be cognizant of the rule interpretation of **term** in addition to its mathematical content. See Chapter 11 for a detailed explanation of the effect of each type of rule and how rules are generated from formulas and rule classes.

Note that if **rule-classes** is **NIL** then the term is stored as a theorem and will be available for **USE** hints (see **PROVE-LEMMA**) but will generate no rules.

12.15. DATA-BASE

General Form:
(DATA-BASE query name)

Example Forms:
(DATA-BASE 'IMMEDIATE-DEPENDENTS 'REVERSE)
and
(DATA-BASE 'IMMEDIATE-SUPPORTERS 'REVERSE)
and
(DATA-BASE 'EVENT-FORM 'REVERSE)

This is a low-level function for getting information about the data base. Before discussing the function we describe the data base in detail.

The data base is an acyclic directed graph. At each node is the following information:

- the name of the node,
- the event command which created the node,
- a number indicating the time at which the node was created (used to order event lists chronologically), and
- a list of satellites—the names of auxiliary axioms, functions, and executable counterparts introduced by the event command that created the node (see the discussion below).

A node has satellites if the creating event introduces a function symbol or introduces more than one axiom. For example, the **GROUND-ZERO** node has as its satellites the list of every primitive function symbol, every executable counterpart, and every rule added by **BOOT-STRAP**. **DCL** nodes have a single satellite that is the executable counterpart of the symbol introduced. **ADD-SHELL** nodes have many satellites: the base object, the recognizer, the accessors, the executable counterparts of all the new symbols, and all the rules generated by the shell axioms. No name is a satellite of more than one node.

We say a Lisp symbol is a *citizen* of the data base if it is either the name of a node or is a satellite of a node. The name of each axiom, theorem, function, and executable counterpart is a citizen. The only other citizens are the names of events such as **TOGGLE**s. Henceforth, whenever we refer to a citizen **x** as a node we mean either the node with name **x**, if such a node exists, or the node of which **x** is a satellite. Every citizen has a *status*, **ENABLED** or **DISABLED**, that is used as the status of the associated rule(s).

The *primary* of a citizen, **fn**, is defined as follows. If **fn** is a satellite of **p**, then **p** is the primary of **fn**. Otherwise, **fn** is a node and is its own primary.

The arcs in the data base graph represent ''immediate dependency.'' The relation is defined below. It is *not* the obvious ''logical dependency'' relation.

We define *immediate dependency* as follows. Every node in the data base is immediately dependent on **GROUND-ZERO**. An **ADD-SHELL** event is immediately dependent upon the primary of every other shell function involved in its definition, i.e., the older recognizers in the type restrictions and the older base objects in the default values. Every node whose event contains a term (including the terms used in hints) is immediately dependent upon the primary of every function symbol in every such term. Every node whose admissibility requires theorem proving (e.g., **DEFN** and **PROVE-LEMMA** nodes) is immediately dependent upon the primary of every citizen reported used in the proofs. Furthermore, every such node is immediately dependent upon the primary of every prior event that generated a type prescription rule about any function used in the proofs.[11] **TOGGLE** events (including **DISABLE**s and **ENABLE**s) are dependent upon the primary of the event enabled or disabled. **TOGGLE-DEFINED-FUNCTIONS** events are dependent only upon **GROUND-ZERO**.

The above definition establishes the meaning of the ''immediate dependents'' of every node. Note that the immediate dependents of a node include the reported uses of all of its satellites. We make the convention that the ''immediate dependents'' of a satellite are just those of its primary. For example, suppose **PUSH** is declared as a constructor with accessors **TOP** and **POP**. Then **PUSH** is the name of a node and **TOP** and **POP** are among its satellites. If **TOP** is used in some subsequent event, the event is recorded among the dependents of **TOP**'s primary, namely **PUSH**. Uses of **POP** and the other functions introduced in the shell declaration are similarly recorded against **PUSH**. We freely mix such dependencies because we do not provide a way to remove one accessor, say, from the data base without removing the entire shell declaration.

Warning: This notion of ''immediate dependency'' is flawed. Not every logical dependent is among the ''immediate dependents.'' The problem is that not all facts *used* by the theorem prover are *reported* used. In particular, type prescription lemmas are not tracked and the executable counterparts for the

---

[11]The rewriter does not keep track of the use of type prescription rules because the type set mechanism is so heavily used. This clause in the definition of dependency is an attempt to compensate for this inadequacy by sweeping into the dependency relation some of the events that *might* have been used. Even this does not guarantee to get all of the rules used. See the warning below.

metafunctions **SUBRP**, **APPLY-SUBRP**, **FORMALS**, **BODY**, **V&C$**, **V&C-APPLY$**, **EVAL$**, **APPLY$**, and **FOR** do not leave tracks when they operate on quoted constants, even though the values of those metafunctions depend not just on the constants but how certain functions are defined.

For example, suppose **FN** is a user-defined function. Then **(SUBRP 'FN)** reduces to **F**. Furthermore, no record is left that the computation depends on **FN**; indeed, in a way, it doesn't depend on the function **FN**, it depends on the atom **(PACK '(70 78 . 0))**. Thus, by using **UNDO-NAME** to remove an event and its dependents it is possible to get the data base into an inconsistent state: define **FN**, prove **'FN** is not a **SUBRP**, undo **FN**, add a shell with **FN** as the constructor, prove **'FN** is a **SUBRP**. Because we do not track the dependency of **(SUBRP 'FN)** on **FN**, undoing **FN** fails to undo the lemma that **'FN** is a non-**SUBRP**.

Moral: Do not rely upon the validity of any formula proved in a session in which **UNDO-NAME** was used. Think of **UNDO-NAME** as a quick and dirty way to approximate the logical theory obtained by removing the given event. When an interactive session has successfully led to the proof of the main theorem, replay the entire sequence of events from **GROUND-ZERO** or some trusted library without any undoing. See also the discussion of **UNDO-BACK-THROUGH**, a trusted way to remove events from the data base.

We now discuss the **DATA-BASE** procedure.

General Form:
(DATA-BASE query name)

The first argument is a Lisp symbol corresponding to one of the queries listed below. If the second argument, **name**, is not a citizen, the result is **NIL**. Otherwise, the result depends upon the **query** as described below.

query	**result**
PRIMARY	If **name** is the name of a node, **DATA-BASE** returns **name**. Otherwise, **name** is the name of a satellite and **DATA-BASE** returns the name of the node to which **name** is attached. Since **NIL** is not an event name, **(DATA-BASE 'PRIMARY x)** is non-**NIL** iff **x** is a citizen. If the result is different than **x**, **x** is a satellite of the result.
EVENT-FORM	Returns an event command equivalent to the one the user submitted to create **name**. **name** may be a satellite of the event returned. The command returned will be a list whose first element is the name of an event command, e.g., **PROVE-LEMMA**, **ADD-SHELL**, **TOGGLE**, etc. All list structures used as terms within the command are the result of translating (with **TRANSLATE**) the user type-in. The

EVENT-FORM returned may differ in other respects from the one typed by the user. For example, the user may type a **DISABLE** command and the **EVENT-FORM** may be an equivalent **TOGGLE**, or the user may have omitted the **hints** argument to a **PROVE-LEMMA** command and the **EVENT-FORM** may supply the **hints** **NIL**.

STATUS Returns the Lisp symbol **DISABLED** if **name** is currently disabled; otherwise returns the symbol **ENABLED**. Recall that if the name of a rule is **DISABLED** the rule is never applied.

SATELLITES Returns a list of all the satellites of the primary of **name**.

FORMULA Roughly speaking, **DATA-BASE** returns the formula named by **name**. However, this is not a well-defined concept since some names (e.g., the name of a **TOGGLE** event) do not have any associated formula, and other names (e.g., the name of a shell constructor) have many associated formulas. The precise specification is as follows: If **name** is the name of an **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** event (including those generated by **AXIOM** and **LEMMA** commands), the translation of the third argument of that event command is returned (i.e., the axiom added or theorem proved). If **name** is the name of a defined function—even a satellite—the defining equation is returned as a formula of the form **(EQUAL (name** x_1 **...** x_n**) body)**. If **name** is anything else, **NIL** is returned.

IMMEDIATE-DEPENDENTS
Returns a list containing the names of the immediate dependents of **name**, ordered chronologically.

ALL-DEPENDENTS
Returns the transitive closure of the immediate dependents relation starting at **name**, ordered chronologically. This is the list of all events that somehow depend upon **name**—in so far as we are able to determine given the incomplete usage reports. It is this list of events, plus **name** itself, that will be removed from the data base if **name** is removed.

IMMEDIATE-SUPPORTERS
Returns the list of all events claiming **name** among their immediate dependents, ordered chronologically.

ALL-SUPPORTERS
Returns the transitive closure of the immediate supporters relation starting at **name**, ordered chronologically. This is

the list of all events upon which **name** somehow depends—in so far as we are able to determine given the incomplete usage reports.

ANCESTORS If **name** is not a function symbol, **NIL** is returned. Otherwise, returns a list of the names of all the **DEFN**, **CONSTRAIN**, **ADD-SHELL** and **DCL** events which introduced the function symbols that are ancestors of **name**. See page 169 for the definition of ''ancestor.''[12]

12.16. DCL

General Form:
(DCL fn args)

Example Form:
(DCL LOC (VAR MEM))

DCL is an event command for introducing an undefined function symbol into the logic. It does not evaluate its arguments. **fn** must be a new function symbol and **args** must be a list of **n** distinct variable symbols. **DCL** declares **fn** to be a function symbol of arity **n**. **fn** is henceforth considered ''old.'' **DCL** adds no axioms to the theory. Unless an error is caused because of improper arguments, **DCL** adds a new event to the data base, prints a time triple, and returns the new event name, **fn**.

If a function symbol has been introduced with **DCL** we say it is an *undefined* function symbol.

The example declaration above declares **LOC** to be a function of arity **2**, provided the name **LOC** has not been previously introduced as an event or function name. After the declaration, it is permitted to use such expressions as **(LOC V M)** and **(LOC (CAAR X) (MEM ST))** as terms.

Undefined function symbols can be used in formulas, e.g., axioms, definitions, and theorems. However, they cannot subsequently be defined. Undefined functions are not explicit value preserving: calls of undefined functions on explicit values cannot be reduced to explicit values by the primitive axioms and definitions. Calls of functions defined in terms of undefined functions are usually not explicit value preserving—not if the undefined functions are neces-

---

[12]The Lisp program **ANCESTORS** takes a list of function symbols and returns the union of the ancestral event names of each of them. **(DATA-BASE 'ANCESTORS name)** is just **(ANCESTORS (LIST name))**. When the ancestors of a clique of function symbols is needed, the routine **ANCESTORS** is much more efficient than multiple calls of **DATA-BASE**.

sarily involved in the computation. Since **DCL** adds no axioms, the primitive metafunctions **SUBRP**, **APPLY-SUBR**, etc., are undefined on **'fn** if they were undefined on **'fn** before the **DCL**. The user may add axioms to characterize **fn**. However, in that case we recommend that you use **CONSTRAIN**, rather than **DCL**, to introduce **fn** and thus avoid the use of **ADD-AXIOM**.

It may be instructive to note that **(DCL fn args)** could be implemented as **(CONSTRAIN fn-INTRO NIL T ((fn (LAMBDA args T))))**, except that the latter creates the event name **fn-INTRO** whereas the former uses **fn** as both the event name and the name of the introduced function.

12.17. DEFN

General Forms:
(DEFN fn args body)
and
(DEFN fn args body hints)

Example Forms:
```
(DEFN LEN (X)
      (IF (NLISTP X)
          0
          (ADD1 (LEN (CDR X)))))
```

and
```
(DEFN UP (I J)
      (IF (LESSP I J)
          (UP (ADD1 I) J)
          T)
      ((LESSP (DIFFERENCE J I))))
```

DEFN is an event command for defining a function symbol. It does not evaluate its arguments. **args** must be a list of **n** variable names v_1, ..., v_n. The command has the same theoretical effect as the axiomatic act:

Definition.
(fn v_1 **...** v_n**) = body**

with the following exception. Admissibility restriction (d) of the principle of definition is that ''there is a term **m** ...'' such that certain derived formulas are theorems. The *implemented* principle of definition further requires that the theorems be provable by the theorem prover's simplification process! If admissible, the theory is augmented by the axioms introduced by the above axiomatic act. In addition, the rules corresponding to those axioms are added to the data base. **DEFN** then computes the induction schemes suggested by the various measured subsets. It also computes a type prescription rule for the function and preprocesses the body of the function to create a definitional replacement rule (named **fn**) for expanding calls of the function. To preprocess the body, **DEFN** expands the calls of all nonrecursive primitive functions in the body (e.g., **AND**, **FIX**) and then normalizes the resulting **IF**-expression so that no **IF** contains an **IF** in its test. This may exponentially increase the size of the body. Finally, **DEFN** creates a new event named **fn** and links it into the data base. Then **DEFN** prints a narrative describing the justification(s) of the definition and the type prescription computed, a time triple is printed, and **fn** is returned.

If the axiomatic act above is inadmissible (or the simplifier cannot prove the required formulas), an appropriate message is printed, an error is caused, and the data base is left as it was before the **DEFN** command was executed.

We have not yet discussed how the appropriate measure term is selected by **DEFN**. The answer is simple: unless otherwise instructed by the **hints** argument, **DEFN** tries **COUNT** on each argument changed in recursion. The optional **hints** argument permits the user-specification of a measure term to be used.

If non-**NIL**, **hints** must be a list of doublets as described below. Each doublet specifies a well-founded relation and a measure term. For each doublet, **DEFN** generates formulas which, if theorems, are sufficient to establish the theorems described in requirement (d). **DEFN** tries to prove each of the formulas for each doublet. If it is successful for any doublet, the definition is admissible. If it is unsuccessful on each, an error is caused. The **hints** argument allows multiple ‘‘justifications’’ of a definition; for example the first doublet might claim that the **COUNT** of the first argument gets smaller, while the second doublet claims that the second argument gets closer to **100**. Having alternative justifications makes the theorem prover more powerful when it is considering what induction should be used on a given conjecture.

Each doublet must be of the form **(wfn m-term)**, where **wfn** is a ‘‘known well-founded relation’’ and **m-term** is a term all of whose variables are among those in **args**, namely $\mathbf{x_1}$, ..., $\mathbf{x_n}$. The known well-founded relations depend upon how **BOOT-STRAP** was called. If booted in **NQTHM** mode, the known well-founded relations are **ORD-LESSP** and **LESSP**.[13] In **THM** mode, the

---

[13]Technically, the logic requires use of **ORD-LESSP** to justify a definition or an induction. But it is easy to see, from the definition of **ORD-LESSP**, that if the analogous measure theorems are proved about **LESSP** then the required theorems could be proved for **ORD-LESSP**. When **LESSP** is used as the relation it is unnecessary to prove that the measure term satisfies **ORDINALP**: since **LESSP** coerces non-**NUMBERP**s to **0**, the measure term used in the required **ORD-LESSP** justification is just the **FIX** of the measure term used in the **LESSP** justification.

known well-founded relations are **LESSP**, **LEX2**, and **LEX3**.

When the system fails to prove the formulas justifying a submitted definition, it prints out an explanatory message that lists the measures tried and the simplified (but unproven) goals generated. If the measure you believe is the correct one was not tried, use the **hints** argument to specify the correct measure. If the measure you believe is the correct one was tried but could not be shown to decrease, prove **REWRITE** rules suitable to establish the printed goals and resubmit the definition.

12.18. DEFTHEORY

General Form:

```
(DEFTHEORY name theory)
```

Example Form:

```
(DEFTHEORY LIST-THEORY
  (APPEND *1*APPEND REVERSE *1*REVERSE
   ASSOC-OF-APPEND APPEND-RIGHT-ID REVERSE-REVERSE))
```

DEFTHEORY is an event command. It does not evaluate its arguments. The first argument, **name**, must be a new name. The second argument, **theory**, must be a non-empty list of citizens. **DEFTHEORY** creates a new event, **name**, that associates **theory** with **name**. Henceforth **name** may be used as a theory name and its ‘‘value’’ is **theory**.

In addition to the theory names defined via **DEFTHEORY** there are two built-in theory names:

- **GROUND-ZERO**: a theory name whose value is the list of citizens introduced by **BOOT-STRAP**, i.e., the satellites of the **GROUND-ZERO** event in the current data base; and
- **T**: a theory name whose value is the list of all citizens in the current data base.

Theory names make it convenient to enable or disable all the names associated with the theory. For example, the event **(DISABLE-THEORY name)** will globally disable all of the citizens in the value of the theory named **name**. There is an analogous hint to **PROVE-LEMMA**, also called **DISABLE-THEORY**, that permits the ‘‘local disabling’’ of all the names in a theory during a proof. The analogous enabling command and hint is called **ENABLE-THEORY**. See **DISABLE-THEORY**, **ENABLE-THEORY**, and the hints of **PROVE-LEMMA** for details. It is also possible to map over all the names in the value of a theory and set the enabled/disabled status of each according to the type of event that introduced the name, e.g., one can enable all the definitions and shell axioms and disable all the proved lemmas in a theory. See **SET-STATUS**. Using the built-in theory **T** and **SET-STATUS** it is possible to ‘‘erect barricades’’ in your data base marking the end of one phase of a project and the beginning of another. For example, by disabling all rules except definitions one ‘‘wipes the slate clean’’ and is able to begin developing a new set of rules about selected definitions just as though the definitions had just been introduced—with the added advantage that when old rules are recognized as useful they can be enabled. See **SET-STATUS**.

Theories are very useful in connection with projects involving several layers. Imagine a project in which there are just two layers, which we shall call the ‘‘low’’ level and the ‘‘high’’ level. The low layer is defined in terms of Nqthm primitives and the high level is defined primarily in terms of the low level functions. Let **LOW-DEFNS** be a theory that contains the low level function names and let **HIGH-DEFNS** be the analogous theory for the high level functions. Suppose a large collection of rules has been developed for each layer. Collect the rule names into two theories, **LOW-RULES** and **HIGH-RULES**. The intention is that the rules for a given layer be sufficient for reasoning about the functions at that layer. Thus, to prove a high level theorem, one should have **HIGH-DEFNS** disabled and **HIGH-RULES** enabled, so the high level functions are not expanded into their low level equivalents and the high level rules are employed to manipulate them. Now suppose a useful new high level rule is discovered and that it is not derivable from the current high level rules. To prove such a rule it is necessary to ‘‘drop down a level’’ and expose the definitions of the high level concepts. Thus, **HIGH-DEFNS** should be enabled, **HIGH-RULES** should be disabled and **LOW-RULES** should be enabled. The proof of the new high level rule is constructed from the rules of the lower level. Theories make it convenient to move between such layers or to ensure that the theorem prover stays at a given layer. This is particularly important when, as always seems to happen, the various layers share certain function symbols and contain different sets of rules for them according to the paradigms of use in a given layer. In these cases it is often the intended use rather than the statement of a theorem that indicates what layer is involved.

12.19. DISABLE

General Form:
`(DISABLE name)`

Example Forms:
`(DISABLE TIMES-2-REWRITE)`
and
`(DISABLE REVERSE)`
and
`(DISABLE *1*REMAINDER)`

`DISABLE` is an event command that sets the status of a citizen to **`DISABLED`**. The command does not evaluate its argument. The argument, **`name`**, must be a citizen. The command creates a new event name, **`new-name`**, adds a new event to the data base with **`new-name`** as its name, and sets the status of **`name`** in the data base to **`DISABLED`**. Then **`DISABLE`** prints a time triple and returns **`new-name`**. **`(DISABLE name)`** is just an abbreviation for **`(TOGGLE new-name name T)`**. Indeed, even though we speak of ‘‘**`DISABLE`** events’’ the data base actually records all **`DISABLE`**, **`ENABLE`**, and **`TOGGLE`** commands as **`TOGGLE`** events.

Roughly speaking, the effect of **`(DISABLE name)`** is to prohibit the use of **`name`** in all subsequent proof attempts until the **`DISABLE`** is overridden with an **`ENABLE`** command or temporarily overridden with an **`ENABLE`** hint in **`PROVE-LEMMA`** or **`new-name`** is undone.

More precisely, suppose the status of **`name`** is **`DISABLED`** in the data base. Then if **`name`** is the name of a rule, the rule is never applied. If **`name`** is the name of a defined function, calls of the function are not expanded.[14] If **`name`** is the name of the executable counterpart of a function, applications of the function to explicit values in formulas being proved are not reduced to explicit values.[15] Note that, while legal, there is no sense in disabling any name other than the name of a rule, a function, or an executable counterpart. For example, disabling the name of a **`DISABLE`** event has no effect.

---

[14] However, if all the arguments of a function call are explicit values, the executable counterpart of the function may be executed. Thus, **`(REVERSE '(1 2 3))`** will be replaced by **`'(3 2 1)`** even if **`REVERSE`** is disabled. To prevent this, disable the executable counterpart, **`*1*REVERSE`**, of the function as well.

[15] Applications of a **`DISABLED`** executable counterpart are reduced when they arise in the execution of some other executable counterpart. For example, suppose **`REVERSE`** is defined in terms of **`APPEND`** and the executable counterpart of **`APPEND`** is **`DISABLED`** as by **`(DISABLE *1*APPEND)`**. Then **`(APPEND '(3 2) '(1))`** will not be reduced to an explicit value when it appears in formulas being proved. But **`(REVERSE '(1 2 3))`** does reduce to **`'(3 2 1)`**, even though it requires the reduction of **`APPEND`** calls.

The status of a name in the data base is either **`ENABLED`** or **`DISABLED`** according to the most recently executed **`ENABLE`** or **`DISABLE`** (or equivalent **`TOGGLE`**) event for that name still in the data base. If no such event is in the data base, the status is **`ENABLED`**.

A citizen can also be disabled ''locally'' (without affecting its status in the data base) via the **`DISABLE`** and **`DISABLE-THEORY`** hints to **`PROVE-LEMMA`**, **`CONSTRAIN`**, and **`FUNCTIONALLY-INSTANTIATE`**.

Disabling is often important in trying to construct large proofs. Some users prefer to operate in the mode in which all names are disabled globally and a name must be explicitly enabled locally to be used in a proof. In this mode the theorem prover usually responds quickly because its options are so limited. The command **`LEMMA`** is a variant of **`PROVE-LEMMA`** that makes it easy to operate in this mode when desired.

Three situations requiring disabling arise so often we will discuss them explicitly here.

12.19.1. A Bolt from the Blue

Scenario: A proof that was supposed to work failed. Upon analysis of the output you see that a key **`REWRITE`** rule was not applied. Upon further analysis you see that it is because the target term to which it was supposed to apply was deformed by the prior application of another rule.

The most common solution to this is to disable the unwanted rule, either ''globally'' with the **`DISABLE`** command or ''locally'' with the **`DISABLE`** hint to **`PROVE-LEMMA`**. Depending on the nature of the unwanted rule it is sometimes more reasonable to re-express the desired rule so that its left-hand side anticipates the action of the other rule. This is particularly true if the ''unwanted'' rule is one of the fundamental rules of your evolving theory and having it disabled is disastrous. It should be noted that the ''unwanted'' rule might be a definition which is unexpectedly expanding.

12.19.2. Nonrecursive Definitions

Scenario: A complicated nonrecursive function has been defined, and you have just finished proving a set of beautiful **`REWRITE`** rules that hide its complexity and let you manipulate calls of the function smoothly. However, none of your rules are ever applied!

The problem here is that nonrecursive functions are (almost) always expanded immediately. Thus, no rule about such a function will ever find a match in a simplified conjecture. The solution, generally, is to disable the nonrecursive function as soon as you have characterized it with **REWRITE** rules. If you use nonrecursive functions extensively and do not want all proofs about them to proceed by simply expanding them into their cases, we advise the following approach: define the function, prove the fundamental properties as rewrite rules, disable the function, and base all subsequent proofs on the fundamental properties. You will frequently discover that you omitted a ‘‘fundamental property.’’ **ENABLE** will be useful then.

12.19.3. Hierarchical Rule Development

Scenario: You have just finished proving a fundamental result in your evolving theory. You expect the result to be used widely in subsequent proofs. However, you find that it is not.

The problem here, often, is that the lemmas proved in order to derive the fundamental result are ‘‘getting in the way.’’ Frequently, the proof of a major theorem requires the proofs of many minor ones that handle special cases. These ‘‘lemmas’’ are often formulated haphazardly simply to permit the derivation of the proof of the main theorem. However, because the lemmas necessarily deal with the same clique of function names as the main theorem, they will often find unanticipated applications outside the proof of the main theorem. In particular, they may well deform the very targets for which the main theorem was intended. The standard solution is to disable these intermediate or inconsequential lemmas once the main theorem has been proved.

12.20. DISABLE-THEORY

General Form:
```
(DISABLE-THEORY theory-name)
```

Example Forms:
```
(DISABLE-THEORY LIST-THEORY)
```

DISABLE-THEORY is an event command. It does not evaluate its argument. The argument, **theory-name**, should be the name of a theory. See **DEFTHEORY**. **DISABLE-THEORY** creates a new event name, **name**, adds a new event to the data base with **name** as its name, and sets the status of every name in the list of citizens denoted by **theory-name** to **DISABLED** unless the name's status already is **DISABLED**. Then **DISABLE-THEORY** prints a time triple and returns **name**. The **DISABLE-THEORY** hint to **PROVE-LEMMA** permits one to disable a theory ''locally.''

(DISABLE-THEORY theory-name) is just an abbreviation for **(SET-STATUS name theory-name ((OTHERWISE DISABLE)))**, which is defined on page 205. The careful reader will note that **theory-name** is therefore also permitted to be an explicit list of citizen names or a dotted-pair naming the endpoints of an interval of citizen names. However, to highlight the symmetry between the global disabling effect of this event and the local disabling effect of the **DISABLE-THEORY** hint to **PROVE-LEMMA**, where the ''arguments'' must be theory names, we prefer to encourage the view that **DISABLE-THEORY** operates on a named theory. The experienced user may wish to use the more general forms of **DISABLE-THEORY**.

See page 152 for a discussion of the effect of enabling or disabling a name and why it is done. See **DEFTHEORY** for a general discussion of why theories are useful. See the documentation of **SET-STATUS** for a warning about the peculiar interaction between **SET-STATUS** and undoing.

12.21. DO-EVENTS

General Form:
(DO-EVENTS events)

Example Forms:
```
(DO-EVENTS '((DEFN LEN (X)
               (IF (NLISTP X)
                   0
                   (ADD1 (LEN (CDR X)))))
             (PROVE-LEMMA LEN-APPEND (REWRITE)
               (EQUAL (LEN (APPEND A B))
                      (PLUS (LEN A) (LEN B))))))
```

```
(DO-EVENTS XXX)
```

DO-EVENTS executes each event form in a list of such forms and is the most common way to replay an edited sequence of undone events or process an unproblematic sequence of newly created commands. The argument, **events**, is evaluated and should be a list of event forms.

DO-EVENTS iterates down **events**, considering each event form in turn. For each form it prettyprints the form, then evaluates the form (which causes additional printing, e.g., of proofs and time triples), prints the value of the form, and then prints a form feed.[16] **DO-EVENTS** terminates either when all events have been processed or some event either causes an **ERROR** or **FATAL ERROR** or some **PROVE-LEMMA** fails. **DO-EVENTS** returns **T** if all events were successfully processed and **NIL** otherwise.

Normally, the output of **DO-EVENTS**, and of the event commands it executes, is printed to the terminal. However, when used by **PROVEALL**, all output, including error messages, goes to the **proofs** and **err** extensions as noted on page 178. In this case, **DO-EVENTS** indicates its progress through **events** by printing out the name of each event form just before the form is executed. It prints out a comma upon completion of the execution of each form.

12.22. DO-FILE

General Form:
(DO-FILE file)

Example Form:
(DO-FILE "region.events")

DO-FILE executes the forms in a file, treating them as though they were Nqthm event forms. The argument is evaluated and should the name of a file containing Nqthm events. The forms in the file are iteratively read, printed to the terminal, and evaluated. If any form causes an error or returns nil then it is considered to have failed and **DO-FILE** terminates and returns **NIL**. **DO-FILE** terminates and returns **T** when it has processed all of forms in the file.

DO-FILE is similar in spirit to **PROVE-FILE** in that it processes a file presumed to contain events. Unlike **PROVE-FILE** it does not check that the file actually contains events and will execute any lisp form. In that sense, **DO-FILE** is similar to the Common Lisp function **LOAD**, except that **DO-FILE** prints the forms before execution and aborts if any form fails. Because of its cavalier attitude toward the forms in the file, **DO-FILE** should not be used for the final ‘‘endorsing’’ replay of a script file. But it is very handy in daily work where, for example, it is not uncommon to replay a segment of a larger event file by writing it to a temporary file and calling **DO-FILE**.

---

[16]Actually, **DO-EVENTS** prints the value of the global Lisp variable **EVENT-SEPARATOR-STRING** to separate one event and its output from the next. The initial value of that variable is **#\Page**.

12.23. ELIM

ELIM is one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. **ELIM** rules are used in the destructor elimination process, which is the second process tried (simplification being the first).

12.24. ENABLE

General Form:
(ENABLE name)

Example Forms:
(ENABLE TIMES-2-REWRITE)
and
(ENABLE REVERSE)
and
(ENABLE *1*REMAINDER)

ENABLE is an event command that sets the status of a citizen to **ENABLED**. The command does not evaluate its argument. The argument, **name**, must be a citizen. The command creates a new event name, **new-name**, adds a new event to the data base with **new-name** as its name, and sets the status of **name** in the data base to **ENABLED**. Then **ENABLE** prints a time triple and returns **new-name**. **(ENABLE name)** is just an abbreviation for **(TOGGLE new-name name NIL)**.

See the discussion of **DISABLE** for details.

A citizen can also be enabled ‘‘locally’’ (without affecting its status in the data base) via the **ENABLE** and **ENABLE-THEORY** hints in **PROVE-LEMMA** **CONSTRAIN**, and **FUNCTIONALLY-INSTANTIATE**.

12.25. ENABLE-THEORY

General Form:
(ENABLE-THEORY theory-name)

Example Forms:
(ENABLE-THEORY LIST-THEORY)

ENABLE-THEORY is an event command. It does not evaluate its argument. The argument, **theory-name**, should be the name of a theory. See **DEFTHEORY**. **ENABLE-THEORY** creates a new event name, **name**, adds a new event to the data base with **name** as its name, and sets the status of every name in the list of citizens denoted by **theory-name** to **ENABLED** unless the name's status already is **ENABLED**. Then **ENABLE-THEORY** prints a time triple and returns **name**. The **ENABLE-THEORY** hint to **PROVE-LEMMA** permits one to enable a theory ''locally.''

(ENABLE-THEORY theory-name) is just an abbreviation for **(SET-STATUS name theory-name ((OTHERWISE ENABLE)))**, which is defined on page 205. The careful reader will note that **theory-name** is therefore also permitted to be an explicit list of citizen names or a dotted-pair naming the endpoints of an interval of citizen names. However, to highlight the symmetry between the global enabling effect of this event and the local enabling effect of the **ENABLE-THEORY** hint to **PROVE-LEMMA**, where the ''arguments'' must be theory names, we prefer to encourage the view that **ENABLE-THEORY** operates on a named theory. The experienced user may wish to use the more general forms of **ENABLE-THEORY**.

See page 152 for a discussion of the effect of enabling or disabling a name and why it is done. See **DEFTHEORY** for a general discussion of why theories are useful. See the documentation of **SET-STATUS** for a warning about the peculiar interaction between **ENABLE-STATUS** and undoing.

12.26. Errors

Each error condition checked and reported by the system is classified into one of three classes:

- **WARNING**: a cautionary message concerning a legal but perhaps unintended aspect of a command, e.g., that the body of a function definition makes no reference to one of the formal parameters.
- **ERROR**: a violation of the preconditions on a command, such as the submission of an ill-formed term, the submission of an inadmissible function definition, the attempted introduction of an event name already in the data base, etc. Generally **ERROR**s can be fixed by editing your type-in, though they sometimes indicate a deeper problem than mere syntax.
- **FATAL ERROR**: a violation of the internal invariants assumed by our code or the exhaustion of some resource (e.g., the variable symbols that may be introduced by destructor elimination). It is generally not possible to fix such an error. Except for resource

errors or situations in which the user has entered our code via some route other than the documented commands, **FATAL ERROR**s should not arise.

The system may cause Lisp resource errors, such as stack overflow due to excessively deep function calling (usually an indication of circularities in the rewrite rules you have added) or the exhaustion of dynamic storage space (usually caused by circular rewriting that enlarges the term, excessive case splitting, or combinatoric explosion in the normalization of propositional expressions). The system should not cause any other kind of error, e.g., ‘‘Trap: Argument given to CAR was not a list, locative, or NIL,’’ while it is being used entirely in accordance with this handbook. If such errors occur, please notify us.

All three types of errors print supposedly self-explanatory messages on the terminal. After **WARNING**s the computation proceeds as though nothing had happened. The other two kinds of errors cause interactive Lisp breaks.

It is not possible to fix an error from within the Lisp break and proceed. Proceeding from the break automatically aborts the computation that led to the error. The only reason we enter a break at all is to give you the opportunity to inspect the state of the computation before it is lost by the abort.

Getting out of the break is a system-dependent operation.

- On a Symbolics Lisp Machine, press the **ABORT** key.
- In Lucid on a Sun, type **:A** to the break prompt.
- In KCL or CMU Common Lisp on a Sun, type **:Q** to the break prompt.
- In Allegro on a Sun, type **:RES** to the break prompt.

These actions should return you to the top level of the Lisp system.

The data base is not disturbed by any command that halts with an **ERROR**. The same cannot be said if a **FATAL ERROR** occurs. See however the discussion of the data base integrity on page 120.

To fix an error you must edit (or retype) the command that caused it. Most Lisp systems support this, and since our code runs in many different implementations of Common Lisp and under many different kinds of host operating systems, we have not tried to implement a uniform error recovery mechanism.

12.27. Event Commands

The *event commands* are those that create nodes in the data base. The event commands are

- **ADD-AXIOM** (with its variant **AXIOM**)
- **ADD-SHELL**
- **BOOT-STRAP**
- **CONSTRAIN**
- **DCL**
- **DEFN**
- **DEFTHEORY**
- **FUNCTIONALLY-INSTANTIATE**
- **PROVE-LEMMA** (with its variant **LEMMA**)
- **SET-STATUS** (with its variants **DISABLE-THEORY** and **ENABLE-THEORY**)
- **TOGGLE** (with its variants **DISABLE** and **ENABLE**)
- **TOGGLE-DEFINED-FUNCTIONS**

The ‘‘variants’’ of the event commands are defined in terms of the basic commands but provide convenient commonly used forms. Thus, for example, a **DISABLE** command actually executes a certain call of **TOGGLE**. It is the **TOGGLE** command that is stored in the data base.

