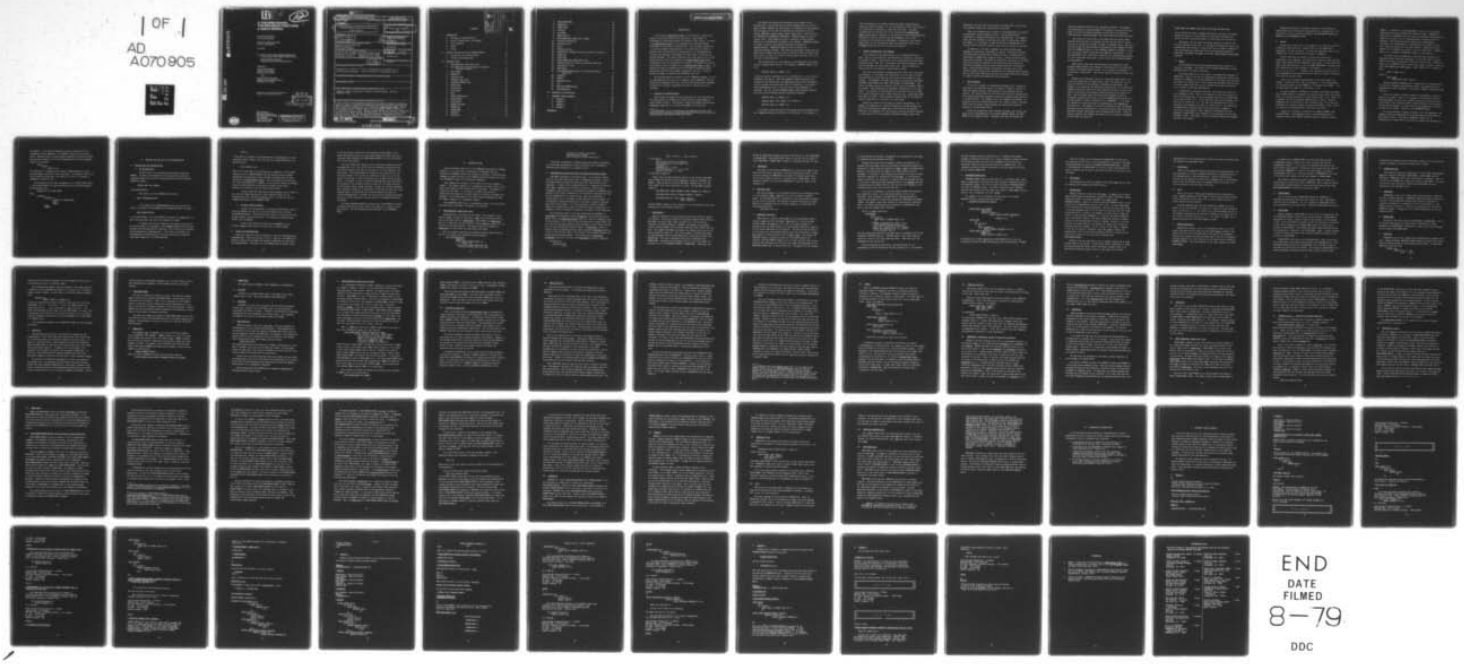
AD-A070 905 SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 9/2
A THEOREM-PROVER FOR RECURSIVE FUNCTIONS: A USER'S MANUAL.(U)
JUN 79 R S BOYER, J S MOORE N00014-75-C-0816
UNCLASSIFIED SRI/CSL-91 NL

| OF |
AD
A070905



END
DATE
FILMED
8-79
DDC

LEVEL II

12

A THEOREM-PROVER FOR RECURSIVE FUNCTIONS: A USER'S MANUAL

Technical Report CSL-91
SRI Projects 4079/6494

Contract No. N00014-75-C-0816
Grant No. MCS-7681425

June 1979

By: Robert S. Boyer, Senior Research Mathematician
J Strother Moore, Senior Research Mathematician

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Office of Naval Research
Department of the Navy
Arlington, Virginia 22217

and

National Science Foundation
Software Systems Science Program
Washington, D.C. 20550

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MNP
TWX: 910-373-1246

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) SRI

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CSL-91

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): (6) A Theorem-Prover for Recursive Functions: A User's Manual

5. TYPE OF REPORT & PERIOD COVERED: (9) Technical Rept.

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): (10) Robert S. Boyer, J. Strother Moore

8. CONTRACT OR GRANT NUMBER(s): (15) N00014-75-C-0816, NSF-MCS-7681425

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Computer Science Laboratory
SRI International
Menlo Park, CA 94025

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
NR 049-378

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
Department of the Navy
Arlington, VA 22217

Software Systems Science
National Science Found.
Washington, D.C. 20550

12. REPORT DATE: (11) June 79

13. NO. OF PAGES: iii+61

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)

(12) 66p.

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Reproduction in whole or in part is permitted for any purpose of the United States Government. It may be released to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer program verification, automatic theorem-proving, induction, INTERLISP, LISP

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A user's manual for an automatic theorem-proving computer program is presented. (Both the theory of recursive functions and the proof techniques underlying the theorem-prover are presented in the forthcoming A Computational Logic, Academic Press, 1979.) We here describe how to use the program, which is written in the INTERLISP dialect of LISP. Matters covered include our syntactic conventions; starting up the theorem-prover; defining functions; proving and storing lemmas; undoing previous work; and examining and saving the theorem-prover's state. More than 30 user commands are defined and described. Simple examples are given.

DD FORM 1 JAN 73 1473
EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410646

CONTENTS

I INTRODUCTION*

In our book A Computational Logic [1] we describe a formal logic based on recursive functions, and we present a large number of techniques for discovering proofs of theorems in that theory. As noted in A Computational Logic we have implemented our theorem-proving techniques in an automatic theorem-prover. This document explains how to use that automatic theorem-prover. We assume that the reader is familiar with the motivation and orientation of our theorem-proving research and completely understands the formal logic in which we are proving theorems. The reader unfamiliar with these aspects of our work should read the first five chapters of A Computational Logic before attempting to use the theorem-prover seriously. Throughout the rest of this manual we refer to Chapter n of A Computational Logic as ACL n. We assume the reader is familiar with the DEC TOPS-20 operating system, particularly with the file naming conventions, (see [2]). We also assume the reader is familiar with INTERLISP (see [3]).

The theorem-prover is actually a set of INTERLISP programs. To use the theorem-prover you log on to a computer, enter INTERLISP, load the appropriate theorem-prover files, and then call theorem-prover programs to define new concepts, axiomatize new types of objects, and prove theorems.

A. Teaching the Theorem-Prover

While using our program you will spend most of your time teaching the system about the concepts you define and their relationships to other defined concepts. The system is taught by defining functions and suggesting lemmas for it to prove and remember for future use.

---

* The development of our theorem-prover was supported in part by the National Science Foundation under Grant MCS-7681425 and by the Office of Naval Research under Contract N0014-75-C-0816.

The system uses axioms and previously proved lemmas in four distinct ways. The system does not decide automatically how to use a given theorem; whenever any new theorem is introduced, you must specify how the lemma is to be used by providing the system with a set of tokens, INTERLISP literal atoms, called "lemma types," taken from the set {REWRITE ELIM GENERALIZE INDUCTION}. REWRITE lemmas are used to rewrite terms. ELIM lemmas are used to eliminate certain function symbols by re-representing variables in the problem. GENERALIZE lemmas are used to restrict the range of variables introduced by generalizations. INDUCTION lemmas are used to justify new induction/recursion schemes. Note that by introducing a lemma with the empty list of lemma types you cause the system to name the formula and save it, but to store it so that it will be used in no way (i.e., it is not used in future proofs).

The theorem-prover is very sensitive to the syntactic form chosen by you to express each new fact. For example, a REWRITE lemma of the form

```
(IMPLIES (AND p q) (EQUAL r s))
```

is used to rewrite (instances of) r to s provided the system can first establish p and then q. Note the asymmetry between hypothesis and conclusion, and between left- and right-hand sides of the conclusion. In fact, because the system must limit the resources it is willing to spend establishing p and q, even the order of the hypotheses is relevant. That is, the above REWRITE lemma causes different behavior than any of the following logically equivalent formulas:

```
(IMPLIES (AND p q) (EQUAL s r))

(IMPLIES (AND p (NOT (EQUAL r s))) (NOT q))

(IMPLIES (AND q p) (EQUAL r s))
```

To become an effective user of the system you must understand how your commands influence the behavior of the system. It is possible to

infer the meaning of the various lemma types after enough hands-on experience with the system. (It is also possible to infer the structure of a brick wall by battering it down with your head.) We highly recommend that the serious user of the system read A Computational Logic, paying special attention to the heuristics controlling lemmas and definitions, and the carefully explained examples. For a brief description of the syntactic requirements on formulas of the various lemma types, see the discussions under each lemma type name in the REFERENCE GUIDE (Section III of this User's Manual).

B. Events, Dependencies, and Commands

Every definition and theorem known to the theorem-prover has a name. The act of introducing a new definition or theorem to the system is called an "event." Some events, such as definitions, are naturally associated with a name (e.g., the name of the function defined); others, such as theorems, are given names by the user. See the subsection Syntax below for the syntactic restrictions on names.

The basic theorem-prover commands are those that create new events: the addition of a new axiom, the introduction of a new type of object, the definition of a new function, and the proof and storage of a new theorem. The INTERLISP functions that create new events are ADD.AXIOM, ADD.SHELL, BOOT.STRAP, DCL, DEFN, PROVE.LEMMA, and MOVE.LEMMA. Some events introduce several new axioms, each of which is assigned a name by the system. These subevents are considered "satellites" of the "main event." For example, the introduction of a new shell type (e.g. CONS) is one event that gives rise to many subevents (e.g., axioms about CONS, CAR, CDR, and LISTP). Every theorem or function name in the system is either a main event or a satellite of a main event.

Events are related to each other by logical dependencies. For example, the admission of a certain formula as a theorem depends upon all of the functions and lemmas used in the proof of the theorem. Similarly, the admission of a new recursive function definition depends not only upon all of the previously introduced concepts used in the

definition, but also upon the functions and lemmas used to prove that the proposed "definition" truly defines a function.

Thus, the theorem-prover's "state" or "knowledge base" is actually a noncircular, directed graph of events. The theorem-prover's performance is largely determined by its knowledge base. For example, there are many theorems it can prove only after it has proved certain key lemmas. It is possible to dump the system's knowledge base to a "library file" to save the system's state from one session to the next.

In addition to the basic commands, the system provides many commands for operating on the graph of events: obtaining the events that depend upon a given event, undoing an event, or editing and re-executing the command that created an event. Several functions delete events from the graph. The graph is always kept consistent in the sense that when an event is deleted all the events that are (directly or indirectly) dependent upon the event being deleted are also deleted. Thus, if after proving several theorems you find that one of your earliest defined concepts was inconveniently or inappropriately defined, you can "undo" that definition and lose only those results whose meaning or logical validity may depend upon that definition.