12.28. EVENTS-SINCE

General Form:
(EVENTS-SINCE name)

Example Form:
(EVENTS-SINCE 'REVERSE)

The argument, **name**, is evaluated and must be the name of an event. **EVENTS-SINCE** returns the list of event forms for all events created since **name**. The list includes the event form for **name** itself and is ordered chronologically, starting with the event form for **name**.

12.29. Executable Counterparts

Corresponding to every function symbol in the logic is a Lisp procedure called the *executable counterpart* of the function. The name of the executable counterpart is obtained by concatenating the string, ''**`*1*`**'' onto the front of the function symbol. Thus, **`*1*APPEND`** is the executable counterpart of **`APPEND`**.

Executable counterparts are used to compute the value of functions on constants. Such computations are the essence of the execution facility provided by **`R-LOOP`** but also arise in the theorem prover proper. Whenever a term of the form **`(fn 't`**$_1$ **`... 't`**$_n$**`)`**, e.g., the application of a function to explicit values, is produced, we use the executable counterpart of **`fn`** to compute the equivalent explicit value, if any.[17] To explain further we must first discuss the notion of explicit value.

Recall that an *explicit value* is a variable-free term involving only shell constructors and base objects with the property that each argument of each constructor satisfies the type restriction of the corresponding argument position. For example, **`(CONS (ADD1 (ZERO)) (ZERO))`** is an explicit value term.

Internally, each explicit value term, **`t`**, is represented by a Lisp data structure of the form **`(QUOTE evg)`**, where **`evg`** is called the *explicit value guts* of **`t`**. The explicit value guts is just the Lisp object corresponding to what we called the ''explicit value descriptor'' in section 4.7.5, page 90.[18]

The explicit value above is represented by **`(QUOTE (1 . 0))`**. The Lisp object **`(1 . 0)`** is the explicit value guts of **`(CONS (ADD1 (ZERO)) (ZERO))`**.

Suppose that **`'e`**$_1$, ..., **`'e`**$_n$ are **`n`** explicit value terms and that their respective explicit value guts are **`e`**$_1$, ..., **`e`**$_n$. Suppose **`fn`** is an **`n`**-ary function. Then **`(fn 'e`**$_1$ **`... 'e`**$_n$**`)`** is a term which is likely to be equal to some explicit value. The executable counterpart of **`fn`**, **`*1*fn`**, can sometimes be used to compute the explicit value equivalent to **`(fn 'e`**$_1$ **`... 'e`**$_n$**`)`** as follows: Apply the Lisp procedure **`*1*fn`** to the **`n`** Lisp objects **`e`**$_1$, ..., **`e`**$_n$. If a result, **`e`**, is returned

---

[17] More precisely, the executable counterpart of **`fn`** is so used, provided either **`fn`** is a shell constructor or base function symbol or it is enabled. Executable counterparts can be disabled individually with **`DISABLE`** or **`TOGGLE`**, as in **`(DISABLE *1*APPEND)`**, in theories with **`DISABLE-THEORY`**, or in regions of the **`CHRONOLOGY`** with **`SET-STATUS`**. All executable counterparts can be disabled with **`TOGGLE-DEFINED-FUNCTIONS`**. Note that shell constructors and base function cannot be effectively disabled; they are used whether disabled or not. The theorem prover's internal form requires that explicit values be kept in a certain normal form.

[18] This is not quite true. A ''dotted s-expression'' in our formal presentation is a sequence whose next-to-last element is the dot character. In Lisp, the explicit value guts corresponding to a ''dotted s-expression'' is in fact a binary tree of Lisp list cells whose right-most cell contains an atom other than **`NIL`**. Such trees are printed by Lisp with a dot. Our dotted s-expressions are read by Lisp and converted into such trees.

then **'e** is the internal representation of an explicit value term equivalent to **(fn 'e$_1$ ... 'e$_n$)**. Otherwise, the execution of ***1*fn** will signal failure (by executing a Lisp **THROW**). Failure is signaled either because an undefined function is encountered on the execution path or, for functions involving **V&C$**, certain resource limits are exhausted. See **REDUCE-TERM-CLOCK**.

Note that we have not said that ***1*fn** computes an equivalent explicit *if there is one*, merely that *if *1*fn computes a value* it represents an equivalent explicit value. However, for a very large class of functions the equivalent explicit value always exists and the executable counterparts compute them.

We say a function **fn** is *explicit value preserving* if either (a) it is one of the primitive functions other than the metafunctions **SUBRP**, **APPLY-SUBR**, **FORMALS**, **BODY**, **V&C$**, **V&C-APPLY$**, **EVAL$**, **APPLY$**, and **FOR** or (b) it is a defined function with the property that every function symbol used in the body, other than **fn** itself, is explicit value preserving. Explicit value preserving functions have the property that for every application to explicit values there exists an equivalent explicit value. Except for those functions which use the metafunctions excluded above and those which use functions introduced with **DCL** or **CONSTRAIN**, every function in our system is explicit value preserving. It is in this sense we used the word ''likely'' above. For all explicit value preserving functions, the executable counterparts always compute the corresponding explicit values.

However, in addition to the syntactically defined class of explicit value preserving functions, there are defined functions which in fact reduce, or often reduce, to explicit values on explicit values even though metafunctions or undefined functions are used on some branches through their definitions. **V&C$** itself is a good example: on many applications to explicit values, e.g., on applications to the quotations of terms composed of explicit value preserving functions, **V&C$** can be reduced computationally to an explicit value.

Every function symbol in the logic has an executable counterpart whether the function is explicit value preserving or not. The **ADD-SHELL** command and the **DEFN** command create the executable counterparts for the symbols they introduce. The **DCL** and **CONSTRAIN** commands create the trivial executable counterpart (''signal failure'') for their function symbols. The command **TOGGLE-DEFINED-FUNCTIONS** can be used to disable all executable counterparts other than shell constructors and base objects. If ***1*fn** is disabled then **(fn 'e$_1$ ... 'e$_n$)** is not reduced to an explicit value by execution of ***1*fn**.

It is possible to compile executable counterparts. This generally makes **R-LOOP** evaluate your input much faster, and it will speed up your proofs if they involve explicit values. See **COMPILE-UNCOMPILED-DEFNS**.

12.30. Explicit Values

Explicit values are the canonical constants in the logic: terms composed entirely of shell constructors and bottom objects with the additional property that every subterm satisfies the type restriction for the constructor argument position occupied. See page 69, in Chapter 4 for a formal definition of explicit values. See page 90 for a formal description of the notation in which they are written. See the discussion of ‘‘executable counterparts,’’ page 161, for a discussion of how we compute on explicit values.

12.31. Extensions

See the discussion of File Names on page 163.

12.32. FAILED-EVENTS

General Form:
`FAILED-EVENTS`

`FAILED-EVENTS` is a Lisp variable that contains all of the event commands submitted in the current session that did not terminate successfully.

12.33. File Names

Generally speaking it is not necessary to know anything about file names the first few times you use the theorem prover: commands are typed to Lisp at the terminal and their output is printed at the terminal. The first use of file names usually occurs when you want to save the current data base to a file, with **`MAKE-LIB`**, and subsequently restore the data base, with **`NOTE-LIB`**. The other common use of file names is at the successful conclusion of a major phase of a project when you will probably want to create an ‘‘endorsed data base’’ (see page 185) and a coherent record of the event commands that construct it and the proofs concerned. This is done by calling the function **`PROVE-FILE`** or the function **`PROVEALL`**.

To use these more sophisticated commands it is necessary to understand file name formats and our file name conventions. The file name format used by the theorem prover is that defined by the host operating system. *We assume you are familiar with the file name formats on your machine.* We make two assumptions about file name formats and then proceed to define our file name conventions. Our assumptions are

- File names are written as Lisp strings, i.e., ASCII character sequences enclosed in double quotation marks, as in **"doe:>nqthm>demo.lib"**.
- A ''full file name'' is composed of two parts, the ''root name'' and the ''extension.'' The root name is separated from the extension by the ASCII dot character, ''**.**''. The format of both the root name and the extension are dictated entirely by the host file system.

In the development of a given subject, e.g., the proof of correctness of the program **FSTRPOS**, the command file, the library files, the proof file, and the error message file all generally have the same root name, which on a Lisp machine might be **"doe:>smith>fstrpos"** and on a Unix system **"/usr/smith/fstrpos"**.

We use the extension of each file name to indicate the particular kind of data contained in the file. Commands that create or read files, such as **NOTE-LIB** and **PROVE-FILE**, take root names as their arguments and extend them by tacking on one of several fixed extension names according to the type of file to be read or written.

For example, proofs generated by **PROVEALL** are written into a file with the **proofs** extension while error messages are written to the **err** extension. There are, in all, nine different extensions used by the system. The names used above, **proofs** and **err**, are just ''generic'' names used in this handbook. The actual strings used to form extensions may be specified by the user, by setting each of six global Lisp variables. Thus, the **proofs** extension of **"/usr/smith/fstrpos"** might be **"/usr/smith/fstrpos.proofs"** on one system and **"doe:>smith>fstrpos.prf"** on another. We make this precise in the definition below.

Given a root name, **file**, we speak of nine different *extensions* of **file**, each known by a generic name. Each extension is formed by concatenating **file**, a dot, and then a character string (or Lisp symbol) associated with the generic name. To each generic name there corresponds a Lisp variable whose value is the associated character string. These variables may be set by the user. The generic names, the kind of data contained in such extensions, the Lisp variable concerned, and its initial setting are described below.

events The **PROVE-FILE** program reads and executes the forms in the **events** extension of its **file** argument. The name used for the **events** extension is the value of the Lisp variable **FILE-EXTENSION-EVENTS**, which is initially **"events"**.

proofs The **PROVEALL** program opens a file with the **proofs** extension to contain a transcript of the event sequence being executed. In particular, each event form is printed into the **proofs** extension, and then the form is executed with all of its output written to the **proofs** extension—error messages, comments on definitions, proofs, time triples, etc.—and then the value of the completed form is printed into the **proofs** extension. See **PROVEALL** for the details. The name used for the **proofs** extension is the value of the Lisp variable **FILE-EXTENSION-PROOFS**, which is initially **"proofs"**. The variant of **PROVE-FILE** that redirects terminal output, **PROVE-FILE-OUT**, also opens a **proofs** extension of its **file** argument and writes the transcript to it.

err The **PROVEALL** program opens a file with the **err** extension to contain all **WARNING**, **ERROR** and **FATAL ERROR** messages generated during the **PROVEALL**. (These messages are also printed into the **proofs** extension as part of the transcript.) The **err** extension contains nothing but such messages and is a convenient summary of the anomalies in the event sequence. The name used for the **err** extension is the value of the Lisp variable **FILE-EXTENSION-ERR**, which is initially **"err"**.

lib The **lib** extension is one of two extensions in which the data base is saved by **MAKE-LIB** and restored by **NOTE-LIB**. The other is the **lisp** extension below. The data in the **lib** extension is written from and read into Lisp variables and property lists. The name used for the **lib** extension is the value of the variable **FILE-EXTENSION-LIB** and is initially **"lib"**.

lisp The **lisp** extension is used in connection with files that contain Common Lisp code. **MAKE-LIB** and **NOTE-LIB** use the the **lisp** extension for a file containing the Common Lisp definitions of the executable counterparts of the functions in the saved data base. **COMPILE-UNCOMPILED-DEFNS** uses the **lisp** extension when it creates a file to be compiled. These uses of the **lisp** extension all concern files created by the system in connection with executable counterparts. But the source files of the theorem prover itself are also Common Lisp files and, as noted in the Installation Guide, Chapter 14, the programs **COMPILE-NQTHM** and **LOAD-NQTHM** also use the **lisp** extension. The name used for the **lisp** extension is the value of the Lisp variable **FILE-EXTENSION-LISP**

and is initially **"lisp"**.

bin A file with the **bin** extension should be the result of compiling the corresponding **lisp** extension. **COMPILE-UNCOMPILE-DEFNS** uses the **bin** extension to load the compiled file it creates. During installation of our system, **COMPILE-NQTHM** and **LOAD-NQTHM** (see Chapter 14) use the **bin** extension to load compiled files. The name used for the **bin** extension is the value of the Lisp variable **FILE-EXTENSION-BIN**. In general, **FILE-EXTENSION-BIN** should be set to the same extension used by the local compiler. That is, if your compiler puts the object code for file **foo** into **foo.o** then **FILE-EXTENSION-BIN** should be set to **"o"**; if your compiler puts the object code for file **foo** into **foo.bin** then **FILE-EXTENSION-BIN** should be set to **"bin"**. However, the only use we make of **FILE-EXTENSION-BIN** is when one of the programs mentioned above **LOAD**s the compiled code for a given root name. As far as we are aware, all implementations of the Common Lisp **LOAD** program now support the convention that, when no extension is used and a compiled file for the given root name exists, then the compiled file is **LOAD**ed. Thus, it is not necessary to know the extension used by the compiler—which is to say, one need not extend the root name to refer explicitly to the compiled file to be loaded. We therefore adopt the convention that when **FILE-EXTENSION-BIN** is **NIL** no extension is tacked on. This is the initial setting for **FILE-EXTENSION-BIN**. If your implementation of Common Lisp does not support the convention, you should set **FILE-EXTENSION-BIN** to the same string used by the local compiler.

started The **PROVE-FILE** program, when observing its flag file protocol, signals the commencement of processing by creating the **started** extension of its **file** argument. The name used for the **started** extension is the value of the variable **FILE-EXTENSION-STARTED** and is initially **"STARTED"**.

proved The **PROVE-FILE** program, when observing its flag file protocol, signals the successful termination of processing by creating the **proved** extension of its **file** argument. The name used for the **proved** extension is the value of the variable **FILE-EXTENSION-PROVED** and is initially **"proved"**.

fail The **fail** extension is a file generated by **PROVEALL** and

PROVE-FILE (when it is observing its flag file protocol) when the event sequence causes an error or a proof fails. The name used for the **fail** extension is the value of the Lisp variable **FILE-EXTENSION-FAIL** and is initially **"fail"**.

We now raise a further complication in the use of file names: most Lisps—or the underlying operating systems—provide default mechanisms by which our ‘‘full file names’’ are further elaborated before a unique file identifier is generated. For example, on Unix the user who is ‘‘connected’’ to the directory **/usr/smith** may find that the **lisp** extension of the root name **"fstrpos"**, namely **"fstrpos.lisp"**, actually identifies the file **"/usr/smith/fstrpos.lisp"**. Similarly, by setting the appropriate ‘‘default path names,’’ the user on the Lisp Machine can arrange for a simple root name such as **"fstrpos"** to identify files with much more elaborate unique names.

The Lisp variable ***DEFAULT-NQTHM-PATH*** may be used to determine the directory from which files are read and written while compiling, loading, or using the theorem prover. The initial value of ***DEFAULT-NQTHM-PATH*** is **NIL**, which means ‘‘do nothing.’’ However, if the value is non-**NIL**, then we merge that value, using the Common Lisp function **MERGE-PATHNAMES**, whenever we **OPEN** a file for reading or writing and whenever we **LOAD** a file. If you specify a full pathname when using functions such as **NOTE-LIB**, **MAKE-LIB**, and **PROVEALL**, then the value of ***DEFAULT-NQTHM-PATH*** is irrelevant. However, if you specify a non-**NIL** value for ***DEFAULT-NQTHM-PATH*** and you specify no directory in a file name passed to **NOTE-LIB**, **MAKE-LIB**, etc., then the file used will be on the directory specified by ***DEFAULT-NQTHM-PATH***. When non-**NIL**, ***DEFAULT-NQTHM-PATH*** should be set to a string (not a pathname) that gives the full name of the relevant directory. It is important that the last character of the string be the characer that separates components of a pathname on your system, e.g., slash on a Unix system, colon on a Macintosh, or the greater-than character on a Symbolics Lisp Machine. Example values of ***DEFAULT-NQTHM-PATH*** are **"/usr/smith/"** and **"doe:>nqthm>"**.

Finally, when Nqthm opens a file for output it explicitly deletes any existing file by that name.

12.34. FUNCTIONALLY-INSTANTIATE

General Forms:

```
(FUNCTIONALLY-INSTANTIATE name rule-classes term
                          old-name fs)
```

and

```
(FUNCTIONALLY-INSTANTIATE name rule-classes term
                          old-name fs hints)
```

Example Form:

```
(FUNCTIONALLY-INSTANTIATE TIMES-PR-IS-TIMES-AC
  (REWRITE)
  (EQUAL (TIMES-AC L Z) (TIMES-PR L Z))
   H-PR-IS-H-AC
   ((H TIMES)
    (H-PR TIMES-PR)
    (H-AC (LAMBDA (X Y) (TIMES-AC X Y)))))
```

FUNCTIONALLY-INSTANTIATE is like **PROVE-LEMMA** in that it proves a theorem, **term**, and adds it to the data base as a theorem under the name **name** and as rules of the classes specified by **rule-classes**; however, **term** must be the functional instantiation under **fs** of some already proved theorem, named **old-name**. Rather than proving **term** directly, **FUNCTIONALLY-INSTANTIATE** attempts to verify that the axioms used in the proof of **old-name** are theorems under **fs**.

FUNCTIONALLY-INSTANTIATE is an event command. It does not evaluate its arguments. **name** must be a new name, **rule-classes** must be a (possibly empty) list of rule classes, and **term** must be a term or the word ***AUTO***. If **old-name** is a symbol, then the **FORMULA DATA-BASE** query for **old-name** must be some non-**NIL** formula **old-term**.[19] **fs** must be a functional substitution; the domain of **fs** must consist of function symbols already introduced into the theory by the user via **CONSTRAIN**, **DCL**, or **DEFN** (i.e., the domain may not include symbols in the **GROUND-ZERO** theory nor functions introduced with **ADD-SHELL**) and the arity of each function in the domain of **fs** must be the arity of the corresponding function in the range. Unless **term** is ***AUTO***, **term** must be the functional instantiation of **old-term** under **fs**, i.e., **old-term****fs**. (If **term** is ***AUTO***, we treat it as though that were an abbreviation for **old-term****fs** below.) Finally, **hints**, if

[19]The case in which **old-name** is not a symbol is rarely used and we discuss it later.

supplied, must be as required by the **PROVE-LEMMA** event (see page 188).

If the above syntactic restrictions are satisfied, **FUNCTIONALLY-INSTANTIATE** undertakes to prove **term** via functional instantiation. In particular, the theorem prover is called (with the **hints** provided) on the conjunction of the **fs** functional instances of the formulas of certain **ADD-AXIOM**, **CONSTRAIN**, and **DEFN** events. Such an event is involved iff (a) it uses as a function symbol some symbol in the domain of **fs** and (b) it is either (i) an **ADD-AXIOM** or (ii) a **CONSTRAIN** or **DEFN** that introduces a function symbol ''ancestral'' in **old-term** or some **ADD-AXIOM**. The user of the system need not understand what we mean by ''ancestral'' since the system determines the relevant formulas and prints them out before undertaking to prove them. However, we define the notion below. If any of the formulas to be instantiated use as a variable some variable that is free in any **LAMBDA** expression in the range of **fs**, then **FUNCTIONALLY-INSTANTIATE** aborts with an error and does not change the data base. Such aborts can always be avoided by choosing new variable names in **fs**. If the proof attempt is successful, **term** is stored in the data base as a theorem and rules of the classes specified by **rule-classes** are generated and stored as well. The name of each rule is **name**. If the proof attempt fails, **FUNCTIONALLY- INSTANTIATE** behaves as does **PROVE-LEMMA**: the ''failure banner'' is printed along with a time triple and **NIL** is returned.

A function symbol **fn** is *ancestral* in a term if and only if **fn** is an ''ancestor'' of some symbol used as a function symbol in the term. We say that **fn** is an *ancestor* of **g** if and only if **g** was introduced by a **DEFN** or **CONSTRAIN** event, **ev**, and either **fn** is one of the function symbols introduced by **ev** (including **g** itself) or **fn** is an ancestor of a function symbol used in the axiom added by **ev**. That this definition of ancestral is sufficient to justify functional instantiation is the subject of the main proof in [2].

We wish to make it convenient to apply the same functional substitution to several different theorems in a sequence of **FUNCTIONALLY-INSTANTIATE** events, without having to prove the same constraints repeatedly. Therefore, **FUNCTIONALLY-INSTANTIATE** does not bother to prove the **fs** instance of a formula if any previous **FUNCTIONALLY-INSTANTIATE** did prove it. This requires searching through a record kept of the proof obligations of all previous **FUNCTIONALLY-INSTANTIATE** events. If there are many such events you may wish to limit the search to some particular set of past events. This can be done by using a non-symbol for **old-name**. If **old-name** is not a symbol then it must be a list of the form **(old-name' name**$_1$ **... name**$_k$**)**, where **old-name'** is a name whose **FORMULA DATA-BASE** query produces the non-**NIL** formula **old-term** in foregoing description, and each **name**$_i$ is the name of a previous **FUNCTIONALLY-INSTANTIATE** event whose proof obligations are to be searched. The search is limited to the **name**$_i$.

12.35. Functional Substitution

General Form:
((f$_1$ g$_1$) ... (f$_n$ g$_n$))

Example Form:
((H PLUS) (G (LAMBDA (X) (TIMES K X))))

As defined formally on page 114, 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 a function symbol or **LAMBDA** expression and the arity of **f** is the arity of **g**. In our implementation, such substitutions are represented by lists of doublets as shown in the general form above. Each **f$_i$** must be a symbol and each **g$_i$** must be a function symbol or **LAMBDA** expression of the form **(LAMBDA (a$_1$ ... a$_n$) body)**, where the **a$_i$** are distinct variable symbols and **body** is a term. The set of the **f$_i$** is called the domain of the substitution and the set of the **g$_i$** is called the range. Functional substitutions have two distinct uses in our implementation and each use puts some additional restrictions on the substitution.

Functional substitutions are used with the **CONSTRAIN** event to supply both arities and witnesses for the functions being introduced. In such use, the **f$_i$** above must be new names and the **LAMBDA** expressions occurring among the **g$_i$** may not contain free variables; **CONSTRAIN** uses the **g$_i$** as witnesses for the **f$_i$** while verifying that the axiom to be added is satisfiable and then **CONSTRAIN** introduces the **f$_i$** as function symbols with the same arities as the corresponding **g$_i$**. The arity of **(LAMBDA (a$_1$ ... a$_n$) body)** is **n**.

Functional substitutions are also used with the **FUNCTIONALLY-INSTANTIATE** event, where they are used to specify how some theorem is to be functionally instantiated. In that usage, the **f$_i$** are required to be function symbols already introduced into the theory by the user via **CONSTRAIN**, **DCL**, or **DEFN** (i.e., the **f$_i$** may not be **GROUND-ZERO** functions nor functions introduced with **ADD-SHELL**) and the arity of **f$_i$** must be the arity of the corresponding **g$_i$**.

12.36. GENERALIZE

GENERALIZE is one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. **GENERALIZE** rules are used in the generalization process, which is the fourth process tried (after simplification, destructor elimination, and use of equalities).

12.37. Hints to PROVE-LEMMA, CONSTRAIN and FUNCTIONALLY-INSTANTIATE

General Form:

```
((USE (name (var term) ... (var term))
      ...
      (name (var term) ... (var term)))
 (EXPAND term ... term)
 (DISABLE name ... name)
 (ENABLE name ... name)
 (DISABLE-THEORY theory-name ... theory-name)
 (ENABLE-THEORY theory-name ... theory-name)
 (HANDS-OFF fn ... fn)
 (INDUCT term)
 (DO-NOT-INDUCT T)
 (DO-NOT-GENERALIZE T)
 (NO-BUILT-IN-ARITH T))
```

The ‘‘general form’’ shown above is meant merely to be suggestive of the form taken by the **hints** argument to **PROVE-LEMMA**, **CONSTRAIN** and **FUNCTIONALLY-INSTANTIATE**. See page 188 for details. The **hints** argument to **DEFN** has a different form and is described on page 149.

12.38. Hints to DEFN

General Form:

```
((relation-name measure-term)
 ...)
```

Example Form:

```
((LESSP (DIFFERENCE (ADD1 MAX) I)))
```

See page 149 for details on the form taken by the **hints** argument to **DEFN**. The **hints** argument to **PROVE-LEMMA**, **CONSTRAIN** and **FUNCTIONALLY-INSTANTIATE** takes a different form, described on page 188.

12.39. LEMMA

General Forms:
(LEMMA name rule-classes term)
and
(LEMMA name rule-classes term hints)

Example Form:
```
(LEMMA LEN-APP (REWRITE)
  (EQUAL (LEN (APP A B)) (PLUS (LEN A) (LEN B)))
  ((ENABLE LEN APP)))
```

Some users prefer to operate in a mode in which all rules are disabled and a name must be explicitly enabled locally to be used in a proof.[20] This mode has the advantage of making the theorem prover respond quickly to most challenges, because there are usually so few options available to it. It also allows one to build up very large sets of rules without so much concern about how they act in concert. The disadvantage is that the user must ‘‘sketch’’ each proof by listing the rules involved. Theories are important in this connection because they allow groups of ‘‘harmonious’’ rules to be conveniently enabled in unison. Users often develop several theories corresponding to different situations that commonly arise in their project and then attack a given problem with the theories felt most appropriate.

The **LEMMA** command makes operation in this mode convenient. The form **(LEMMA name rule-classes term)** is just an abbreviation for

```
(PROVE-LEMMA name rule-classes term
             ((ENABLE-THEORY GROUND-ZERO)
              (DISABLE-THEORY T))).
```

That is, when used without a **hints** argument **LEMMA** is like **PROVE-LEMMA** except that all non-primitive rule names are disabled. Little of interest can be proved by **LEMMA** when no hints are given. When a hint argument is supplied to **LEMMA**, a possibly modified version of it is passed to **PROVE-LEMMA**. See page 188 for a description of the **hints** argument to **PROVE-LEMMA**. Roughly speaking, the modified hint disables everything except the names specifically enabled by the user-supplied hint. Let **hints** be the user-supplied hint. We describe below the modified hint argument supplied to **PROVE-LEMMA**, **hints'**.

[20] This is sometimes called ‘‘Bevier-mode’’ after Bill Bevier who first used it extensively in [1].

- If **hints** contains no **DISABLE-THEORY** or **ENABLE-THEORY** hints, then **hints'** is obtained from **hints** by adding **(ENABLE-THEORY GROUND-ZERO)** and **(DISABLE-THEORY T)**.
- If **hints** contains any **DISABLE-THEORY** hint or an **(ENABLE-THEORY T)** hint, then **hints'** is **hints**. That is, if the supplied hints disable any theories or enable the universal theory, then **LEMMA** is just **PROVE-LEMMA**.
- Otherwise, **hints** contains an **ENABLE-THEORY** hint (but not **(ENABLE-THEORY T)**) and does not contain a **DISABLE-THEORY** hint. Then **hints'** is obtained from **hints** by modifying the **ENABLE-THEORY** entry to include **GROUND-ZERO** (unless it is already there) and adding the hint **(DISABLE-THEORY T)**.

Note that it is possible to operate in both modes in the same session. Most users try to maintain a coherent and effective global data base of rules, using the global theory commands to keep the ‘‘right’’ set of rules enabled for each phase of the project. But when more control is needed, **LEMMA** can be used.

By studying the modified hints **LEMMA** passes to **PROVE-LEMMA** (e.g., by reading the description above or by using **PPE** to print the **PROVE-LEMMA** event generated by a sample **LEMMA** command), the user can see how to achieve the same effect in the less commonly used commands **CONSTRAIN** and **FUNCTIONALLY-INSTANTIATE**.

12.40. MAINTAIN-REWRITE-PATH

General Form:
(MAINTAIN-REWRITE-PATH flg)

Example Form:
(MAINTAIN-REWRITE-PATH T)

This routine turns on and off the maintenance of the rewrite path (see page 131 for a discussion of rewrite paths and see the entries for **ACCUMULATED-PERSISTENCE**, **BREAK-LEMMA**, and **BREAK-REWRITE** for related features). The argument, **flg**, is evaluated and should be either **T** or **NIL**. If **T**, the rewrite path is henceforth maintained. If **NIL**, the rewrite path is not maintained. Maintenance of the stack involves storing into a data structure certain information about every call of the rewriter and degrades the performance to about 60% of its normal speed in proofs involving heavy use of replacement rules. Nevertheless, in deep proof developments we frequently maintain the rewrite path because of the information it provides us.

BREAK-LEMMA calls **MAINTAIN-REWRITE-PATH** the first time a rule is monitored since the path must be maintained in case an interactive break occurs. **UNBREAK-LEMMA** does *not* disable path maintenance when the last rule is removed from the list of monitored lemmas. It merely prints a message that path maintenance is still enabled. The reason is that you may still be planning to use **BREAK-REWRITE** or **ACCUMULATED-PERSISTENCE**. If not, you can regain the normal efficiency by turning off the maintenance of the rewrite path. If path maintenance is disabled by the user while monitors are still installed on some lemma names, then a break will occur when those lemmas are activated, but path information will not be available.

The rewrite path is not the Lisp control stack but a separate data structure. It persists even after an aborted computation. Therefore, if you are maintaining the rewrite path and abort some proof attempt (say, out of frustration over the time it is taking or because a stack overflow occurs) you can still inspect the path as it stood at the time of the abort. See **BREAK-REWRITE**.

Warning: It is dangerous to interrupt the theorem prover in the middle of a proof and call **MAINTAIN-REWRITE-PATH**. This can lead to hard errors later in the proof attempt because, when the path is being maintained assumptions are made about its structure and these assumptions are false if the path has not been maintained from the initial entry into the rewriter. If the theorem prover is running silently for long periods and you wish to poke around with **BREAK-REWRITE** to see what is going on, and the rewrite path is not enabled, you are advised to abort the proof, enable path maintenance, and then start the proof from scratch.

12.41. MAKE-LIB

General Forms:
(MAKE-LIB file)
and
(MAKE-LIB file compile-flg)

Example Form:
(MAKE-LIB "demo")

MAKE-LIB is the command-level routine for saving the data base so that it can be restored by **NOTE-LIB**. The arguments are evaluated. The first argument, **file**, should be a valid file root name (see page 163). The second argument, if supplied, should be **T** or **NIL**; if not supplied, it defaults to **NIL**. The data base is saved to a pair of files, namely the **lib** and **lisp** extensions of **file** (see page 163).

Into the **lisp** extension of **file** **MAKE-LIB** writes the Lisp source code for the executable counterparts of all functions in the current data base. Into the **lib** extension it writes everything else in the data base. **MAKE-LIB** does not alter the data base. When the two files have been written, they are closed. If **compile-flg** is **T**, the Lisp source code for the executable counterparts is then compiled and the compiled code is then loaded. This will create a **bin** file extension corresponding to the just-created **lisp** file extension of **file**. **MAKE-LIB** returns a list containing the **lib** and **lisp** file names created.

12.42. META

META is the first element in one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. The general form of the token is **(META** $\mathbf{fn_1}$ **...** $\mathbf{fn_n}$**)**. **META** rules are used in the simplification process, which is the first process tried.

12.43. Names—Events, Functions, and Variables

In the formal treatment of the logic in Chapter 4 we defined the variable and function symbols of our logic as follows:

Terminology. A sequence of characters, s, is a *symbol* if and only if (a) s is nonempty, (b) 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, (c) 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}.
```

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

However, in the extended syntax we introduced the notion of ‘‘well-formed’’ s-expressions and their ‘‘translations’’ to formal terms. We then adopted the policy that all terms displayed thereafter (and used in the implementation) would in fact be well-formed s-expressions standing for their translations.

The result of this is that there is a discrepancy between the set of legal names in the formal syntax and the set of names you can actually use in formulas. For example, **QUOTE** is a legal function name, but there is no way in the implementation to write down an application of the function symbol **QUOTE**, and an error is caused if you try to define **QUOTE**.

We therefore give here the definitions of the *legal names* as that concept is practically defined by the extended syntax.

The variable names permitted in the implementation are the symbols, excluding **T**, **F**, and **NIL** as above.

The function names and event names permitted in the implementation are the symbols, as defined above, excluding the following:

- **CASE**, **COND**, **F**, **LET**, **LIST**, **LIST***, **NIL**, **QUOTE**, and **T**, and
- any symbol, such as **CADDR**, which starts with a **C**, ends with an **R** and contains only **A**s and **D**s in between.

A function or event name is *old* (or *new*) iff it is (or is not) a citizen of the data base (see page 143).

12.44. NOTE-LIB

General Forms:
(NOTE-LIB file)
and
(NOTE-LIB file compile-flg)

Example Form:
(NOTE-LIB "demo")

NOTE-LIB is the routine for restoring a data base saved by **MAKE-LIB**. The arguments are evaluated. The first argument, **file**, should be a valid file root name. The second argument, **compile-flg**, should be **T** or **NIL**; it defaults to **NIL** if not supplied. The **lib** and **lisp** extensions of **file** must exist and should have been created by **MAKE-LIB** (see the discussion of File Names, page 163). If **compile-flg** is **T**, the **bin** extension of **file** must exist and should have been created by the same call of **MAKE-LIB** (with **compile-flg T**) that created the **lisp** extension of **file**.

NOTE-LIB destroys the data base extant when it is executed and configures the data base to be that which was extant when the corresponding **MAKE-LIB** for **file** was executed. If **compile-flg** is **T**, **NOTE-LIB** loads the compiled code for the executable counterparts of the functions in the incoming data base.

Because Nqthm changes (slowly) over time, it is sometimes the case that a library file created by one release of Nqthm is in a different format than those created by another release. This is called a *library incompatibility*. When it occurs, i.e., when **`NOTE-LIB`** detects that the incoming library is in the wrong format, a message to that effect is printed. However, **`NOTE-LIB`** attempts to load the library anyway. While you should not trust the resulting data base, it can often be used to recover a list of events (e.g., **`EVENTS-SINCE`**) from which a new library can be constructed from scratch.

`NOTE-LIB` returns a list containing the names of the two files read.

Uncompiled Nqthm library files are portable from one host to another. That is, the brand of Common Lisp in use during **`MAKE-LIB`** is irrelevant to **`NOTE-LIB`**. A similar remark holds about the host processor. Thus, for example, library files created a Sparc workstation can be noted by a user running on a Mac II. Of course, the compiled file created when **`MAKE-LIB`** is given a second argument of **`T`** is generally not portable between Common Lisps or host processors. Thus, when noting a file from another system, do not give **`NOTE-LIB`** a second argument of **`T`**.

12.45. NQTHM Mode

`NQTHM` mode refers to the set of axioms added to the data base by **`BOOT-STRAP`**. See the discussion of **`BOOT-STRAP`** on page 127.

12.46. PPE

General Form:
`(PPE name)`

Example Form:
`(PPE 'REVERSE)`

`PPE` stands for ‘‘PrettyPrint Event.’’ The argument, **`name`**, is evaluated and should be a citizen of the data base. The command prints out the **`EVENT-FORM`** of **`name`**.

12.47. PROVE

General Form:
(PROVE term)

Example Form:
(PROVE '(PROPERP (REVERSE X)))

PROVE *is* the theorem prover. The argument, **term**, is evaluated by Lisp and should be a term. **PROVE** attempts to prove **term**; it returns **PROVED** if the proof succeeds and **NIL** if the proof fails. The output is written to the terminal. There is no provision for hints. **PROVE** does not alter the data base. To silence **PROVE** so that no output is produced, execute **(SETQ IO-FN #'(LAMBDA () NIL))**. That is, bind or set the Common Lisp variable **IO-FN** to a no-op function of no arguments. To restore normal printing, restore **IO-FN** to its initial value.

12.48. PROVEALL

General Forms:
(PROVEALL file events)
and
(PROVEALL file events compile-flg)

Example Form:
```
(PROVEALL "demo"
  '((DEFN REV1 (X A)
      (IF (NLISTP X)
          A
          (REV1 (CDR X)
                (CONS (CAR X) A))))
    (PROVE-LEMMA REV1-IS-REV (REWRITE)
                 (EQUAL (REV1 X A)
                        (APPEND (REVERSE X) A)))))
```

Note. At the time the first edition of this handbook was written, **PROVEALL** was the standard way to execute a sequence of event commands and collect the output into a file. It suffers however from the problem that its second argument, **events**, is a list of events and those events have to be created somehow, usually by reading them from a file. That creation process opens the possibility that Nqthm will somehow be corrupted. It also leaves open the possibility that the syntax of the events as displayed in the permanent record, i.e., the file from which they were read, is not strictly in Nqthm's extended syntax. For example, the source file might employ user-defined or implementation-dependent Common Lisp read macros. From such a permanent record it is often impossible to reconstruct what was actually proved. For the second edition, therefore, we introduced **PROVE-FILE** (see page 180) which is similar to **PROVEALL** but which also takes responsibility for reading the events from a specified file, enforces the extended syntax, and also enforces restrictions intended to ensure that the resulting data base is ''endorsed.'' We now prefer to use **PROVE-FILE** instead of **PROVEALL**. We have left **PROVEALL** in the system for the sake of compatibility.

The arguments to **PROVEALL** are evaluated. The first argument, **file**, should be a file root name (see page 163). Five or six extensions of **file** will be created, namely the **proofs**, **err**, **lib**, **lisp**, and, possibly, the **bin** and **fail** extensions. The second argument, **events**, should be a list of event forms to be executed. The third argument, **compile-flg**, should be **T** or **NIL** if supplied; if not supplied, it defaults to **NIL**.

PROVEALL first opens the **proofs** and **err** extensions of **file** and arranges for the event commands and **DO-EVENTS** to print all their ''normal'' output to the **proofs** extension and all error messages to the **err** extension. **PROVEALL** then prints a header into the **proofs** extension, giving the current time and date.

PROVEALL next uses **DO-EVENTS** to execute the event forms in **events**. **DO-EVENTS** executes each event form in turn until either they have all been executed or some form either causes an **ERROR** or **FATAL ERROR** or some **PROVE-LEMMA** fails. **DO-EVENTS** creates a complete transcript of the session in the **proofs** extension of **file**. In particular, the **proofs** extension will contain each event form executed, any **WARNING** or error messages caused, the justifications of each definition, the proof attempt for each **PROVE-LEMMA**, the time triple for each event, and the value of each event. The **err** extension will contain every **WARNING**, **ERROR**, and **FATAL ERROR** message printed in the **proofs** extension. Because the normal output of the event commands is being written to files, **DO-EVENTS** indicates its progress through **events** by printing just the event names to the terminal, each name being printed upon the commencement of the execution of the corresponding event.

Upon termination of **DO-EVENTS**, **PROVEALL** prints a ''system description'' into the **proofs** extension. This description gives the names of the files that were loaded to produce the version of the theorem prover used in the proveall. The format used differs from one host system to the next. **PROVEALL** also prints to the **proofs** extension the total amount of time consumed by the run, by summing (componentwise) the time triples printed.

PROVEALL then uses **MAKE-LIB** to save the data base, passing it the

compile-flg argument. This creates both the **lib** and **lisp** extensions of **file**, and the **bin** extension if **compile-flg** is **T**. Note that if **DO-EVENTS** terminates prematurely because of an error or unsuccessful proof, the data base saved will be the one in which the terminal event was executed. The output from that event will be in the **proofs** extension. The failure can be reproduced by using **NOTE-LIB** to restore the data base created by **PROVEALL** and then executing the terminal event.

Finally, if **DO-EVENTS** terminated prematurely, **PROVEALL** creates the **fail** extension of **file** and writes into it (and into the **err** extension) an **ERROR** message containing the event form that caused the failure.

12.49. PROVE-FILE

General Forms:

```
(PROVE-FILE file)
```

and

```
(PROVE-FILE file :CHECK-SYNTAX      cs-flg
                 :WRITE-FLAG-FILES  wf-flg
                 :PETITIO-PRINCIPII pp-flg)
```

Example Forms:

```
(PROVE-FILE "basic")
```

and

```
(PROVE-FILE "basic"
            :PETITIO-PRINCIPII T
            :WRITE-FLAG-FILES NIL)
```

PROVE-FILE is the standard way to create an ‘‘endorsed’’ data base from a file purporting to be a sequence of Nqthm event commands.[21] We define what we mean by an ‘‘endorsed’’ data base below, but the key property of such a data base is that we believe the formulas labeled ‘‘theorems’’ in it are actually theorems.

PROVE-FILE evaluates its arguments. The first argument, **file**, should be a file root name (see page 163), **file** must not end with the substring **"tmp"**, and **file.events**, the **events** extension of **file**, should exist and contain Nqthm event commands as described below. In accordance with Common Lisp,

---

[21] We now prefer **PROVE-FILE** to the older **PROVEALL**, for the reasons given in the discussion of **PROVEALL**, page 178.

the optional ‘‘keyword arguments’’ may be supplied in any order or be omitted. When omitted, the **:CHECK-SYNTAX** and **:WRITE-FLAG-FILES** arguments default to **T**; **PETITIO-PRINCIPII** defaults to **NIL**.

PROVE-FILE is a read-eval-print loop similar to Common Lisp’s **LOAD** but enforcing certain requirements on the forms read and evaluated. **PROVE-FILE** repeatedly reads a form from **file.events**, prints the form, checks the syntax of the form, executes the form and prints its value. If all the forms in **file.events** are successfully processed, **PROVE-FILE** returns **T**. If an unacceptable form is read, **PROVE-FILE** calls **(MAKE-LIB file)** to save the data base and returns **NIL**.

By making the last form in **file.events** a **MAKE-LIB** form the user can arrange for successful termination to produce a library.

All output produced during **PROVE-FILE** is directed to Common Lisp’s ***STANDARD-OUTPUT***, which is normally the terminal. Because **PROVE-FILE** prints the forms and their values, the text written to ***STANDARD-OUTPUT*** resembles an interactive session. The user wishing to direct the output to a file should see **PROVE-FILE-OUT**.

PROVE-FILE observes a certain protocol of creating and deleting ‘‘flag files’’ to signal success or failure to a user who might be ‘‘watching’’ a **PROVE-FILE** in the background of some other activity. This protocol is explained below.

Each form in the **file.events** must be one of those described below. In addition, the first form must be a **BOOT-STRAP** or **NOTE-LIB** and only the last form is permitted to be a **MAKE-LIB**. A form, **form**, may occur in **file.events** only if:

- **form** is **(BOOT-STRAP)**, **(BOOT-STRAP NQTHM)** or **(BOOT-STRAP THM)**;
- **form** is **(NOTE-LIB str)** or **(NOTE-LIB str flg)**, where **str** is a string and **flg** is either **T** or **NIL**;
- **form** is a call of **ADD-AXIOM**, **ADD-SHELL**, **AXIOM**, **COMMENT**, **CONSTRAIN**, **DCL**, **DEFN**, **DEFTHEORY**, **DISABLE**, **DISABLE-THEORY**, **ENABLE**, **ENABLE-THEORY**, **FUNCTIONALLY-INSTANTIATE**, **LEMMA**, **PROVE-LEMMA**, **SET-STATUS**, **TOGGLE**, **TOGGLE-DEFINED-FUNCTIONS** or **UBT**;
- **form** is **(COMPILE-UNCOMPILED-DEFNS "tmp")**[22];

---

[22]**PROVE-FILE** may not be used with files that end in **"tmp"** so as to avoid the possibility of a collision of the library file created for **file**, namely **file.lisp**, and the temporary file created by **COMPILE-UNCOMPILE-DEFNS**, **tmp.lisp**. Such a collision is possible, even when **file** is not **"tmp"** since **file** might contain directory prefixes.

- **form** is **(SETQ REDUCE-TERM-CLOCK n)**, where **n** is an integer;
- **form** is **(SETQ *COMPILE-FUNCTIONS-FLG* flg)**, where **flg** is either **T** or **NIL**; or
- **form** is **(MAKE-LIB str)** or **(MAKE-LIB str flg)**, where **str** is a string and is equal to the first argument of **PROVE-FILE**, **file**, and **flg** is either **T** or **NIL**.

All of the **event** files in our collection of Nqthm benchmarks in the **examples** directory (page 226) are processed successfully by **PROVE-FILE** and therefore illustrate the allowed format.

The readtable used while reading forms from **file.events** supports only the Nqthm extended syntax (page 70). Thus, some legal Common Lisp forms, such as **#3r102** (the integer eleven written in base three) and **'#,(lisp-fn)** (a **QUOTE**d constant created by executing the Common Lisp program **lisp-fn**), are disallowed in **file.events**; errors will be signalled when such forms are read.

Caution. Because **PROVE-FILE** binds ***READTABLE*** to implement the Nqthm syntax, you may find it difficult to get out of an interactive break under **PROVE-FILE**. This is because our readtable prevents the use of keywords such as **:q** or **:a**, which are often chosen by implementors as the escape commands for interactive breaks. We try to restore the readtable appropriately before every error break, but circumstances beyond our control sometime prevent this. If an error occurs during **PROVE-FILE** and you are left in an interactive break and have difficulty (e.g., errors arise when you type a colon), we recommend that you execute **(SETQ *READTABLE* (COPY-READTABLE NIL))**, which will restore the host Lisp's readtable and allow you to type normally thereafter. The reason we prevent colons from occuring in Nqthm syntax is that the symbol **SI::CAR**, for example, might be **CAR** in some Lisps and not be **CAR** in others, depending on what symbols were imported into the **SI** package.

If the **:CHECK-SYNTAX** keyword argument is supplied and **NIL**, then we do not use the Nqthm readtable or enforce restrictions on the forms in the **file.events**.

A ''flag file'' protocol is observed by **PROVE-FILE** if the keyword argument **:WRITE-FLAG-FILES** is non-**NIL**, as it is by default. The protocol uses three extensions of **file**, **started**, **fail**, and **proved**. When called, **PROVE-FILE** creates **file.started** and deletes both **file.proved** and **file.fail** (when they exist). An error is caused if **file.proved** exists and cannot be deleted. Upon successful termination, **PROVE-FILE** creates **file.proved** and deletes **file.started**. Upon unsuccessful termination, **PROVE-FILE** creates **file.fail**.

Thus, barring interference from other users, a sure way to check for successful termination of a background execution of **PROVE-FILE** is to confirm that **file.proved** exists and was created after the **PROVE-FILE** commenced (since it is possible that the attempt to delete the file failed and left an error message on the terminal of the processor on which it was running). The existence of a **file.fail** created after the **PROVE-FILE** commenced is a sure way to confirm the failure of **PROVE-FILE**. But failures may manifest themselves by nontermination or resource exhaustion that are not detectable with the flag file protocol.

When **file.proved** is created, certain interesting statistics are written into it. These include the total time used in processing **file.events**, a note on the incompatibility of the library files noted if such an incompatibility has occurred in the session (see **NOTE-LIB**), the names of all nondefinitional axioms involved (including those in library files noted), and the names of all events targeted by the undo-back-through command **UBT** during the run.

The **:PETITIO-PRINCIPII** argument, when non-**NIL**, makes **PROVE-FILE** skip all proof obligations. Thus, with this argument set, the forms in **file.events** are merely *assumed* to be admissible (though their syntax is checked). This is a fast way to almost reconstruct the data base created by a sequence of events. The reconstruction is not perfect because the logical dependencies discovered during proofs are not recorded. It is a handy way to experiment or to recover from system crashes. However,

Warning. Nqthm is unsound when the **:PETITIO-PRINCIPII** argument to **PROVE-FILE** is used. No record of the use of **:PETITIO-PRINCIPII** is left in the data base or in libraries created by the user from the data base constructed by **PROVE-FILE**.

When **PETITIO-PRINCIPII** is non-**NIL**, **PROVE-FILE** does not observe the flag file protocol, and hence, does not create **file.proved** even if the run is successful. Furthermore, no **MAKE-LIB** is done by such a **PROVE-FILE** and no output occurs. The routine **SKIM-FILE** is a convenient way to call **PROVE-FILE** with **PETITIO-PRINCIPII** set.

We conclude with some motivational remarks about **PROVE-FILE** and the process of creating endorsed data bases.

PROVE-FILE is a weak, file-based read-eval-print loop. Why not stick with the all-powerful Common Lisp read-eval-print loop and its file-based counterpart, **load**?

During the formalization and proof development phase of an Nqthm project, we prefer the flexibility and sense of freedom one gets from dealing directly with Lisp, rather than with some pea-brained and constraining ‘‘user interface.’’ For example, we often find ourselves executing raw Common Lisp forms to ‘‘compute’’ the Nqthm forms we wish to evaluate. We enjoy being able to type arbitrary Common Lisp forms, define Lisp utilities and readmacros, and otherwise tailor the environment to our particular tastes. We thus recommend against the casual use of **PROVE-FILE** in the development phase of a project.

But the Common Lisp interface is ''too powerful'' when it comes to soundness. Executing **(SETQ TRUE FALSE)** in the host Lisp will render Nqthm unsound. More subtle forms of unsoundness can creep in, e.g., by the accidental corruption of Nqthm's data base caused by destructive list processing operations performed by utility routines written by the user. As noted on page 214, one of Nqthm's own utilities, **UNDO-NAME**, can render the system unsound. Therefore, we do not endorse as a theorem anything ''proved'' by Nqthm in a normal interactive session. When a proving project is entirely complete, we prefer to know that no idiotic, surreptitious, or untrusted Lisp code is present that would render our efforts meaningless. Thus it makes sense to have, as we do in **PROVE-FILE**, a loader that prohibits the intermingling of possibly dangerous Lisp forms with trusted Nqthm forms.

When we have reached the end of a project we have often constructed a single **events** file, say **project.events**, that starts with a **BOOT-STRAP**. If we wish to preserve the final data base of the project, as we would if other projects build upon it, we insert a **(MAKE-LIB "project")** at the end of the file. We then undertake to ''certify'' the event list. We start by checking that there is no file named **project.proved** on the directory. Then we invoke **(PROVE-FILE "project")**. If this terminates with the creation of **project.proved** we inspect its contents to see that no inadvertent axioms were added.[23] We also look for **UBT**s in the script because the presence of undoing in an event list can lead to such confusing results as a single name being defined two different ways (at different times).

Because projects that use Nqthm can be very large, taking years to complete, and can be worked on by many people, there is often the practical necessity to divide the events into a collection of several files which are chained together by **NOTE-LIB**s and **MAKE-LIB**s. To 'certify root and branch' a collection of such files using **PROVE-FILE**, we start by making sure that there are no **lib**, **lisp**, **bin**, **proved**, or **started** file extensions on the directories in question, including those referred to in the **NOTE-LIB** forms that occur at the beginnings of the files to be certified. (Alternatively, we check that every file ''in sight'' has extension **events**.) This check will prevent the use of older libraries and thus avoid the risk of using libraries that are uncertified or otherwise out of date with respect to their source files. Then we execute a sequence of **PROVE-FILE**

[23] One can develop an Nqthm proof in a top-down fashion by using **ADD-AXIOM** to state the lemmas one intends later to prove. One then proceeds to derive the main result from these unproven ''lemmas.'' Upon completing the main proof, one is then obligated to replace each of the **ADD-AXIOM**s with a suitable event list concluding in a **PROVE-LEMMA**. Occasionally an **ADD-AXIOM** is forgotten or overlooked and finds its way into what the user thought was the completed script.

commands on the event files in question in an order consistent with the inner dependencies (calling **PROVE-FILE** on each file exactly once). If at the end there is a **proved** file for each of the **event** files, the certification is complete. Obviously, there should be no changes permitted to any of the files in sight (except, of course, the creation of new files by **PROVE-FILE**) during the execution of the **PROVE-FILE**s.

The collection of example event files distributed with Nqthm illustrates this discipline. We recommend these examples to the reader. To maintain this collection we have mechanized the certification procedure with a Lisp script, distributed as **driver.lisp** in the **examples** subdirectory.

Technically speaking, an *endorsed data base* is a data base created, in a fresh Nqthm, by a successful execution of **PROVE-FILE**, provided (a) the first event in the **events** file is either a **BOOT-STRAP** or a **NOTE-LIB** of an endorsed data base, and (b) no ‘‘library incompatibilities’’ were noted (see **NOTE-LIB**). Modulo the difficulties discussed below, we believe the ‘‘theorems’’ in an endorsed data base are indeed theorems. In particular, consider any **(PROVE-LEMMA name rule-classes term hints)** in the endorsed data base. Let h be the sequence of axiomatic acts in the data base prior to the event in question. Then we claim h is a history (i.e., the foregoing axiomatic acts are all admissible) and **term** can be proved directly from the axioms of some definitional/constrain extension of h.

Inadequacies. We currently know of three senses in which **PROVE-FILE** fails to do fully the job that we intend. (1) Common Lisp syntax for integers is not entirely fixed by the Common Lisp standards. For example, **1.7J** is given as an explicit example, on p. 341 of [53], of a character sequence that is a ‘‘potential number,’’ that is, a sequence that some Common Lisp might read as a number. If a Common Lisp were to read **1.7J** as an integer, e.g., as the integer 17, then by running Nqthm in that Common Lisp one could prove the theorem **(EQUAL 1.7J 17)**. But **(EQUAL 1.7J 17)** might be false, or might even cause an error in Nqthm in another Common Lisp. The solution to this problem is to check with your Lisp provider that no character strings that are ‘‘potential numbers’’ read as integers, except those satisfying the normal syntax for integers shared by all Common Lisp implementations. (2) It is not clear from the Common Lisp standard what files can be created by the compilation process. Typically, if you compile **project.lisp**, then **project.***x* is created, for some extension *x*. However, AKCL takes this freedom to a rather liberal extent: when the variable **COMPILER::*SPLIT-FILES*** is non-**NIL**, as we have been forced to set it when doing some proofs, files with names **0project.o**, **1project.o**, **2project.o**, ..., may be created during the compilation of **project.lisp**. Such names create a potential for collision with the compiled library of another events file, say one named **0project.events**. Thus the AKCL user of **PROVE-FILE** should check that the name of no events file begins with a digit. We do not make this check mechanically, as we do the check for names that end in **"tmp"**, because it is possible that some prefix of a name supplied to **PROVE-FILE** is a mere directory name, e.g., **2nd-try/project**. (3) Unbalanced close parentheses at the top level are treated by Common Lisp's **READ** in an undefined, implementation dependent way. Most Lisps we use ignore unbalanced close parentheses, but some cause an error. Such parentheses are ''invalid'' according to [53]. A perfect implementation of **PROVE-FILE**, we think, would check for such parentheses and cause an error.

12.50. PROVE-FILE-OUT

General Forms:

```
(PROVE-FILE-OUT file)
```

and

```
(PROVE-FILE-OUT file
                :CHECK-SYNTAX      cs-flg
                :WRITE-FLAG-FILES  wf-flg
                :PETITIO-PRINCIPII pp-flg)
```

Example Forms:

```
(PROVE-FILE-OUT "basic")
```

and

```
(PROVE-FILE-OUT "basic"
                :PETITIO-PRINCIPII T
                :WRITE-FLAG-FILES NIL)
```

PROVE-FILE-OUT is like **PROVE-FILE** except it arranges for all output to go to the **proofs** extension of **file**. That is, **PROVE-FILE-OUT** opens the file **file.proofs** and calls **PROVE-FILE** (on the same arguments given to **PROVE-FILE-OUT**) in such a way as to ensure that all output is written to **file.proofs**. The output file is closed upon termination and represents a complete transcript of the processing of **file.events**. Nothing is printed to the terminal. See **PROVE-FILE**.

12.51. PROVE-LEMMA

General Forms:

```
(PROVE-LEMMA name rule-classes term)
```

and

```
(PROVE-LEMMA name rule-classes term hints)
```

Example Form:

```
(PROVE-LEMMA REVERSE-REVERSE (REWRITE)
   (IMPLIES (PROPER X)
            (EQUAL (REVERSE (REVERSE X)) X)))
```

and

```
(PROVE-LEMMA TIMES-LIST-EQUAL-FACT (REWRITE)
   (IMPLIES (PERM (POSITIVES N) L)
            (EQUAL (TIMES-LIST L) (FACT N)))

   ;Hints:
   ((USE (PERM-TIMES-LIST (L1 (POSITIVES N))
                          (L2 L)))
    (DISABLE PERM-TIMES-LIST)))
```

PROVE-LEMMA is an event command for proving a theorem and storing it and possibly some generated rules in the data base. It does not evaluate its arguments. **name** must be a new name and will be made the name of the event in the data base, **rule-classes** must be a (possibly empty) list of rule classes and **term** must be a well-formed term (see **TRANSLATE**). **PROVE-LEMMA** applies the theorem prover to **term**, printing commentary as the proof attempt proceeds. If the proof attempt is successful, **term** is stored in the data base as a theorem; in addition, rules of the classes specified by **rule-classes** are generated from **term** and stored. The name of each rule is **name**. An error is caused if **term** is unsuitable for some class in **rule-classes** and no change is made in the data base.

When formulating input for **PROVE-LEMMA** you must be cognizant of the rule interpretation of **term** in addition to its mathematical content. See Chapter 11 for a detailed explanation of the effect of each type of rule and how rules are generated from formulas and rule classes. See Chapter 13 for advice about how to use rules.

Note that if **rule-classes** is **NIL** then the term is stored as a theorem and will be available for **USE** hints (see below) but will generate no rules.

If the proof attempt succeeds, **PROVE-LEMMA** prints ‘‘**Q.E.D.**’’ and a time triple for the event and returns the name of the new event, **name**.

If the proof attempt fails, **PROVE-LEMMA** prints a ‘‘failure banner’’

```
************** F A I L E D **************
```

and then a time triple. Then it returns **NIL**.

When the theorem prover fails the only claim made is the trivial one that our heuristics did not lead to a proof. The system’s remarks should not be interpreted to mean that the input formula is not a theorem, even if the last formula printed is **F**: the heuristics might have generalized the input formula before failing. That said, however, two caveats are in order.

First, conjectures submitted for proof are frequently not theorems. Do not be too quick to dismiss a failed proof simply because the theorem prover is so weak. It is indeed weak, but you can’t fool it either. If a point were awarded the opposition each time a player asserted an inconsistent belief, the theorem prover would never lose and would, in fact, win almost every session.

Second, when it fails on *theorems* it is frequently because the proof you had in mind involves some step that has not been made explicit by your previously proved lemmas—or at least by their interpretation as rules. See Chapter 13 where we discussed the notion of the crucial ‘‘check points’’ in a proof attempt and what you should do at each. If worse comes to worse, you can essentially tell the system your proof via the **hint** argument to **PROVE-LEMMA**, which we describe at length below.

The **hints** argument permits the user to give the theorem prover some guidance in its proof attempt. If nonempty, **hints** must be a list of tuples. The **CAR** of each tuple must be one of the symbols listed below, where we describe the form and effect of each acceptable tuple. An error is caused if any element of **hints** fails to have the correct form or if there is more than one occurrence of any kind of hint.

12.51.1. USE Hints

General Form:

$$\texttt{(USE (name}_1\ \texttt{(v}_{1,1}\ \texttt{t}_{1,1}\texttt{)}\ \ldots\ \texttt{(v}_{1,n_1}\ \texttt{t}_{1,n_1}\texttt{))}$$
$$\ldots$$
$$\texttt{(name}_k\ \texttt{(v}_{k,1}\ \texttt{t}_{k,1}\texttt{)}\ \ldots\ \texttt{(v}_{k,n_k}\ \texttt{t}_{k,n_k}\texttt{)))}$$

Example Form:

```
(USE (LEMMA1 (X (CAR A)) (Y 0))
     (LEMMA2 (X (SUB1 X)) (I X) (K (FACT X))))
```

A **USE** hint essentially forces the use of one or more instances of one or more previously proved theorems, definitions, or axioms. Each pair following the symbol **USE** specifies the name of a formula and a substitution. The **USE** hint creates the indicated instantiation of each formula and adds it as an explicit hypothesis to the conjecture, **term**, submitted to the theorem prover.

More precisely, the conditions on **USE** hints are as follows. Each $\mathbf{name_i}$ must be a citizen of the data base with a non-**NIL** associated **FORMULA** in **DATA-BASE**. It does not matter if $\mathbf{name_i}$ is enabled or disabled or if any rules were generated for $\mathbf{name_i}$. Each $\mathbf{v_{i,j}}$ must be a variable symbol, each $\mathbf{t_{i,j}}$ must be a term, and $\mathbf{v_{i,j}}$ must be distinct from $\mathbf{v_{i,k}}$ if $\mathbf{j \neq k}$.

Provided the above conditions are met, the effect of such a **USE** hint on the behavior of **PROVE-LEMMA** is to cause a modified (but provably equivalent) conjecture to be submitted to the theorem prover. Rather than submitting **term**, **PROVE-LEMMA** submits

$$\mathtt{(IMPLIES\ (AND\ thm_1\ \ldots\ thm_k)\ term)}$$

where $\mathbf{thm_i}$ is obtained by instantiating the **FORMULA** for $\mathbf{name_i}$, replacing $\mathbf{v_{i,j}}$ by $\mathbf{t_{i,j}}$.

Observe that each $\mathbf{thm_i}$ is provably non-**F**—each is an instance of a theorem, definition, or axiom. Hence, the submitted formula is equivalent to the input one.

However, the careful reader might also note that if $\mathbf{name_i}$ is enabled and is a well-constructed replacement rule, there is a very high likelihood that it will be used to eliminate $\mathbf{thm_i}$ from the modified conjecture. This renders the hint useless. It is generally a good idea to **DISABLE** any **REWRITE** rules instantiated with a **USE** hint.

The modified conjecture is often easier to prove because the necessary instances of the necessary theorems are explicitly present. Indeed, it is possible to reduce any noninductive proof in the logic to propositional calculus and equality reasoning after supplying all the necessary instances of lemmas.[24]

12.51.2. EXPAND Hints

General Form:
$\mathtt{(EXPAND\ t_1\ \ldots\ t_n)}$

Example Form:
```
(EXPAND (PRIME1 X (SUB1 X)))
```

---

[24] We do not recommend such a wholesale use of **USE** simply because the number of lemmas and instances needed would often produce a modified conjecture that was too large for the combinatorial processing done by all known propositional decision engines.

Each $\mathbf{t_i}$ must be a term of the form **(fn $\mathbf{x_1}$ ... $\mathbf{x_k}$)** where **fn** is a defined function. That is, each $\mathbf{t_i}$ must be a call of a defined function. Variable symbols, calls of shell functions, and calls of undefined or constrained functions are prohibited. The **EXPAND** hint forces the expansion of each $\mathbf{t_i}$ whenever it is encountered by the rewriter.

The **EXPAND** hint is implemented simply by putting the (translated and with certain calls of **IDENTITY** expanded, as described below) $\mathbf{t_i}$ on a list that is known to the rewriter. Whenever a function call is considered for expansion the list is checked. Thus, it is not necessary that the $\mathbf{t_i}$ occur in the input conjecture. However, it is crucial that the *exact* form of the intended term be written in the hint. We give an example below.

Suppose **(SECOND X)** is defined to be **(CADR X)**. Suppose also that **(TIMES I (SECOND X))** occurs in the input conjecture and the intended proof requires its expansion but the system does not expand it automatically. An **EXPAND** hint is called for. The hint **(EXPAND (TIMES I (SECOND X)))** will *not* work. By the time the rewriter considers expanding the **TIMES** expression, it will have become **(TIMES I (CADR X))**. Thus, to force the expansion of **(TIMES I (SECOND X))** it is necessary to give the hint **(EXPAND (TIMES I (CADR X)))**! This is not generally a problem. The reason is that you will generally add **EXPAND** hints only in response to the failure of an earlier proof attempt. In that earlier attempt the term **(TIMES I (CADR X))** will have appeared at the checkpoint where you expected its expansion to appear and you will have realized ‘‘the theorem prover needs to expand *that* term.’’

As with the left-hand sides of **REWRITE** rules, it is sometimes desired to write large constants in the terms to be expanded. An alternative is to write a variable-free expression that reduces to the desired constant and to embed that expression in a call to the **IDENTITY** function. The **EXPAND** hint replaces such expressions by their reduced values before storing them on the list inspected by the rewriter. Thus, when used in an **EXPAND** hint, **(SUM A B 4294967296)** and **(SUM A B (IDENTITY (EXPT 2 32)))** cause identical behavior, provided **EXPT** is defined as the exponentiation function. The **IDENTITY** function is given special treatment only here, in the **EXPAND** hint, and in the preprocessing of **REWRITE** rules. Otherwise, **IDENTITY** is not given any special treatment by the system.

12.51.3. DISABLE and ENABLE Hints

General Forms:
(DISABLE name$_1$ ... name$_n$)
and
(ENABLE name$_1$ ... name$_n$)

Example Form:
(DISABLE LEMMA1 TIMES-2-REWRITE)

The **DISABLE** hint has the effect of disabling the listed rule names for the duration of the proof attempt in question. The **ENABLE** hint is analogous.

We sometimes call these hints ‘‘local’’ enables and disables because they do not affect the status of the names in the data base. The effects of these hints are visible only during the proof attempt for which the hints were supplied.

Each **name$_i$** must be a citizen of the data base. See page 152 for a discussion of the effect of enabling or disabling a name and why it is done. Roughly speaking, the effect of **DISABLE** is to prevent the theorem prover from using any of the named rules during the current proof attempt. **ENABLE** permits the named rules to be used even though they might be **DISABLE**d in the current data base.

Using the **DISABLE-THEORY** and **ENABLE-THEORY** hints, all the names in a theory can be ‘‘locally’’ disabled or enabled. This raises questions of priority, e.g., ‘‘What if a name is **DISABLE**d locally with a **DISABLE** hint but is a member of a theory that is enabled locally?’’ Such questions are addressed in the following discussion of the **DISABLE-THEORY** and **ENABLE-THEORY** hints.

12.51.4. DISABLE-THEORY and ENABLE-THEORY Hints

General Forms:
(DISABLE-THEORY theory-name$_1$ ... theory-name$_n$)
and
(ENABLE-THEORY theory-name$_1$ ... theory-name$_n$)

Example Form:
(DISABLE-THEORY LIST-THEORY BAG-THEORY)

The **DISABLE-THEORY** hint has the effect of disabling all the names in the listed theories. The **ENABLE-THEORY** hint is analogous.

We sometimes call these hints ‘‘local’’ enables and disables because they do not affect the status of the names in the data base. The effects of these hints are visible only during the proof attempt for which the hints were supplied.

Each **theory-name$_i$** must be a theory name, i.e., a name introduced with **DEFTHEORY** or one of the two builtin theory names, **GROUND-ZERO** and **T**. See page 152 for a discussion of the effect of enabling or disabling a name and why it is done.

If the theory name **T** is used then no other theory name is permitted in the hint. Furthermore, if the hint **(DISABLE-THEORY T)** is supplied, then **(ENABLE-THEORY T)** may not be supplied and vice versa.

The local status of a name during a proof may thus be affected by any of four hints, **DISABLE**, **ENABLE**, **DISABLE-THEORY**, and **ENABLE-THEORY**. The following algorithm is used to determine the status of a name during a proof attempt governed by any of these hints.

- If the given name is included in an **ENABLE** hint, consider it enabled.
- Otherwise, if the given name is included in a **DISABLE** hint, consider it disabled.
- Otherwise, if the given name belongs to some theory named in an **ENABLE-THEORY** hint, consider it enabled.
- Otherwise, if the given name belongs to some theory named in an **DISABLE-THEORY** hint, consider it disabled.
- Otherwise, if there is an **(ENABLE-THEORY T)** hint (respectively, a **(DISABLE-THEORY T)** hint), consider it enabled (respectively, disabled).
- Otherwise, simply check whether it is enabled or disabled in the global database.

Thus, roughly speaking, the hints which explicitly mention rule names, **DISABLE** and **ENABLE**, are given priority over the theory-level hints, **DISABLE-THEORY** and **ENABLE-THEORY**; enabling is given priority over disabling. For example, one may give a hint that enables several relevant theories but then one may explicitly disable certain names within those theories.

12.51.5. HANDS-OFF Hints

General Form:
(HANDS-OFF $\mathbf{fn_1}$ **...** $\mathbf{fn_n}$**)**

Example Form:
(HANDS-OFF PLUS TIMES QUOTIENT REMAINDER)

The **HANDS-OFF** hint prevents the rewriter from trying any rule that rewrites a term beginning with one of the function symbols listed in the hint. Each $\mathbf{fn_i}$ must be a function symbol. Each such name is added to a list known to the rewriter. Roughly speaking, if a function symbol is on this list then every every **REWRITE** stored under that symbol, including its definition, is locally disabled. Because the documentation of the **HANDS-OFF** feature was grossly incorrect in the first edition of this handbook, we describe it somewhat more carefully.

When rewriting the target term **(fn** $\mathbf{a_1}$ **...** $\mathbf{a_k}$**)**, the rewriter first rewrites each $\mathbf{a_i}$, say to $\mathbf{a_i'}$. The rewriter then shifts its attention to **(fn** $\mathbf{a_1'}$ **...** $\mathbf{a_k'}$**)**, which we here call the ''intermediate term.'' If all the $\mathbf{a_i'}$ are explicit values and **fn** has an enabled executable counterpart, rewriter returns the explicit value obtained by reducing the intermediate term. If **fn** is **EQUAL** or a shell recognizer such as **NUMBERP**, built in rules are tried on the intermediate term. If the intermediate term is one of several special forms having to do with **V&C$**, certain built-in rules are tried. See the Reference Guide entries for **REWRITE-APPLY$**, **REWRITE-APPLY-SUBR**, **REWRITE-CAR-V&C$**, **REWRITE-CAR-V&C-APPLY$**, **REWRITE-EVAL$**, **REWRITE-V&C$**, and **REWRITE-V&C-APPLY$**, starting on page 202. Otherwise, the rewriter determines whether **fn** is among the $\mathbf{fn_i}$ noted in the **HANDS-OFF** hint. If so, the intermediate term is returned as the result of rewriting the target term. Otherwise, the rewriter tries to apply enabled **REWRITE** rules matching the intermediate term, including the definition of **fn**.

12.51.6. INDUCT Hints

General Form:
(INDUCT (fn $\mathbf{v_1}$ **...** $\mathbf{v_n}$**))**

Example Form:
(INDUCT (PRIME1 X Y))

The **INDUCT** hint forces the theorem prover immediately into the induction process. Furthermore, rather than select an induction using its heuristics, the induction used is that suggested by **(fn** $\mathbf{v_1}$ **...** $\mathbf{v_n}$**)**. **fn** must be a recursively defined function, and the $\mathbf{v_i}$ must be distinct variables. The induction suggested has exactly the same case analysis as in the body of the definition of **fn**. Those branches not leading to recursive calls are base cases. Those branches leading to $\mathbf{k}$>**0** recursive calls **(fn** $\mathbf{t_{1,1}}$ **...** $\mathbf{t_{1,n}}$**)**, ..., **(fn** $\mathbf{t_{k,1}}$ **...** $\mathbf{t_{k,n}}$**)** have **k** induction hypotheses. The $\mathbf{i}^{th}$ induction hypothesis is obtained by instantiating each $\mathbf{v_j}$ by the corresponding term $\mathbf{t_{i,j}}$ from the $\mathbf{i}^{th}$ recursive call.

The output produced by the theorem prover in response to an **INDUCT** hint is very unenlightening: it prints nothing at all but simply produces the various cases as described above and proceeds to simplify each.

12.51.7. DO-NOT-INDUCT

General Form and Example Form:
(DO-NOT-INDUCT T)

If present, the **DO-NOT-INDUCT T** hint causes the theorem prover to abort with failure as soon as it encounters a goal for which it would otherwise enter the induction process. (The second element of the hint can, in fact, be any Lisp object and is ignored.)

12.51.8. DO-NOT-GENERALIZE

General Form and Example Form:
(DO-NOT-GENERALIZE T)

If present, the **DO-NOT-GENERALIZE T** hint disables the entire generalization process. (The second element of the hint can, in fact, be any Lisp object and is used as the value of an internal flag. If the flag setting is **NIL** the hint has no effect at all. Thus, the ‘‘General Form’’ is also the ‘‘Example Form.’’)

12.51.9. NO-BUILT-IN-ARITH

General Form and Example Form:
(NO-BUILT-IN-ARITH T)

If present, the **(NO-BUILT-IN-ARITH T)** hint prevents the built-in linear arithmetic procedure from being used. We use this hint only to demonstrate the ability (or inability) of the system to prove by heuristic means the elementary theorems about **LESSP** and **PLUS**. (The second element of the hint can, in fact, be any Lisp object and is used as the value of an internal flag. If the flag setting is **NIL** the hint has no effect at all. Thus, the ‘‘General Form’’ is also the ‘‘Example Form.’’)

12.52. R-LOOP

General and Example Form:
(R-LOOP)

R-LOOP permits the evaluation of terms in the logic. It is a Lisp style ‘‘read-eval-print loop’’ providing certain convenient nonlogical features through so-called ‘‘special forms.’’ (Noninteractive entrances to **R-LOOP** are provided via the functions **R** and **S** described at the end of this section.)

Upon entry to **R-LOOP** the system prints a header indicating the values of certain ‘‘switch settings’’ explained below. Then **R-LOOP** prompts the user for type-in by printing a ‘‘*****’’ and then waiting until an s-expression, **t**, has been typed. As usual in our system, one is actually typing to the standard Lisp reader. Line editing, redisplay of previously typed input, etc., are all available as however provided by the host Lisp system. In some Lisp systems, the user must type a carriage return after the s-expression before the system actually reads it; in other systems, the final balancing close parenthesis initiates the read.

Once **t** has been read it is evaluated under the axioms and definitions as described below, unless **t** is one of the special forms also noted below. The process of evaluation will yield one of three possible outcomes: an error message indicating that **t** is not well-formed, a message indicating that **t** cannot be reduced to an explicit value, or an explicit value equivalent to **t**. In the latter case the value is printed. In all cases, **R-LOOP** then iterates by prompting for another s-expression.

Here is a sample exchange:

```
*(APPEND '(1 2 3) '(4 5 6))    ;user type-in
 '(1 2 3 4 5 6)                ;R-LOOP's response
```

Users familiar with the Lisp ‘‘read-eval-print’’ loop will notice a slight but disconcerting difference: in Lisp the exchange would have been

```
*(APPEND '(1 2 3) '(4 5 6))    ;user type-in
 (1 2 3 4 5 6)                 ;Lisp's response
```

Note the absence of the **'** in Lisp’s response. **R-LOOP** deals exclusively with *terms*—the ‘‘values’’ printed are terms, they just happen to all be explicit values and thus, when printed in our syntax, usually are preceded by the **'** mark. (Numbers, **T**, **F**, and **NIL** are the only explicit values displayed without use of **QUOTE** notation.)

By ''evaluation'' we mean the reduction of a term to an explicit value by a call-by-value style interpretation of the axioms and definitions. For example, to evaluate a term of the form **(PERM x y)**, **R-LOOP** recursively reduces **x** and **y** to explicit values and then (effectively) instantiates the body of **PERM** with those values and reduces the result.[25]

The evaluation performed by **R-LOOP** takes place under an assignment of explicit values to variable symbols. **R-LOOP** permits the user to set this assignment with the **SETQ** special form. This feature is provided so that the user can save the value of one evaluation for possible input to another. Thus, if **X** has the value **7** and **Y** has the value **8** then the evaluation of **(TIMES X Y)** is **56**.

If the term **t** is submitted for evaluation and produces the explicit value **v**, then it is a theorem that **(EQUAL t v)**, under the equality hypotheses implicit in the current asignment of values to variables.

Not all terms can be reduced to explicit values. Examples are variables not assigned values in the current **R-LOOP** assignment (i.e., ''unbound variables'' in the programmer's sense, ''undefined abbreviations'' in the mathematician's) and calls of undefined or constrained functions. Some terms which are provably equal to explicit values cannot be reduced to that explicit value by evaluation. For example, **(IF V 3 3)** is provably equal to **3** but does not reduce to **3** if **V** is ''unbound.'' Similarly, **(V&C$ T '(RUSSELL) NIL)** is provably equal to **F**, if **RUSSELL** is defined as on page 26, but will not reduce to **F**. When a term submitted for evaluation cannot be reduced to an explicit value by **R-LOOP**, the answer ''**(NOT REDUCIBLE)**'' is printed.

R-LOOP has two mode switches that can be set by several special forms. One of the switches determines whether evaluation is ''traced'' and, if so, how. The other determines whether abbreviations are used when values are printed out.

The *trace mode* may be *full*, *partial*, or *off*. When trace mode is off, no printing occurs during the evaluation of terms. The value is printed when the evaluation has completed successfully. When trace mode is not off, printing occurs during the evaluation of each term. Trace mode prints a step-by-step proof of the equivalence of the input term and the derived value. The difference between full and partial trace mode is only how large the printed steps are. In full trace mode, every application of substitutions of equals for equals into the input term is printed. In partial trace mode, we print only the ''major'' steps associated with function expansion.

Suppose we define **APP** as **APPEND**:

---

[25] The ''**R**'' in ''**R-LOOP**'' in fact stands for ''reduce'' as we defined it in [7].

Definition.

```
(APP X Y)
   =
(IF (NLISTP X)
    Y
    (CONS (CAR X)
          (APP (CDR X) Y)))
```

Below we illustrate the evaluation of **(APP ’(A) ’(1 2 3))**, first with trace mode off, then in full trace mode, and then in partial trace mode. Then, we illustrate **(APP ’(A B C) ’(1 2 3))** in partial trace mode.

```
(R-LOOP)
Trace Mode: Off   Abbreviated Output Mode:  On
Type ? for help.

*(APP ’(A) ’(1 2 3))
 ’(A 1 2 3)


*FULL-TRACE
Trace Mode:  Full

*(APP ’(A) ’(1 2 3))
=(IF (NLISTP ’(A))
     ’(1 2 3)
     (CONS (CAR ’(A))
           (APP (CDR ’(A)) ’(1 2 3))))
=(IF F
     ’(1 2 3)
     (CONS (CAR ’(A))
           (APP (CDR ’(A)) ’(1 2 3))))
=(CONS (CAR ’(A))
       (APP (CDR ’(A)) ’(1 2 3)))
=(CONS ’A (APP (CDR ’(A)) ’(1 2 3)))
=(CONS ’A (APP NIL ’(1 2 3)))
=(CONS ’A
       (IF (NLISTP NIL)
           ’(1 2 3)
           (CONS (CAR NIL)
                 (APP (CDR NIL) ’(1 2 3)))))
=(CONS ’A
       (IF T
           ’(1 2 3)
           (CONS (CAR NIL)
```

```
                 (APP (CDR NIL) '(1 2 3)))))
='(A 1 2 3)


*TRACE
Trace Mode:  Partial

*(APP '(A) '(1 2 3))
=(CONS 'A (APP NIL '(1 2 3)))
='(A 1 2 3)


*(APP '(A B C) '(1 2 3))
=(CONS 'A (APP '(B C) '(1 2 3)))
=(CONS 'A
       (CONS 'B (APP '(C) '(1 2 3))))
=(CONS 'A
       (CONS 'B
             (CONS 'C (APP NIL '(1 2 3)))))
='(A B C 1 2 3)


*OK
Exiting R-LOOP.
NIL
```

The *output abbreviation mode* may be either *on* or *off*. The mode determines how the final explicit value of a term is displayed. When output abbreviation is on, explicit values are displayed in a way that does not require use of the ugly ``**1***'' prefix. When output abbreviation mode is off, explicit values are displayed in **QUOTE** notation. For example,

```
(R-LOOP)
Trace Mode: Off   Abbreviated Output Mode: Off
Type ? for help.

*(PAIRLIST '(A B C) (LIST 1 2 3))
 '((A . 1) (B . 2) (C . 3))
*(PAIRLIST '(A B C) (LIST 0 T F))
 '((A . 0)
   (B . *1*TRUE)
   (C . *1*FALSE))


*ABBREV
```

```
Abbreviated Output Mode:  On
*(PAIRLIST '(A B C) (LIST 1 2 3))
 '((A . 1) (B . 2) (C . 3))
*(PAIRLIST '(A B C) (LIST 0 T F))
 (LIST '(A . 0)
       (CONS 'B T)
       (CONS 'C F))


*OK
Exiting R-LOOP.
NIL
```

In the session above we start with abbreviated output mode off. We evaluate two **PAIRLIST**s. The values of both are printed in **QUOTE** notation. But the value of the second one requires the use of ***1*TRUE** and ***1*FALSE**.

Then we enter abbreviated output mode and re-evaluate both **PAIRLIST**s. The value of the first one is printed just as before. But the value of the second one is not printed in **QUOTE** notation. Instead, **LIST** and **CONS** are used to ‘‘unquote’’ enough of the value to expose those parts that would otherwise require use of the ‘‘***1***’’ prefix.

Below we list the special forms recognized by **R-LOOP**. The special forms are, in general, well-formed terms that are simply not treated as terms when read at the top level by **R-LOOP**.

OK Exit **R-LOOP** and return to its caller. If you invoked **(R-LOOP)** from the Lisp command interpreter (as opposed to from within some other Lisp program), **OK** returns you to the Lisp command interpreter. No harm results from exiting **R-LOOP** via Lisp interrupt characters (see ‘‘Aborting Commands’’ page 119). Variable assignments in the **R-LOOP** assignment are maintained from one call of **R-LOOP** to the next.

(SETQ var term)
In any s-expression of this form, **var** should be a variable symbol and **term** should be a term. This special form causes **term** to be evaluated under the current **R-LOOP** assignment and, if the evaluation succeeds and produces **val**, the current assignment is changed so that **var** has the value **val**. In addition, **val** is printed. If the evaluation of **term** fails, no change is made to the **R-LOOP** assignment.

TRACE Set trace mode to ‘‘partial.’’

FULL-TRACE Set trace mode to ‘‘full.’’

UNTRACE	Set trace mode to ‘‘off.’’
ABBREV	Turn on abbreviated output mode.
UNABBREV	Turn off abbreviated output mode.
?	Causes **R-LOOP** to print out a brief summary of the available special forms.

Special forms are given special treatment only when they are the top-level s-expression read by **R-LOOP**. If, for example, the **SETQ** special form is used as a proper subterm in an s-expression, it is evaluated in the usual sense of the logic. For example, suppose **X** is **7** in the **R-LOOP** assignment, and the user types the s-expression **(PLUS (SETQ X 3) X)** to **R-LOOP**. Unless the user has defined the function **SETQ** in the logic, this s-expression is ill-formed. If **SETQ** has been defined, the value of this s-expression is the value of **(PLUS (SETQ 7 3) 7)**.

The **R-LOOP** assignment is maintained from one call of **R-LOOP** to the next within a given session with the theorem prover. A typical use of **R-LOOP** in a session is to enter it periodically after function definitions to test the suitability of the definitions or the validity of a conjecture being formulated. The variable assignment feature is often used to store test data, e.g., the state argument of an interpreter for a programming language being formalized.

The variable assignment is not written to the library files created by **MAKE-LIB**. Unless otherwise saved, **R-LOOP** assignments are lost when the Lisp session terminates. The assignment is stored in an internal form as the value of the Lisp variable **R-ALIST**. This value may be printed and read back in via the usual Lisp print and read programs.

The Lisp routine **R** is the heart of **R-LOOP**. **R** takes a single s-expression as its input, translates it, reduces it with respect to **R-ALIST**, and returns either the special value **’(NOT REDUCIBLE)** or the result of introducing abbreviations back into the reduced value. The Lisp routine **S** takes two arguments. The first must be a Lisp symbol representing a variable in the logic and the second must be an s-expression whose translation is an explicit value. **S** modifies **R-ALIST** so that the value of the variable is the given explicit value.

12.53. REDUCE-TERM-CLOCK

General Form:
REDUCE-TERM-CLOCK

This Lisp variable may be set to any integer value. Its initial value is 100. The variable is used to prevent ''infinite loops'' in the executable counterparts of partial functions. In particular, the variable specifies the number of times such a function is permitted to recur before the execution is aborted.

For example, suppose that we introduce the function **APP** with

```
(DEFN APP (X Y)
  (EVAL$ T '(IF (EQUAL X (QUOTE NIL))
                Y
                (CONS (CAR X) (APP (CDR X) Y)))
           (LIST (CONS 'X X) (CONS 'Y Y))))
```

Then suppose we enter **R-LOOP** and try to evaluate **(APP '(A B C) '(1 2 3))**. The answer, **'(A B C 1 2 3)**, is quickly computed. However, if we try to evaluate **(APP 0 1)**) we get the message: **APP aborted** and **R-LOOP** reports that the term is **(NOT REDUCIBLE)**.

Because **APP** was introduced without a proof of termination, its executable counterpart, ***1*APP**, is coded so that no more than **REDUCE-TERM-CLOCK** recursions are permitted. Because **(APP 0 1)** is undefined by the above recursion, the limit on the number of recursions is exceeded. Whenever the limit is exceeded the message **APP aborted** is printed by ***1*APP** and the term involving that computation is considered irreducible. While this seems perfectly sensible in the case of an infinite loop, like **(APP 0 1)**, it is less attractive in cases where **APP** is defined but the resource limitation imposed by **REDUCE-TERM-CLOCK** is insufficient to permit the computation to conclude. For example, **(APP '(1 2 3 ... 100) NIL)** cannot be computed if the **REDUCE-TERM-CLOCK** is set to 100.

Because the executable counterparts of defined functions are also called during proof attempts, the message that a computation has been **aborted** may appear in the middle of a proof attempt—even a successful proof attempt. This message is extremely annoying because it is not coordinated with the rest of the theorem prover's output and hence destroys the organization of that output. Furthermore, when the limit is set very high and a term involving an infinite loop occurs in the theorem, much time is spent in the needless attempt to evaluate the executable counterpart. In this case, time can be saved by setting **REDUCE-TERM-CLOCK** to some small positive integer—ideally to the depth of the maximum computation required in the proof. But we eschew such hackery because we don't like finding proofs only because of resource limitations. When we are tempted to play with the setting of **REDUCE-TERM-CLOCK**, we more often disable the executable counterpart of the function involved, after proving **REWRITE** rules that exhibit the values of the necessary computations.

If **REDUCE-TERM-CLOCK** is set to -1 then no attempt is made to limit the recursion. This may result in Lisp stack overflows during the proofs of theorems.

The value of **REDUCE-TERM-CLOCK** is not saved in library files created by **MAKE-LIB**. Thus, its value is not properly restored by **NOTE-LIB**. Neither is its value restored by **UNDO-BACK-THROUGH**. These omissions are essentially due to the fact that we do not provide an event for manipulating the value of **REDUCE-TERM-CLOCK**, primarily because we think it would be so rarely used. In any case, the user is entirely responsible for maintaining the setting of **REDUCE-TERM-CLOCK** and failure to maintain it properly can conceivably prevent previously successful events from replaying.

12.54. REWRITE

REWRITE is one of the rule class tokens that may be given to **ADD-AXIOM**, **CONSTRAIN**, **FUNCTIONALLY-INSTANTIATE** or **PROVE-LEMMA** to indicate what kinds of rules should be generated from a formula. **REWRITE** rules actually come in four different forms: type prescription rules, compound recognizer rules, replacement rules, and linear arithmetic rules. All four forms are used in the simplification process, which is the first process tried. In addition, type prescription rules are used throughout the system to help determine the type of a term.

It is perhaps a mistake for us to lump together into the class ‘‘**REWRITE**’’ all four kinds of rules. Only replacement rules cause terms to be rewritten in the way traditionally associated with term rewrite systems. If a theorem built in as a **REWRITE** rule fails to cause rewriting, it may be that the rules generated from the theorem were not replacement rules.

12.55. REWRITE-APPLY$

REWRITE-APPLY$** is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(APPLY$ (QUOTE fn) args). Roughly speaking, if **fn** is a symbol that names a total function, then **(APPLY$ (QUOTE fn) args)** is rewritten to **(fn (CAR args) ... (CAD...DR args))**, where the appropriate number of arguments for **fn** are taken from **args**. This simplification is justified by theorems proved in [11]. If the name **REWRITE-APPLY$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.56. REWRITE-APPLY-SUBR

REWRITE-APPLY-SUBR is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(APPLY-SUBR (QUOTE fn) args)**. If **fn** is a function symbol, **(SUBRP 'fn)** is **T** and **fn** is not ''erratic'' as defined below, then **(APPLY-SUBR (QUOTE fn) args)** is rewritten to **(fn (CAR args) ... (CAD...DR args))**, where the appropriate number of arguments for **fn** are taken from **args**. If **(SUBRP 'fn)** is **F**, **(APPLY-SUBR (QUOTE fn) args)** is rewritten to **F**. These simplifications are justified by **SUBRP** axiom and **Axiom 55**.

The notion of ''erratic'' is quite subtle. At first sight, one is tempted to agree that if **(APPLY-SUBR 'fn args)** is **(fn (CAR args) ... (CAD...DR args))**, if **(SUBRP 'fn)** is **T**. Certainly it is true for the **SUBRP**s that are introduced implicitly by our axiomatic acts, because we add the corresponding axiom about **APPLY-SUBR** as part of the ''**SUBRP** axiom'' (page 106). But it is consistent for the user to declare a new function symbol, say, **G**, of no arguments, and to add axioms specifying that **(SUBRP 'G)** is **T** but **(APPLY-SUBR 'G ARGS)** is **(NOT (G))**. Then clearly it would be incorrect to simplify **(APPLY-SUBR 'G args)** to **(G)**.

To avoid this we introduce the notion of ''erratic'' functions. A function symbol is *erratic* if it was introduced with **DCL** or **CONSTRAIN** or else mentions an erratic function in its **BODY**. Note that we sweepingly consider all declared functions erratic, rather than more carefully analyze the available **APPLY-SUBR** axioms.

If the name **REWRITE-APPLY-SUBR** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.57. REWRITE-CAR-V&C$

REWRITE-CAR-V&C$** is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **(CAR (V&C$ T (QUOTE term) a)). Roughly speaking, if **term** is a term containing no erratic symbols and the domain of the alist **a** is manifest, we simplify **(CAR (V&C$ T (QUOTE term) a))** to the appropriate instance of **term**, provided **(V&C$ T (QUOTE term) a)** is non-**F**, and to **0** otherwise. This simplification is justified by theorems proved in [11]. If the name **REWRITE-CAR-V&C$** is disabled, this simplifier is not used. The name is a citizen (a satellite of **GROUND-ZERO**) so it can be enabled and disabled. It is initially enabled.

12.58. REWRITE-CAR-V&C-APPLY$

`REWRITE-CAR-V&C-APPLY$` is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **`(CAR (V&C-APPLY$ (QUOTE fn) args))`**. Roughly speaking, if **`F`** is not a member of **`args`** and **`fn`** is a total function other than **`IF`**, then **`(CAR (V&C-APPLY$ (QUOTE fn) args))`** simplifies to **`(fn (CAAR args) ... (CAAD...DR args))`**. This simplification is justified by theorems proved in [11]. If the name **`REWRITE-CAR-V&C-APPLY$`** is disabled, this simplifier is not used. The name is a citizen (a satellite of **`GROUND-ZERO`**) so it can be enabled and disabled. It is initially enabled.

12.59. REWRITE-EVAL$

`REWRITE-EVAL$` is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **`(EVAL$ T (QUOTE term) a)`** and some expressions of the form **`(EVAL$ T (CONS (QUOTE fn) args) a)`**. For example, if **`term`** is tame and the domain of the alist **`a`** is manifest, then **`(EVAL$ T (QUOTE term) a)`** is just a certain instance of **`term`**. This simplification is justified by theorems proved in [11]. If the name **`REWRITE-EVAL$`** is disabled, this simplifier is not used. The name is a citizen (a satellite of **`GROUND-ZERO`**) so it can be enabled and disabled. It is initially enabled.

12.60. REWRITE-V&C$

`REWRITE-V&C$` is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **`(V&C$ flg form a)`**, where **`flg`** is **`'LIST`** or **`form`** is a **`QUOTE`**d tame term. The simplification just replaces the expression by non-**`F`** in situations where propositional equivalence is sufficient. This is justified by theorems proved in [11]. If the name **`REWRITE-V&C$`** is disabled, this simplifier is not used. The name is a citizen (a satellite of **`GROUND-ZERO`**) so it can be enabled and disabled. It is initially enabled.

12.61. REWRITE-V&C-APPLY$

`REWRITE-V&C-APPLY$` is the name of a special-purpose simplifier built into Nqthm. It simplifies some expressions of the form **`(V&C-APPLY$ (QUOTE fn) args)`**. For example, if **`fn`** is a total function other than **`IF`** and only propositional equivalence is of interest, then **`(V&C-APPLY$ (QUOTE fn) args)`** is non-**`F`** provided **`F`** is not in **`args`**. This is justified by theorems proved in [11]. If the name **`REWRITE-V&C-APPLY$`** is disabled, this simplifier is not used. The name is a citizen (a satellite of **`GROUND-ZERO`**) so it can be enabled and disabled. It is initially enabled.

12.62. Root Names

See the discussion of File Names on page 163.

12.63. Rule Classes

A *rule class* is one of the symbols **`REWRITE`**, **`ELIM`**, or **`GENERALIZE`** or else a list of the form **`(META`** $\mathbf{fn_1}$ **`...`** $\mathbf{fn_n}$**`)`** where each $\mathbf{fn_i}$ is the name of a function in the logic. Rule classes are used to specify how a given axiom or theorem is to be stored as a rule in the data base. In Chapter 11 we describe in detail how rules are generated from formulas.

12.64. SET-STATUS

General Form:

```
(SET-STATUS name citizens action-alist)
```

Example Form:

```
(SET-STATUS CLOSE-OFF-LAYER1 T
  ((BOOT-STRAP INITIAL)
   (ADD-SHELL ENABLE)
   ((DEFN *1*DEFN) ENABLE)
   (OTHERWISE DISABLE)))
```

`SET-STATUS` is an event command that sweeps over a list of citizens and performs an action that may alter the enabled/disabled status of the citizen as a function of the type of event that introduced the citizen. See **DISABLE** for a general discussion of the effect and motivation behind enabling or disabling a name. See **DEFTHEORY** for a general discussion of the use of theories.

SET-STATUS does not evaluate its arguments. The first argument, **name**, must be a new name. The second argument, **citizens**, denotes some list of citizens and must be of one of the forms below:

- a theory name (as defined with **DEFTHEORY** or one of the builtin theory names **GROUND-ZERO** and **T**), denoting the value of the theory;
- a list of citizens denoting itself; or
- a pair of the form **(ev_1 . ev_2)** containing two event names, ev_1 and ev_2, denoting all of the citizens introduced in the inclusive chronological interval bounded on one end by the event ev_1 and on the other by ev_2.

Note that when an event pair is used, choice of order is unimportant. Finally, the third argument, **action-alist**, must be a list of doublets, each of the form **(key val)**, where each **key** is either a member of the set of words {**BOOT-STRAP ADD-SHELL DEFN *1*DEFN ADD-AXIOM PROVE-LEMMA CONSTRAIN FUNCTIONALLY-INSTANTIATE OTHERWISE**} or else is a proper list of members of that set, and each **val** is a member of the set {**ENABLE DISABLE AS-IS INITIAL**}, and where the following restrictions apply:

- no **key** word may occur twice in **action-alist**,
- the **val INITIAL** may be paired only with the **key BOOT-STRAP**, and
- the last element of **action-alist** must have **OTHERWISE** as its **key**.

The **action-alist** is used to determine a ‘‘status action’’ for citizens of the data base, as follows. If the citizen is the name of the executable counterpart of a function introduced with **DEFN**, then the status action is the **val** associated with the **key *1*DEFN** in **action-alist**, if there is one, and otherwise is the **val** associated with the **key OTHERWISE**. If the citizen is not such an executable counterpart, then its status action is the **val** associated with the kind of event command that created the citizen, if that **key** appears in the **action-alist**, and otherwise is the **val** associated with **OTHERWISE**.

SET-STATUS creates a new event, named **name**. The only other effect of **SET-STATUS** is possibly to change the enable/disable status of the citizens in the list denoted by **citizens**. The new status is determined by the status action for the name. If the action is **ENABLE**, the citizen is made enabled by **SET-STATUS**. If the action is **DISABLE**, the citizen is made disabled by **SET-STATUS**. If the action is **AS-IS** the citizen's status is unchanged. Finally, if the action is **INITIAL** then the citizen was created by **BOOT-STRAP** and its status is restored to what it was at the conclusion of **BOOT-STRAP**.

For example, the example command

```
(SET-STATUS EXAMPLE (PRIME-FACTORS . GROUND-ZERO)
  ((BOOT-STRAP INITIAL)
   (ADD-SHELL ENABLE)
   ((DEFN *1*DEFN) ENABLE)
   ((PROVE-LEMMA ADD-AXIOM) DISABLE)
   (OTHERWISE AS-IS)))
```

will create a new event named **EXAMPLE** and will possibly affect the status of all the names introduced in the inclusive interval from **GROUND-ZERO** through **PRIME-FACTORS**. In particular, it will leave all the satellites of **GROUND-ZERO** with the same status they had after the **BOOT-STRAP** that created them, it will enable all the satellites of all **ADD-SHELL**s, it will enable all defined functions and their executable counterparts, it will disable all **PROVE-LEMMA**s and **ADD-AXIOM**s, and it will leave all other citizens unchanged. Thus, for example, no **FUNCTIONALLY-INSTANTIATE** event is affected by this **SET-STATUS**

A common use of **SET-STATUS** is in its guise as **DISABLE-THEORY** and **ENABLE-THEORY**, where it is used to set the status of all the names in a given theory. We find that big proof projects often proceed in stages. During one particular stage, a certain collection of theories is heavily used, while in a later stage, a different collection of theories is needed. While one can manage this with local hints, it may be preferable (depending on the extent of each stage) to use global commands to configure the system to use the right rules automatically. At the end of a stage one then disables the now uninteresting theories and develops and enables the appropriate new ones.

In such projects we find useful such **SET-STATUS** commands as the following one.

```
(SET-STATUS END-OF-STAGE-1 T
  ((BOOT-STRAP INITIAL)
   ((ADD-SHELL DEFN *1*DEFN) ENABLE)
   (OTHERWISE DISABLE)))
```

This command essentially erects a barrier between one stage and the next. It restores the primitive rules (including shell rules) to their normal status, enables all definitions and executable counterparts, and disables everything else. Thus, the possibly hundreds of rules proved during ‘‘stage 1’’ are now inactive and we are ready to begin a completely new stage. If we find that we need rules that were proved during stage 1, we enable them explicitly when we discover they are needed. The reason we enable function symbols is that if we mention them in our subsequent work we will likely want them to behave as though we had just defined them; and if we do not mention them, their being enabled does not hurt us. Of course, we might want to **`DISABLE`** explicitly certain functions.

Warning. In an effort to be efficient, **`SET-STATUS`** does not touch the status recorded for a name unless it must change it. This has a bizarre interaction with **`UNDO-NAME`** that is best explained by illustration. First, forget about **`SET-STATUS`** and consider how **`DISABLE`** events (actually, **`TOGGLE`** events) affect the status of a name. Suppose **`NAME`** is initially enabled and that at event **`a`** it is disabled by **`(DISABLE NAME)`**. Suppose some theorems are proved after **`a`** but no status events occur. Finally, suppose that another **`(DISABLE NAME)`** occurs at event **`b`**. Clearly, **`NAME`** is disabled after **`b`**. But suppose event **`a`** is undone with **`UNDO-NAME`**. What then is the status of **`NAME`**? The answer is that **`NAME`** is still disabled because event **`b`** disabled it and has not been undone or overridden.

Now consider the same scenario but this time use two **`SET-STATUS`** commands instead of the two **`DISABLE`** commands. For example, event **`a`** might be **`(SET-STATUS a (NAME . NAME) ((OTHERWISE DISABLE)))`** and **`b`** might be analogous. Because **`NAME`** is already disabled when event **`b`** is executed, event **`b`** has no effect on **`NAME`**. Thus, when **`a`** is undone with **`UNDO-NAME`** the status of **`NAME`** changes from disabled to enabled even though there is a subsequent **`SET-STATUS`** at event **`b`** that would have disabled it had event **`b`** actually been executed in the new context. We could have implemented **`SET-STATUS`** like **`TOGGLE`** so that it records the desired status regardless of the old status. We chose this implementation for efficiency: **`SET-STATUS`** is often used to map over thousands of names, many of which have already been disabled by previous sweeping **`SET-STATUS`** commands. We felt free to implement it this way, despite the difficulties with **`UNDO-NAME`**, because **`UNDO-NAME`** is already known not to maintain the integrity of the data base (see 144).

12.65. SET-TIME-LIMIT

General Forms:
`(SET-TIME-LIMIT)`
and
`(SET-TIME-LIMIT n)`

Example Form:
(SET-TIME-LIMIT 10)

SET-TIME-LIMIT is a Lisp routine that sets a limit on the amount of time that may be spent in the theorem prover. When the system notices that it has exceeded the limit, an error is signalled. When **n** is supplied, it must be a positive integer representing the time limit in seconds. **(SET-TIME-LIMIT n)** installs **n** as the limit. **(SET-TIME-LIMIT)** removes the limit. The system initially has no limit. The limit is not part of the data base and is not saved in library files. Thus, if you want a time limit during a given session, you must invoke **SET-TIME-LIMIT** in that session. Time limits are checked only during rewriting. There are other time-consuming loops in our system, such as ‘‘clausification.’’ Because of this, it is possible (but unusual) for the system to run considerably longer than the allotted time.

12.66. SKIM-FILE

General Forms:
(SKIM-FILE file)
and
(SKIM-FILE file :CHECK-SYNTAX cs-flg)

Example Form:
(SKIM-FILE "basic")

SKIM-FILE is a convenient way to call **PROVE-FILE** with the **:PETITIO-PRINCIPII** keyword argument set to **T** and the **:WRITE-FLAG-FILES** keyword argument set to **NIL**. **SKIM-FILE** prints a message to the terminal before calling **PROVE-FILE**; the message reminds the user that **SKIM-FILE** may render Nqthm inconsistent and that output is suppressed while skimming. See **PROVE-FILE** (page 180) for details and pay special attention to the warning about **:PETITIO-PRINCIPII**.

Roughly speaking, the effect of **SKIM-FILE** is to process all the event forms in **file**, merely *assuming* their admissibility. **SKIM-FILE** evaluates its arguments. The first argument, **file**, should be a file root name (see page 163) and **file.events**, the **events** extension of **file**, should exist and contain Nqthm event commands as described in the documentation of **PROVE-FILE**. The keyword argument is optional; its meaning is as described in **PROVE-FILE**.

12.67. Time Triple

A *time triple* is a triple of numbers printed at the end of event commands and indicating the amount of time used. The printed form of the triple is