C. Error Handling

If you try to execute an inappropriate command (e.g., assign the same name to two different events, or attempt to define a function in terms of unknown concepts) self-explanatory error messages will be printed. The system checks for over 100 errors and has a sophisticated error handling mechanism designed to keep the theorem-proving machine in a consistent state. For example, when a new command is processed, all possible errors are checked before the first change is made to the data base, since an abortion midway through the update would leave the machine in an unacceptable state.

Errors are grouped into three classes, warnings, soft errors, and fatal errors (distinguished by the headers WARNING, ERROR, and FATAL ERROR in the message printed). Warnings arise when the system has

detected something unusual but not logically incorrect. For example, the system prints a warning message if you define a function but do not refer to one of the formal parameters in the body of the function. After printing a warning message, the system continues normal execution.

Soft errors are true errors in the sense that the system cannot continue until the error is repaired, but they are the type of errors that can be repaired by editing a formula or changing a name. When such an error occurs the system prints an explanatory error message and then calls the INTERLISP editor on the offending command if possible. If you exit the editor normally (with OK) the command is re-executed. If you exit abnormally (e.g., with STOP or CTRL-D), the theorem-proving machine is left in the same clean state it was in before the offending command was encountered.

Fatal errors occur when system resources are exhausted or when internal checks indicate the presence of inconsistency in the data base or bugs in the theorem-prover itself. It is usually not possible to proceed past a fatal error. Such errors should be reported to Boyer and Moore.

Despite our precautions, there is a relatively simple way for you to get the system into an illegal state: abort with control B, D, or E while the system is in the process of updating its data base. For example, if you interrupt the definition process while it is iteratively computing the output type of the newly defined function, the system may appear to believe in an inaccurate characterization of the function's type. Such an illegal state will more usually manifest itself by ultimately causing a fatal error or even an INTERLISP error (which is to our system what a "trap at location n" is to INTERLISP). It is often not possible to recover from such an interruption, even immediately after the fact. For example if you abort a definition after the system has begun to store facts about it, you cannot properly undo the aborted definition because insufficient undo information was stored. Since you cannot in general tell whether the system is in the process of updating the data base, the moral is:

NEVER ABORT ANY COMMAND THAT MIGHT BE UPDATING THE DATA BASE.

If you realize you typed a command incorrectly, wait until the system detects an error in it (after which you can abort from within the editor) or until the system accepts it (after which you can undo it).

Eventually, we will invest the effort necessary to protect the critical sections of our code. At the moment we assume the users to be sophisticated and friendly -- and we don't put our stamp of approval on any proof except one constructed from an uninterrupted sequence of BOOT.STRAP, ADD.AXIOM, ADD.SHELL, DCL, DEFN, PROVE.LEMMA and MOVE.LEMMA commands.

D. Output

The theorem-prover prints an English description of what it is doing as it proceeds. The values of the variables PROVE.FILE and TTY: determine where its output is printed.

The system directs all of its output, including error messages, to PROVE.FILE. Thus, if you are not interested in seeing the output from a sequence of commands, you can set PROVE.FILE to the name of an open file and look at the output later. This is particularly useful when you want to collect some proofs that you are confident the system can perform (e.g., because it has done them before).

If PROVE.FILE is different from TTY:, the system also prints its error messages to TTY:. Thus, if you wanted to write your proofs to a disk file but would like to see any messages that arise, set PROVE.FILE to the disk and TTY: to T. If you would like even error messages to be directed to a disk file (i.e., you want the theorem-prover to print nothing to the terminal), set TTY: to a disk file too. By setting TTY: and PROVE.FILE to different disk files you will get a complete transcript of the system's output (possibly interleaved with error messages) in PROVE.FILE, and a separate file containing just the error messages in TTY:. This makes it easy to determine whether any messages occurred. None should.

Rigging the system to run without printing to the terminal is useful when you are running the theorem-prover in a detached job. For the details of how to run while detached, see REDO.UNDONE.EVENTS and PROVEALL (in the REFERENCE GUIDE).

E. Syntax

As you are no doubt aware, all formulas in our logic are written in a LISP-like prefix notation. Indeed, they are just INTERLISP S-expressions. We have no parser. We find this convenient since at the command level in our system you are typing to INTERLISP.

Certain rules must be obeyed in writing down names and S-expressions in our language. All names (e.g., function names, variable names, event names) must be INTERLISP literal atoms. You may only form names using the characters in the list LEGAL.NAME.CHARS. Currently that list contains upper case A through Z, 0 through 9, ., -, /, \, >, <, =, @, +, *, &, %, $, #, ~, ^. Finally, every name must start with one of the 26 alphabetic characters and must not end with a period, and must be more than one character long.

T and F are acceptable abbreviations for the terms (TRUE) and (FALSE). You may not use T and F as variables. 0 is an abbreviation for the bottom object of type NUMBERP: the constant term (ZERO). 1 is an abbreviation for (ADD1 0), 2 for (ADD1 1), etc.

As noted in ACL III, LITATOMs in our theory are either the bottom-most LITATOM (which is returned by the constant function NIL and written (NIL)) or are constructed by the unary function PACK. Thus, (PACK x) is a LITATOM. NIL is an acceptable abbreviation for (NIL). (Because INTERLISP does not allow NIL to have a property list, NIL is not used internally as a function symbol; instead we use NIHIL). Recall that in our logic NIL is different from F, the false truth value.

In ACL III, we present a convention for naming LITATOMs using quotation marks. Since ACL was written, we have abandoned that convention and we have adopted another convention, using the syntax

(QUOTE ...) familiar to LISP programmers. That is, in our current theorem-prover we write (QUOTE atm) instead of "atm". (QUOTE atm), where atm is an INTERLISP literal atom is an acceptable abbreviation for (PACK n), for some n; the n's corresponding to different atm's are different. The system arbitrarily decides the correspondence. (Currently, the correspondence is determinced by the order of first use.) Thus you should never use PACK directly. (QUOTE NIL) is NIL. (QUOTE T) is a LITATOM, and is not T, the true truth value. (QUOTE x), where x is a nonnegative INTERLISP integer, is the same as x.

The reason that we abandoned the quotation mark convention for denoting literal atoms used in ACL was to obtain the effect that the QUOTE in LISP has on "arguments" other than literal atoms. (QUOTE x), where x is an INTERLISP list structure (car . cdr), is taken to mean the term (CONS (QUOTE car) (QUOTE cdr)). Thus, (QUOTE (0 1)) is taken to mean:

```
(CONS 0 (CONS 1 NIL))
```

which is

```
(CONS (ZERO)
      (CONS (ADD1 (ZERO)) (NIL))).
```

Function symbols beginning with C, ending with R, containing only A's and D's in between, and differing from CR, are abbreviations for compositions of CAR's and CDR's. For example, (CADDR X) means (CAR (CDR (CDR X))).

Before you mention a function symbol in an expression it must be known to the system. Thus, if you misspell a function name (and the resulting symbol is not in fact a known function) an error will occur. If you wish to use a function symbol without defining or introducing it via the shell principle, you must first use the program DCL to declare the name to the system. (See DCL in the REFERENCE GUIDE.)

It is not permitted to apply a function to too few arguments. For example, giving the theorem-prover an S-expression involving (CONS X) will cause an error. Note that in INTERLISP, (CONS X) is an abbreviation for (CONS X NIL); but we do not support that convention in

our theory. It is also not permitted to give a function of 0 or 1 arguments too many arguments. For example, (CAR A B) will cause an error. However we have a rather unusual handling of the case in which you provide a function of 2 or more arguments with too many arguments. (PLUS X Y Z U) means:

```
(PLUS Z
      (PLUS Y
            (PLUS Z U)))
```

For functions of 2 arguments this default right-association in the second argument position is quite convenient. For example, (AND P Q R) is just what you would want, and (CONS A B C D NIL) is the same as INTERLISP's (LIST A B C D).

For functions of more than 2 arguments, e.g., BIGPLUS which takes 4, the second argument is right-associated and the trailing arguments are duplicated, e.g.,

```
(BIGPLUS X Y U V BASE CARRY)
```

means

```
(BIGPLUS X
         (BIGPLUS Y
                  (BIGPLUS U V BASE CARRY)
                  BASE
                  CARRY)
         BASE
         CARRY).
```

II GETTING INTO AND OUT OF THE THEOREM-PROVER

A. Getting Into the Theorem-Prover

1. On the SRI KL-10

On SRI's KL-10, the theorem-prover resides on directory <MOORE>. To start the theorem-prover you should first inform the operating system that you will be requiring files found on directory <MOORE> by typing:

```
@DEFINE DSK: DSK:,<MOORE>
```

to the TOPS-20 EXEC.

Then start up a fresh INTERLISP and execute:

```
(LOAD '<MOORE>CODE.INIT)
```

If the SYSOUT file <MOORE>THM.EXE exists, you can get the effect of starting a fresh INTERLISP and loading CODE.INIT by typing:

```
@RUN <MOORE>THM.EXE
```

to the TOPS-20 EXEC. When this SYSOUT file exists it is much faster to enter the system this way than by the sequence of LOADs.

To complete the initial loading of the system, you must call the function INIT. As explained in the REFERENCE GUIDE section of this document, INIT performs several functions, one of which is to initialize the theorem-prover's knowledge base. If you wish to initialize the theorem-prover's knowledge base to the most primitive one available (described in ACL III), you should execute:

(INIT T)

If you wish to initialize the theorem-prover's knowledge base to some previously saved "library file" (see the user command MAKE.LIB), you should execute:

(INIT (QUOTE file))

where file is the name of the library file in question. The standard library we use is <MOORE>PROVEALL.LIB and contains several hundred definitions and previously proved theorems (including most of those described in A Computational Logic). You are welcome to use this file for early experimentation. However, if you intend to apply the theorem-prover to some particular and difficult problem you may prefer eventually to construct your own library from scratch.

Having called INIT with some non-NIL argument you are ready to go: your core image contains the latest "blessed" version of the theorem-prover and a specified knowledge base. You may then use the commands described in the REFERENCE GUIDE.

2. On Other TOPS-20 Systems

To move the theorem-prover to another TOPS-20 system, create a directory <MOORE> and copy to it the files named in the file [SRI-KL]<MOORE>CODE.FILES. The files should be copied to files with the same version numbers as the source files. The files may be obtained from SRI-KL over the ARPANET.

If you cannot obtain the files via the ARPANET, we will provide a magnetic tape copy of the files for a nominal charge.

B. Saving the Knowledge Base

When your session is over you may want to save the theorem-prover's knowledge base. There are two ways to do it. One is to use MAKE.LIB to produce a library file suitable for giving to INIT when you next enter the theorem-prover. The other is to use EVENTS.SINCE to obtain a list

of all the axioms, definitions, and theorems you have added to the system during this session, to save those formulas on some file, and to reprocess those formulas with REDO.UNDONE.EVENTS as the first official act of your next session.