```
[ pre-post prove output ]
```

where **pre-post** is the number of seconds spent verifying the syntactic preconditions of the command and generating and adding the appropriate rules to the data base, **prove** is the number of seconds spent in formal proof, and **output** is the number of seconds spent printing information to the user.

For example, in a **DEFN** command, **pre-post** includes the time taken to check that the body of the definition involves no free variables, etc., as well as the time to preprocess the definition to compute such information as the induction schemes suggested; **prove** is the time taken to prove the admissibility formulas; and **output** is the time taken to print the message associated with acceptance of a definition.

12.68. THM Mode

THM mode refers to the set of axioms added by **BOOT-STRAP**. The system can be brought up so that it supports a constructive variant of the logic described here. This is in fact the logic supported by our system before we introduced the nonconstructive function **V&C$** and the quantifier **FOR**. See the discussion of **BOOT-STRAP** on page 127.

12.69. TOGGLE

General Form:
(TOGGLE new-name old-name flg)

Example Forms:
(TOGGLE TIMES-2-REWRITE-OFF TIMES-2-REWRITE T)
and
(TOGGLE REVERSE-ON REVERSE NIL)

TOGGLE is an event command for setting the status of a citizen in the data base. **new-name** must be a new event name, **old-name** must be a citizen, and **flg** should be **T** or **NIL**. The name of the new event created is **new-name**. The effect of the event is to set the status of **old-name** to **DISABLED** if **flg** is **T** and to **ENABLED** if **flg** is **NIL**. **TOGGLE** then prints a time triple and returns

new-name.

The event commands **DISABLE** and **ENABLE** are both abbreviations for **TOGGLE** events. For details on the effect of disabling and enabling names, see the discussion of **DISABLE**.

12.70. TOGGLE-DEFINED-FUNCTIONS

General Form:
(TOGGLE-DEFINED-FUNCTIONS new-name flg)

Example Form:
(TOGGLE-DEFINED-FUNCTIONS DISABLE-ALL-COUNTERPARTS T)

TOGGLE-DEFINED-FUNCTIONS is an event command. Its arguments are not evaluated. Roughly speaking, this command allows you to disable, globally, all executable counterparts, i.e., prevent all functions from automatically evaluating on explicit values.

The first argument, **new-name**, must be a new event name. The second argument, **flg**, must be either **T** or **NIL**. The command sets the ‘‘executable counterparts disabled’’ flag of the data base to **flg**, creates a new event named **new-name**, prints a time triple, and returns **new-name**. The effect of the ‘‘executable counterparts disabled’’ flag is that, when non-**NIL**, no executable counterpart, other than shell constructors and base functions, are used during proofs. Thus, if the executable counterparts disabled flag is **T**, the expression **(PLUS 3 4)**, for example, would not simplify to **7**, except by repeated expansion of the definitional axiom or by **REWRITE** rules.

Technically speaking, the flag is not stored explicitly in the data base. Instead, the node for **new-name** is stored, and the most recently executed **TOGGLE-DEFINED-FUNCTIONS** event in the graph determines the value of the flag.

12.71. TRANSLATE

General Form:
(TRANSLATE x)

Example Form:
(TRANSLATE ’(TIMES X Y (CADDR U)))

TRANSLATE is the Lisp procedure that maps user type-in into a well-formed term, or causes an error. The argument is evaluated. The value is then explored, abbreviations are expanded, and the result is checked for well-formedness. **TRANSLATE** returns the internal representation of the given term. We include **TRANSLATE** in this documentation simply so that you can experiment with the features of the Lisp reader to which you are typing.

In Appendix I and example file 11 we provide a formal definition of a function named **TRANSLATE**, corresponding to our notion in Chapter 4 of ‘‘well-formedness’’ and ‘‘translation.’’ The Lisp procedure **TRANSLATE** and the logical function **TRANSLATE** in Appendix I are closely related but are not identical. Both check for well-formedness. The Lisp procedure signals its absence with an error while the logical function returns **F**. But when the input is well-formed, the Lisp translator returns a term in our internal form while the logical function returns the formal term it denotes. In our internal form, all explicit value terms are represented in **QUOTE** notation. But the whole purpose of the logical translator is to explicate that notation by exhibiting the (often huge) nest of formal constructors. Thus, when **'(ADD1 1)** is given to the Lisp translator, **(QUOTE 2)** is returned. But when **'(ADD1 1)** is given to the logical translator, **(ADD1 (ADD1 (ZERO)))** is returned.

12.72. UBT

General Forms:
(UBT)
and
(UBT x)

Example Forms:
(UBT REVERSE)
and
(UBT 3)

UBT is an abbreviated form of **UNDO-BACK-THROUGH**. The argument, **x**, is not evaluated and must be either an event name or a nonnegative integer. If **x** is an integer it defaults to the $\mathbf{x}^{th}$ element of **CHRONOLOGY** (**0**-based counting, starting with the most recent event name). If no argument is provided, **x** defaults to the most recent event name. **UBT** removes from the data base every event from the most recent through **x**. The list of events forms removed, in chronological order, is stored in the global Lisp variable **UNDONE-EVENTS**. The list of events is suitable as the **events** argument to **PROVEALL** or **DO-EVENTS**; re-evaluating those events will reconstruct the data base. **UBT** returns the name of the oldest event removed, i.e., **x** or the name to which **x** defaulted.

UBT is guaranteed to restore the data base to the state it was in immediately before event **`x`** was created. Using **`UBT`** thus preserves the integrity of the data base. **`UNDO-NAME`** may not. See page 144.

12.73. UNBREAK-LEMMA

General Forms:
`(UNBREAK-LEMMA x)`
or
`(UNBREAK-LEMMA)`

Example Form:
`(UNBREAK-LEMMA 'REVERSE-REVERSE)`

The argument, **`x`**, is optional. If supplied, it is evaluated. If **`x`** is the name of a rule that is being monitored, it is removed from the list of monitored rules. See **`BREAK-LEMMA`**. If **`x`** is **`NIL`** or not supplied, all rules are removed from the list of monitored rules.

12.74. UNDO-BACK-THROUGH

General Form:
`(UNDO-BACK-THROUGH x)`

Example Form:
`(UNDO-BACK-THROUGH 'REVERSE)`

The argument, **`x`**, is evaluated and must be an event name. **`UNDO-BACK-THROUGH`** removes from the data base every event from the most recent through **`x`**. **`UNDO-BACK-THROUGH`** returns the list of event forms removed from the data base, in chronological order. The list returned is suitable as the **`events`** argument to **`PROVEALL`** or **`DO-EVENTS`**; re-evaluating those events will recreate the data base.

`UNDO-BACK-THROUGH` is guaranteed to restore the data base to the state it was in immediately before event **`x`** was created. Using **`UNDO-BACK-THROUGH`** thus preserves the integrity of the data base. **`UNDO-NAME`** may not. See page 144.

12.75. UNDO-NAME

General Form:
(UNDO-NAME name)

Example Form:
(UNDO-NAME 'REVERSE)

The argument, **x**, is evaluated and must be an event name. **UNDO-NAME** removes from the data base **x** and **ALL-DEPENDENTS** of **x** (see the discussion of **DATA-BASE** on page 143). **UNDO-NAME** returns the list of event forms removed from the data base, in chronological order. The list returned is suitable as the **events** argument of **DO-EVENTS**.

Wishfully speaking, **UNDO-NAME** removes from the data base **x** and all of its logical dependents. This is in fact often the case, but we do not guarantee that it is always the case. The problem, as noted on page 144, is that the theorem prover cannot track every use of certain kinds of rules and hence **ALL-DEPENDENTS** may not actually include all dependents. For this reason, **UNDO-NAME** prints a rather tiresome **WARNING** message every time it is called upon to remove any but the most recent event. **UNDO-NAME** is in fact correct when it removes only the most recent event and, more generally, when the events removed are those erased by **UNDO-BACK-THROUGH**.

Warning: Because **UNDO-NAME** may not maintain the integrity of the data base, we do not endorse any proof constructed in a session in which **UNDO-NAME** has been used. We discuss the notion of an ''endorsed data base'' on page 185.

Notwithstanding the foregoing caveat, it should be said that in day-to-day interaction with the theorem prover we make heavy use of **UNDO-NAME**. A common situation is to have defined a function, **fn**, differently than intended but not discover it until several dependent events (e.g., functions using **fn**) have been created. When we discover the discrepancy our usual response is to execute

```
(SETQ XXX (UNDO-NAME 'fn))
```

which undoes the introduction of **fn** and all of its dependents but leaves in place any events not dependent on **fn**. The Lisp variable **XXX** above is set to the list of events undone, starting with **fn**. Then we define **fn** ''properly.'' Finally, we execute

```
(DO-EVENTS (CDR XXX))
```

which attempts to reconstruct the dependents of the old **fn** on top of the new definition of **fn**. The **CDR** above is necessary to skip past the first event of **XXX**, which is the old definition of **fn**. Of course, the **DO-EVENTS** may not succeed: the new definition of **fn** may not have the right properties. More often when the **DO-EVENTS** fails it is because proofs constructed the first time those events were processed cannot be found now because of additions to the data base that were not undone. The most common situation is that names that were enabled when the original events were processed are now disabled, or vice versa. It sometimes happens that rules added to the data base since the original definition of **fn** now lead the theorem prover astray.

In general, we make free use of **UNDO-NAME**, behaving exactly as though it satisfied its wishful specification. However, after several days of such interactive work, and always upon reaching our final goal, we replay the entire list of events created since the last event in some endorsed data base. One way to ‘‘replay’’ an appropriate event sequence is to use **UNDO-BACK-THROUGH** to roll back to some endorsed event and simultaneously collect the event forms since then. Then **PROVEALL** or **DO-EVENTS** may be used to re-evaluate the events.

However, our standard way of ensuring that Nqthm endorses our final goal is to use **PROVE-FILE** on the script file that we build as the permanent record of the project. See **PROVE-FILE** (page 180) for a discussion of the issues surrounding ‘‘endorsement.’’ **PROVE-FILE** also creates a coherent file containing all the proofs.

We have never been burned by our use of **UNDO-NAME** and recommend its use provided its shortcomings are fully understood.

To illustrate how badly **UNDO-NAME** can burn you we offer the following command sequence which concludes with a proof that **T** = **F**.

```
(DEFN FN () 23)

(PROVE-LEMMA FN-ARITY-IS-0 NIL (NLISTP (FORMALS 'FN)))

(UNDO-NAME 'FN)

(DEFN FN (X) 45)

(PROVE-LEMMA FN-ARITY-IS-NON-0 NIL
             (LISTP (FORMALS 'FN)))

(PROVE-LEMMA SURPRISE NIL (EQUAL T F)
             ((USE (FN-ARITY-IS-0)
                   (FN-ARITY-IS-NON-0))))
```

The problem, of course, is that the **`UNDO-NAME`** above does not remove from the data base the theorem **`FN-ARITY-IS-0`** because its dependence on **`FN`** is not recorded since neither the statement of the theorem nor the proof involve the function symbol **`FN`**.

14 Installing Nqthm

In this chapter, we describe how to build an executable version of Nqthm.

14.1. The Source Files

Information about obtaining the source files for Nqthm may be obtained by sending an email message to **`nqthm-request@cli.com`** or by writing to Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, Texas 78703 U.S.A. There is no fee if you obtain the files by anonymous ftp.

The 1988 edition of *A Computational Logic Handbook* indicated that Nqthm was distributed without copyright or license. However, the current sources for Nqthm are copyrighted and there is a license. That license may be found at the beginning of the file **`basis.lisp`**, and also in the files **`nqthm-public-software-license.doc`** and **`nqthm-public-software-license.ps`**. You must read and agree to the terms of this license before using Nqthm.

Here is a listing of the source file names and sizes. The files **`defn-sk.lisp`** and **`infix.lisp`** are not necessary to build Nqthm. Their utility is described later. The file **`sloop.lisp`** is a version of the MIT Loop facility written by Bill Schelter.

```
basis.lisp          107676
code-1-a.lisp       132573
code-b-d.lisp       174693
code-e-m.lisp       143518
```

```
code-n-r.lisp        227375
code-s-z.lisp        120269
defn-sk.lisp          51271
events.lisp           30172
genfact.lisp          11539
infix.lisp           108951
io.lisp               61267
nqthm.lisp            10655
ppr.lisp               8648
sloop.lisp            41551
```

Many other files are distributed with Nqthm, including example files (in the subdirectory **examples** and its subdirectories) and installation utilities (in the subdirectory **make**). However, only the above ***.lisp** files are necessary to build Nqthm from scratch.

The source files for Nqthm are written in the language Common Lisp, cf. *Common Lisp The Language* [53, 54]. Nqthm runs under implementations conforming to either [53] or [54]. We have run Nqthm under Austin Kyoto Common Lisp, Allegro Common Lisp, Lucid Common Lisp, CMU Common Lisp, Macintosh Common Lisp and Symbolics Common Lisp.

SPARC executable image shortcut. As a convenience to a large subset of potential Nqthm users, among the files that we distribute with Nqthm is an executable image of Nqthm built under Austin Kyoto Common Lisp for running on a Sparc. If you intend to run on a Sparc and are happy to use AKCL, you can simply execute the file **sparc-akcl-nqthm-image** and skip the remaining installation instructions. Kyoto Common Lisp (KCL) was developed by Taiichi Yuasa and Masami Hagiya at Kyoto University. Austin Kyoto Common Lisp contains some enhancements and ports by Bill Schelter. Use of KCL requires mailing in a license. See the file **kcl.license**. To obtain the sources for AKCL, see the file **kcl.faq**. Obviously, future changes to the Sparc architecture or operating system may render this image useless. In that case, please follow the instructions for building Nqthm from scratch below.

Unix make shortcut. Although compiling and loading Nqthm is relatively simple, compiling Lisp code so that it will execute with maximum speed and building an executable image are things that require a bit of Lisp expertise and whose details vary from Lisp to Lisp. If you will not be using the Sparc-AKCL image that we distribute with Nqthm, but you will run under Unix and use one of AKCL, Lucid, Allegro, or CMU Common Lisps, then we suggest that you next consider using the **make** command to build an executable Nqthm image. Simply connect to the directory containing the Nqthm sources and execute

```
 % make LISP=xxx
```

where **xxx** is whatever command you use to invoke your Lisp. If successful, this command will compile Nqthm and create an executable file named **nqthm-1992** on the same directory. At successful conclusion, it will print out a message indicating that compilation, saving, and very simple testing of Nqthm-1992 worked ok. Subsequently and optionally, the tiny file **nqthm-1992** may be copied to a **bin** directory for easy invocation.

If this simple **make** command does not work, or if you are not running under Unix, you may easily build Nqthm with the very simple commands below.

To build Nqthm it is necessary to *compile* the source files and then to *load* the object files produced by the compilation.

14.2. Compilation

In order to compile Nqthm, you need to know how to start up your Common Lisp. You also need to have enough room on the directory with the source files to hold the compiled object files. Two megabytes of free disk space should suffice.

Loading the file nqthm.lisp. Under some operating systems, such as Unix, there exists the notion of the default directory associated with a process. When running under such an operating system, *connect* to the directory holding the Nqthm sources, for example by using the **cd** command, start up your Common Lisp, and execute the command **(LOAD "nqthm.lisp")**. On the other hand, if you are running under an operating system without such a notion, first start Common Lisp, then execute the Lisp command **(LOAD "...nqthm.lisp")**, where **...** is a suitable sequence of characters to identify the directory with the Nqthm sources, and then execute the command **(SETQ USER::*DEFAULT-NQTHM-PATH* "...")**.

Getting into the USER package. Next, execute the form:

```
(IN-PACKAGE "USER")
```

We emphasize that we really do mean package **"USER"**, not **"COMMON-LISP-USER"**, even on those Common Lisps that do not have a **"USER"** package initially. In Lisps that conform to the second edition of [54], there may initially be no **"USER"** package. However, loading of the file **nqthm.lisp** will then create a **"USER"** package. All Nqthm commands will be found in the **"USER"** package.

Setting the compiler optimization parameters. There is no single command we know that works across all Common Lisp versions and implementations to generate code that runs as fast as possible. Although it is not absolutely necessary, we recommend that you execute some command such as the following to generate efficient Nqthm compiled code.

```
(PROCLAIM
 (QUOTE (OPTIMIZE
         (SAFETY 0)
         (SPACE #+CMU 1 #-CMU 0)
         (SPEED 3)
         #+Allegro (DEBUG 0)
         #+(OR CMU Lucid) (COMPILATION-SPEED 0)
         #+CMU (EXTENSIONS:INHIBIT-WARNINGS 3))))
```

Compiling. Finally, execute

```
(COMPILE-NQTHM)
```

The compilation may take from a few minutes to an hour depending on the Lisp and machine you are using.

14.2.1. Example of Compilation

For example, to compile Nqthm under AKCL on a Sun, assuming that the source files for the theorem prover have been placed on the directory **`/usr/smith/nqthm-1992/`**, we could execute these commands:

```
% cd /usr/smith/nqthm-1992
% akcl
> (LOAD "nqthm.lisp")
> (IN-PACKAGE "USER")
> (COMPILE-NQTHM)
```

Obviously, the first two commands are Unix commands (hence the **`%`** prompt) and the last three commands are fed to Lisp. We have omitted any compiler optimization parameter setting in this example because under AKCL these parameters come set for maximum speed by default.

14.3. Loading

After Nqthm has been compiled, one may then *load* it into Common Lisp. To load Nqthm one merely (a) executes all of the commands above for compiling Nqthm except the final **`(COMPILE-NQTHM)`** command and then (b) executes **`(LOAD-NQTHM)`**. One may then begin using Nqthm commands, such as **`(BOOT-STRAP NQTHM)`**.

14.3.1. Example of Loading

Suppose that you have compiled Nqthm under Austin Kyoto Common Lisp on a Sun when connected to **/usr/smith/nqthm-1992/**. Then these commands will load Nqthm:

```
% cd /usr/smith/nqthm-1992
% akcl
> (LOAD "nqthm.lisp")
> (IN-PACKAGE "USER")
> (LOAD-NQTHM)
```

14.4. Executable Images

Once Nqthm-1992 has been compiled, building an executable image is generally a good idea because it makes starting up Nqthm very quick and convenient. Unfortunately, building an executable image requires a bit of expertise. See the file **make/save.lisp** for some examples of how to do this well under various Lisps.

14.5. Installation Problems

Although we have installed Nqthm on quite a variety of machines under quite a variety of Lisps, there are naturally problems that will arise.

One common problem is a shortage of disk space. This problem can arise during compilation, while saving an image, or while running examples. Unfortunately, one does not always get helpful diagnostic messages indicating that disk space is the underlying problem. For example, disk space shortage problems can appear as process death problems or as compilation problems. Here are some suggestions for how to begin to diagnose disk space problems. A good Lisp function to use to find out about Lisp space usage is **room**. Good Unix commands to use to help check for running out of disk space are **df**, **du**, and **pwd**. A good Unix command to use to help check for running out of virtual memory is **pstat -s**. A good Unix command to use to check process size is **ps -uax**. The latter two commands have to be executed while a process is running, of course, not after it has crashed.

If during the compilation (or loading) of Nqthm *collisions* between Nqthm's symbols and the symbols of the underlying Common Lisp are reported, then Nqthm may not work in the Lisp in question. For example, in AKCL such a collision would occur if we had defined a function named **CLINES** in our

Nqthm sources. **CLINES** is not listed in [53] or [54] and so we might believe it to be a name we could define. But in fact, in the current version of AKCL, **CLINES** is defined as a function. Loading a definition of it provokes the message **Warning: CLINES is being redefined.** We suspect that we have finally eliminated all such collisions from the lisps we use (Austin Kyoto, Allegro, Lucid, CMU, Macintosh, and Symbolics Common Lisps), but the variety of implementations and the ongoing development of Lisp leave us uncertain. In general, if any such collision is detected, you are advised not to trust the resulting Nqthm or Lisp. However, consultation with experts may well confirm that the collision is of no important consequence.

If the Lisp in question does not use the standard ASCII character codes for the printing ASCII characters (for example **(CHAR-CODE #\A)** is not 65), then Nqthm will not work. We check this during compilation.

Via Bill Schelter's **SLOOP** facility, we use fixnum arithmetic by default for certain iterations.. Schelter's code is conservative and checks to make sure that numbers said to be fixnums are fixnums. Should a small range of fixnums result in an error report, one may work around this problem by setting **SLOOP::*AUTOMATIC-DECLARATIONS*** to **NIL** before compilation.

Proof-checking the entire set Nqthm examples is a fairly large data processing crunch by the standards of 1992 workstations. For example, if we run these test on a Sparc 2 under AKCL, it takes about three days of cpu time. About 270 megabytes of files are created. Under some Lisps a process with a virtual memory size of over 87 megabytes may be created. So your machine and your Lisp may, therefore, perhaps be pressed to a unusual extent with respect to the utilization of virtual memory, disk space, stack size, heap size, etc. Dealing with such problems is exceedingly operating system and Lisp implementation specific. AKCL must return to its top-level read-eval-print loop after this command is executed in order for the command to take effect. Thus, this command cannot be included in the event file itself. On some Risc machines, e.g., Sparc, the AKCL compiler may temporarily need over 100 megabytes of disk space when compiling some example ***.lisp** files. One solution to this is to cause AKCL to split the source files into smaller chunks and to compile the chunks. See the documentation for the variable **COMPILER::*SPLIT-FILES*** in the AKCL file **doc/DOC**.

Under some Lisps, there is likely to arise during compilation a very large number of warnings about functions that are called but not defined. These warnings may be ignored. The functions in question are defined in later files.

Version control can be a perilous problem. In the past, version control problems with Nqthm have been minor because, roughly speaking, each new version of Nqthm was issued in a different programming language: POP-2, Interlisp, Zetalisp, and Common Lisp. This chapter is concerned with the version of Nqthm known as Nqthm-1992, which is written in Common Lisp. The version of Nqthm that was distributed at the time of publication of the 1988 edition of the book *A Computational Logic Handbook* is now sometimes referred to as Nqthm-1987. Nqthm-1987 was also written in Common Lisp. To help distinguish between the 1987 and 1992 versions of Nqthm, we have arranged that each file of the Nqthm-1992 version of Nqthm contains the form **(CHK-BASE-AND-PACKAGE-1992 10 *PACKAGE*)**. No file of the 1987 version of Nqthm contains this form.

To check for possible transmission errors, at least when running under Unix, execute the following checksumming command:

```
% cat *.lisp | sum
```

which should produce the answer **25988 1202**.

Help from the Nqthm-users group. An online mailing list for Nqthm users exists. News about Nqthm, interesting new applications, and advice about how to build and use the system are available via this mailing list. To join the mailing list, send a message to **nqthm-users-request@cli.com**. To send a message to everyone on this list, send a message to **nqthm-users@cli.com**.

14.6. Shakedown

As a very simple-minded test that the compilation and loading are successful, we recommend that you invoke **(BOOT-STRAP NQTHM)** and **(ASSOC-OF-APP)**. The result should be a couple of pages of readable output, printed to the terminal, defining the function **APP** analogously to **APPEND** and proving its associativity. This only takes a few cpu seconds.

As a more thorough check that the theorem prover is running as we expect it to, we recommend that you use **PROVE-FILE-OUT** to process at least one of the many example files provided. A good choice is example file 14 (the file **examples/basic/proveall.events**), which corresponds closely to Appendix A of [5]. To run this particular test follow these two steps.

- Start Nqthm, either by running a save image or by loading Nqthm following the instructions above. Set the variable ***DEFAULT-NQTHM-PATH*** to a string (not a pathname) that names the **examples** directory for your copy of Nqthm. For example, on a Unix system you might execute

```
(SETQ *DEFAULT-NQTHM-PATH*
      "/usr/smith/nqthm-1992/examples/")
```

- Invoke the Lisp form **(PROVE-FILE-OUT "proveall")**.

> Evaluation of this form takes about a quarter of an hour running in AKCL on a Sparc-2, and should return **`T`**. The evaluation should also produce the files **`proveall.proofs`** (about 1.5 megabytes of proofs and definitions), **`proveall.lib`**, **`proveall.lisp`**, a compiled version of **`proveall.lisp`**, and the short text file **`proveall.proved`**, which you may read to check that evaluation was successful.

For a much more exhaustive test of Nqthm, a test representing many person-years of substantial Nqthm use, one may run the entire suite of examples provided on the subdirectories of the directory **`examples`**. To run this set of examples under a Unix for which the **`make`** instruction suggested above worked for Nqthm installation, see the instructions in the file **`makefile`**. On other systems, see the files **`examples/driver.lisp`** and **`examples/README`** for instructions.

14.7. Printing Nqthm Events in Conventional Notation

We provide a utility that will translate a file of Nqthm events into a LaTex [39] or Scribe [19] source document. The utility, which may be found in **`sinfix/sinfix.lisp`**, also converts Nqthm formulas into conventional (''infix'') notation. For example, the Nqthm event

```
(prove-lemma boolean-sample (rewrite)
  (implies (and (lessp a b)
                (lessp b (length c)))
           (zerop (difference a (length c)))))
```

may be displayed as

THEOREM: boolean-sample
$a < b \wedge b < \text{length}\,(c) \rightarrow a - \text{length}\,(c) \cong \mathbf{0}$

when the LaTex or Scribe produced by our utility is processed by the appropriate document preparation facility.

The main utility provided is **`infix-file`**, a Common Lisp routine which takes the name of an Nqthm event file (and some other arguments) and produces a corresponding LaTex or Scribe file. Typically the user then includes this file in some larger LaTex or Scribe document that contains the source text for a paper or presentation.

Because Nqthm event files frequently have Lisp-style comments in them (delimited by semicolons at the left and end-of-line at the right), **`infix-file`** copies comments to the output LaTex or Scribe file, deleting the semicolons. Thus, the user can include text and formatting commands in the event file, but must know which document preparation facility—LaTex or Scribe—will be used to process the text.

Unfortunately, ''conventional'' notation is remarkably unconventional as soon as one begins dealing with non-arithmetic operators (and even with arithmetic operators if one must distinguish, say, Peano addition from rational addition). For example, the user who is making heavy use of the notion of permutations may find it natural to display **`(PERM X Y)`** as ''$x \approx y$'' while another user may be happier with the more conventional ''perm(x, y).'' Thus, our infix utility allows the user to specify how to display given operators. By appropriately loading the tables, sophisticated notation can be obtained from Nqthm events.

Here is a more elaborate example. The text below might be found in an Nqthm event file.

```
; @b<Example 2>: A more complex example showing indenting,
; processing of '!' formatting commands in comments, and
; extensions to the basic formatting functions. The
; functions !qTrains !qbig-delta and !qdelta2
; are defined to print special forms by the 'theory' file.
; The function !qTrains prints as a record selector.
; The constant function !qbig-delta prints as Δ.
; The function !qdelta2 is given a relatively fancy print image.

(defn train-entry (j v a time ss)
  (implies (and (lessp (time (last ss)) time)
                (not (member j (trains (last ss))))
                (lessp (big-delta) (abs (delta2 j a v)))
                (block-not-in-use (car ts) ss))
           (state-seqp (append ss (update-state time j v a)))))
```

When this text is processed by **`infix-file`** and then the result is processed by Scribe the following is produced:

Example 2: A more complex example showing indenting, processing of '!' formatting commands in comments, and extensions to the basic formatting functions. The functions 'trains' 'big-delta' and 'delta2' are defined to print special forms by the 'theory' file. The function 'trains' prints as a record selector. The constant function 'big-delta' prints as Δ. The function 'delta2' is given a relatively fancy print image.

DEFINITION:
train-entry (j, v, a, $time$, ss)
$=$

$$\begin{aligned}
& \text{time}(\text{last}(ss)) < time \\
\land\ & j \notin \text{last}(ss).trains \\
\land\ & \Delta < |\Delta_j^a(v)| \\
\land\ & \text{block-not-in-use}(\text{car}(ts), ss) \\
\rightarrow\ & \text{state-seqp}(ss \mathbin{@} \text{update-state}(time, j, v, a))
\end{aligned}$$

Documentation for **`infix-file`** and its associated routines is provided in **`sinfix/sinfix.lisp`**. The utility was initially written by Robert S. Boyer to produce LaTex output and was then extended by Mike Smith to produce Scribe output as well.

14.8. Example Event Files

We now describe the current contents of the **`examples`** directory. In each entry we give a file name, followed in parentheses by the author(s) and possibly a citation, followed by a very brief description of the work. When no citation is given, no published description of the work is available and the interested reader should look at the **`events`** file itself. Many of the files have explanatory comments. Each file has been successfully processed by **`PROVE-FILE`**. The files are listed in alphabetical order.

1. **`examples/basic/alternating.events`**

   (Boyer) a formalization and correctness proof of the ‘‘Gilbreath Trick’’ [22, 23], a card trick having to do with the outcome of shuffling a deck of cards that has been previously arranged into alternating colors; the Nqthm attack on this problem was inspired by Gerard Huet’s use of the COQ theorem prover to do the proof [28]

2. **`examples/basic/async18.events`**

   (Moore, [43]) a model of asynchronous communication and a proof of the reliability of the biphase mark communications protocol

3. **`examples/basic/binomial.events`**

   (Boyer and Moore, [11]) the binomial theorem expressed with **`FOR`** and a proof thereof

4. **`examples/bronstein/*.events`**

   (Bronstein and Talcott, [15, 16, 17, 18]) a collection of twenty six **`event`** files that are described in the four papers cited above; the work includes a formalization of ‘‘string-functional semantics’’ for circuit descriptions and its use to verify the correctness properties of many circuits, including the Saxe-Leiserson retimed correlator, a pipelined ripple adder, and an abstract pipelined cpu

5. **`examples/basic/controller.events`**

   (Boyer, Moore, and Green, [4]) a model of the problem of controlling a vehicle’s course and a proof that under certain conditions a particular program keeps the vehicle within a certain corridor of the desired course and, under more ideal conditions, homes to the course

6. **`examples/basic/fibsums.events`**

   (Cowles) proofs of several interesting theorems about the sums of Fibonacci numbers

7. **`examples/basic/fortran.events`**

   (Boyer and Moore, [6]) supporting definitions for a Fortran verification condition generator

8. **`examples/basic/fs-examples.events`**

   (Boyer, Goldschlag, Kaufmann, and Moore, [3]) illustrations of the use of constrained functions and functional instantiation

9. **`examples/basic/gauss.events`**

   (Russinoff, [48]) the original Nqthm proof of Gauss’ law of quadratic reciprocity

10. **`examples/basic/new-gauss.events`**

    (Russinoff, [48]) an improved proof of Gauss’ law of quadratic reciprocity (after all, Gauss proved it eight times!)

11. **`examples/basic/parser.events`**

    (Boyer and Moore, Appendix I of this book) a formalization of the syntax and abbreviation conventions of the Nqthm extended logic, expressed as a function from lists of ASCII character codes to the quotations of formal terms

12. **`examples/basic/peter.events`**

    (Boyer) a sequence of lemmas describing the relationship between Ackermann's original function and R. Peter's version of it

13. **`examples/basic/pr.events`**

    (Boyer) a proof of the existence of nonprimitive recursive functions

14. **`examples/basic/proveall.events`**

    (Boyer and Moore, Appendix A of [5]) elementary list processing, number theory through Euclid's theorem and prime factorization, soundness and completeness of a tautology checker, correctness of the **`CANCEL`** metafunction, correctness of a simple assembly language program, correctness of a simple optimizing expression compiler

15. **`examples/basic/quant.events`**

    (Boyer and Moore, [11]) illustrations of the use of **`V&C$`** and **`FOR`**, including a study of several partial functions and functions, such as the ''91 function,'' that recurse on the value of their own recursive calls

16. **`examples/basic/rsa.events`**

    (Boyer and Moore, [8]) proof of the invertibility of the Rivest/Shamir/Adleman public key encryption algorithm

17. **`examples/basic/small-machine.events`**

    (Moore) a simple operational semantics and its use to prove program properties directly and via the so-called ''functional'' and ''inductive assertion'' methods

18. **`examples/basic/tic-tac-toe.events`**

    (Moore) a formalization of what it means for a program to play non-losing tic-tac-toe, the proof that a certain algorithm does so, and the successive refinement of the algorithm into the functional expression of an iterative number-crunching program

19. **`examples/basic/tmi.events`**

    (Boyer and Moore, [9]) proof of the Turing completeness of Pure Lisp

20. **`examples/basic/unsolv.events`**

    (Boyer and Moore, [10]) proof of the unsolvability of the halting problem for Pure Lisp

21. **`examples/basic/wilson.events`**

    (Russinoff, [47]) proof of Wilson's theorem

22. **`examples/basic/ztak.events`**

    (Moore, [41]) proof of the termination of Takeuchi's function

23. **`examples/bevier/kit.events`**

    (Bevier, [1]) the formalization, implementation and proof that a simple separation kernel (implementing multiprocessing on a uniprocessor) provides process scheduling, error handling, message passing, and interfaces to asynchronous devices

24. **`examples/cowles/intro-eg.events`**

    (Cowles) a brief introduction to Nqthm intended for mathematicians and a proof of a theorem about factorial

25. **`examples/cowles/shell.events`**

    (Cowles) alternative ways to decompose sequences and a study of Nqthm's shell principle

26. **`examples/flatau/app-c-d-e.events`**

    (Flatau, [20]) the development and proof of correctness of a compiler and runtime system for a subset of the Nqthm language (including **`IF`**, **`CONS`**, and subroutine call) requiring dynamic storage allocation; this event list corresponds to Appendices C, D, and E of [20] and deals with a runtime system that does not provide a garbage collector.

27. **`examples/flatau/app-f.events`**

    (Flatau, [20]) this event is analogous to example file 26 but corresponds to Appendix F of [20] and deals with a runtime system including a reference counting garbage collector.

28. **`examples/fortran-vcg/fortran.events`**

    (Boyer and Moore, [6]) the same file as **`basic/fortran`**, above, which is duplicated on this subdirectory for technical reasons

29. **`examples/fortran-vcg/fsrch.events`**

    (Boyer and Moore, [6]) proofs of the verification conditions for a Fortran implementation of the Boyer-Moore fast string searching algorithm

30. **`examples/fortran-vcg/isqrt.events`**

    (Boyer and Moore, [4]) proofs of the verification conditions for a Fortran implementation of the integer version of Newton's square root algorithm

31. **`examples/fortran-vcg/mjrty.events`**

    (Boyer and Moore, [12]) proofs of the verification conditions for a Fortran implementation of a linear-time majority vote algorithm

32. **`examples/hunt/fm8501.events`**

    (Hunt, [29]) formalizations of the machine code for the 16-bit FM8501 microprocessor, a register transfer model of a microcoded implementation of the machine, and a proof of their correspondence

33. **`examples/kaufmann/expr-compiler.events`**

    (Kaufmann, see Young [60]) the proof of correctness of a simple expression compiler, designed as an exercise for beginners

34. `examples/kaufmann/foldr.events`

    (Kaufmann) an illustration of a method of proving permutation-independence of list processing functions

35. `examples/kaufmann/generalize-all.events`

    (Kaufmann, [33]) the correctness of a generalization algorithm that operates in the presence of free variables

36. `examples/kaufmann/koenig.events`

    (Kaufmann, [36]) a proof of Koenig's tree lemma

37. `examples/kaufmann/locking.events`

    (Kaufmann, [34]) a model of a simple data base against which read and write transactions can occur

38. `examples/kaufmann/mergesort-demo.events`

    (Kaufmann) the correctness of a merge sort function

39. `examples/kaufmann/note-100.events`

    (Kaufmann, [35]) the proof of Ramsey's theorem for exponent 2, finite case, described in a style intended to assist those wishing to improve their effectiveness with Nqthm

40. `examples/kaufmann/partial.events`

    (Kaufmann) an approach to handling partial functions with Nqthm

41. `examples/kaufmann/permutationp-subbagp.events`

    (Kaufmann) a formalization of the notion of permutation via bags

42. `examples/kaufmann/ramsey.events`

    (Kaufmann, [36]) a proof of Ramsey's theorem for the infinite case

43. `examples/kaufmann/rotate.events`

    (Kaufmann, [32]) a proof about rotations of lists, intended as an introduction to Nqthm

44. **`examples/kaufmann/rpn.events`**

    (Kaufmann and Jamsek) an exercise in reverse Polish notation evaluation

45. **`examples/kaufmann/shuffle.events`**

    (Kaufmann) another solution to the Gilbreath card trick challenge (see example file 1)

46. **`examples/kunen/ack.events`**

    (Kunen) an illustrative definition of Ackermann's function

47. **`examples/kunen/new-prime.events`**

    (Kunen) an alternative proof of the fundamental theorem of arithmetic that—unlike the one presented in [5]—does not use concepts not involved in the statement of the theorem

48. **`examples/numbers/bags.events`**

    (Bevier) a library of useful definitions and lemmas about bags

49. **`examples/numbers/extras.events`**

    (Wilding) a trivial extension of the **`integers`** library used in **`fib2`** below

50. **`examples/numbers/fib2.events`**

    (Wilding, [58]) a proof of Matijasevich's lemma about Fibonacci numbers

51. **`examples/numbers/integers.events`**

    (Bevier, Kaufmann, and Wilding, [45]) a library of useful definitions and lemmas about the integers

52. **`examples/numbers/naturals.events`**

    (Bevier, Kaufmann, and Wilding, [46]) a library of useful definitions and lemmas about the natural numbers

53. **`examples/numbers/nim.events`**

    (Wilding) a formalization of the game of Nim and a proof that a certain algorithm implements a winning strategy

54. **`examples/shankar/church-rosser.events`**

    (Shankar, [51]) a proof of the Church-Rosser theorem for lambda-calculus

55. **`examples/shankar/goedel.events`**

    (Shankar, [50]) a proof of Gödel's incompleteness theorem for Shoenfield's first order logic extended with Cohen's axioms for hereditarily finite set theory, Z2

56. **`examples/shankar/tautology.events`**

    (Shankar, [49]) a proof that every tautology has a proof in Shoenfield's propositional logic

57. **`examples/talcott/mutex-atomic.events`**

    (Nagayama and Talcott, [44]) a proof of the local correctness of a mutual exclusion algorithm under a certain ‘‘atomicity assumption’’

58. **`examples/talcott/mutex-molecular.events`**

    (Nagayama and Talcott, [44]) a proof of the local correctness of a mutual exclusion algorithm without the ‘‘atomicity assumption’’ mentioned above

59. **`examples/yu/amax.events`**

    (Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program that finds the maximum value in an integer array

60. **`examples/yu/asm.events`**

    (Yu, [62]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a trivial C program that uses embedded assembly code (the object being to demonstrate that embedded assembly code can be handled)

61. **`examples/yu/bsearch.events`**

(Yu, [62]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a binary search program written in C

62. **`examples/yu/cstring.events`**

(Yu, [62]) the correctness proofs for the MC68020 machine code produced by the Gnu C compiler for 21 of the 22 C String Library functions from the Berkeley Unix C string library; the proof for each function is broken into two ‘‘phases;’’ the first phase establishes the correspondence of the machine code and a suitable recursive function and the second phase establishes that the recursive function has the specified properties; the file **`examples/yu/cstring.events`** actually contains the second phase proofs for all of the string functions handled; the first phase proof for each string function is contained in a separate **`events`** file named for the string function, e.g., **`examples/yu/memchr.events`**, **`examples/yu/memcmp.events`**, **`examples/yu/strnspn.events`**, etc.

63. **`examples/yu/fixnum-gcd.events`**

(Yu) the correctness proof for the MC68020 machine code produced by the AKCL compiler for a Common Lisp program that computes the greatest common divisor of two **`FIXNUM`**s

64. **`examples/yu/fmax.events`**

(Yu, [62]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a trivial C program to compute the maximum of two integer according to a supplied comparison function (the object being to demonstrate that C ‘‘function pointers’’ are handled)

65. **`examples/yu/gcd.events`**

(Yu, [62]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for Euclid’s greatest common divisor algorithm written in C

66. **`examples/yu/gcd3.events`**

(Yu, [62]) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program consisting of two nested calls of the GCD program (the object being to demonstrate that procedure call is handled in a way that allows hierarchical verification)

67. **`examples/yu/group.events`**

    (Yu, [61]) proofs of two theorems in finite group theory, the first is about kernels of homomorphisms and the second is the Lagrange theorem

68. **`examples/yu/isqrt.events`**

    (Yu) the correctness proof for the MC68020 machine code produced by the GNU C compiler for C program for computing integer square roots via Newton's method

69. **`examples/yu/isqrt-ada.events`**

    (Yu) the correctness proof for the MC68020 machine code produced by the Verdix Ada compiler from an Ada program for computing integer square roots via Newton's method

70. **`examples/yu/log2.events`**

    (Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program for computing integer logarithms (base 2) e.g., repeated division by 2

71. **`examples/yu/mc20-0.events`**

    (Yu, [62]) some utilities involved in the formal specification of the MC68020

72. **`examples/yu/mc20-1.events`**

    (Yu, [14]) the formal specification of about 80% of the user available instructions for the Motorola MC68020 microprocessor

73. **`examples/yu/mc20-2.events`**

    (Yu, [62]) a library of useful definitions and lemmas about the formalization of the MC68020

74. **examples/yu/mjrty.events**

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a linear time majority vote algorithm written in C

75. **examples/yu/qsort.events**

(Yu) the correctness proof for the MC68020 machine code produced by the Gnu C compiler for Hoare's *in situ* quick sort program written in C; the source code is that on page 87 of [38] except that inline code is used rather than the subroutine **swap**; Yu reports (private communication) that this change was made only because he was, at the time, investigating methods of proving recursive programs correct and did not want to be distracted by other subroutine calls

76. **examples/yu/switch.events**

(Yu, [62]) the correctness proof for the MC68020 machine code produced by the GNU C compiler for a trivial C program that employs the **switch** statement (the object being to demonstrate the technique used to compile **switch** can be handled)

77. **examples/yu/zero.events**

(Yu) the correctness proof for the MC68020 machine code produced by the GNU C compiler for a C program that zeros an integer array (the object being to demonstrate that writes to memory can be handled)

14.9. Infix

The file **infix.lisp**, which is distributed with the Nqthm sources, is a pretty printer that produces Latex output for Nqthm formulas. The formulas produced are in conventional mathematical notation rather than in the Lisp-like syntax of Nqthm. See the comments at the beginning of that file for instructions.

14.10. DEFN-SK

The file **`defn-sk.lisp`**, which is distributed with the Nqthm sources, implements a *skolemizer*. Via this facility, one can handle first order logic quantifiers within Nqthm. See Kaufmann's paper [36] for further details. See the beginning of the file **`defn-sk.lisp`** for instructions. This facility is the work of Matt Kaufmann, and he has incorporated it into his Pc-Nqthm system. We include **`defn-sk.lisp`** with the distribution of Nqthm because it is used in the following example event files in subdirectory **`examples/yu`**: **`cstring`**, **`qsort`**, and **`group`**.

Appendix I

A Parser for the Syntax

Below we reproduce verbatim example file 11 in which we formalize the extended syntax in the logic itself. We offer this event list as

- an alternative equivalent specification of the extended syntax,
- an example of formalization in the logic,
- an example of a file suitable for processing by **`PROVE-FILE`**, and
- an example of the tactics used to admit complicated definitions.

Furthermore, the data base created by processing this file allows the Nqthm user to experiment with the syntax, i.e., to obtain the formal term denoted by the display of any well-formed term in the extended syntax.

All of the text below is new to the second edition.

```
; The Formalized Extended Syntax


; SECTION: Introduction

; This file, /examples/basic/parser.events, is a formalization of
; the extended syntax as presented in Chapter 4.  We formalize the
; syntax in the logic itself.  Roughly speaking, for every occur-
; rence of ‘‘Terminology’’ in Chapter 4 there is an admissible
; function definition here that formalizes the concept defined in
; Chapter 4.  For example, we formally define what it is to be a
```

```
; ''numeric sequence'' by introducing a function which returns T or
; F according to whether its argument is such a sequence.

; The outline of our presentation is as follows.  We first develop
; the notion of a ''token tree.''  Token trees in the logic are re-
; presented by integers, literal atoms, and list structures.  The
; text in Chapter 4 then develops the idea of how to ''display'' a
; token tree.  Here we do the inverse: we formalize the notion of
; how to parse a token tree from a sequence of ASCII character
; codes.  This culminates in the definition of READ-TOKEN-TREE
; which takes a list of character codes and returns either F (in-
; dicating that the input sequence is unparsable) or a token tree.
; We then return to the text of Chapter 4 and formalize what it is
; for a token tree to be ''readable,'' what ''readmacro-expansion''
; is, and what an ''s-expression'' is.  This part of the formal-
; ization may illuminate backquote notation for those unfamiliar
; with it.  We finally define what it is for an s-expression to be
; ''well-formed'' and what the ''translation'' of such an
; s-expression is.

; This presentation is meant to clarify the extended syntax.  Thus,
; it would hardly do to express the formalization in terms of the
; extended syntax.  We therefore largely confine ourselves here to
; the formal syntax with the following exceptions:

; 1. We permit ourselves to write natural numbers, e.g., 2, in
;    place of their formal equivalents, e.g., (ADD1 (ADD1 (ZERO))).

; 2. We permit ourselves to use QUOTE notation to denote literal
;    atom constants.  Thus, we write 'ABC where
;      (PACK (CONS 65 (CONS 66 (CONS 67 0))))
;    would otherwise be needed.

; 3. We occasionally display list constants in QUOTE notation,
;    e.g., '(UPPER-B UPPER-O UPPER-X) is written in place of
;    (CONS 'UPPER-B (CONS 'UPPER-O (CONS 'UPPER-X 'NIL))).  Two
;    large association lists are displayed with QUOTE notation.
;    These displays are in conjunction with the definitions of
;    utility functions whose semantics is intuitively clear, namely
;    a function that maps from our name for an ASCII character to
;    its ASCII code and a function that maps from the name of a
;    function to its arity.

; 4. We permit ourselves to write comments.

; It should be stressed that these are ''exceptions'' only in the
; sense that they are not part of the formal syntax.  These con-
; ventions are entirely supported by the extended syntax and we
; note their use only because they are exceptions to our self-
; imposed restriction to the formal syntax in this file.

; We have followed the text's style of the definition very closely.
; This results in a somewhat ''inefficient'' formalization.  For
```

```
; example, it is possible to implement the parsing/readmacro-expan-
; sion process in a single pass, but we have not done so.  While we
; feel our use of the logic here is illustrative of ''good usage''
; the critical reader must keep in mind that we are trying to
; formalize the ideas as presented in the text and not merely code
; a parser in the logic.

; This file may be processed with PROVE-FILE and the resulting
; library may be noted and used in R-LOOP to experiment with the
; syntax.  We recommend that the library be compiled.  If the Nqthm
; installer has so processed the file, the library file may already
; exist as /examples/basic/parser.  This source file also contains
; ''commented out'' Common Lisp code that permits more convenient
; experimentation.  Search for occurrences of ''defun'' to find the
; two regions in question.

; Finally, this file serves as a good example of a fairly substan-
; tial Nqthm formalization effort.  We recommend it even to readers
; who know the syntax but who wish to see how Nqthm is used to
; formalize ideas that are already precisely (but informally)
; understood.  We urge such readers to compare the formal defini-
; tions with the corresponding informal definitions of Chapter 4.

; We start by initializing Nqthm's data base to the GROUND-ZERO
; theory.

(BOOT-STRAP NQTHM)


; SECTION: Conventions Concerning Characters

; In the text we imagine that we have integers, characters and
; sequences of characters as our atoms.  We then build token trees
; as sequences of these atoms and other token trees.  Thus, the
; sequence consisting of 65, 66, and 67 is uniquely recognized as
; being a sequence of integers, while the sequence consisting of
; the characters A, B, and C, is recognized as being a word.

; But in this formalization, we do not have characters and in fact
; we use their ASCII codes instead.  But the ASCII codes are them-
; selves integers.  Thus, the sequence containing 65, 66, and 67 is
; ambiguous as a token tree: is it a tree containing three integers
; or is it a tree containing the word ABC?  Fortunately, characters
; never enter token trees except as the elements of words and the
; various tokens.  Therefore, in this formalization we use LITATOMs
; to represent the words and the tokens.  That is, where the text
; represents the word ABC as a sequence consisting of the char-
; acters A, B, and C, the formalization will represent it as a
; LITATOM obtained by PACKing the ASCII codes for A, B, and C
; (around a 0).  Similarly, the backquote token in the text is the
; sequence consisting of the backquote character, but here it is
; the LITATOM obtained by packing the ASCII code for that character
; (around a 0).
```

```
; We first make it convenient to refer to the ASCII character code
; of all the printable characters.  We will invent a name for each
; character, e.g., UPPER-A, LOWER-B, OPEN-PAREN, and define (ASCII
; name) to be the ASCII integer code for the named character.  Thus
; (ASCII 'UPPER-A) will be 65.  We will also define ASCII so that
; when given a list of names it returns the list of ASCII codes for
; the characters named in the list.  Thus, (ASCII '(UPPER-A
; LOWER-A)) will be '(65 97).

; The names and their ASCII codes are given in the following
; association list:
(DEFN ASCII-TABLE NIL
  '((TAB                . 9)
    (NEWLINE            . 10)
    (PAGE               . 12)
    (SPACE              . 32)
    (RUBOUT             . 127)
    (EXCLAMATION-POINT  . 33)           ; !
    (DOUBLE-QUOTE       . 34)           ; "
    (NUMBER-SIGN        . 35)           ; #
    (DOLLAR-SIGN        . 36)           ; $
    (PERCENT-SIGN       . 37)           ; %
    (AMPERSAND          . 38)           ; &
    (SINGLE-QUOTE       . 39)           ; '
    (OPEN-PAREN         . 40)           ; (
    (CLOSE-PAREN        . 41)           ; )
    (ASTERISK           . 42)           ; *
    (PLUS-SIGN          . 43)           ; +
    (COMMA              . 44)           ; ,
    (MINUS-SIGN         . 45)           ; -
    (DOT                . 46)           ; .
    (SLASH              . 47)           ; /
    (DIGIT-ZERO         . 48)           ; 0
    (DIGIT-ONE          . 49)           ; 1
    (DIGIT-TWO          . 50)           ; 2
    (DIGIT-THREE        . 51)           ; 3
    (DIGIT-FOUR         . 52)           ; 4
    (DIGIT-FIVE         . 53)           ; 5
    (DIGIT-SIX          . 54)           ; 6
    (DIGIT-SEVEN        . 55)           ; 7
    (DIGIT-EIGHT        . 56)           ; 8
    (DIGIT-NINE         . 57)           ; 9
    (COLON              . 58)           ; :
    (SEMICOLON          . 59)           ; ;
    (LESS-THAN-SIGN     . 60)           ; <
    (EQUAL-SIGN         . 61)           ; =
    (GREATER-THAN-SIGN  . 62)           ; >
    (QUESTION-MARK      . 63)           ; ?
    (AT-SIGN            . 64)           ; @
    (UPPER-A            . 65)           ; A
    (UPPER-B            . 66)           ; B
    (UPPER-C            . 67)           ; C
    (UPPER-D            . 68)           ; D
```

```
(UPPER-E          . 69)          ; E
(UPPER-F          . 70)          ; F
(UPPER-G          . 71)          ; G
(UPPER-H          . 72)          ; H
(UPPER-I          . 73)          ; I
(UPPER-J          . 74)          ; J
(UPPER-K          . 75)          ; K
(UPPER-L          . 76)          ; L
(UPPER-M          . 77)          ; M
(UPPER-N          . 78)          ; N
(UPPER-O          . 79)          ; O
(UPPER-P          . 80)          ; P
(UPPER-Q          . 81)          ; Q
(UPPER-R          . 82)          ; R
(UPPER-S          . 83)          ; S
(UPPER-T          . 84)          ; T
(UPPER-U          . 85)          ; U
(UPPER-V          . 86)          ; V
(UPPER-W          . 87)          ; W
(UPPER-X          . 88)          ; X
(UPPER-Y          . 89)          ; Y
(UPPER-Z          . 90)          ; Z
(OPEN-BRACKET     . 91)          ; [
(BACKSLASH        . 92)          ; \
(CLOSE-BRACKET    . 93)          ; ]
(UPARROW          . 94)          ; ^
(UNDERSCORE       . 95)          ; _
(BACKQUOTE        . 96)          ; `
(LOWER-A          . 97)          ; a
(LOWER-B          . 98)          ; b
(LOWER-C          . 99)          ; c
(LOWER-D          . 100)         ; d
(LOWER-E          . 101)         ; e
(LOWER-F          . 102)         ; f
(LOWER-G          . 103)         ; g
(LOWER-H          . 104)         ; h
(LOWER-I          . 105)         ; i
(LOWER-J          . 106)         ; j
(LOWER-K          . 107)         ; k
(LOWER-L          . 108)         ; l
(LOWER-M          . 109)         ; m
(LOWER-N          . 110)         ; n
(LOWER-O          . 111)         ; o
(LOWER-P          . 112)         ; p
(LOWER-Q          . 113)         ; q
(LOWER-R          . 114)         ; r
(LOWER-S          . 115)         ; s
(LOWER-T          . 116)         ; t
(LOWER-U          . 117)         ; u
(LOWER-V          . 118)         ; v
(LOWER-W          . 119)         ; w
(LOWER-X          . 120)         ; x
(LOWER-Y          . 121)         ; y
```

```
    (LOWER-Z            . 122)          ; z
    (OPEN-BRACE         . 123)          ; {
    (VERTICAL-BAR       . 124)          ; |
    (CLOSE-BRACE        . 125)          ; }
    (TILDE              . 126)))        ; ~
(DEFN ASCII-LIST (LST)
  (IF (NLISTP LST)
      NIL
      (CONS (CDR (ASSOC (CAR LST) (ASCII-TABLE)))
            (ASCII-LIST (CDR LST)))))
(DEFN ASCII (X)
  (IF (LITATOM X)
      (CDR (ASSOC X (ASCII-TABLE)))
      (ASCII-LIST X)))

; So now we can write (ASCII 'UPPER-A) for 65 and
; (ASCII '(UPPER-A LOWER-A)) for '(65 97).


; SECTION: Token Trees
(DEFN UPPER-DIGITS NIL
  (ASCII '(DIGIT-ZERO
           DIGIT-ONE
           DIGIT-TWO
           DIGIT-THREE
           DIGIT-FOUR
           DIGIT-FIVE
           DIGIT-SIX
           DIGIT-SEVEN
           DIGIT-EIGHT
           DIGIT-NINE
           UPPER-A
           UPPER-B
           UPPER-C
           UPPER-D
           UPPER-E
           UPPER-F)))

; We define (CADRN n LST) to be equivalent to (CADD...DR LST) where
; the number of D's is n.  Thus, (CADRN 2 LST) is (CADDR LST).  An-
; other way to think about CADRN is that it returns the Nth element
; of LST using 0-based enumeration, e.g., (CADRN 3 '(A B C D E F))
; is 'D.
(DEFN CDRN (N LST)
  (IF (ZEROP N)
      LST
      (CDRN (SUB1 N) (CDR LST))))
(DEFN CADRN (N LST) (CAR (CDRN N LST)))

; It is also convenient to have
(DEFN LIST1 (X) (CONS X NIL))
(DEFN LIST2 (X Y) (CONS X (LIST1 Y)))
(DEFN LIST3 (X Y Z) (CONS X (LIST2 Y Z)))
```

```
; because we are eschewing the use of the abbreviation LIST.
(DEFN FIRST-N (N LST)
  (IF (ZEROP N)
      NIL
      (CONS (CAR LST)
            (FIRST-N (SUB1 N) (CDR LST)))))
(DEFN BASE-N-DIGIT-CHARACTER (N C)
  (AND (LEQ N 16)
       (MEMBER C (FIRST-N N (UPPER-DIGITS)))))
(DEFN POSITION (X LST)
  (IF (NLISTP LST)
      0
      (IF (EQUAL X (CAR LST))
          0
          (ADD1 (POSITION X (CDR LST))))))
(DEFN BASE-N-DIGIT-VALUE (C)
  (POSITION C (UPPER-DIGITS)))
(DEFN ALL-BASE-N-DIGIT-CHARACTERS (N LST)
  (IF (NLISTP LST)
      T
      (AND (BASE-N-DIGIT-CHARACTER N (CAR LST))
           (ALL-BASE-N-DIGIT-CHARACTERS N (CDR LST)))))
(DEFN BASE-N-DIGIT-SEQUENCE (N LST)
  (AND (LISTP LST)
       (ALL-BASE-N-DIGIT-CHARACTERS N LST)))
(DEFN OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE (N LST)
  (OR (BASE-N-DIGIT-SEQUENCE N LST)
      (AND (LISTP LST)
       (AND (OR (EQUAL (CAR LST) (ASCII 'PLUS-SIGN))
                (EQUAL (CAR LST) (ASCII 'MINUS-SIGN)))
            (BASE-N-DIGIT-SEQUENCE N (CDR LST))))))
(DEFN LENGTH (LST)
  (IF (NLISTP LST)
      0
      (ADD1 (LENGTH (CDR LST)))))
(DEFN EXP (I J)
  (IF (ZEROP J)
      1
      (TIMES I (EXP I (SUB1 J)))))
(DEFN BASE-N-VALUE (N LST)
  (IF (NLISTP LST)
      0
      (PLUS (TIMES (BASE-N-DIGIT-VALUE (CAR LST))
                   (EXP N (LENGTH (CDR LST))))
            (BASE-N-VALUE N (CDR LST)))))
(DEFN NUMERATOR-SEQUENCE (LST)
  (IF (NLISTP LST)
      NIL
      (IF (EQUAL (CAR LST) (ASCII 'SLASH))
          NIL
          (CONS (CAR LST)
                (NUMERATOR-SEQUENCE (CDR LST))))))
(DEFN DENOMINATOR-SEQUENCE (LST)
```

```
  (IF (NLISTP LST)
      NIL
      (IF (EQUAL (CAR LST) (ASCII 'SLASH))
          (CDR LST)
          (DENOMINATOR-SEQUENCE (CDR LST)))))
(DEFN BASE-N-SIGNED-VALUE (N LST)
  (IF (EQUAL (CAR LST) (ASCII 'MINUS-SIGN))
      (MINUS (BASE-N-VALUE N (CDR LST)))
      (IF (EQUAL (CAR LST) (ASCII 'PLUS-SIGN))
          (BASE-N-VALUE N (CDR LST))
          (BASE-N-VALUE N LST))))
(DEFN NUMBER-SIGN-SEQUENCE (LST)
  (AND (GEQ (LENGTH LST) 3)
       (EQUAL (CAR LST) (ASCII 'NUMBER-SIGN))
       (MEMBER (CADRN 1 LST)
               (ASCII '(UPPER-B UPPER-O UPPER-X)))
       (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE
        (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-B))
            2
            (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-O))
                8
                16))
        (CDR (CDR LST)))))
(DEFN LAST (LST)
  (IF (NLISTP LST)
      LST
      (IF (NLISTP (CDR LST))
          LST
          (LAST (CDR LST)))))
(DEFN ALL-BUT-LAST (LST)
  (IF (NLISTP LST)
      NIL
      (IF (NLISTP (CDR LST))
          NIL
          (CONS (CAR LST)
                (ALL-BUT-LAST (CDR LST))))))
(DEFN NUMERIC-SEQUENCE (LST)
  (OR (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE 10 LST)
      (OR (AND (EQUAL (CAR (LAST LST)) (ASCII 'DOT))
               (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE
                10
                (ALL-BUT-LAST LST)))
          (NUMBER-SIGN-SEQUENCE LST))))
           (DEFN NUMERIC-VALUE (LST)
  (IF (OPTIONALLY-SIGNED-BASE-N-DIGIT-SEQUENCE 10 LST)
      (BASE-N-SIGNED-VALUE 10 LST)
      (IF (EQUAL (CAR (LAST LST)) (ASCII 'DOT))
          (BASE-N-SIGNED-VALUE 10 (ALL-BUT-LAST LST))
          (BASE-N-SIGNED-VALUE
           (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-B))
               2
               (IF (EQUAL (CADRN 1 LST) (ASCII 'UPPER-O))
                   8
```

```
                   16))
           (CDR (CDR LST))))))
(DEFN SINGLE-QUOTE-TOKEN  () (PACK (CONS (ASCII 'SINGLE-QUOTE) 0)))
(DEFN BACKQUOTE-TOKEN     () (PACK (CONS (ASCII 'BACKQUOTE) 0)))
(DEFN DOT-TOKEN           () (PACK (CONS (ASCII 'DOT) 0)))
(DEFN COMMA-TOKEN         () (PACK (CONS (ASCII 'COMMA) 0)))
(DEFN COMMA-AT-SIGN-TOKEN () (PACK (CONS (ASCII 'COMMA)
                                    (CONS (ASCII 'AT-SIGN) 0))))
(DEFN COMMA-DOT-TOKEN     () (PACK (CONS (ASCII 'COMMA)
                                    (CONS (ASCII 'DOT) 0))))
(DEFN WORD-CHARACTERS ()
  (ASCII
   '(UPPER-A UPPER-B UPPER-C UPPER-D UPPER-E UPPER-F UPPER-G
     UPPER-H UPPER-I UPPER-J UPPER-K UPPER-L UPPER-M UPPER-N
     UPPER-O UPPER-P UPPER-Q UPPER-R UPPER-S UPPER-T UPPER-U
     UPPER-V UPPER-W UPPER-X UPPER-Y UPPER-Z
     DIGIT-ZERO DIGIT-ONE DIGIT-TWO DIGIT-THREE DIGIT-FOUR
     DIGIT-FIVE DIGIT-SIX DIGIT-SEVEN DIGIT-EIGHT DIGIT-NINE
     DOLLAR-SIGN UPARROW AMPERSAND ASTERISK UNDERSCORE
     MINUS-SIGN PLUS-SIGN EQUAL-SIGN TILDE OPEN-BRACE CLOSE-BRACE
     QUESTION-MARK LESS-THAN-SIGN GREATER-THAN-SIGN)))
(DEFN SUBSETP (X Y)
  (IF (NLISTP X)
      T
      (IF (MEMBER (CAR X) Y)
          (SUBSETP (CDR X) Y)
          F)))
(DEFN WORD (S)
  (AND (LITATOM S)
       (OR (NUMERIC-SEQUENCE (UNPACK S))
           (AND (LISTP (UNPACK S))
                (SUBSETP (UNPACK S) (WORD-CHARACTERS))))))

; It is convenient to be able to recognize those words that are
; numeric sequences and to talk about their numeric values, without
; having to think about unpacking them.  So we define NUMERIC-WORD
; and NUMERIC-WORD-VALUE and use them below where the text would
; have us use ''numeric sequence'' and ''numeric value.''
(DEFN NUMERIC-WORD (S)
  (AND (LITATOM S)
       (NUMERIC-SEQUENCE (UNPACK S))))
(DEFN NUMERIC-WORD-VALUE (S)
  (NUMERIC-VALUE (UNPACK S)))
(DEFN INTEGERP (X)
  (OR (NUMBERP X)
      (NEGATIVEP X)))
(DEFN SPECIAL-TOKEN (X)
  (OR (EQUAL X (SINGLE-QUOTE-TOKEN))
   (OR (EQUAL X (BACKQUOTE-TOKEN))
    (OR (EQUAL X (COMMA-TOKEN))
     (OR (EQUAL X (COMMA-AT-SIGN-TOKEN))
         (EQUAL X (COMMA-DOT-TOKEN)))))))
```

```
; The following function can be used as follows.  If we test
; (EQLEN X 3) then we know that lst is of the form (x1 x2 x3).
(DEFN EQLEN (LST N)
  (IF (ZEROP N)
      (EQUAL LST NIL)
      (IF (NLISTP LST)
          F
          (EQLEN (CDR LST) (SUB1 N)))))
(DEFN DOTTED-PAIR (X)
  (AND (EQLEN X 3)
       (EQUAL (CADRN 1 X) (DOT-TOKEN))))
(DEFN DOTTED-S-EXPRESSION (X)
  (IF (NLISTP X)
      F
      (IF (DOTTED-PAIR X)
          T
          (DOTTED-S-EXPRESSION (CDR X)))))
(DEFN SINGLETON (X)
  (EQLEN X 1))

; We use the following lemma to make the CADRNs go away during the
; termination proof for TOKEN-TREE.

(PROVE-LEMMA CDRN-EXPANDER (REWRITE)
  (EQUAL (CDRN (ADD1 N) X) (CDRN N (CDR X))))

; Now we define token trees formally.  We use the logic's LISTPs,
; together with the integers, words and tokens (the last two being
; LITATOMs), to represent token trees.  We are faithful to the
; text's style of representing dotted token trees by lists whose
; second-to-last elements are the dot token.  That is, we don't
; represent such trees by dotted pairs in the logic.  (This would
; not work because the token tree (A B . (C D E)) is legitimate and
; is different from (A B C D E).  The text permits such token trees
; since they may be typed.  They readmacro-expand to the same
; s-expression.)  However, we do not define ''token tree'' in quite
; the same style as the text, though the result is the same.  The
; text recognizes undotted and dotted token trees ''from the
; outside'' -- e.g., an undotted token tree is a nonempty sequence
; of token trees and a dotted one is similar but has a dot as its
; second-to-last element.  To define token trees formally this way
; we would need to use mutual recursion.  Rather than do that, we
; just recurse down dotted and undotted lists and catch the dot
; when we come to it.
(DEFN TOKEN-TREE (X)
  (IF (NLISTP X)
      (OR (INTEGERP X)
          (WORD X))
  (IF (AND (SPECIAL-TOKEN (CAR X))
           (EQUAL (LENGTH X) 2))
      (TOKEN-TREE (CADRN 1 X))
  (IF (DOTTED-PAIR X)
      (AND (TOKEN-TREE (CAR X))
```

```
          (TOKEN-TREE (CADRN 2 X)))
  (IF (SINGLETON X)
      (TOKEN-TREE (CAR X))
      (AND (TOKEN-TREE (CAR X))
           (TOKEN-TREE (CDR X)))))))

; We define a few functions to make token tree manipulation and
; recognition easier.
(DEFN SPECIAL-TOKEN-TREE (X)
  (SPECIAL-TOKEN (CAR X)))
(DEFN SINGLE-QUOTE-TOKEN-TREE (X)
  (EQUAL (CAR X) (SINGLE-QUOTE-TOKEN)))
(DEFN BACKQUOTE-TOKEN-TREE (X)
  (EQUAL (CAR X) (BACKQUOTE-TOKEN)))
(DEFN COMMA-ESCAPE-TOKEN-TREE (X)
  (EQUAL (CAR X) (COMMA-TOKEN)))
(DEFN SPLICE-ESCAPE-TOKEN-TREE (X)
  (OR (EQUAL (CAR X) (COMMA-AT-SIGN-TOKEN))
      (EQUAL (CAR X) (COMMA-DOT-TOKEN))))
(DEFN CONSTITUENT (X) (CADRN 1 X))


; SECTION:  Reading Token Trees

; At this point in the text, we define how to display a token tree.
; We will skip that here and instead formalize the process of pars-
; ing a token tree from a sequence of characters (i.e., from a dis-
; play of one).

; Our parser is reminiscent of Lisp's read routine with several im-
; portant exceptions.  We do not deal with readmacros in our parser
; -- that is the job of readmacro-expansion (defined later).  We
; also do not produce dotted pairs but rather lists containing the
; dot token.  Finally, we can produce ''unreadable'' token trees
; such as parsed from (PLUS ,X B).

; The parser is called READ-TOKEN-TREE and it is implemented in two
; passes.  The first pass transforms a list of ASCII character
; codes into a list of lexemes.  In our case, all the lexemes are
; LITATOMs, including the numeric sequences that look like inte-
; gers, i.e., we resolve the ambiguity in ''display'' by always
; using numeric sequences.  The second pass parses the lexemes into
; a tree.  In fact, the parser can produce trees that are not token
; trees, such as the one produced by parsing (PLUS # B).  This
; ''pseudo-token tree'' fails to be a token tree because one of the
; atoms in it that ''should'' be a word is not a word.  READ-TOKEN-
; TREE therefore calls the predicate TOKEN-TREE on the parsed tree
; to determine whether the parse was successful.


; SUBSECTION: Pass I of the Reader

; We give convenient names to the lexemes for open and close
```

```
; parentheses.  We could call these ''tokens'' but don't want to
; confuse them with the tokens of the text.  The lexemes for the
; tokens, e.g., the dot lexeme, will be the tokens themselves,
; e.g., the value of (DOT-TOKEN).
(DEFN OPEN-PAREN   () (PACK (CONS (ASCII 'OPEN-PAREN) 0)))
(DEFN CLOSE-PAREN  () (PACK (CONS (ASCII 'CLOSE-PAREN) 0)))

; The following function scans past characters until it has passed
; the first newline character.  It is used to scan past semicolon
; comment.  It returns the stream starting just after that newline.
; If no newline is found, it returns the empty stream.  The ques-
; tion then arises: was the comment that started this scan well-
; formed or not?  It is missing its final newline.  It turns out
; that Common Lisp's answer to this is that it is ok.  That is, a
; file can end without there being a final newline on the end of a
; comment on the last line of the file.  The text of Chapter 4 does
; not deal with this issue.
(DEFN SKIP-PAST-NEWLINE (STREAM)
  (IF (NLISTP STREAM)
      STREAM
      (IF (EQUAL (CAR STREAM) (ASCII 'NEWLINE))
          (CDR STREAM)
          (SKIP-PAST-NEWLINE (CDR STREAM)))))

; The lexical analyzer will sometimes recurse using this function
; to ''decrement'' the input stream.  We must prove that the stream
; returned is weakly smaller (the semicolon that starts the
; comment will have already been stripped off by an explicit CDR).
; We measure size here with LENGTH instead of COUNT because COUNT
; doesn't decrease for all of the recursions (see the next function
; definition).