The first method may consume lots of disk space because the entire knowledge base (i.e., the old one plus the extension performed during the current session) will have to be saved; but it only requires a few CPU seconds. The second method of saving your work requires only the disk space needed to write down the formulas you typed in; but it may require a good deal of time to reprocess those formulas upon your next entry. For example, to save the entire knowledge base after proving all the theorems in our standard library requires 122 file pages and about 30 CPU seconds (total) coming and going. Saving the formulas themselves only requires 22 pages but it takes about 2 CPU hours to prove all those formulas again. We generally use the second method only when our initial knowledge base is already very large and the extension performed during the current session is no more than 10 or 20 simple definitions and lemmas.

The usual INTERLISP method of saving state, using SYSOUT to produce a runnable EXE file with your current core image in it, should be used with caution. See the discussion under NOTE.FILE in the REFERENCE GUIDE.

III REFERENCE GUIDE

Unless otherwise noted, all of our INTERLISP functions are "lambda-spread" -- i.e., they take a fixed number of arguments that are evaluated by INTERLISP before the function is executed. Since most user commands are typed directly to the INTERLISP top-level executive, we use the "EVALQUOTE" syntax to illustrate the commands.

Each command that creates an event takes as its last argument a comment. The comment argument to a command must be either NIL or a list whose first element is the literal atom * (asterisk). The comment is stored with the event. It is useful to comment your function definitions and lemmas for the same reason it is useful to comment your programs. In the description of the event-creating commands, we omit the comment argument for typographic reasons.

This REFERENCE GUIDE lists, in alphabetical order, all of the user-level functions, variables, and tokens in our system.

A. ADD.AXIOM(name lemma.types term)

ADD.AXIOM adds a new axiom to the system. The statement of the axiom is term, the name of the axiom is name, and the axiom is used in each of the ways listed in lemma.types. Each element of lemma.types must be a lemma type, and the syntactic form of term must permit its use in each of the ways specified. The restrictions on the members of lemma.types are given under the individual type names.

Here is a sample axiom about an undefined function APPLY (see DCL for how to introduce new, undefined function symbols):

```
_ADD.AXIOM(APPLY.PLUS
           (REWRITE)
           (EQUAL (APPLY (QUOTE PLUS) X Y)
                  (PLUS X Y))
           (* This axiom, named APPLY.PLUS and
              used as a rewrite rule only, says
```

```
                that for all X and Y, the result
                of APPLYing the LITATOM
                PLUS to X and Y is their Peano sum.))
```

We provide a mechanism for adding axioms with extreme reluctance. It is very easy and painless to beg the question with a mechanical theorem-prover by asking it to assume too much.

B. ADD.SHELL(shell.name btm.object recognizer destructor.tuples)

ADD.SHELL axiomatizes a new shell class. In our theory, shells play the role that "data types" play in programming language. Shells are typed ordered n-tuples, possibly with type restrictions on the components. A shell class is characterized by a "constructor" function that takes n arguments and returns an n-tuple of a unique type; a "recognizer" that returns T or F according to whether its argument is an element of that type; n "destructor" (or "accessor") functions that dig out the components of an n-tuple of that type; and, optionally, a "bottom object" (i.e., an object of that type but not an n-tuple). It is possible to specify restrictions on the types of objects occupying each slot of the n-tuple. It is also possible to specify "default values" to be used when an object of the wrong type is supplied for a component or when an accessor is applied to an object of the wrong type.

The constructor name for the type axiomatized by ADD.SHELL is shell.name; the recognizer name is recognizer; and the destructors are given in the list destructor.tuples, which is a list of n elements, each of the form (ac tr dv). The ac's are the destructor function names, the tr's are the type restrictions on each component in the n-tuple, and the dv's are the corresponding default values. If btm.object is NIL, no default object is supplied with the type; otherwise, btm.object must be a list of the form (b), where b is the name of the constant function for constructing the bottom object. The restrictions on shell.name, recognizer, the ac's, tr's, dv's, and btm.object are given in ACL III.

The effect of:

```
_ADD.SHELL(const
           (b)
```

```
                    r
                    ((acl trl dvl) ... (acn trn dvn)))
```

is the same as:

> Add the shell const of n arguments
> with (optionally, bottom object (b),)
> recognizer r,
> accessors acl, ..., acn,
> type restrictions trl, ..., trn, and
> default values dvl, ..., dvn.

as described in ACL III.

Recall that the most primitive version of the system's knowledge base includes the axioms for the nonnegative integers, LITATOMs, and CONSes. These axioms are added by ADD.SHELL commands built into the "boot strap" that occurs when INIT is called. The ADD.SHELL commands used are:

```
_ADD.SHELL(ADD1 (ZERO) NUMBERP ((SUB1 (NUMBERP X1) (ZERO))))

_ADD.SHELL(PACK (NIHIL) LITATOM ((UNPACK T (ZERO))))

_ADD.SHELL(CONS NIL LISTP ((CAR T (NIHIL))
                           (CDR T (NIHIL))))
```

(We use (NIHIL) instead of (NIL) because it is not possible to alter the property list of the literal atom NIL.)

C. BOOT.STRAP()

ACL III describes the most primitive theory with which our system will operate. The theory includes the functions symbols TRUE, FALSE, IF and EQUAL; the shells ADD1, PACK, and CONS (i.e., the axioms and functions for Peano arithmetic, LITATOMs, and CONSes); the standard measure function COUNT; and the defined functions LESSP, ZEROP, FIX, and PLUS. The command BOOT.STRAP initializes the system's data base to the theory described in ACL III. All of the axioms and definitions added are made dependent upon the BOOT.STRAP event, which has the event name GROUND.ZERO. Thus every subsequent event will be dependent upon GROUND.ZERO. One method of obtaining a list of all of the events in a data base is to call DEPENDENT.EVENTS on GROUND.ZERO. Similarly, one

method of completely erasing the system's data base is to call UNDO.NAME on GROUND.ZERO. Before BOOT.STRAP begins to add the initial axioms it calls UNDO.NAME on GROUND.ZERO to erase the existing data base (if any).

D. CHRONOLOGY

The value of the variable CHRONOLOGY is a list of the names of the events in the current data base, in reverse chronological order. Thus, the first element of CHRONOLOGY is the name of the most recent event, and the last element is name of the oldest event, GROUND.ZERO. After loading a library, you can print CHRONOLOGY to learn the names of the events in it.

E. DCL(name args)

DCL declares name to be an undefined function of n arguments, where n is the length of args. args must be a list of distinct variable names. An undefined function, once DCLed, can be constrained by axioms, but can never be defined. For example, to extend the syntax by adding APPLY as an undefined function of three arguments, one would make the following declaration:

```
_DCL(APPLY (FN X Y))
```

F. DEFN(name args body)

DEFN defines a function named name, with formal argument names as listed in args, and with body body. Args must be a list of distinct variable names, and body must be a term. The system attempts to establish that the restrictions on the definitional principle of ACL III are met before admitting the new function. The most severe restriction is that there be a well-founded relation such that some measure of args gets smaller in every recursive call of name in body. The system uses INDUCTION lemmas to guide its search for a suitable measure and relation, and uses REWRITE lemmas to try to prove that the tests in the function body governing each recursive call imply the hypotheses in the INDUCTION lemmas used to justify each call. See ACL III for the details

of the definitional principle, and ACL XIV for the details of the search for measures and well-founded relations.

Together with some trivial syntactic checks, the existence of a measure and well-founded relation justifying a definitional equation is sufficient to ensure that the equation of (name . args) with body indeed defines a function and may be added to the theory without loss of soundness. In addition, these measures and relations are used to invent induction arguments for conjectures involving name. If the system is unable to establish the existence of a measure and well-founded function it will print a warning message explaining that it has assumed that such a measure and well-founded relation exist and that the soundness of the resulting theory rests entirely with the user.

It is to your advantage to teach the system enough about measures and relations to allow it to find justifications of all of your functions. This teaching is accomplished by defining measures as functions and having the system prove appropriate INDUCTION and REWRITE lemmas. The system is much more flexible in inventing good inductions when it is working with measures and relations it "understands" than when it has only assumed their existence.

Here is a sample definition.

```
_DEFN(APPEND
      (X Y)
      (IF (LISTP X)
          (CONS (CAR X) (APPEND (CDR X) Y))
          Y)
      (* APPEND concatenates two lists.  It is
         a well-defined function because the
         size of X, as measured by the function
         COUNT, gets smaller according to LESSP
         in the recursive call.))
```

Note the difference between the way DEFN takes its arguments and the way INTERLISP's DEFINEQ takes its. DEFINEQ takes a list, each element of which is a triple of the form (name args body). DEFN takes just the components of one such triple.

After processing the definition, the system prints out an explanation of why the definition has been accepted (i.e., the measures

and well-founded relations justifying it). In addition the system attempts to determine the types (i.e., shell classes) of output returned by the function and prints out this information in the form of an "observed" lemma that has also been stored as a REWRITE rule. The information printed by DEFN is directed to the file named by the value of the variable PROVE.FILE.

G. DEPENDENT.EVENTS(name)

DEPENDENT.EVENTS takes an event name (i.e., a function name, axiom name, or lemma name) and returns the list of events reachable from it in the dependency graph. The list of events returned by DEPENDENT.EVENTS is a list of INTERLISP forms. The CAR of each form is the INTERLISP function that created the event name (e.g., DEFN or PROVE.LEMMA) and the CDR is a list of arguments for that command.

For example, if you initialize the data base to our current PROVEALL.LIB file, the prettyprinted value of:

```
_DEPENDENT.EVENTS(LESSP.LENGTH)
```

is:

```
((PROVE.LEMMA LESSP.LENGTH
              (INDUCTION)
              (IMPLIES (LISTP L)
                       (LESSP (LENGTH (DELETE (MAXIMUM L)
                                              L))
                              (LENGTH L))))
 (DEFN DSORT
       (L)
       (IF (NLISTP L)
           NIL
           (CONS (MAXIMUM L)
                 (DSORT (DELETE (MAXIMUM L) L)))))
 (PROVE.LEMMA DSORT.SORT2
              (REWRITE)
              (EQUAL (DSORT X) (SORT2 X))))
```

The definition of DSORT depends upon LESSP.LENGTH and in turn, the statement and proof of DSORT.SORT2 depends upon the definition of DSORT.

Since all events can be reached from GROUND.ZERO, you will get a total picture of the state of the theorem-prover's "acquired" knowledge by prettyprinting the value of DEPENDENT.EVENTS(GROUND.ZERO). Looking at the dependents of GROUND.ZERO is one way to begin investigating the kind of requests that we type in and which functions you can use in your functions.

H. EDITC(name)

With EDITC you can edit the comment of the event name and not incur the undoing (and redoing) that EDITEV entails.

I. EDITEV(name)

EDITEV allows you to edit the event name. If name is not an event name, EDITEV prints an appropriate message and exits. Otherwise, EDITEV obtains the list of all events dependent upon name (using DEPENDENT.EVENTS), copies it, and then calls the INTERLISP editor on the entire list of dependencies with the first event -- the one named name -- as the editor's "current expression." You may edit that event or any of the ones dependent upon it (it is often the case that when you change the definition of a function, for example, you also wish to change the "calls" of that function elsewhere). If you exit the editor abnormally (e.g., with STOP or CTRL-D), even after making some tentative changes, no part of the theorem-prover's state is changed. However, if you exit with OK, the system checks to see whether the edited list of events is different from the original. If not, no part of the state is changed. If the list of events is different, then event name is undone (and consequently so are all the events dependent upon it), and each of the events in the result of your edit are re-executed with REDO.UNDONE.EVENTS.

EDITEV is the only safe way for you to change a definition or lemma already added to the system. In particular, after a definition or lemma has been processed and "built into" the system, trying to edit it by destructively editing property lists or other parts of the actual

representation of the theorem-prover's state will almost certainly leave the system in an unreliable state.

J. EDITV(name)

EDITV is the standard Interlisp function for editing variables. We have redefined it so that it behaves differently on variables used in the implementation of the theorem-prover. However, EDITV behaves normally on any variable not used in the implementation of the theorem-prover, with the exception that the message "Should you declare it?" will be printed at the conclusion of the edit.

K. ELIM

ELIM is one of the "lemma types" specifying how a lemma can be used. Roughly speaking, an ELIM lemma is used to replace some variable in the conjecture by a new term, so as to allow certain rewrites to remove certain function symbols.

An ELIM lemma must have the form (IMPLIES hyp (EQUAL lhs var)), where (1) var is a variable, (2) there is at least one proper subterm of lhs of the form (d v1 ... vn), where d is a function symbol and the vi are distinct variables and are the only variables in the lemma, and (3) var occurs in lhs only in such (d v1 ... vn) terms. See ACL X for a detailed description of how ELIM lemmas are used.

L. EVENTS.SINCE(event)

EVENTS.SINCE returns a list, in chronological order, of all the events that have occurred since the event named event occurred. Each event on the list is a form whose CAR is the name of the INTERLISP function that created the event and whose CDR is the list of arguments to that function (i.e., the events are in the same form as returned by DEPENDENT.EVENTS). The list includes, as its first element, the event named event.

A typical use of EVENTS.SINCE is to ascertain what you have accomplished during a session (i.e., what you have done and not subsequently undone). It is up to you to determine the name of the first event of the session. It may be of use to know that the value of the variable CHRONOLOGY is a list of all event names, in reverse chronological order (i.e., the first element of CHRONOLOGY is the name of the last event created, and the last element is GROUND.ZERO).

One method of saving the system's logical state is to save the events since you last did an INIT or NOTE.FILE, and to bring the system up next time by initializing to the same old data base and then using REDO.UNDONE.EVENTS to re-execute the events saved.

M. EVENT.FORM(x)

If x is an event name, EVENT.FORM returns the INTERLISP S-expression representing the event that created it (e.g., DEFN CONSed onto the list of arguments). If x is a satellite, EVENT.FORM returns the S-expression for its main event. Otherwise, EVENT.FORM returns NIL.

N. FAILED.THMS

Roughly speaking, FAILED.THMS is a list of all the conjectures that the system has failed to prove in the current session. FAILED.THMS is maintained as follows. When you start up the system it is initialized to NIL. Every time PROVE is entered it adds the conjecture to be proved to the list of conjectures in FAILED.THMS (unless the conjecture is already in FAILED.THMS). When PROVE terminates normally and has proved the conjecture, it removes it from FAILED.THMS. Thus, if midway through a proof you abort with CTRL-E, the conjecture PROVE was called on will still be on FAILED.THMS.

FAILED.THMS is not part of the knowledge base. Thus, if you start up a fresh copy of the system and initialize the knowledge base to some previously saved library file, FAILED.THMS is not restored to what it was when the library file was created. The purpose of FAILED.THMS is to help you remember theorems you were trying to prove (before you got

distracted by trying to prove the necessary lemmas), and to save you the trouble of having to type those theorems back in again.

O. FORMULA.OF(name)

This function returns the S-expression of the formula named name if name was created by ADD.AXIOM, PROVE.LEMMA, or MOVE.LEMMA, and NIL otherwise. FORMULA.OF cannot find a formula for a satellite. Thus, if you ask for the FORMULA.OF a lemma name created by ADD.SHELL, the result will be NIL, since the axioms created during ADD.SHELL are mere satellites of the ADD.SHELL event.

P. GENERALIZE

GENERALIZE is one of the "lemma types" specifying how a lemma can be used. Roughly speaking, a GENERALIZE lemma mentioning an instance of the term t is used when the theorem-prover decides to generalize a conjecture containing t by replacing it with a new variable v. The lemma is used to restrict v. There are no constraints on the form of GENERALIZE lemmas. See ACL XII for a detailed description of how such lemmas are used.

Q. GROUND.ZERO

The name of the event created by BOOT.STRAP is GROUND.ZERO. All of the axioms, definitions, and shells added by BOOT.STRAP are made dependent upon GROUND.ZERO. Consequently, every event in any data base can be reached from GROUND.ZERO.

R. INDUCTION

INDUCTION is one of the "lemma types" specifying how a lemma can be used. Roughly speaking, INDUCTION lemmas inform the system that a given measure decreases under the well-founded function LESSP. An induction lemma must be of the form:

```
(IMPLIES hyp
         (LESSP (m t1 ... tn)
                (m x1 ... xn)))
```

where the xi are distinct variables, all the variables in hyp occur in the conclusion, and m is a function symbol.

The theorem-prover makes a special exception to the above rule for a lemma of the form (IMPLIES hyp (LESSP t1 x1)) when t1 is a term that always returns a number. In this case, it treats the lemma as though it were:

```
(IMPLIES hyp
         (LESSP (COUNT t1) (COUNT x1)))
```

If hyp is a conjunction of terms, and one of the terms is of the form (NOT (EQUAL (m t1 ... tn) (m x1 ... xn))), the lemma informs the system that (m1 t1 ... tn) is less than or equal to (m x1 ... xn) under the hypotheses in hyp, except for the (NOT (EQUAL --)) term. Such a hypothesis is the only way to so inform the system (i.e., the system does not recognize a conclusion of the form (LESSEQP & &) or (OR (LESSP & &) (EQUAL & &))).

For a detailed discussion of how INDUCTION lemmas are used read ACL XIV and XV.

S. INIT(file)

INIT performs the last step in the process of bringing up the theorem-prover. INIT first loads some "patch files" that correct bugs in or add new features to the system contained in the main files. If file is T, INIT then loads a block compiled version of the simplifier and calls BOOT.STRAP to initialize the system's data base to the formal theory described in ACL III. If file is not T and not NIL, it is assumed to be the name of a library file (produced by MAKE.LIB). In this case, INIT loads the block compiled simplifier and then calls NOTE.FILE on file to initialize the data base to that contained in file.

The option of calling INIT with file set to NIL is not useful to the average user. The resulting system contains the patch files but neither the block compiled simplifier nor a data base. Because of the lack of a data base none of the theorem-prover functions can be called without causing errors (e.g., all function symbols are unknown). We use

the NIL option for development purposes (e.g., if all we want to do is edit theorem-prover programs, it not necessary to have a data base loaded).

T. LEGAL.NAME.CHARS

The list of characters permitted in event, function and variable names is kept in the variable LEGAL.NAME.CHARS. The character set is restricted for two reasons. First, it is convenient for the theorem-prover to know that no user variable has certain characters in it, so that it can use such "illegal" variables for internal purposes. Second, PUBLISH can only distinguish English commentary from references to event names if event names are somehow restricted.

You should not change the setting of LEGAL.NAME.CHARS, as that will not suffice to inform the theorem-prover that its internal names must be changed. You may inspect the value of the list to see what characters names may have.

U. LEMMAS(fns)

The argument, fns, is assumed to be a list of function symbols (e.g., TAUTOLOGYP, GCD, MEMBER). LEMMAS returns, as a list, the names of all lemmas that mention all of the names in fns. LEMMAS is useful when you know you proved a lemma about some function symbol but cannot remember its name. For example, if you recalled that there was some lemma involving both MEMBER and STRPOS and you wanted to know all such lemmas you could execute:

```
_LEMMAS((MEMBER STRPOS))
```

Under the current PROVEALL.LIB data base the result would be:

```
(DELTA1.LEMMA DELTA1.LESSP.IFF.MEMBER STRPOS.LIST.APPEND)
```

V. LEMMA TYPES

The lemma types are REWRITE, ELIM, GENERALIZE, and INDUCTION.

W. LIB.FILE

LIB.FILE is a variable whose value is the name of the current library file, if any. See the discussion under NOTE.FILE.

X. LIB.PROPS

LIB.PROPS is a list of the property list keys that the theorem-prover uses. Property lists are where almost all the information associated with events is stored. (However, see NOTE.FILE for important information about our management of property lists.) The remaining information is stored on the variables named in LIB.VARS.

Y. MAKE.LIB(file)

MAKE.LIB creates a new file, named file. The file contains the theorem-prover's entire current knowledge base. Calling NOTE.FILE on the created file will make the system's knowledge base exactly what it was when file was created with MAKE.LIB. For example, if you began your session with the state contained in the file TP.DATA.1, then defined some functions and proved some lemmas, and then called:

```
_MAKE.LIB(TP.DATA)
```

the system would create a new version of TP.DATA, containing all of the data in TP.DATA.1 plus your new functions and lemmas. In particular, unless you undo some of the events in TP.DATA.1, the new file will be at least as large as the data base you started with.

Library files (as these data base files are called) are text files. By listing such a file on the line printer and inspecting it you can get a good idea of what information the system extracts from each logical event and how it is represented internally.

See the discussion under NOTE.FILE for important implementation information about library files.

Z. MOVE.LEMMA(name lemma.types oldname)

It is sometimes convenient (indeed, necessary) to cause the system to "forget" a lemma or to use it in a way different from the ways specified when it was created. However, the logical nature of events prevents us from being able to change the way a lemma is stored after the fact. MOVE.LEMMA is a way of achieving the desired effect by the following slightly devious means. The first two arguments to MOVE.LEMMA are identical to the first two arguments of PROVE.LEMMA, namely the name of an event to be created and a list of lemma types. The third argument, oldname, is the name of an existing axiom or lemma introduced by ADD.AXIOM, PROVE.LEMMA, or MOVE.LEMMA. MOVE.LEMMA disables oldname (i.e., prevents the lemma named from being used in any way whatsoever) and then adds the formula associated with oldname as a new lemma, with event name name, to be used in the ways specified by lemma.types. The new event is made dependent upon the old, just as though the old had been (the only lemma) used in the proof of the new.