(PROVE-LEMMA LESSP-SKIP-PAST-NEWLINE (REWRITE)
  (NOT (LESSP (LENGTH STREAM)
              (LENGTH (SKIP-PAST-NEWLINE STREAM)))))

; The next function scans for the end of a just-opened #-comment.
; The function returns the stream just past the closing sequence of
; the comment.  The variable I below counts the number of ''open''
; #| sequences we have seen.  We do not stop until we see a |#
; sequence that closes that one, i.e., decrements I to 0.  I is
; initially 1 because we call this function after reading the
; opening sequence.  Note that this function will scan the entire
; input stream if the balancing sequence is not present.  This is
; treated as an error by Common Lisp.  We therefore have the
; problem of signalling this error.  If we return the empty stream,
; it is indistinguishable from a well-formed comment that ends at
; the very end of the stream.  We therefore use a trick: we return
; the stream containing a single open parenthesis character.  This
; character will cause the lexical analyzer to put an unbalanced
; open parenthesis lexeme as the last lexeme in the stream fed to
; the parser.  That, in turn, will cause an error.  We also signal
; such an error if we find too many |# sequences.  A minor tech-
```

```
; nical problem arises: to insure that the lexical analyzer can
; recurse with this skipping function, we have to make sure the
; size of the stream we return is less than the one the lexical
; analyzer started with.  Since the analyzer will have CDRd past
; the opening #| our adding a single open parenthesis in the place
; of those two characters will not matter -- if we measure the size
; of the stream with LENGTH!  If we were to measure with COUNT, we
; would have to worry about the size of the various ASCII codes and
; about the object in the final CDR of the stream.
(DEFN SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN (STREAM I)
  (IF (NLISTP STREAM)
      (CONS (ASCII 'OPEN-PAREN) STREAM)
  (IF (AND (EQUAL (CAR STREAM) (ASCII 'NUMBER-SIGN))
           (EQUAL (CADRN 1 STREAM) (ASCII 'VERTICAL-BAR)))
      (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
       (CDRN 2 STREAM)
       (ADD1 I))
  (IF (AND (EQUAL (CAR STREAM) (ASCII 'VERTICAL-BAR))
           (EQUAL (CADRN 1 STREAM) (ASCII 'NUMBER-SIGN)))
      (IF (EQUAL I 0)
          (CONS (ASCII 'OPEN-PAREN) (CDRN 2 STREAM))
          (IF (EQUAL I 1)
              (CDRN 2 STREAM)
              (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
               (CDRN 2 STREAM)
               (SUB1 I))))
      (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
       (CDR STREAM)
       I)))))

; This is the key inductive fact about the relative LENGTHs of the
; input and output of the function above:

(PROVE-LEMMA SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN-LEMMA
  (REWRITE)
  (NOT (LESSP
        (ADD1 (LENGTH STREAM))
        (LENGTH (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
                 STREAM
                 I)))))

; However, this is the actual theorem we need to admit the lexical
; analyzer (eventually) below:

(PROVE-LEMMA LESSP-SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
  (REWRITE)
  (IMPLIES (AND (LISTP STREAM)
                (AND (EQUAL (CAR STREAM) (ASCII 'NUMBER-SIGN))
                     (EQUAL (CADR STREAM) (ASCII 'VERTICAL-BAR))))
           (LESSP
            (LENGTH
             (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
              (CDDR STREAM)
```

```
             1))
           (LENGTH STREAM))))

; To recognize white space we need:
(DEFN WHITE-SPACEP (C)
  (MEMBER C (ASCII '(SPACE TAB NEWLINE))))

; We will accumulate the characters in a lexeme in a list (with a 0
; at the bottom) and when we have a completed lexeme we will add it
; to a growing list of lexemes.  The characters are accumulated in
; reverse order.  The empty lexeme is never added to the list.
(DEFN REV1 (LST ANS)
  (IF (NLISTP LST)
      ANS
      (REV1 (CDR LST) (CONS (CAR LST) ANS))))
(DEFN EMIT (PNAME LST)
  (IF (EQUAL PNAME 0)
      LST
      (CONS (PACK (REV1 PNAME 0)) LST)))

; The following function maps lower case alphabetic characters into
; their upper case counterparts.  Thus, (UPCASE (ASCII 'LOWER-A))
; is (ASCII 'UPPER-A), etc.  UPCASE is the identity function on
; characters other than LOWER-A through LOWER-Z.
(DEFN UPCASE (C)
  (IF (AND (LEQ (ASCII 'LOWER-A) C)
           (LEQ C (ASCII 'LOWER-Z)))
      (DIFFERENCE C (DIFFERENCE (ASCII 'LOWER-A) (ASCII 'UPPER-A)))
      C))

; Here then is the lexical analyzer, pass I of the parser.  The
; function returns a list of lexemes parsed from stream, which is a
; list of ASCII codes.  PNAME is the lexeme currently being
; assembled.  It is 0 when ``empty.''
(DEFN LEXEMES (STREAM PNAME)
  (IF (NLISTP STREAM)
      (EMIT PNAME NIL)
  (IF (EQUAL (CAR STREAM) (ASCII 'SEMICOLON))

; A semicolon terminates the accumulating lexeme and causes us to
; skip to the next line.

      (EMIT PNAME
            (LEXEMES (SKIP-PAST-NEWLINE (CDR STREAM))
                     0))

; Backquote, single quote, and the two parens terminate the lexeme
; and also emit the corresponding token or lexeme.

  (IF (MEMBER (CAR STREAM)
              (ASCII '(BACKQUOTE
                       SINGLE-QUOTE
                       OPEN-PAREN
```

```
                       CLOSE-PAREN)))
      (EMIT PNAME
            (EMIT (CONS (CAR STREAM) 0)
                  (LEXEMES (CDR STREAM) 0)))

  (IF (EQUAL (CAR STREAM) (ASCII 'COMMA))

; Comma terminates the lexeme but emits the comma, comma at-sign,
; or comma dot token, depending on the next character.

      (IF (OR (EQUAL (CADRN 1 STREAM) (ASCII 'AT-SIGN))
              (EQUAL (CADRN 1 STREAM) (ASCII 'DOT)))
          (EMIT PNAME
                (EMIT (CONS (CADRN 1 STREAM) (CONS (CAR STREAM) 0))
                      (LEXEMES (CDR (CDR STREAM)) 0)))
          (EMIT PNAME
                (EMIT (CONS (CAR STREAM) 0)
                      (LEXEMES (CDR STREAM) 0))))

  (IF (WHITE-SPACEP (CAR STREAM))

; White space terminates the lexeme and emits nothing.

      (EMIT PNAME (LEXEMES (CDR STREAM) 0))

  (IF (AND (EQUAL PNAME 0)
       (AND (EQUAL (CAR STREAM) (ASCII 'NUMBER-SIGN))
            (EQUAL (CADRN 1 STREAM) (ASCII 'VERTICAL-BAR))))

; Finally, number-sign does not terminate the lexeme, but if it
; occurs while no lexeme is being accumulated and is immediately
; followed by vertical bar, it signals the beginning of a comment.

      (LEXEMES (SKIP-PAST-BALANCING-VERTICAL-BAR-NUMBER-SIGN
                (CDRN 2 STREAM)
                1)
               0)

; Otherwise, we just accumulate the character onto the lexeme being
; assembled.  This case includes DOT, which is not a break char-
; acter and thus must appear in isolation to be parsed as a dot
; token.

      (LEXEMES (CDR STREAM)
               (CONS (UPCASE (CAR STREAM)) PNAME)))))))
  ((LESSP (LENGTH STREAM))))

; SUBSECTION:  Pass II of the Reader

; We now develop the parser, which maps a list of lexemes into a
; tree -- or returns F if the list does not parse.  One example of
; a list of lexemes that doesn't parse is one containing unbalanced
; parentheses.  Another example is one that contains more lexemes
```

```
; than necessary to describe exactly one tree.  That is, our parser
; balks if there are lexemes left over when one tree has been
; parsed.

; The formalization below is just a simple push-down stack auto-
; maton.  It scans the lexemes one at a time from the right all the
; way to the end of the list.  It has a stack of trees it is build-
; ing up.  When it sees an open parenthesis, it pushes an empty
; frame onto the stack.  When it sees a normal lexeme, like the
; word PLUS or the integer 123, it accumulates this onto the right
; end of the top-most frame.  When it sees a close parenthesis, it
; pops the top frame and accumulates it onto the right end of the
; frame below.  We code it so that if any of these stack operations
; is ill-formed, e.g., we pop an empty stack, the result is F.
; Furthermore, we arrange for an F stack to be propagated, i.e.,
; pushing something onto F produces F.  Thus, if a parsing error
; arises early in the scan of the lexemes, e.g., an unbalanced
; close parenthesis is seen, the stack becomes F and even though
; the scan continues until the last lexeme has been processed, the
; F is preserved as the signal that an error occurred.

; When all the lexemes have been processed we inspect the stack and
; verify that it contains a single frame.  If not, then we return
; F.  (If the stack contains less than one frame, then the stack is
; in fact F and an error was detected earlier.  If the stack
; contains more than one frame, we have unbalanced open paren-
; theses.)

; To make this work we have to start the stack as though we are
; accumulating a list, i.e., with one empty frame on the stack.
; Then when we are done we have to check that the list we accumu-
; lated has exactly one element.  If it has none, the input was
; empty, which is an error.  If it has more than one, the input
; contained more than one tree and that is an error.

; To handle the special tokens single-quote, backquote, comma,
; comma-at-sign, and comma-dot, we generalize the stack slightly so
; that when these tokens are read they push a new frame that con-
; tains the token rather than the empty list.  Then we continue
; parsing.  When the next tree is assembled it is ‘‘accumulated
; onto the right end of the frame below,’’ where we actually take
; into account the special tokens that might mark the frame below.
; For example, if x is to be accumulated onto the frame below, and
; the frame below contains the single-quote token, then we create
; the single-quote tree (’ x) and recursively accumulate it onto
; the frame below that.

; Finally, the dot token is afforded no such special handling
; except that when a list is accumulated onto another (or returned
; as the answer) we verify that if it contains the dot token as an
; element then the token occurs as the next-to-last element.

; Of course, this whole process is much simpler than the Lisp
```

```
; reader because we are not implementing readmacros here, nor do we
; have to do the checks for ''readability.''  We just parse a tree
; and return it for the rest of this system to inspect.

; So here are the functions implementing our stacks
(DEFN TOP-PSTK (STACK) (IF (LISTP STACK) (CAR STACK) F))
(DEFN POP-PSTK (STACK) (IF (LISTP STACK) (CDR STACK) F))
(DEFN PUSH-PSTK (X STACK) (IF (EQUAL STACK F) F (CONS X STACK)))
(DEFN EMPTY-PSTK (X) (NLISTP X))

; Here is the predicate that verifies that a dotted tree is prop-
; erly formed.  We permit x to be the dot token, even though that
; is not a token tree, so that it can be accumulated just like
; other atoms.  We will filter out the isolated dot at the top.  So
; what we want to check here is that if X is a list and the dot
; token appears as an element then it appears in the next-to-last
; position and there is at least one element before it.  The vari-
; able I below just counts the number of elements we've scanned
; past.
(DEFN DOT-CRITERION (I X)
  (IF (NLISTP X)
      T
      (IF (EQUAL (CAR X) (DOT-TOKEN))
          (AND (NOT (ZEROP I))
           (AND (LISTP (CDR X))
                (NLISTP (CDR (CDR X)))))
          (DOT-CRITERION (ADD1 I) (CDR X)))))

; Here is how we accumulate X ''onto the frame below,'' where the
; ''frame below'' is the top of the stack passed into this func-
; tion.  We check first that X satisfies the dot token criterion
; and cause an error if it doesn't.  Then we handle the special
; tokens.
(DEFN ADD-ELEMENT-TO-TOP (X STACK)
  (IF (EMPTY-PSTK STACK)
      F
  (IF (NOT (DOT-CRITERION 0 X))
      F
  (IF (SPECIAL-TOKEN (TOP-PSTK STACK))
      (ADD-ELEMENT-TO-TOP (LIST2 (TOP-PSTK STACK) X)
                          (POP-PSTK STACK))
      (PUSH-PSTK (APPEND (TOP-PSTK STACK) (LIST1 X))
                 (POP-PSTK STACK))))))

; When we are all done, the following function is used to verify
; that exactly one whole object was constructed and that it is not
; an isolated dot-token.
(DEFN STOP (STACK)
  (IF (AND (EQLEN STACK 1)
       (AND (EQLEN (TOP-PSTK STACK) 1)
            (NOT (EQUAL (CAR (TOP-PSTK STACK)) (DOT-TOKEN)))))
      (CAR (TOP-PSTK STACK))
      F))
```

```
; Finally, here is the parser.  The proper top-level call of this
; function is (PARSE lexemes (CONS NIL NIL)).  It returns either a
; tree or F.  F indicates that the lexemes either did not parse in-
; to a complete tree or parsed into more than one.  The tree may be
; ''unreadable.''
(DEFN PARSE (LEXEMES STACK)
  (IF (NLISTP LEXEMES)
      (STOP STACK)
  (IF (EQUAL (CAR LEXEMES) (OPEN-PAREN))
      (PARSE (CDR LEXEMES) (PUSH-PSTK NIL STACK))
  (IF (SPECIAL-TOKEN (CAR LEXEMES))
      (PARSE (CDR LEXEMES) (PUSH-PSTK (CAR LEXEMES) STACK))
  (IF (EQUAL (CAR LEXEMES) (CLOSE-PAREN))
      (PARSE (CDR LEXEMES)
             (ADD-ELEMENT-TO-TOP (TOP-PSTK STACK)
                                 (POP-PSTK STACK)))
      (PARSE (CDR LEXEMES)
             (ADD-ELEMENT-TO-TOP (CAR LEXEMES) STACK)))))))


; SUBSECTION:  Putting the Two Passes Together

; As noted, the parser may not produce a token tree because the
; lexical analyzer can produce non-word, non-token lexemes.  For
; example, the stream (ASCII '(SPACE NUMBER-SIGN SPACE)) parses as
; the ''word'' we might display as # and which is logically repre-
; sented by the literal atom (PACK (CONS 35 0)).  But this is not a
; word, technically, because the number sign character is not among
; the word characters.  Therefore, after we have parsed the lexemes
; we check that the result is a token tree and return F if it is
; not.  Thus, this function may return F because
; (a) the lexical analyzer found fault with the character stream
;     (e.g., an unbalanced #-comment)
; (b) the parser found fault with the lexeme stream (e.g., an un-
;     balanced open parenthesis), or
; (c) the final tree failed to be a token tree.
(DEFN READ-TOKEN-TREE (STREAM)
  (IF (TOKEN-TREE (PARSE (LEXEMES STREAM 0) (CONS NIL NIL)))
      (PARSE (LEXEMES STREAM 0) (CONS NIL NIL))
      F))

; SECTION:  Readable Token Trees and S-Expressions

; We now resume our formalization of the text.  The definition of
; ''readable from depth'' N assumes X is a token tree or F.  We
; allow F as a possible input only because that is the error signal
; presented by READ-TOKEN-TREE.  F is not a token tree -- the only
; atomic token trees are integers and words.  Our formalization of
; ''readable'' announces that F is not readable, passing the
; ''error'' up.
(DEFN READABLE (X N)
  (IF (NLISTP X)
      (NOT (EQUAL X F))
```

```
  (IF (SINGLE-QUOTE-TOKEN-TREE X)
      (READABLE (CONSTITUENT X) N)
  (IF (BACKQUOTE-TOKEN-TREE X)
      (AND (READABLE (CONSTITUENT X) (ADD1 N))
           (NOT (SPLICE-ESCAPE-TOKEN-TREE (CONSTITUENT X))))
  (IF (OR (COMMA-ESCAPE-TOKEN-TREE X)
          (SPLICE-ESCAPE-TOKEN-TREE X))
      (AND (LESSP 0 N)
           (READABLE (CONSTITUENT X) (SUB1 N)))
  (IF (DOTTED-PAIR X)
      (AND (READABLE (CAR X) N)
       (AND (READABLE (CADRN 2 X) N)
            (NOT (SPLICE-ESCAPE-TOKEN-TREE (CADRN 2 X)))))
  (IF (SINGLETON X)
      (READABLE (CAR X) N)
      (AND (READABLE (CAR X) N)
           (READABLE (CDR X) N)))))))))

; Like our formalization of ''readable,'' our formalization of ''s-
; expression'' allows its argument to be either a token tree or F
; and returns F on F.  Thus, if you apply S-EXPRESSION to the out-
; put of READ-TOKEN-TREE you will get F if the read ''caused an
; error.''  In actual use, we only apply S-EXPRESSION to the output
; of READMACRO-EXPANSION and we only apply that to READABLE token
; trees.  Thus, this aspect of our formalization of s-expressions
; is irrelevant.  We preserve it because it is sometimes nice,
; while testing these definitions, to run S-EXPRESSION on the
; output of the reader.
(DEFN S-EXPRESSION (X)
  (IF (NLISTP X)
      (IF (EQUAL X F)
          F
          (NOT (NUMERIC-WORD X)))
  (IF (SPECIAL-TOKEN-TREE X)
      F
  (IF (DOTTED-PAIR X)
      (AND (S-EXPRESSION (CAR X))
           (S-EXPRESSION (CADRN 2 X)))
  (IF (SINGLETON X)
      (S-EXPRESSION (CAR X))
      (AND (S-EXPRESSION (CAR X))
           (S-EXPRESSION (CDR X))))))))


; SECTION:  Backquote Expansion

; At this point in the text we give the definition of readmacro-
; expansion.  However, it is presented without first defining the
; notion of backquote expansion.  We therefore skip ahead in the
; text and formalize backquote expansion now so we can then present
; readmacro-expansion.

; Of course, backquote expansion and backquote-list expansion are
```

```
; mutually recursive.  If FLG below is T we are defining backquote
; expansion.  Otherwise, we are defining backquote-list expansion.
(DEFN BACKQUOTE-EXPANSION (FLG X)
  (IF FLG
      (IF (NLISTP X)
          (LIST2 'QUOTE X)
          (IF (OR (COMMA-ESCAPE-TOKEN-TREE X)
                  (SPLICE-ESCAPE-TOKEN-TREE X))
              (CONSTITUENT X)
              (BACKQUOTE-EXPANSION F X)))
      (IF (NLISTP X)
          'IMPOSSIBLE-IF-X-IS-A-DOTTED-OR-UNDOTTED-TOKEN-TREE
          (LIST3 (IF (SPLICE-ESCAPE-TOKEN-TREE (CAR X))
                     'APPEND
                     'CONS)
                 (BACKQUOTE-EXPANSION T (CAR X))
                 (IF (SINGLETON X)
                     (LIST2 'QUOTE 'NIL)
                     (IF (DOTTED-PAIR X)
                         (BACKQUOTE-EXPANSION T (CADRN 2 X))
                         (BACKQUOTE-EXPANSION F (CDR X)))))))
  ((ORD-LESSP (CONS (ADD1 (COUNT X)) (IF FLG 1 0)))))


; SECTION:  Readmacro Expansion

; Our definition of readmacro-expansion differs from the text in
; that we do all four passes at once.  It is so easy in English to
; say ''replace every ...'' and its formalization requires a recur-
; sive sweep through the tree.  Even if we had higher order func-
; tions (or used V&C$) we would spend about as much space defining
; the generic sweep as we do below just doing it.
(DEFN READMACRO-EXPANSION (X)
  (IF (NLISTP X)
      (IF (NUMERIC-WORD X)
          (NUMERIC-WORD-VALUE X)
          X)
  (IF (SINGLE-QUOTE-TOKEN-TREE X)
      (LIST2 'QUOTE (READMACRO-EXPANSION (CONSTITUENT X)))
  (IF (BACKQUOTE-TOKEN-TREE X)
      (BACKQUOTE-EXPANSION T
       (READMACRO-EXPANSION (CONSTITUENT X)))
  (IF (DOTTED-PAIR X)
      (IF (OR (AND (NLISTP (READMACRO-EXPANSION (CADRN 2 X)))
                   (NOT (EQUAL (READMACRO-EXPANSION (CADRN 2 X))
                               'NIL)))
              (SPECIAL-TOKEN-TREE
               (READMACRO-EXPANSION (CADRN 2 X))))
          (LIST3 (READMACRO-EXPANSION (CAR X))
                 (DOT-TOKEN)
                 (READMACRO-EXPANSION (CADRN 2 X)))
          (CONS (READMACRO-EXPANSION (CAR X))
                (READMACRO-EXPANSION (CADRN 2 X))))
```

```
  (IF (SINGLETON X)
      (LIST1 (READMACRO-EXPANSION (CAR X)))
      (CONS (READMACRO-EXPANSION (CAR X))
            (READMACRO-EXPANSION (CDR X))))))))


; SECTION:  Some Common Lisp

#|

; This entire section is commented out.  It is here only as a con-
; venience for those wishing to test the token tree reader and
; readmacro-expansion.  The following function reads a token tree
; from a stream of ASCII character codes.  If it did not parse or
; is ‘‘unreadable,’’ suitable messages are returned.  Otherwise,
; the token tree is readmacro-expanded.  If an s-expression does
; not result, a suitable message is returned.  THIS ERROR MESSAGE
; SHOULD NEVER HAPPEN because the readmacro-expansion of a readable
; token tree supposedly always produces an s-expression!  If no
; errors are reported, the function returns the resulting s-
; expression.  Thus, this function is essentially a formalization
; of the the Lisp read routine.
(DEFN TEST-READER (STREAM)
  (LET ((X (READ-TOKEN-TREE STREAM)))
    (COND ((NOT X) ’DID-NOT-PARSE)
          ((NOT (READABLE X 0)) ’NOT-READABLE-FROM-0)
          (T (LET ((Y (READMACRO-EXPANSION X)))
               (COND
                ((S-EXPRESSION Y)
                 Y)
                (T (LIST
                    ’READMACRO-EXPANSION-PRODUCED-NON-S-EXPRESSION
                    Y))))))))

; The following defines a Lisp routine, not a function in the
; logic.  The routine’s name is test-reader and it is just a
; convenient interface to the logical function defined above.  You
; give it a Lisp string and it converts it into a list of ASCII
; codes.  This just lets us type things like (test-reader
; "(PLUS X Y)") to Common Lisp -- not to R-LOOP -- to execute the
; logic’s TEST-READER.

 (defun ascii (string)
   (mapcar (function char-code)
           (coerce string ’list)))

 (defun test-reader (string)
   (*1*test-reader (ascii string)))

; We now return to the main flow of this development of the
; extended syntax.

|#
```

```
; SECTION: Some Terminology Preliminary to Translation

; We now define the concepts used to describe the process of
; translation from an s-expression to a formal term.
(DEFN CORRESPONDING-NUMBERP (N)
  (IF (ZEROP N)
      (LIST1 'ZERO)
      (LIST2 'ADD1 (CORRESPONDING-NUMBERP (SUB1 N)))))
(DEFN CORRESPONDING-NEGATIVEP (N)
  (LIST2 'MINUS (CORRESPONDING-NUMBERP (NEGATIVE-GUTS N))))
(DEFN A-D-SEQUENCEP (LST)
  (IF (NLISTP LST)
      T
      (IF (OR (EQUAL (CAR LST) (ASCII 'UPPER-A))
              (EQUAL (CAR LST) (ASCII 'UPPER-D)))
          (A-D-SEQUENCEP (CDR LST))
          F)))
(DEFN CAR-CDR-SYMBOLP (X)
  (AND (LITATOM X)
   (AND (EQUAL (CAR (UNPACK X)) (ASCII 'UPPER-C))
    (AND (EQUAL (CAR (LAST (UNPACK X))) (ASCII 'UPPER-R))
         (A-D-SEQUENCEP (ALL-BUT-LAST (CDR (UNPACK X))))))))

; While the text defines the A/D sequence of a CAR/CDR symbol to be
; a sequence of A's and D's, we define it to be a sequence of the
; ASCII codes of A and of D.
(DEFN A-D-SEQUENCE (X)
  (ALL-BUT-LAST (CDR (UNPACK X))))
(DEFN CAR-CDR-NEST (LST X)
  (IF (NLISTP LST)
      X
      (IF (EQUAL (CAR LST) (ASCII 'UPPER-A))
          (LIST2 'CAR (CAR-CDR-NEST (CDR LST) X))
          (LIST2 'CDR (CAR-CDR-NEST (CDR LST) X)))))

; Ultimately we will define the translation process.  This will
; involve us in consing together the translations of various com-
; ponents of an s-expression, after making sure those components
; are well-formed.  We signal ill-formed input by returning F in-
; stead of (the quotation of) the formal term.  To make the neces-
; sary well-formedness checks less distracting we define the fol-
; lowing function which we use to create the formal terms.  It is
; like CONS, but observes the convention that if either argument is
; F the result is F.
(DEFN FCONS (X Y)
  (IF (AND X Y)
      (CONS X Y)
      F))

; We also provide ourselves with several LIST-like functions that
; respect this convention.  'Tis a pity we do not have macros in
; the Nqthm logic because we really just want to define a ''func-
; tion'' symbol that takes an arbitrary number of arguments.
```

```
(DEFN FLIST1 (X) (FCONS X NIL))
(DEFN FLIST2 (X Y) (FCONS X (FLIST1 Y)))
(DEFN FLIST3 (X Y Z) (FCONS X (FLIST2 Y Z)))
(DEFN FLIST4 (X Y Z W) (FCONS X (FLIST3 Y Z W)))
(DEFN FLIST5 (X Y Z W V) (FCONS X (FLIST4 Y Z W V)))
(DEFN FLIST6 (X Y Z W V U) (FCONS X (FLIST5 Y Z W V U)))
(DEFN FLIST7 (X Y Z W V U R) (FCONS X (FLIST6 Y Z W V U R)))

; Here is the formalization of the ''fn nest around x for lst'',
; which is presented in the text before we get to the extended
; syntax.  Note that we use our FCONS convention so that if X or
; any element of LST is F, the result is F.
(DEFN FN-NEST (FN LST X)
  (IF (NLISTP LST)
      X
      (FLIST3 FN (CAR LST) (FN-NEST FN (CDR LST) X))))
(DEFN CORRESPONDING-NUMBERPS (LST)
  (IF (NLISTP LST)
      NIL
      (CONS (CORRESPONDING-NUMBERP (CAR LST))
            (CORRESPONDING-NUMBERPS (CDR LST)))))
(DEFN EXPLOSION (LST)
  (FN-NEST 'CONS
           (CORRESPONDING-NUMBERPS LST)
           (LIST1 'ZERO)))
(DEFN CORRESPONDING-LITATOM (X)
  (LIST2 'PACK
         (EXPLOSION (UNPACK X))))

; The following function tacks the *1* prefix onto the front of
; LITATOMs.  Thus, (star-one-star 'true) is the LITATOM '*1*TRUE --
; EXCEPT we can't write that literal atom that way because it is
; not a symbolp.  It must be written as (PACK '(42 49 42 84 82 85
; 69 . 0)) or, equivalently, '(*1*QUOTE PACK (42 49 42 84 82 85 69
; . 0)).
(DEFN STAR-ONE-STAR (X)
  (PACK (APPEND (ASCII '(ASTERISK DIGIT-ONE ASTERISK))
                (UNPACK X))))

; We need to define what it is to be a symbol.  The text does this
; quite early, while introducing formal terms.  Here is the
; formalization.
(DEFN ASCII-UPPER-ALPHABETICS NIL
  (ASCII '(UPPER-A UPPER-B UPPER-C UPPER-D UPPER-E UPPER-F
           UPPER-G UPPER-H UPPER-I UPPER-J UPPER-K UPPER-L
           UPPER-M UPPER-N UPPER-O UPPER-P UPPER-Q UPPER-R
           UPPER-S UPPER-T UPPER-U UPPER-V UPPER-W UPPER-X
           UPPER-Y UPPER-Z)))
(DEFN ASCII-DIGITS-AND-SIGNS NIL
  (ASCII '(DIGIT-ONE DIGIT-TWO DIGIT-THREE DIGIT-FOUR DIGIT-FIVE
           DIGIT-SIX DIGIT-SEVEN DIGIT-EIGHT DIGIT-NINE
           DOLLAR-SIGN UPARROW AMPERSAND ASTERISK UNDERSCORE
           MINUS-SIGN PLUS-SIGN EQUAL-SIGN TILDE OPEN-BRACE
```

```
           CLOSE-BRACE QUESTION-MARK LESS-THAN-SIGN
           GREATER-THAN-SIGN)))
(DEFN ALL-UPPER-ALPHABETICS-DIGITS-OR-SIGNS (L)
  (IF (NLISTP L)
      T
      (AND (OR (MEMBER (CAR L) (ASCII-UPPER-ALPHABETICS))
               (MEMBER (CAR L) (ASCII-DIGITS-AND-SIGNS)))
           (ALL-UPPER-ALPHABETICS-DIGITS-OR-SIGNS (CDR L)))))
(DEFN LEGAL-CHAR-CODE-SEQ (L)
  (AND (LISTP L)
   (AND (EQUAL (CDR (LAST L)) 0)
    (AND (MEMBER (CAR L) (ASCII-UPPER-ALPHABETICS))
         (ALL-UPPER-ALPHABETICS-DIGITS-OR-SIGNS (CDR L))))))
(DEFN SYMBOLP (X)
  (AND (LITATOM X)
       (LEGAL-CHAR-CODE-SEQ (UNPACK X))))


; SECTION:  Formalizing Aspects of the History

; At this point in the text we define well-formedness and trans-
; lation.  But the text delays until later the definitions of
; several concepts used.  These delayed concepts are: QUOTE
; notation, the *1*QUOTE escape mechanism, and abbreviated FORs.
; We have to formalize these before we can define translation.
; Unfortunately, before we can formalize *1*QUOTE notation we must
; formalize certain concepts relating to the ''history'' in which
; the translation is occurring.  For example, (FN X Y) is well-
; formed and has a translation only if FN is a function symbol of
; arity 2.  We must formalize arity.  We must also formalize the
; notions of what it is to be a shell constructor, a base object,
; and to satisfy the ''type- restrictions'' of a shell.

; But functions that depend upon the ''current history'' cannot be
; defined unless we wish to make the ''current history'' an ex-
; plicit object in the formalization.  That is a reasonable thing
; to do but is beyond the scope of this book because the only DEFN
; events (for example) permitted in a history are those that are
; admissible, which involves the notion of proof.  Thus, to formal-
; ize histories accurately we would have to formalize the rules of
; inference (not just the syntax) and what it is to be a theorem.
; This has been done for other logics in the Nqthm logic.  See for
; example Shankar's work in /examples/shankar or the bibliographic
; entries for Shankar.

; Rather than formalize histories in order to formalize the syntax,
; we will formalize the syntax for a particular history.  The his-
; tory we choose is the GROUND-ZERO history extended with one il-
; lustrative user-defined shell, the stacks as illustrated in the
; text, plus one illustrative user-defined function, CONCAT, which
; is just APPEND by another name.  We introduce these functions be-
; low into the theory.  They are nowhere used in this development
; and thus by searching for references to them you can see every-
```

```
; thing that must be known about a function symbol for it to be
; usable in the extended syntax.

; An irrelevant shell added to illustrate user-defined shells:

(ADD-SHELL PUSH EMPTY-STACK STACKP
           ((TOP (ONE-OF NUMBERP) ZERO)
            (POP (ONE-OF STACKP) EMPTY-STACK)))

; An irrelevant function added to illustrate user-defined
; functions:
(DEFN CONCAT (X Y)
  (IF (NLISTP X)
      Y
      (CONS (CAR X) (CONCAT (CDR X) Y))))

; So consider the GROUND-ZERO logic extended with the two logical
; acts above.  We will now define some functions that answer ques-
; tions about that history, e.g., what are the base function
; symbols?

; The following function returns non-F if its argument is (the quo-
; tation of) a shell base function symbol.  When it returns non-F
; it actually returns (the quotation of) the recognizer for the
; relevant shell.
(DEFN SHELL-BASE-FUNCTION (X)
  (IF (EQUAL X 'TRUE) 'TRUEP
      (IF (EQUAL X 'FALSE) 'FALSEP
          (IF (EQUAL X 'ZERO) 'NUMBERP
              (IF (EQUAL X 'EMPTY-STACK) 'STACKP
                  F)))))

; The next function is the analogous one for shell constructor
; function symbols.
(DEFN SHELL-CONSTRUCTOR-FUNCTION (X)
  (IF (EQUAL X 'ADD1) 'NUMBERP
      (IF (EQUAL X 'CONS) 'LISTP
          (IF (EQUAL X 'PACK) 'LITATOM
              (IF (EQUAL X 'MINUS) 'NEGATIVEP
                  (IF (EQUAL X 'PUSH) 'STACKP
                      F))))))

; The following function returns the list of type restrictions for
; a shell constructor.  The list is in 1:1 correspondence with the
; accessor names, i.e., the arguments of the constructor.  For ex-
; ample, for PUSH the result is '((ONE-OF NUMBERP) (ONE-OF STACKP))
; which tells us the first argument to PUSH must be a NUMBERP and
; the second must be a STACKP.
(DEFN SHELL-TYPE-RESTRICTIONS (X)
  (IF (OR (EQUAL X 'ADD1)
          (EQUAL X 'MINUS))
      (LIST1 (LIST2 'ONE-OF 'NUMBERP))
      (IF (EQUAL X 'CONS)
```

```
          (LIST2 (LIST1 'NONE-OF) (LIST1 'NONE-OF))
          (IF (EQUAL X 'PACK)
              (LIST1 (LIST1 'NONE-OF))
              (IF (EQUAL X 'PUSH)
                  (LIST2 (LIST2 'ONE-OF 'NUMBERP)
                         (LIST2 'ONE-OF 'STACKP))
                  F)))))

; This function takes a shell constructor or base function symbol
; and determines whether it satisfies a given type-restriction.
; Thus, FN might be ADD1 and TYPE-RESTRICTION might be '(ONE-OF
; NUMBERP) -- in which case SATISFIES returns T.  If the type
; restriction were '(ONE-OF STACKP), SATISFIES would return F for
; FN 'ADD1.
(DEFN SATISFIES (FN TYPE-RESTRICTION)
  (IF (EQUAL (CAR TYPE-RESTRICTION) 'ONE-OF)
      (MEMBER (IF (SHELL-BASE-FUNCTION FN)
                  (SHELL-BASE-FUNCTION FN)
                  (SHELL-CONSTRUCTOR-FUNCTION FN))
              (CDR TYPE-RESTRICTION))
      (NOT (MEMBER (IF (SHELL-BASE-FUNCTION FN)
                       (SHELL-BASE-FUNCTION FN)
                       (SHELL-CONSTRUCTOR-FUNCTION FN))
                   (CDR TYPE-RESTRICTION)))))

; This function takes a list of function symbols and a list of type
; restrictions in 1:1 correspondence and determines whether all of
; the type restrictions are satisfied by the corresponding function
; symbols.
(DEFN ALL-SATISFY (FN-LST TYPE-RESTRICTION-LST)
  (IF (NLISTP FN-LST)
      T
      (AND (SATISFIES (CAR FN-LST) (CAR TYPE-RESTRICTION-LST))
           (ALL-SATISFY (CDR FN-LST) (CDR TYPE-RESTRICTION-LST)))))

; The next two functions define the arity of each of the function
; symbols in the particular history we are considering.  We use the
; extended syntax here, namely QUOTE notation, to write down the
; alist.  Readers using this formalization as a means of mastering
; the notation should simply understand that as a result of the
; definition below, (ARITY 'IF) = 3, (ARITY 'STACKP) = 1 and (ARITY
; 'ANY-NEW-SYMBOL) = F.  That is, if (fn . n) appears in the defi-
; nition of ARITY-ALIST then (ARITY 'fn) = n.  Otherwise, (ARITY
; 'fn) = F.
(DEFN ARITY-ALIST NIL
  '((IF . 3) (EQUAL . 2) (COUNT . 1) (FALSE . 0)
    (FALSEP . 1) (TRUE . 0) (TRUEP . 1) (NOT . 1)
    (AND . 2) (OR . 2) (IMPLIES . 2) (ADD1 . 1)
    (NUMBERP . 1) (SUB1 . 1) (ZERO . 0) (LESSP . 2)
    (GREATERP . 2) (LEQ . 2) (GEQ . 2) (ZEROP . 1)
    (FIX . 1) (PLUS . 2) (PACK . 1) (LITATOM . 1)
    (UNPACK . 1) (CONS . 2) (LISTP . 1) (CAR . 1)
    (CDR . 1) (NLISTP . 1) (MINUS . 1) (NEGATIVEP . 1)
```

```
    (NEGATIVE-GUTS . 1) (DIFFERENCE . 2) (TIMES . 2)
    (QUOTIENT . 2) (REMAINDER . 2) (MEMBER . 2)
    (IFF . 2) (ORD-LESSP . 2) (ORDINALP . 1)
    (ASSOC . 2) (PAIRLIST . 2) (SUBRP . 1)
    (APPLY-SUBR . 2) (FORMALS . 1) (BODY . 1)
    (FIX-COST . 2) (STRIP-CARS . 1) (SUM-CDRS . 1)
    (V&C$ . 3) (V&C-APPLY$ . 2) (APPLY$ . 2)
    (EVAL$ . 3) (QUANTIFIER-INITIAL-VALUE . 1)
    (ADD-TO-SET . 2) (APPEND . 2) (MAX . 2)
    (UNION . 2) (QUANTIFIER-OPERATION . 3) (FOR . 6)
    (PUSH . 2) (EMPTY-STACK . 0) (STACKP . 1) (TOP . 1)
    (POP . 2) (CONCAT . 2)))
(DEFN ARITY (FN)
  (IF (ASSOC FN (ARITY-ALIST))
      (CDR (ASSOC FN (ARITY-ALIST)))
      F))

; That concludes the definitions of history dependent concepts.  We
; now return to the text.