Thus, if LEMMA1 was stored as a REWRITE lemma and you now wish it were also a GENERALIZE lemma, you could execute:

```
_MOVE.LEMMA(LEMMA2 (REWRITE GENERALIZE) LEMMA1
            (* This creates a new event, named LEMMA2,
               that has exactly the same formula as
               LEMMA1, and is stored as both a REWRITE
               lemma and a GENERALIZE lemma.
               In addition, LEMMA1 is disabled.))
```

The new event is named LEMMA2 and adds three facts to the world: LEMMA1 cannot be used, the formula named LEMMA2 (which happens to be the same as the formula named LEMMA1) may be used as a rewrite rule, and the formula named LEMMA2 may be used as a generalize lemma. Future proofs might employ LEMMA2 in either of the two ways specified. If you later used UNDO.NAME to undo the MOVE.LEMMA (i.e., the event named LEMMA2) the system would undo those events that depended upon LEMMA2 and erase the three facts added to the world by the MOVE.LEMMA. Note in particular that undoing LEMMA2 permits LEMMA1 to be used again.

If you wished to prevent the system from using the fact expressed by LEMMA1 at all, you could execute:

```
_MOVE.LEMMA(LEMMA2 NIL LEMMA1)
```

which disables LEMMA1, creates the event LEMMA2 with the same formula as LEMMA1, and stores it so that it can not be used. Subsequently undoing LEMMA2 will restore the potency of LEMMA1.

Note the difference between using MOVE.LEMMA to disable LEMMA1 and using UNDO.NAME to undo LEMMA1. The first does not disable results derived from LEMMA1 while the second eliminates LEMMA1 and all of its dependents. Also, the first is an event and can be undone, while the second is not an event and leaves no trace of LEMMA1.

AA. NO.BUILT.IN.ARITH.FLG

After we finished writing A Computational Logic we implemented a linear arithmetic subroutine in the simplifier. The theorem-prover on <MOORE> includes this simplifier. The subroutine knows about 0, ADD1, SUB1, NUMBERP, LESSP, and PLUS and can determine for some sets of linear inequalities that they are unsatisfiable. The implementation includes heuristics which instantiate and apply REWRITE type lemmas, both as rewrite rules and as additional inequalities. See the discussion of the REWRITE lemma type for more details on which rewrite rules are used.

The linear arithmetic subroutine enables the system is able to prove many facts of linear integer arithmetic very quickly. However, it is sometimes instructive to see how those facts would be approached by the version of the system that knows nothing of arithmetic beyond Peano's axioms. Therefore we provided a switch with which the subroutine could be turned on and off.

If NO.BUILT.IN.ARITH.FLG is set to NIL then the linear arithmetic subroutine is turned on. enabled. REWRITE lemmas of a special form (described under REWRITE) are stored in a special way and not as rewrite rules. If the flag is set to T, then the subroutine is turned off. While the flag is set to T, REWRITE lemmas of the special form are stored as rewrite rules. Initially NO.BUILT.IN.ARITH.FLG is NIL.

AB. NOTE.FILE(file)

This function initializes the system's knowledge base to that contained in the file file. It is assumed that file was produced by MAKE.LIB.

There must not be a knowledge base at the time of the NOTE.FILE; attempting to bring in a second knowledge base will cause an error (and your state will be not in fact have been changed). This restriction is enforced because we have no means of ensuring that the facts in one knowledge base are consistent with those in another. Thus, if you have been proving theorems and wish to restore the system's state to some previously saved one, you must first erase your current knowledge base by calling UNDO.NAME on GROUND.ZERO, upon which depend the initial axioms (and, hence, all subsequent definitions and proofs). After undoing GROUND.ZERO the system will be in its virgin state (e.g., it will not even recognize the function symbol IF) and you can then call NOTE.FILE on the saved library file. (Alternatively, and usually more efficiently, you may start up an entirely fresh version of the theorem-prover and INIT the desired file.)

Most of the system's data is stored on the property lists of function symbols and theorem names. Effectively, MAKE.LIB writes all these properties out, and NOTE.FILE loads them in. However, to save space, NOTE.FILE only stores the name of the file to be "loaded" in the variable LIB.FILE and arranges for its properties to be loaded incrementally, one property at a time, as they are called for. This is done by hanging a note on each atom involved, pointing to the byte address (implicitly in LIB.FILE) at which the properties for that atom are stored. Instead of accessing properties with GETPROP and PUTPROP we access them with our own special functions GET1 and PUT1.

Thus, NOTE.FILE does not take very much time to "notice" a library file. Furthermore, noticing a file does not take much space, even for a very large library. But the main advantages of this implementation accrue from the fact that properties are never brought in unless they are asked for, and in a typical session, many properties never are. For

example, unless you undo an event, the information indicating what must be undone is never in main memory. Undo and dependency information represents about 40% of the data in a library file. Another savings comes from the observation that lemmas about one clique of function symbols may never be referenced in proofs about another clique. For example, if you are proving theorems about list processing functions, you may never bring in the function symbols and lemmas about prime numbers.

The theorem-prover actually uses MAKE.LIB and NOTE.FILE together to swap out its data base when the total number of CONSes tied up in property lists exceeds a certain number (the value of MAX.PROP.CNT, initially set to 20000). When this happens, the system calls MAKE.LIB to create a new library file with the name SWAPPEDLIB. Then it erases all properties and calls NOTE.FILE on the SWAPPEDLIB file. The result is that no property is in memory and the list space can be reclaimed. GET1 will swap in the properties one by one as they are called for. For example, when we drive the theorem-prover through the proofs of the several hundred theorems in its standard data base, it runs out of property list space twice and twice swaps the data base out and continues. Of course, after each swap subsequent processing brings back in some of the old properties, but not all of them are brought back in because the system is also progressing through cliques of function symbols and is not undoing events or inspecting dependencies of old ones.

To keep your directory from being cluttered with SWAPPEDLIB files, swapping destroys the old SWAPPEDLIB. The rule used is that if a swap occurs while the current LIB.FILE is a SWAPPEDLIB file, the old file is killed by smashing its length to 0 and deleting it. (This frees the disk space without doing a TOPS-20 EXPUNGE.) Because of this behavior you should never create a library file with main name SWAPPEDLIB, nor should you ever intend to keep a SWAPPEDLIB file from one day to the next. If for some reason you want to keep a SWAPPEDLIB file you should rename it.

Because the theorem-prover may need to create a swapped library to avoid running out of list space, you should not run the theorem-prover unless you have several hundred free pages of disk space. If you will not tolerate the creation of such a file, set the variable MAX.PROP.CNT to 300000.

The use of swapped library files makes it inconvenient to use the INTERLISP SYSOUT command to save the state of the theorem-prover. If you insist on using SYSOUT, here is the recommended procedure. First see what the value of LIB.FILE is. If it is NOBIND (which means there is no library file and all properties are in memory) or anything other than the name of a SWAPPEDLIB file, you are safe. You can do the SYSOUT, creating an EXE file that can be restarted with the TOPS-20 RUN command. However, at the time you run the EXE file, make sure that the file named by a non-NOBIND LIB.FILE is on the connected directory under exactly the same name (thus you must save both the EXE file and the file named by LIB.FILE).* The LIB.FILE will be opened automatically the first time it is referenced. If, when you want to call SYSOUT, LIB.FILE is a SWAPPEDLIB file you should first rename it. Failure to do so will mean that future sessions based on that EXE file may kill the swapped library upon which the EXE file was based and you will not be able to use the EXE file a second time. To rename LIB.FILE proceed as follows: (CLOSEF LIB.FILE) to close the file, CTRL-C up to the monitor; use the TOPS-20 RENAME command to rename the file just closed to some new name, newlib; CONTINUE back into INTERLISP, and (SETQ LIB.FILE (INPUT (INFILE 'newlib))) to reset the value of LIB.FILE to the new name. Then you are free to make and use the SYSOUT just as though you were not running with a swapped library.

---

* In connection with this problem, note that if at the time of your SYSOUT LIB.FILE is set to <MOORE>PROVEALL.LIB, you are implicitly relying on us to keep that particular PROVEALL.LIB file around. However, we cannot guarantee to do so, since we generally provide a new library file every time we release a new version of the system. Thus, you should take responsibility for saving that file by copying it to your directory and renaming the LIB.FILE before the SYSOUT as described below.

AC. PPE(x)

PPE is an NLAMBDA-nospread INTERLISP function (it takes an indefinite number of arguments that are not evaluated before the function is entered). The arguments should be names of events or satellites (i.e., function, axiom, or lemma names). For each member of x, PPE prettyprints (to the primary output file) the corresponding event. Here is an example:

```
_PPE(TIMES TIMES.ZERO PLUS POP.PUSH FOO)
(DEFN TIMES
      (I J)
      (IF (ZEROP I)
          0
          (PLUS J (TIMES (SUB1 I) J))))

(PROVE.LEMMA TIMES.ZERO
             (REWRITE)
             (EQUAL (TIMES X 0) 0))

(***** PLUS is a satellite of
       (BOOT.STRAP))

(***** POP.PUSH is a satellite of
       (ADD.SHELL PUSH NIL STACKP
                  ((TOP T 0) (POP T 0))))

(***** FOO is neither an event nor satellite)
```

No mechanical method is provided for obtaining the formula corresponding to a satellite name. For example, as noted in the example above, POP.PUSH is a satellite of a certain ADD.SHELL command. In fact, POP.PUSH is the system generated name of the rewrite rule (EQUAL (POP (PUSH X Y)) Y). The only way a user can determine the formula corresponding to a satellite is to consult A Computational Logic. Appendix C of the book gives, in schematic form, the axioms added by ADD.SHELL and their names. All of the names introduced during the initial boot strap (e.g., IF, TRUE, ADD1, CONS, PLUS) are considered satellites of GROUND.ZERO. ACL III describes the formal theory "created" by the boot strap.

AD. PPR(fmla pprfile)

PPR is the common entry to our prettyprint routine. It takes a term in our theory and a file open for output, and prettyprints the term to the file. PPR returns NIL.

There are three advantages to using PPR instead of the INTERLISP prettyprinter. First, PPR knows how to introduce the abbreviations permitted by our theory. For example,

```
(APPLY (PACK (ADD1 (ZERO)))
       (ADD1 (ADD1 (ZERO)))
       (CAR (CDR X)))
```

is printed as:

```
(APPLY (QUOTE PUSHV) 2 (CADR X))
```

(in the current PROVEALL.LIB data base where (QUOTE PUSHV) happens to be assigned (PACK 1)). Second, PPR produces more readable output and usually puts the formula on fewer lines than the INTERLISP prettyprinter because PPR computes, rather that guesses at, the exact amount of space a subexpression will require. Third, PPR is faster than the INTERLISP prettyprinter, especially on large expressions.