; SECTION:  QUOTE Notation and the *1*QUOTE Escape

; The essence of the QUOTE notation is the notion of ‘‘explicit
; value descriptor.’’  In the text, that notion is defined in terms
; of ‘‘explicit value escape descriptor’’ without the latter notion
; being defined.  We then define the latter notion and its defi-
; nition involves the former.  Thus, the two are mutually recur-
; sive.  We formalize the mutual recursion with our traditional FLG
; argument.  Normally, one would expect one value of the flag to
; indicate we were defining ‘‘explicit value descriptor’’ and the
; other value to indicate ‘‘explicit value escape descriptor.’’
; But it turns out that the basic form of mutual recursion here is
; ‘‘explicit value descriptor’’ versus ‘‘list of explicit value
; descriptors’’ and the notion of ‘‘explicit value escape de-
; scriptor’’ is written ‘‘in-line.’’

; So when FLG is T below, we check that X is an explicit value
; descriptor and if so we return the formal term it denotes.  We
; return F if X is not such a descriptor.  When FLG is F we check
; that X is a list of explicit value descriptors and we return
; either the list of denoted formal terms or F if X fails to be a
; list of descriptors.

; The following three events have no logical significance.  They
; do, however, permit Nqthm to process the next definition without
; inordinate delay.  We disable all function definitions except
; CADRN and CDRN.

(SET-STATUS PRE-EXPLICIT-VALUE-DESCRIPTOR
            T
            ((DEFN DISABLE) (OTHERWISE AS-IS)))

(ENABLE CADRN)
```

```
(ENABLE CDRN)
(DEFN EXPLICIT-VALUE-DESCRIPTOR (FLG X)
  (IF FLG

; Here we define what it is for X to be an explicit value de-
; scriptor and what formal term it denotes.

      (IF (NLISTP X)
          (IF (INTEGERP X)
              (IF (NUMBERP X)
                  (CORRESPONDING-NUMBERP X)
                  (CORRESPONDING-NEGATIVEP X))
          (IF (EQUAL X (STAR-ONE-STAR 'TRUE))
              (LIST1 'TRUE)
          (IF (EQUAL X (STAR-ONE-STAR 'FALSE))
              (LIST1 'FALSE)
          (IF (SYMBOLP X)
              (CORRESPONDING-LITATOM X)
              F))))
      (IF (EQUAL (CAR X) (STAR-ONE-STAR 'QUOTE))

; The test of the following IF contains the formalization of
; ''explicit value escape descriptor.''

          (IF (AND (OR (SHELL-CONSTRUCTOR-FUNCTION (CADRN 1 X))
                       (SHELL-BASE-FUNCTION (CADRN 1 X)))
               (AND (EQLEN (CDRN 2 X) (ARITY (CADRN 1 x)))
                (AND (EQUAL (CDR (LAST X)) NIL)
                 (AND (NOT (EQUAL (CADRN 1 X) 'ADD1))
                  (AND (NOT (EQUAL (CADRN 1 X) 'ZERO))
                   (AND (NOT (EQUAL (CADRN 1 X) 'CONS))
                    (AND (EXPLICIT-VALUE-DESCRIPTOR F (CDRN 2 X))
                     (AND
                      (IF (SHELL-CONSTRUCTOR-FUNCTION (CADRN 1 X))
                          (ALL-SATISFY
                           (STRIP-CARS
                            (EXPLICIT-VALUE-DESCRIPTOR F
                             (CDRN 2 X)))
                           (SHELL-TYPE-RESTRICTIONS (CADRN 1 X)))
                          T)
                      (IF (EQUAL (CADRN 1 X) 'PACK)
                          (NOT (LEGAL-CHAR-CODE-SEQ (CADRN 2 X)))
                          (IF (EQUAL (CADRN 1 X) 'MINUS)
                              (EQUAL (CADRN 2 X) (ZERO))
                              T))))))))))
              (CONS (CADRN 1 X)
                    (EXPLICIT-VALUE-DESCRIPTOR F (CDRN 2 X)))
              F)
      (IF (DOTTED-PAIR X)
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (EXPLICIT-VALUE-DESCRIPTOR T (CADRN 2 X)))
      (IF (SINGLETON X)
```

```
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (CORRESPONDING-LITATOM NIL))
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (EXPLICIT-VALUE-DESCRIPTOR T (CDR X)))))))

; Here we define what it is for X to be a list of explicit value
; descriptors and the list of terms denoted by them.

      (IF (NLISTP X)
          NIL
          (FCONS (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                 (EXPLICIT-VALUE-DESCRIPTOR F (CDR X))))))

(SET-STATUS POST-EXPLICIT-VALUE-DESCRIPTOR
            T
            ((DEFN ENABLE) (OTHERWISE AS-IS)))

; The function QT is just a convenient way to refer to the explicit
; value term denoted by an explicit value descriptor (or F if its
; argument is not such a descriptor).  Thus, (QT 'ABC) is '(PACK
; (CONS 65 (CONS 66 (CONS 67 0)))), except that the integers are
; actually ADD1 nests.  Read ''quotation'' for QT.
(DEFN QT (X) (EXPLICIT-VALUE-DESCRIPTOR T X))

; SECTION:  In Support of COND, CASE, and LET

; The next batch of functions are all involved in the translation
; of COND, CASE, and LET.  We are interested in recognizing lists
; of doublets, e.g., ((w1 v1) (w2 v2) ...), the absence of
; duplication among the wi, etc.
(DEFN DOUBLETS (LST)
  (IF (NLISTP LST)
      (EQUAL LST NIL)
      (AND (EQLEN (CAR LST) 2)
           (DOUBLETS (CDR LST)))))
(DEFN DUPLICATESP (LST)
  (IF (NLISTP LST)
      F
      (IF (MEMBER (CAR LST) (CDR LST))
          T
          (DUPLICATESP (CDR LST)))))
(DEFN STRIP-CADRS (LST)
  (IF (NLISTP LST)
      NIL
      (CONS (CADRN 1 (CAR LST))
            (STRIP-CADRS (CDR LST)))))
(DEFN SYMBOLPS (LST)
  (IF (NLISTP LST)
      T
      (AND (SYMBOLP (CAR LST))
           (SYMBOLPS (CDR LST)))))
```

```
; This function applies the substitution ALIST to TERM (FLG=T) or
; to a list of terms (FLG=F).
(DEFN SUBLIS-VAR (FLG ALIST TERM)
  (IF FLG
      (IF (NLISTP TERM)
          (IF (ASSOC TERM ALIST)
              (CDR (ASSOC TERM ALIST))
              TERM)
          (IF (EQUAL (CAR TERM) 'QUOTE)
              TERM
              (CONS (CAR TERM)
                    (SUBLIS-VAR F ALIST (CDR TERM)))))
      (IF (NLISTP TERM)
          NIL
          (CONS (SUBLIS-VAR T ALIST (CAR TERM))
                (SUBLIS-VAR F ALIST (CDR TERM)))))
  ((LESSP (COUNT TERM))))


; SECTION:  In Support of FOR

; The text delays the discussion of FOR statements until after V&C$
; has been presented.  We have to deal with them now.  The fol-
; lowing functions access or check certain parts of an abbreviated
; FOR.
(DEFN ABBREVIATED-FOR-VAR (X) (CADRN 1 X))
(DEFN ABBREVIATED-FOR-RANGE (X) (CADRN 3 X))
(DEFN ABBREVIATED-FOR-WHEN (X)
  (IF (EQLEN X 8)
      (CADRN 5 X)
      'T))
(DEFN ABBREVIATED-FOR-OP (X)
  (IF (EQLEN X 8)
      (CADRN 6 X)
      (CADRN 4 X)))
(DEFN ABBREVIATED-FOR-BODY (X) (CAR (LAST X)))
(DEFN FOR-OPERATIONP (X)
  (OR (EQUAL X 'ADD-TO-SET)
   (OR (EQUAL X 'ALWAYS)
    (OR (EQUAL X 'APPEND)
     (OR (EQUAL X 'COLLECT)
      (OR (EQUAL X 'COUNT)
       (OR (EQUAL X 'DO-RETURN)
        (OR (EQUAL X 'EXISTS)
         (OR (EQUAL X 'MAX)
          (OR (EQUAL X 'SUM)
           (OR (EQUAL X 'MULTIPLY)
               (EQUAL X 'UNION))))))))))))

; We now define the function that recognizes an abbreviated FOR.
(DEFN ABBREVIATED-FORP (X)
  (AND (LISTP X)
   (AND (EQUAL (CAR X) 'FOR)
```

```
    (AND (OR (EQLEN X 8)
             (EQLEN X 6))
     (AND (SYMBOLP (ABBREVIATED-FOR-VAR X))
      (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) NIL))
       (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) 'T))
        (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) 'F))
         (AND (EQUAL (CADRN 2 X) 'IN)
          (AND (OR (EQLEN X 6)
                   (EQUAL (CADRN 4 x) 'WHEN))
               (FOR-OPERATIONP
                (ABBREVIATED-FOR-OP X))))))))))))

; To translate an abbreviated FOR we must sort the list of vari-
; ables used in the WHEN clause and the BODY.
(DEFN ALPHABETIC-LESSP1 (L1 L2)
  (IF (NLISTP L1)
      T
      (IF (NLISTP L2)
          F
          (IF (LESSP (CAR L1) (CAR L2))
              T
              (IF (EQUAL (CAR L1) (CAR L2))
                  (ALPHABETIC-LESSP1 (CDR L1) (CDR L2))
                  F)))))
(DEFN ALPHABETIC-LESSP (X Y)
  (ALPHABETIC-LESSP1 (UNPACK X)
                     (UNPACK Y)))
(DEFN ALPHABETIC-INSERT (X L)
  (IF (NLISTP L)
      (LIST1 X)
      (IF (ALPHABETIC-LESSP X (CAR L))
          (CONS X L)
          (CONS (CAR L)
                (ALPHABETIC-INSERT X (CDR L))))))
(DEFN ALPHABETIZE (L)
  (IF (NLISTP L)
      L
      (ALPHABETIC-INSERT (CAR L)
                         (ALPHABETIZE (CDR L)))))

; To collect the variable symbols that occur in a term (or list of
; terms) we use ALL-VARS.
(DEFN ALL-VARS (FLG X)
  (IF FLG
      (IF (NLISTP X)
          (CONS X NIL)
          (ALL-VARS F (CDR X)))
      (IF (NLISTP X)
          NIL
          (UNION (ALL-VARS T (CAR X))
                 (ALL-VARS F (CDR X))))))
(DEFN STANDARD-ALIST (VARS)
  (IF (NLISTP VARS)
```

```
      (QT NIL)
      (LIST3 'CONS
             (LIST3 'CONS
                    (QT (CAR VARS))
                    (CAR VARS))
             (STANDARD-ALIST (CDR VARS)))))
(DEFN DELETE (X L)
  (IF (NLISTP L)
      L
      (IF (EQUAL X (CAR L))
          (CDR L)
          (CONS (CAR L) (DELETE X (CDR L))))))
(DEFN MAKE-ALIST (VAR WHEN BODY)
  (STANDARD-ALIST
   (ALPHABETIZE
    (DELETE VAR
            (UNION (ALL-VARS T WHEN)
                   (ALL-VARS T BODY))))))

; The following lemmas are used in the justification of the defi-
; nition of TRANSLATE.

(PROVE-LEMMA LESSP-ABBREVIATED-FOR-RANGE (REWRITE)
  (IMPLIES (EQUAL (CAR X) 'FOR)
           (LESSP (COUNT (ABBREVIATED-FOR-RANGE X)) (COUNT X))))

(PROVE-LEMMA LESSP-ABBREVIATED-FOR-WHEN (REWRITE)
  (IMPLIES (EQUAL (CAR X) 'FOR)
           (LESSP (COUNT (ABBREVIATED-FOR-WHEN X)) (COUNT X))))

(PROVE-LEMMA LESSP-LAST (REWRITE)
  (NOT (LESSP (COUNT X) (COUNT (LAST X)))))

(PROVE-LEMMA LESSP-ABBREVIATED-FOR-BODY (REWRITE)
  (IMPLIES (EQUAL (CAR X) 'FOR)
           (LESSP (COUNT (ABBREVIATED-FOR-BODY X)) (COUNT X))))

(PROVE-LEMMA LESSP-COUNT-STRIP-CARS (REWRITE)
  (IMPLIES (DOUBLETS LST)
           (NOT (LESSP (COUNT LST)
                       (COUNT (STRIP-CARS LST))))))

(PROVE-LEMMA LESSP-COUNT-STRIP-CADRS (REWRITE)
  (IMPLIES (DOUBLETS LST)
           (NOT (LESSP (COUNT LST)
                       (COUNT (STRIP-CADRS LST))))))

(PROVE-LEMMA LISTP-CDDR-X-COUNT-X (REWRITE)
  (IMPLIES (LISTP (CDDR X))
           (EQUAL (COUNT X)
                  (ADD1
                   (ADD1
                    (PLUS (COUNT (CAR X))
```

```
                          (PLUS (COUNT (CADR X))
                                (COUNT (CDDR X)))))))))

(PROVE-LEMMA LISTP-CDDDR-X-COUNT-X (REWRITE)
  (IMPLIES (LISTP (CDDDR X))
           (EQUAL (COUNT X)
                  (ADD1
                   (ADD1
                    (ADD1
                     (PLUS (COUNT (CAR X))
                           (PLUS (COUNT (CADR X))
                                 (PLUS (COUNT (CADDR X))
                                       (COUNT (CDDDR X))))))))))


; SECTION:  Translation

; We disable all definitions except CADRN and CDRN.

(SET-STATUS PRE-TRANSLATE T ((DEFN DISABLE)(OTHERWISE AS-IS)))
(ENABLE CADRN)
(ENABLE CDRN)

; Here, finally, is the formalization of what it is to be well-
; formed and what the translation of a well-formed term is.  If
; TRANSLATE (FLG=T) returns F, then X is not well-formed; other-
; wise, TRANSLATE (FLG=T) returns the (quotation of the) formal
; term denoted by X.  Because a formal term is either a variable
; symbol (i.e., LITATOM) or function application (i.e., LISTP), an
; answer of F unambiguously identifies the input as ill-formed.
; When FLG=F, TRANSLATE operates on a list of purported terms and
; returns either F, meaning at least one of the elements is
; ill-formed, or returns the list of their translations.
(DEFN TRANSLATE (FLG X)
  (IF FLG

; When FLG = T, we are translating a single term, X.

      (IF (NLISTP X)
          (IF (INTEGERP X)
              (QT X)
          (IF (SYMBOLP X)
              (IF (EQUAL X 'T) (LIST1 'TRUE)
              (IF (EQUAL X 'F) (LIST1 'FALSE)
              (IF (EQUAL X NIL) (QT NIL)
                  X)))
              F))
      (IF (DOTTED-S-EXPRESSION X)
          F
      (IF (EQUAL (CAR X) 'QUOTE)
          (IF (EQLEN X 2)
              (QT (CADRN 1 X))
              F)
```

```
      (IF (EQUAL (CAR X) 'COND)
          (IF (AND (EQLEN X 2)
                   (AND (EQLEN (CADRN 1 X) 2)
                        (EQUAL (CAR (CADRN 1 X)) 'T)))
              (TRANSLATE T (CADRN 1 (CADRN 1 X)))
              (IF (AND (EQLEN (CADRN 1 X) 2)
                       (AND (NOT (EQUAL (CAR (CADRN 1 X)) 'T))
                            (LISTP (CDRN 2 X))))
                  (FLIST4 'IF
                          (TRANSLATE T (CAR (CADRN 1 X)))
                          (TRANSLATE T (CADRN 1 (CADRN 1 X)))
                          (TRANSLATE T (CONS 'COND (CDRN 2 X))))
                  F))
      (IF (EQUAL (CAR X) 'CASE)
          (IF (AND (EQLEN X 3)
                   (AND (EQLEN (CADRN 2 X) 2)
                        (AND (EQUAL (CAR (CADRN 2 X)) 'OTHERWISE)
                             (TRANSLATE T (CADRN 1 X)))))
              (TRANSLATE T (CADRN 1 (CADRN 2 X)))
              (IF (AND (EQLEN (CADRN 2 X) 2)
                   (AND (LISTP (CDRN 3 X))
                        (NOT (MEMBER (CAR (CADRN 2 X))
                                     (STRIP-CARS (CDRN 3 X))))))
                  (FLIST4 'IF
                          (FLIST3 'EQUAL
                                  (TRANSLATE T (CADRN 1 X))
                                  (QT (CAR (CADRN 2 X))))
                          (TRANSLATE T (CADRN 1 (CADRN 2 X)))
                          (TRANSLATE T (CONS 'CASE
                                             (CONS (CADRN 1 X)
                                                   (CDRN 3 X)))))
                  F))
      (IF (EQUAL (CAR X) 'LET)

; The first test below, on DOUBLETS, belongs inside the AND nest.
; But if it is there it is not among the governing terms of the
; recursions in that nest and the termination argument breaks down.

          (IF (DOUBLETS (CADRN 1 X))
              (IF (AND (EQLEN X 3)
                   (AND (TRANSLATE F (STRIP-CARS (CADRN 1 X)))
                    (AND (TRANSLATE F (STRIP-CADRS (CADRN 1 X)))
                     (AND (TRANSLATE T (CADRN 2 X))
                      (AND
                       (SYMBOLPS
                        (TRANSLATE F (STRIP-CARS (CADRN 1 X))))
                       (NOT
                        (DUPLICATESP
                         (TRANSLATE
                          F (STRIP-CARS (CADRN 1 X)))))))))))
                  (SUBLIS-VAR
                   T
                   (PAIRLIST
```

```
                (TRANSLATE F (STRIP-CARS (CADRN 1 X)))
                (TRANSLATE F (STRIP-CADRS (CADRN 1 X))))
               (TRANSLATE T (CADRN 2 X)))
              F)
          F)
  (IF (NOT (TRANSLATE F (CDR X)))
      F
  (IF (OR (EQUAL (CAR X) NIL)
          (OR (EQUAL (CAR X) 'T)
              (EQUAL (CAR X) 'F)))
      F
  (IF (EQUAL (CAR X) 'LIST)
      (FN-NEST 'CONS
               (TRANSLATE F (CDR X))
               (QT NIL))
  (IF (EQUAL (CAR X) 'LIST*)
      (IF (EQLEN X 1)
          F
          (FN-NEST 'CONS
                   (ALL-BUT-LAST (TRANSLATE F (CDR X)))
                   (CAR (LAST (TRANSLATE F (CDR X))))))
  (IF (CAR-CDR-SYMBOLP (CAR X))
      (IF (EQLEN X 2)
          (CAR-CDR-NEST (A-D-SEQUENCE (CAR X))
                        (TRANSLATE T (CADRN 1 X)))
          F)
  (IF (EQLEN (CDR X) (ARITY (CAR X)))
      (FCONS (CAR X) (TRANSLATE F (CDR X)))
  (IF (EQUAL (CAR X) 'FOR)
      (IF (ABBREVIATED-FORP X)
          (FLIST7 'FOR
                  (QT (ABBREVIATED-FOR-VAR X))
                  (TRANSLATE T (ABBREVIATED-FOR-RANGE X))
                  (QT (TRANSLATE T
                                 (ABBREVIATED-FOR-WHEN X)))
                  (QT (ABBREVIATED-FOR-OP X))
                  (QT (TRANSLATE T
                                 (ABBREVIATED-FOR-BODY X)))
                  (MAKE-ALIST
                   (ABBREVIATED-FOR-VAR X)
                   (TRANSLATE T
                              (ABBREVIATED-FOR-WHEN X))
                   (TRANSLATE T
                              (ABBREVIATED-FOR-BODY X))))
          F)
      (IF (AND (LESSP 2 (LENGTH (CDR X)))
               (OR (EQUAL (CAR X) 'AND)
                   (OR (EQUAL (CAR X) 'OR)
                       (OR (EQUAL (CAR X) 'PLUS)
                           (EQUAL (CAR X) 'TIMES)))))
          (FN-NEST (CAR X)
                   (ALL-BUT-LAST (TRANSLATE F (CDR X)))
                   (CAR (LAST (TRANSLATE F (CDR X)))))
```

```
              F))))))))))))))

; This is the case where FLG = F and we are translating a list of
; terms, X.

      (IF (NLISTP X)
          NIL
          (FCONS (TRANSLATE T (CAR X))
                 (TRANSLATE F (CDR X)))))
  ((LESSP (COUNT X))))

(SET-STATUS POST-TRANSLATE T ((DEFN ENABLE)(OTHERWISE AS-IS)))


; SECTION:  The Extended Syntax

; Finally, here is EXSYN, which reads an s-expression from a stream
; of ASCII character codes and translates it into a formal term or
; returns F if the stream is not the display of a term in the ex-
; tended syntax.
(DEFN EXSYN (STREAM)
  (IF (READABLE (READ-TOKEN-TREE STREAM) 0)
      (TRANSLATE T (READMACRO-EXPANSION (READ-TOKEN-TREE STREAM)))
      F))

; EXSYN returns F if the stream cannot be parsed.  The explanation
; of this remark is that an unparsable stream causes READ-TOKEN-
; TREE to return F and READABLE returns F on that input.

; SECTION:  Slightly Abbreviated Formal Terms

; It is exceedingly difficult to read the output of TRANSLATE and
; EXSYN because quoted literal atoms and numbers are exploded.
; Thus 'ABC translates to (PACK (CONS (ADD1 ...)  ...))  where the
; ellipses are very large nests of CONSes and ADD1s.  Below, we
; develop a function that can be used to massage the output of
; TRANSLATE to introduce QUOTE notation for LITATOMs and to intro-
; duce the normal decimal representation for ADD1 nests.  While
; this convention is employed in Chapter 4, e.g., when we exhibit
; two token trees with the same translation, the concepts
; formalized below are not defined in the text.

; If X is an ADD1-nest n deep with a (ZERO) at the bottom, we
; return n; otherwise F.  The variable I is used as an accumulator
; and should be 0 at the top-level.
(DEFN ADD1-NESTP (X I)
  (IF (NLISTP X)
      F
      (IF (AND (EQUAL (CAR X) 'ZERO)
               (EQLEN X 1))
          I
          (IF (AND (EQUAL (CAR X) 'ADD1)
                   (EQLEN X 2))
```

```
              (ADD1-NESTP (CADRN 1 X) (ADD1 I))
              F))))
               (DEFN CONS-ADD1-NESTP (X)
  (IF (NLISTP X)
      F
      (IF (AND (EQUAL (CAR X) 'ZERO)
               (EQLEN X 1))
          0
          (IF (AND (EQUAL (CAR X) 'CONS)
                   (EQLEN X 3))
              (FCONS (ADD1-NESTP (CADRN 1 X) 0)
                     (CONS-ADD1-NESTP (CADRN 2 X)))
              F))))
              (DEFN EXPLODED-LITATOM (X)
  (IF (AND (EQLEN X 2)
        (AND (EQUAL (CAR X) 'PACK)
             (CONS-ADD1-NESTP (CADRN 1 X))))
      (LIST2 'QUOTE (PACK (CONS-ADD1-NESTP (CADRN 1 X))))
      F))
(DEFN ABBREV (FLG X)
  (IF FLG
      (IF (NLISTP X)
          X
          (IF (ADD1-NESTP X 0)
              (ADD1-NESTP X 0)
              (IF (EXPLODED-LITATOM X)
                  (EXPLODED-LITATOM X)
                  (CONS (CAR X)
                        (ABBREV F (CDR X))))))
      (IF (NLISTP X)
          NIL
          (CONS (ABBREV T (CAR X))
                (ABBREV F (CDR X))))))

; Here is a version of EXSYN that uses abbreviations.
(DEFN AEXSYN
 (STREAM)
  (IF (READABLE (READ-TOKEN-TREE STREAM) 0)
      (ABBREV T
       (TRANSLATE T
        (READMACRO-EXPANSION (READ-TOKEN-TREE STREAM))))
      F))

#|

Here are Common Lisp interfaces to EXSYN and AEXSYN.

 (defun exsyn (string)
  (*1*exsyn (ascii string)))

 (defun aexsyn (string)
  (*1*aexsyn (ascii string)))
```

```
|#

; We conclude by making a compiled library containing the current
; data base.  If one executes (NOTE-LIB ".../examples/basic/parser"
; T) in Nqthm, where the ellipsis is meant to be the local
; directory containing our examples subdirectory, then one can use
; R-LOOP to execute these function.  For example, one can type to
; R-LOOP:

;(AEXSYN
; (ASCII
; '(OPEN-PAREN
;   LOWER-A LOWER-D LOWER-D DIGIT-ONE
;   SPACE
;   NUMBER-SIGN VERTICAL-BAR
;   UPPER-C LOWER-O LOWER-M LOWER-M LOWER-E LOWER-N LOWER-T
;   VERTICAL-BAR NUMBER-SIGN
;   LOWER-X
;   CLOSE-PAREN)))

; and get the result '(ADD X).

; This is sufficiently cumbersome that we find the Lisp interface
; functions much more convenient.  If the ''defuns'' in this file
; are executed in an acceptable Common Lisp, then it is possible to
; type to Common Lisp (rather than R-LOOP):

; (aexsyn "(add1 #|Comment|#x)")

; and get the result (ADD1 X).  The Lisp routine aexsyn actually
; executes our logical function AEXSYN but it first converts the
; string argument into a list of ASCII characters.

(MAKE-LIB "parser" T)
```

References

1. W. Bevier. *A Verified Operating System Kernel*. Ph.D. Th., University of Texas at Austin, 1987.

2. R.S. Boyer, D.M. Goldschlag, M. Kaufmann, J S. Moore. Functional Instantiation in First-Order Logic. Tech. Rept. Technical Report 44, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, October, 1989.

3. 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.

4. R. S. Boyer, M. W. Green and J S. Moore. The Use of a Formal Simulator to Verify a Simple Real Time Control Program. In *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, Springer-Verlag Texts and Monographs in Computer Science, 1990", pp. 54-66.

5. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

6. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

7. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

8. R. S. Boyer and J S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm". *American Mathematical Monthly 91*, 3 (1984), 181-189.

9. R. S. Boyer and J S. Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In *Automated Theorem Proving: After 25 Years*, W.W. Bledsoe and D.W. Loveland, Eds., American Mathematical Society, Providence, R.I., 1984, pp. 133-167.

10. R. S. Boyer and J S. Moore. "A Mechanical Proof of the Unsolvability of the Halting Problem". *JACM 31*, 3 (1984), 441-458.

11. R. S. Boyer and J S. Moore. "The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover". *Journal of Automated Reasoning 4,* 2 (1988), 117-172.

12. R. S. Boyer and J S. Moore. MJRTY - A Fast Majority Vote Algorithm. In R.S. Boyer, Ed., *Automated Reasoning: Essays in Honor of Woody Bledsoe*, AutomatedReasoningSeries, 1991", pp. 105-117.

13. R.S. Boyer and Y. Yu. Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor. In *11th Conference on Automated Deduction, Lecture Notes in Computer Science*, D. Kapur, Ed., Springer-Verlag, 1992, pp. 416-430.

14. R.S. Boyer and Y. Yu. A Formal Specification of Some User Mode Instructions for the Motorola 68020. Tech. Rept. TR-92-04, Computer Sciences Department, University of Texas, Austin, February, 1992.

15. A. Bronstein and C. Talcott. String-Functional Semantics for Formal Verification of Synchronous Circuits, Report No. STAN-CS-88-1210. Computer Science Department, Stanford University, 1988.

16. A. Bronstein. *MLP: String-functional semantics and Boyer-Moore mechanization for the formal verification of synchronous circuits*. Ph.D. Th., Stanford University, 1989.

17. A. Bronstein and C. Talcott. Formal Verification of Synchronous Circuits based on String-Functional Semantics: The 7 Paillet Circuits in Boyer-Moore. C-Cube 1989 Workshop on Automatic Verification Methods for Finite State Systems. LNCS 407, 1989, pp. 317-333.

18. A. Bronstein and C. Talcott. Formal Verification of Pipelines based on String-Functional Semantics. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, 1989.

19. Cygnet. *Scribe: Document Production Software User Manual.* 1991.

20. A. Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation*. Ph.D. Th., University of Texas, 1992. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

21. R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, American Mathematical Society, Providence, Rhode Island, 1967, pp. 19-32.

22. M. Gardner. "Mathematical Recreation Column". *Scientific American 203*, 2 (August, 1960), 149-154.

23. N. Gilbreath. "Magnetic Colors". *The Linking Ring 38*, 5 (1958), 60.

24. D.M. Goldschlag. Mechanizing Unity. In *Programming Concepts and Methods*, M. Broy and C. B. Jones, Eds., North Holland, Amsterdam, 1990.

25. David M. Goldschlag. "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover". *IEEE Transactions on Software Engineering 16* (September 1990).

26. M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. Tech. Rept. 103, University of Cambridge, Computer Laboratory, 1987.

27. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *Comm. ACM 12*, 10 (1969), 576-583.

28. G. Huet. The Gilbreath Trick: A Case Study in Axiomatization and Proof Development in the COQ Proof Assistant. Tech. Rept. Technical Report 1511, INRIA, Domaine de Voluceau, Rocquencourt B.P. 105, 78153 Le Chesnay Cedex, France, 1991.

29. W.A. Hunt, Jr. *FM8501: A Verified Microprocessor*. Ph.D. Th., University of Texas at Austin, 1985.

30. W.A. Hunt, Jr. and B. Brock . "A Formal HDL and its use in the FM9001 Verification". *Proceedings of the Royal Society (to appear, April 1992)* (1992).

31. M. Kaufmann. Addition of Free Variables to an Interactive Enchancement of the Boyer-Moore Theorem Prover. Tech. Rept. 42, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, 1990.

32. M. Kaufmann. An Instructive Example for Beginning Users of the Boyer-Moore Theorem Prover. Tech. Rept. Internal Note 185, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, April, 1990.

33. M. Kaufmann. "Generalization in the Presence of Free Variables: A Mechanically-Checked Proof for One Algorithm". *Journal of Automated Reasoning 7*, 1 (March 1991).

34. M. Kaufmann. A Simple Example for Nqthm: Modeling Locking. Tech. Rept. Internal Note 216, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, February, 1991.

35. M. Kaufmann. An Example in Nqthm: Ramsey's Theorem. Tech. Rept. Internal Note 100, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, May, 1991.

36. Matt Kaufmann. "An Extension of the Boyer-Moore Theorem Prover to Support First-Order Quantification". *J. Automated Reasoning 9,* 3 (December 1992), 355-372.

37. Matthew Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Technical Report 19, Computational Logic, Inc., May, 1988.

38. B.W. Kernighan and D.M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

39. L. Lamport. *LaTex: A Document Preparation System.* Addison-Wesley Publishing Company, 1986.

40. J. McCarthy. Towards a Mathematical Science of Computation. Proceedings of IFIP Congress, 1962, pp. 21-28.

41. J S. Moore. "A Mechanical Proof of the Termination of Takeuchi's Function". *Information Processing Letters 9,* 4 (1979), 176-181.

42. J S. Moore. Piton: A Verified Assembly-Level Language. Tech. Rept. CLI-22, Computational Logic, Inc., Austin, Tx, June, 1988.

43. J S. Moore. A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol. Tech. Rept. CLI Technical Report 68, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, August, 1991.

44. M. Nagayama and C. Talcott. An NQTHM Mechanization of ''An Exercise in the Verification of Multi-Process Programs''. Tech. Rept. STAN-CS-91-1370, Computer Science Department, Stanford University, 1991.

45. M. Kaufmann. An Integer Library for Nqthm. Tech. Rept. Internal Note 182, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, March, 1990.

46. W.R. Bevier. A Library for Hardware Verification. Tech. Rept. Internal Note 57, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, August, 1988.

47. D.M. Russinoff. A Mechanical Proof of Wilson's Theorem. Department of Computer Sciences, University of Texas at Austin, 1983.

48. D.M. Russinoff. "A Mechanical Proof of Quadratic Reciprocity". *Journal of Automated Reasoning 8,* 1 (1992), 3-21.

49. N. Shankar. "Towards Mechanical Metamathematics". *Journal of Automated Reasoning 1*, 4 (1985), 407-434.

50. N. Shankar. *Proof Checking Metamathematics*. Ph.D. Th., University of Texas at Austin, 1986.

51. N. Shankar. "A Mechanical Proof of the Church-Rosser Theorem". *JACM 35*, 3 (1988), 475-522.

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

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

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

55. W. Teitelman. INTERLISP Reference Manual. Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, Ca., 1978.

56. D.A. Turner. "A New Implementation Technique for Applicative Languages". *Software -- Practice and Experience 9* (1979), 31-49.

57. D. Verkest, L. Claesen, H. De Man. On the use of the Boyer-Moore theorem prover for correctness proofs of parameterized hardware modules. In *Formal VLSI Specification and Synthesis*, L. Claesen, Eds., Elsevier Science Publishers (North Holland), 1990, pp. 99-116.

58. M. Wilding. "Proving Matijasevich's Lemma with a Default Arithmetic Strategy". *Journal of Automated Reasoning 7*, 3 (1991), 439-446.

59. W.D. Young. A Verified Code-Generator for a Subset of Gypsy. University of Texas at Austin, 1988. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

60. W.D. Young. A Simple Expression Compiler: A Programming and Proof Example for an Nqthm Course. Tech. Rept. Internal Note 210, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, November, 1990.

61. Y. Yu. "Computer Proofs in Group Theory". *Journal of Automated Reasoning 6*, 3 (1990).

62. Y. Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. Ph.D. Th., University of Texas, 1992.

Index

V

W

Z

Contents