AE. PPRIND(fmla leftmargin rparcnt ppr.macro.lst pprfile)

PPRIND is the most flexible entry into our prettyprint routine. It prettyprints fmla, in the columns between leftmargin and (LINELENGTH) of file pprfile. PPRIND considers the leftmost character position on a line to be column 0. Thus column leftmargin has leftmargin characters to the left of it. PPRIND assumes that when called the "print head" (or file pointer) is positioned at the point at which you want the first character to come out. Thus, if you want the formula printed starting in column 5, you should print 5 spaces and then call PPRIND with leftmargin set to 5. Recognizing the occasional need to output characters immediately after the last character printed, PPRIND will guarantee to leave at least rparcnt spaces between the last character printed and (LINELENGTH). Thus, if you wanted to prettyprint the formula, but wanted to make sure there was at least enough space on the last line to print a comma, you would call PPRIND with rparcnt set to 1.

The list ppr.macro.lst provides a facility for "macro expanding" forms before they are printed. ppr.macro.lst is assumed to be a list of ordered pairs of the form (hd . fn). When PPRIND encounters a form to be printed that starts with an atom in the hd position of such a pair, it applies fn to the form and prettyprints the result instead of the original form. It is with ppr.macro.lst that PPR introduces its abbreviations (i.e., a "ppr macro" transforms (ADD1 (ADD1 (ZERO))) to 2, but leaves (ADD1 (ADD1 X)) alone).

AF. PROVE(thm)

PROVE attempts to prove the conjecture thm, using all the theorem-proving techniques at the system's disposal. PROVE prints an English description of the proof attempt in real-time, so you can monitor the progress of the attempt. The proof description is printed to the file named by the value of the variable PROVE.FILE. Initially, PROVE.FILE is set to T, so that proofs are printed to the terminal. By setting PROVE.FILE to a disk file you can have a proof printed to a file so that it can be inspected at your leisure.

It is often the case that you will decide the theorem-prover is failing before it decides that it is failing. If you decide the system is failing you should abort the proof by typing CTRL-E, get the system to prove the necessary lemmas, and then try to prove the "difficult" theorem again. Note that it is permitted to abort PROVE with CTRL-E or CTRL-D without risking messing up the system's state because PROVE does not change the state.

You may abort the printing of any formula, without aborting the proof attempt, by typing CTRL-K.

The result of PROVE is either the INTERLISP literal atom PROVED or a very large failure message string. When the failure message string is returned, the most you are entitled to conclude is that the system was incapable of finding a proof. In particular, the failure message does not mean that the formula is not a theorem. It is often the case that the failure message is precipitated by the system's discovery that it is

trying to prove a goal that is falsifiable. However, even in this case you may not conclude that the input conjecture was a nontheorem since during the proof attempt the system might have abandoned the original formula in favor of a mechanically generated generalization (that might not in fact be a theorem).

AG. PROVE.FILE

The value of the variable PROVE.FILE should be the name of an open file to which you want proofs to be written. Both DEFN and PROVE.LEMMA direct their output to PROVE.FILE. Initially PROVE.FILE is set to T (the terminal). If PROVE.FILE is set to a name other than T when DEFN or PROVE.LEMMA or REDO.UNDONE.EVENTS is called and the file named is closed, the system sets PROVE.FILE to T. For example a typical scenario is to set PROVE.FILE to the name of a just opened file, call PROVE.LEMMA to prove a theorem and write the proof to the disk, and then call CLOSEALL or (CLOSEF PROVE.FILE) to close the disk file. The next time you call one of the functions that writes to PROVE.FILE it will reset it to the terminal.

AH. PROVE.LEMMA(name lemma.types term)

PROVE.LEMMA attempts to prove the conjecture term and remember it as a lemma named name, available for use in the ways specified by the list of lemma types in lemma.types. PROVE.LEMMA first checks to see whether term is acceptable as a lemma of the types listed (so that such an error can be reported before the system has spent the time necessary to prove term). Provided term is acceptable as a lemma of the types in lemma.types, PROVE.LEMMA calls PROVE on term. If the result is PROVED, PROVE.LEMMA creates a new event named name, associates the formula term with name, and makes it available as a lemma to be used in the ways specified by lemma.types. The event is made dependent upon all of the events used in the proof of term.

Note that even though PROVE can be aborted safely with CTRL-E or CTRL-D, PROVE.LEMMA cannot. The reason is that after calling PROVE to

prove the formula, PROVE.LEMMA updates the state. It is possible (though extremely unlikely) that you would decide that the proof is failing and abort when in fact it had already proved the theorem and was beginning to update the state while your terminal was still printing out the formulas. The only approved way to abort PROVE.LEMMA is to type CTRL-H and wait for a soft INTERLISP interrupt, use BT in the resulting break to make sure that the system is still under the call to PROVE, and if so to type STOP or CTRL-D to abort (and if not, type OK to continue the final stages of the updating).

AI. PROVEALL(event.lst detach.flg filename sysoutflg)

PROVEALL is a convenient way to process a command sequence and write the theorem-prover output to a file. PROVEALL executes

(REDO.UNDONE.EVENTS event.lst T 'A detach.flg)

after obtaining 30000 free words of list space, setting the GC message to "." (which will be changed to the empty string if the job is to be detached), setting PROVE.FILE to filename.PROOFS and TTY: to file.name.TTY:. After the sequence of commands has been processed, PROVEALL closes both PROVE.FILE and TTY: and, if sysoutflg is non-NIL, makes a SYSOUT file (called filename.EXE).

Thus, if you have a sequence of events, event.lst, to be executed and you wish to watch the progress of the job from your terminal, but have the proofs go to the file DEMO.PROOFS, you would execute (PROVEALL event.lst NIL 'DEMO). During the proofs your terminal would display only the successive event names as they were encountered during the processing, separated by commas, and interspersed with dots indicating the GCs. If you wished to have the events executed in a detached job and have a SYSOUT file produced afterwards so you could poke around if you wished (e.g., to produce a library file), you should execute (PROVEALL event.lst T 'DEMO T). After seeing the detach message, you could then turn off your terminal and walk away while the system proceeded to do the proofs. You can re-attach your job at any time by typing:

@ATT user password jobno

to the TOPS-20 EXEC, where user is your user name, password is your password, and jobno is the number of the job running the theorem-prover. If you re-attach while the PROVEALL is still running, it will resume printing out the successive event names separated by commas (but the garbage collect message will be the empty string). If you re-attach after the PROVEALL has terminated, INTERLISP will print the value of the call to PROVEALL and you may proceed as usual. It is good practice after a detached PROVEALL to inspect the value of the variable FAILED.THMS to see if any proof failed. It is also good to remember that no library file is made by PROVEALL; if you want to save the system's state you should do so after re-attaching.

See REDO.UNDONE.EVENTS for the details of the system's behavior during and after a detached run.

AJ. PUBLISH(file title)

The file argument is assumed to be the name of a PROVE.FILE written to the disk by PROVEALL (or, actually, by REDO.UNDONE.EVENTS when the two final flag arguments are both NIL). That is, the file should consist of the header as printed by REDO.UNDONE.EVENTS followed by a sequence of commands, theorem-prover output, and values, separated by formfeeds. PUBLISH opens file for reading; creates a new version for writing; and then writes in sequence a title page (with title centered on it above the system identification header printed by REDO.UNDONE.EVENTS), a table of contents listing each event name and the page on which it became defined, each command and its output and value, a cross-referenced index (listing, for each name mentioned in the file, every page on which the name appears), and the summation of the time and CONS usage statistics for the events. PUBLISH then closes both files.

If you list the created file with the TOPS-20 LIST command the page numbers printed will agree with with those in the table of contents and index. Events requiring several physical pages of output are LISTed with page numbers such as 7, 7:1, 7:2, 7:3, ...

AK. REDO!(name)

REDO! uses UNDO.NAME to undo the event named name and then uses REDO.UNDONE.EVENTS to reprocess each of the events returned. It is mainly used to cause the system to go through a proof that just flashed by on the screen or that you have decided to repeat for demonstration or documentation purposes (e.g., between the first proof and the REDO! you might have set PROVE.FILE to be a disk file so you could capture the proof).

AL. REDO.UNDONE.EVENTS(events all.flg failure.action detach.flg)

REDO.UNDONE.EVENTS is used to process a list of theorem-prover commands. The arguments allow you to select, interactively, the commands to be executed, to specify the action to be taken should a command fail, to have your job detached while the commands are being executed, and to control the printing of certain header information.

The list events is assumed to be a list of events as returned by UNDO.NAME. That is, each element of events is a form whose CAR is one of the atoms BOOT.STRAP, ADD.AXIOM, ADD.SHELL, DCL, DEFN, PROVE.LEMMA or MOVE.LEMMA, and whose CDR is an appropriate list of arguments. REDO.UNDONE.EVENTS applies the CAR of each event to the CDR. There are three typical sources of commands for REDO.UNDONE.EVENTS. The first is the list of events returned by UNDO.NAME. For example, if after escaping from several blind alleys you have finally gotten the system to prove your "main theorem," you might undo and redo a sequence of events to write the entire sequence of proofs to a file for documentation purposes. Because of the uncertainty surrounding the validity of the theorem-prover's state after aborted commands and the doubts about the correctness of UNDO.NAME, we recommend such a final uninterrupted "proveall" as the only sure evidence that your formulas are theorems.

The second typical source of input for REDO.UNDONE.EVENTS is a list of events on a disk file, written during a previous session (the result of printing the value of EVENTS.SINCE, perhaps).

The third typical source of events is a mechanical conjecture generator, such as a verification condition generator. We find it convenient for our vcg programs to produce a sequence of axioms, definitions, declarations and conjectures to prove, and then to use REDO.UNDONE.EVENTS to drive our sweep through the list.

Before the first command is executed, REDO.UNDONE.EVENTS prints a header indicating the date and the versions of the LISP and theorem-prover files in use.* This information is printed to PROVE.FILE. If PROVE.FILE is T, the header is two lines of information. Otherwise, it is centered on a page delimited by formfeeds.

As REDO.UNDONE.EVENTS sweeps through the list of events, it indicates which event is being processed. To the file named by PROVE.FILE, REDO.UNDONE.EVENTS prints a formfeed followed by the entire form about to be executed**. After each command is executed, REDO.UNDONE.EVENTS prints the value of the event to PROVE.FILE. Thus, PROVE.FILE will contain each command, the theorem-prover's output in response to the command, and the value returned by the command, with formfeeds separating the commands. The command PUBLISH can be used to process such a file to add a title page, table of contents, and cross-referenced index.

If PROVE.FILE is not the terminal, but you are attached to the job, then it is useful to get some indication as to how the job is progressing. In this case, REDO.UNDONE.EVENTS prints to the terminal the name of each event (i.e., the CADR of the form) before the command is processed and a comma after each event is processed. Thus, if you

---

* REDO.UNDONE.EVENTS actually has two additional arguments. The sixth argument of REDO.UNDONE.EVENTS is don't.print.date.line.flg; if it is T, the printing of the header will be omitted.

** The fifth argument of REDO.UNDONE.EVENTS is don't.print.first.event.flg; if it is T, REDO.UNDONE.EVENTS does not print the first command on events; this option is used by EDITEV when it calls REDO.UNDONE.EVENTS to reprocess an edited event and those events depending on it. In this case, it is assumed that you know what the edited event is and would consider it an inconvenience to see the result of your edit printed back out at you.

are dumping the proofs to a file, but have remained attached to your job, your terminal will slowly print a sequence of event names, separated by commas, and interspersed with GC messages.

If one of the commands in events causes an error, the usual error message will be printed and then (provided you are attached to the job) you will be put into the editor editing the tail of events starting at the offending event. Exiting the editor with STOP or CTRL-D will leave you in the safe state the system was in before it encountered the offending command. Exiting the editor with OK will cause resumption of the command processing in the edited tail. The tail you edit is physically that of the events you supply REDO.UNDONE.EVENTS. Thus, when REDO.UNDONE.EVENTS eventually terminates, the list of events supplied will reflect any edits you performed during the processing.

The tail of events yet to be processed is kept in the variable UNDONE.EVENTS. If you abort REDO.UNDONE.EVENTS you will find the list of events that have not been redone in UNDONE.EVENTS. However, many of the theorem-prover commands call REDO.UNDONE.EVENTS internally -- in fact all of the event creation commands such as DEFN and PROVE.LEMMA actually call REDO.UNDONE.EVENTS because of its error handling. Thus, if after an aborted call to REDO.UNDONE.EVENTS you call one of the standard theorem-prover commands like EDITEV or PROVE.LEMMA, you will find that UNDONE.EVENTS has been reset and no longer contains the tail of the events that was aborted. You should therefore save UNDONE.EVENTS immediately after REDO.UNDONE.EVENTS has been aborted if you wish to save it.

Our usual protocol for getting a sequence of commands processed is to set XXX to the sequence and call REDO.UNDONE.EVENTS on XXX. When we see a proof fail we abort REDO.UNDONE.EVENTS and call EDITV on UNDONE.EVENTS to either correct the offending event or precede it with what we believe are the necessary lemmas, and then call RESTART. The advantage of this technique is that when we finally reach the end of the original XXX it has been edited along the way to contain the unanticipated lemmas required.

The second argument to REDO.UNDONE.EVENTS, all.flg, determines whether all or only some of the events in events are redone. If all.flg is NIL, then when REDO.UNDONE.EVENTS encounters a non-DEFN event in events it asks you whether you wish to redo that event. If all.flg is non-NIL, all events are redone. The all.flg option of REDO.UNDONE.EVENTS is used by EDITEV so that when you reconstruct the sequence of events depending on an undone, edited definition, the functions that call the edited function as a "subroutine" are automatically reprocessed and you have the freedom to specify which dependent lemmas the system should try to prove again.

The third argument, failure.action, is used to specify the system's action should it fail to prove some lemma in the sequence of events. The failure to prove a lemma deserves special attention because it may be impossible for the system to prove the succeeding lemmas if it does not have the failed lemma built in. If failure.action is the literal atom Q (for "Quit"), a failed proof causes REDO.UNDONE.EVENT to abort. If failure.action is the atom C (for "Continue") a failed proof is ignored, in the sense that the rest of the events on events are attempted as though nothing unusual had occurred. If failure.action is the atom A (for "ADD.AXIOM"), the lemma is added as an axiom so that subsequent proofs will not fail because of the absence of the single lemma the system failed to re-establish. If failure.action is NIL, then the system asks you (with ASKUSER) what failure.action should be; the response "?" typed to its question will prompt the system to list the alternatives given above.

The fourth argument, detach.flg, if T, causes the system to detach your job and then start processing events. However, the job will hang if it ever tries to do input or output to the terminal while detached. Thus, before running in detached mode the theorem-prover should be "rigged for silent running." In particular, PROVE.FILE and TTY: must be set to some file other than the terminal, and the INTERLISP garbage collector must be "gagged" to prevent it from printing its usual GC message. Thus, if detach.flg is T, REDO.UNDONE.EVENTS gags the garbage

collector and checks that PROVE.FILE and TTY: are appropriately set. If they are not appropriately set, it asks you to specify the desired file names. Once the system has been rigged for silent running, REDO.UNDONE.EVENTS automatically detaches the job and then begins to process the commands as described above. When you see the system print the message that your job has been detached, you are free to turn your terminal off and walk away.

To enable you to monitor the progress of your detached job, REDO.UNDONE.EVENTS changes the name of the job from "LISP" to a name of the form "n/m", where n is the number of events in events still to be processed, and m is the number of lemmas that the system has failed to prove. The job name is updated every time REDO.UNDONE.EVENTS steps to a new entry in events. When the last event has been processed, the job name is changed to DONE.

If at any time you want to see the job name, connect to the operating system (not necessarily logging in) and type:

@sys user

where user is your user name (or else the number of the job running the theorem-prover).

You can re-attach your job at any time by typing:

@ATT user password jobno

to the TOPS-20 EXEC, where user is your user name, password is your password, and jobno is the number of the job running the theorem-prover. If you re-attach while REDO.UNDONE.EVENTS is still running, it will resume printing out the successive event names separated by commas (but the garbage collections will be silent). If you re-attach after REDO.UNDONE.EVENTS has terminated, INTERLISP will print the value of the call to REDO.UNDONE.EVENTS and you may proceed as usual. It is good practice after re-attaching a detached REDO.UNDONE.EVENTS to inspect the value of the variable FAILED.THMS to see if any proofs failed. It is also good to consider whether to save the system's state after the REDO.UNDONE.EVENTS.

If an error arises during a detached run, the system prints the appropriate message to the TTY: file, changes the job name to ERROR, and sleeps until the job is re-attached. When you re-attach the job you should type CTRL-E (to signal to the sleeping job that you are now there to field the error). The system then reprints the message to the terminal and enters the usual interactive break. (Sleeping is subtly different from the "hanging" that occurs should the job try to read from or write to the terminal; a job that is so hung is automatically logged out by the system after a certain grace period. The sleep induced by an error permits the job to avoid the inactive job reaper indefinitely.)

It is usually the case that when the system crashes, you lose all files created but not closed before the crash. This happens because the directory entry for a file being written is usually not updated until the file is closed. Since it is not unusual for REDO.UNDONE.EVENTS to run for a long time compared to the mean time between SRI-KL crashes (e.g., reprocessing all of the events in our PROVEALL.LIB library requires about 2 CPU hours) REDO.UNDONE.EVENTS uses JSYS calls to update the directory entries for PROVE.FILE and TTY: every time it steps to the next event. Thus, if the system crashes while you are running a detached job, you will lose the state but you will at least have the system output up to the proof during which the system crashed.

AM. RESTART(x)

(RESTART X) is just (REDO.UNDONE.EVENTS (OR X UNDONE.EVENTS) T 'Q). This function is a handy way to continue an interrupted REDO.UNDONE.EVENTS. Recall that when REDO.UNDONE.EVENTS terminates the variable UNDONE.EVENTS is set to the tail of the event list still to be processed. RESTART just calls REDO.UNDONE.EVENTS again, giving it UNDONE.EVENTS as the event list. However, note that UNDONE.EVENTS is liable to be reset by any command that creates an event (e.g., DEFN or PROVE.LEMMA).

Our usual protocol for using REDO.UNDONE.EVENTS and RESTART is to abort REDO.UNDONE.EVENTS when we see a proof fail, to call EDITV on

UNDONE.EVENTS to either correct the offending event or precede it with what we believe are the necessary lemmas, and then to call RESTART. The advantage of this technique is that when you finally reach the end of the original events it has been edited along the way to contain the unanticipated lemmas required.

AN. REWRITE

REWRITE is one of the "lemma types" that specifies how a lemma may be used. Roughly speaking, REWRITE lemmas are used by the simplifier to rewrite terms conditionally. Let the lemma in question be term. If term is of the form (IMPLIES hyp concl), concl is used as a rewrite rule when the system can establish hyp; otherwise, term is treated as though it were (IMPLIES T term). If the conclusion is (EQUAL r s), instances of r are rewritten to s. A conclusion of the form (NOT r) is used just as though it were (EQUAL r F). A conclusion of the form r, where the function symbol is neither EQUAL, NOT, nor LESSP is used to establish that instances of r are non-FALSE. See ACL VII for complete details of how REWRITE lemmas are used.

We have recently added a linear arithmetic package to the simplifier. This package causes certain lemmas with a conclusion of the form (LESSP a b) or (NOT (LESSP a b)) to be treated specially. Here is the restriction on a and b. Consider a and b as PLUS-trees (i.e., trees of non-PLUS terms connected by the function symbol PLUS). Then at least one of the non-PLUS tips of one of the two trees must be a term that includes every variable in the lemma. Thus, if X and Y are the only variables in a lemma and the conclusion is (LESSP X (TIMES X Y)), the lemma meets the above restriction. A conclusion of (LESSP (FN X) (FN Y)) or (LESSP X (PLUS (FN X) (FN Y))) does not meet the restriction. This restriction is present because such "maximal addends" are keyed on by the linear arithmetic package and trigger the instantiation of the lemma (instantiating all its variables) and the addition of the instance to the "pot" of linear inequalities being processed.

To disable the linear arithmetic package and to prevent future REWRITE lemmas from being stored in it, set NO.BUILT.IN.ARITH.FLG to T. Note that this will not suddenly cause previously stored linear lemmas to be used as rewrite rules. To get a lemma with a LESSP conclusion stored and used as a rewrite rule while the linear package is still enabled you can hide the LESSP by writing the conclusion as (EQUAL (LESSP a b) T) or (EQUAL (LESSP a b) F) as appropriate.

AO. TRANSLATE(term)

TRANSLATE is the program that checks for syntax errors in S-expressions and removes the abbreviations permitted in our theory. For example,

```
_TRANSLATE((APPLY (QUOTE PUSHV) 2 (CADR X)))
```

returns the value

```
(APPLY (PACK (ADD1 (ZERO)))
       (ADD1 (ADD1 (ZERO)))
       (CAR (CDR X)))
```

(provided (QUOTE PUSHV) is the abbreviation for the literal atom (PACK 1)). TRANSLATE causes errors when it finds illegal syntax such as too few arguments or unknown function symbols.

All of the event creating routines in our system (e.g., DEFN and PROVE.LEMMA) call TRANSLATE on user-supplied terms. None of the internal routines in the system know about abbreviations (e.g., if you type in a term involving 99, it is expanded to a nest of 99 ADD1's).

AP. TTY:

The value of the variable TTY: is supposed to be the name of an open file to which error and warning messages are printed. Initially TTY: is set to T (the terminal).

Each error message is first printed to PROVE.FILE. Then, if PROVE.FILE and TTY: are set to different file names, the error message is printed to TTY: as well. Thus, by setting PROVE.FILE to a disk file and TTY: to the terminal you can have a complete record of the system's

output on the disk and have error messages (also) printed to your terminal. By setting TTY: and PROVE.FILE to two different disk files you can have no output to the terminal and should inspect TTY: later to see whether any errors occurred.

AQ. UNDO.BACK.THROUGH(name)

This command "rolls back" the theorem-prover's state to the one that existed just before the event named name was created. It works by undoing, in reverse chronological order, name and every event created after name.

AR. UNDO.NAME(name)

UNDO.NAME removes from the event graph every event reachable from the one named name. (If name is a satellite, its main event is used instead.) Roughly speaking, this puts you into the state you would have been in now had you done all the events except the event being undone and those events that depend upon it. UNDO.NAME undoes exactly the events listed by DEPENDENT.EVENTS, so you can use DEPENDENT.EVENTS to ascertain the amount of damage that would be caused by undoing a given event. UNDO.NAME returns exactly the same list that DEPENDENT.EVENTS does; such a list can be given to REDO.UNDONE.EVENTS to attempt to reconstruct the undone events.

UNDO.NAME is not like the INTERLISP undo facility in several respects. The main difference is that UNDO.NAME is a quasi-logical act: not only are the effects of the named event erased but the effects of all subsequent events that depended on it are erased. A second important difference is that after an UNDO.NAME all traces of the undone events are completely wiped out: it is not possible to undo an UNDO.NAME (except by reprocessing the original events, e.g., rediscovering the proofs of the lemmas killed).

WARNING -- We believe, but have not yet proved, that UNDO.NAME is implemented as specified above. It is difficult to ascertain all of the events that are logically dependent

upon a given event because, for efficiency reasons, the theorem-prover does not record uses of rewrite rules stored as type prescriptions. When UNDO.NAME traces the dependencies of such a rewrite rule, it must determine which events might have appealed to that lemma. We believe that our implementation of DEPENDENT.EVENTS is correct in the sense that it returns a superset of the logical dependents of its argument; however, we have not yet entirely convinced ourselves that the computation is correct. Therefore, if you have constructed a sequence of definitions and proofs, and have availed yourself of UNDO.NAME to undo your missteps along the way, you should have the system reproduce the final sequence of events (with REDO!, REDO.UNDONE.EVENTS or PROVEALL) without any UNDO.NAMEs. Until we prove UNDO.NAME satisfies the above specifications, the careful user must consider it (and the commands that depend upon it such as EDITEV) as a mere "programmer's assistant" command, rather than a "logician's assistant" command.

UNDO.NAME is definitely correct when the event undone is the last event created. Thus, undoing all of the events back through a given one, in reverse chronological order, is guaranteed to have the specified effect. The function UNDO.BACK.THROUGH provides such a chronological undoing. Note however the difference between undoing backwards from the fringe reachable from a node (which is what UNDO.NAME does) and undoing all events since a node was added (which is what UNDO.BACK.THROUGH does).

IV REPORTING OF DIFFICULTIES

If you have difficulties getting the theorem-prover to prove a theorem and you wish to obtain our advice, please provide the following information to help us reproduce your situation:

* A description of how you INITed the theorem-prover, in particular which library file, if any you are using.
* A description of any UNDO.NAMEs or EDITEVs that changed the state described by that library file.
* A disk file containing a list of the event producing commands that you have executed after the INIT. Such a list can be obtained by calling EVENTS.SINCE on the name of the first event created after the INIT.
* An informal sketch of why your conjecture is a theorem (e.g., "This conjecture follows immediately from the previously proved REWRITE rules foo & bar.")

V EXTREMELY SIMPLE EXAMPLES

The first time you use a new computing system is usually painful. One reason for this difficulty is the arbitrariness of the syntax of most operating system command languages, which are among the most poorly designed of computer programming languages. TOPS-20 is no exception. Another reason is that even the simplest use of a system may require saving and retrieving information from a permanent storage facility such as a disk file; but IO is the most complex aspect of any computing language. To help you get started, we present four extremely simple example sessions with the theorem-prover.

We underline those characters that are typed by the user. We demark comments lines that are not part of the session by beginning the lines with semicolons. To increase legibility, we have added some blank lines. Some user-typed carriage returns that may not be obvious are written "<cr>".

A. Example 1

```
^C

 SRI-KL, TOPS-20 Monitor 101B(120)
 System shutdown scheduled for Wed 11-Apr-79 23:00:00,
 Up again at Thu 12-Apr-79 04:00:00
 There are 17+10 jobs and the load av. is  0.80

@LOG USERNAME(unechoed password)ACCOUNT<cr>

 Job 18 on TTY250 9-Apr-79 07:21
 Previous login: 9-Apr-79 06:42 from TTY131


@DEF DSK: DSK:, <MOORE><cr>

@THM<cr>

(<MOORE>THM.EXE.1 . <LISP>LISP.EXE.128)
```

```
2←INIT(T)

compiled on  7-Apr-79 22:54:15
FILE CREATED  7-Apr-79 22:54:12
CODE1COMS
FILE CREATED  7-Apr-79 18:09:07
DATA1COMS
compiled on  7-Apr-79 17:56:06
(GROUND.ZERO)

3←DEFN(APPEND (X Y) (IF (LISTP Y) (CONS (CAR) (APPEND
(CDR X) Y)) Y]

ERROR: CAR is a reserved abbreviation for a CAR/CDR nest and
must be given exactly one argument.

tty:

1*PP<cr>

;We are thrown into the INTERLISP Editor.  The command lines
;containing the asterisk prompt character are editor commands.

(DEFN APPEND (X Y)
      (IF (LISTP Y)
          (CONS (CAR)
                (APPEND (CDR X)
                        Y))
          Y)
      NIL)

1*(R (CAR) (CAR X))

;We replace "(CAR)" with "(CAR X)".

2*OK<cr>

;And proceed.

WARNING:  The admissibility of APPEND has not been
established.  We will assume the function to be
well-defined.  This may render the theory inconsistent.  An
induction principle for this function has been assumed,
corresponding to the obvious subgoal induction for the
function.

Observe that (OR (LISTP (APPEND X Y)) (EQUAL (APPEND X Y)
Y)) is a theorem.

**********************************************************************
***                                                                ***
***                    F A I L E D !                               ***
***                                                                ***
**********************************************************************
```

```
Load average during proof:  1.097168
Elapsed time:  7.162 seconds
CPU time (devoted to theorem proving):  1.016 seconds
GC time:  .322 seconds
IO time:  0.0 seconds
CONSes consumed:  855


(
"

**********************************************************************
***                                                                ***
***                    F  A  I  L  E  D !                          ***
***                                                                ***
**********************************************************************
")

4←EDITEV(APPEND)

tty:

3*PP

(DEFN APPEND (X Y)
      (IF (LISTP Y)
          (CONS (CAR X)
                (APPEND (CDR X)
                        Y))
          Y))

;We should have made sure X was a LISTP before CDRing it
;in recursion.  Instead, we checked Y.

3*(R (LISTP Y) (LISTP X))

4*OK

     The lemma CDR.LESSP establishes that (COUNT X)
decreases according to the well-founded function LESSP in
each recursive call.  Hence, APPEND is accepted under the
definitional principle.  Observe that:
      (OR (LISTP (APPEND X Y))
          (EQUAL (APPEND X Y) Y))
is a theorem.

Load average during definition:  1.074789
Elapsed time:  4.271 seconds
CPU time (devoted to theorem proving):  .665 seconds
```

GC time: .348 seconds
IO time: 0.0 seconds
CONSes consumed: 504

APPEND

5← DEFN(PLISTP (X) (IF (LISTP X) (PLISTP (CDR X)) (EQUAL X NIL]

The lemma CDR.LESSP can be used to establish that (COUNT X) decreases according to the well-founded function LESSP in each recursive call. Hence, PLISTP is accepted under the definitional principle. Observe that:

```
(OR (FALSEP (PLISTP X))
    (TRUEP (PLISTP X)))
```

is a theorem.

Load average during definition: 1.046235
Elapsed time: 2.998 seconds
CPU time (devoted to theorem proving): .579 seconds
GC time: .342 seconds
IO time: 0.0 seconds
CONSes consumed: 435

PLISTP

6← DEFN(REVERSE (X) (IF (LISTP X) (APPEND (REVERSE (CDR X)) (CONS (CAR X) NIL)) NIL]

The lemma CDR.LESSP establishes that (COUNT X) decreases according to the well-founded function LESSP in each recursive call. Hence, REVERSE is accepted under the definitional principle. Observe that:

```
(OR (LITATOM (REVERSE X))
    (LISTP (REVERSE X)))
```

is a theorem.

Load average during definition: 1.022781
Elapsed time: 5.0 seconds
CPU time (devoted to theorem proving): .777 seconds
GC time: .316 seconds
IO time: 0.0 seconds
CONSes consumed: 538

REVERSE

7← PPE(APPEND PLISTP REVERSE)

```
(DEFN APPEND
      (X Y)
      (IF (LISTP X)
          (CONS (CAR X) (APPEND (CDR X) Y))
          Y))

(DEFN PLISTP
      (X)
      (IF (LISTP X)
          (PLISTP (CDR X))
          (EQUAL X NIL)))

(DEFN REVERSE
      (X)
      (IF (LISTP X)
          (APPEND (REVERSE (CDR X))
                  (CONS (CAR X) NIL))
          NIL))

NIL

8. PROVE.LEMMA(REVERSE.REVERSE (REWRITE) (IMPLIES (PLISTP X)
(EQUAL (REVERSE (REVERSE X)) X]
     Name the conjecture *1.


      Let us appeal to the induction principle.

;We omit the meat of the proof.

     That finishes the proof of *1.1, which, consequently,
finishes the proof of *1.  Q.E.D.

Load average during proof:  1.165718
Elapsed time:  58.373 seconds
CPU time (devoted to theorem proving):  6.372 seconds
GC time:  4.403 seconds
IO time:  2.571 seconds
CONSes consumed:  7781


PROVED

9. (SETQ FOO (EVENTS.SINCE 'APPEND))

((DEFN APPEND (X Y) (IF (LISTP X) (CONS (CAR X) (APPEND (CDR
X)  Y)) Y)) (DEFN PLISTP (X) (IF (LISTP X) (PLISTP (CDR X))
(EQUAL X NIL))) (DEFN REVERSE (X) (IF (LISTP X) (APPEND
(REVERSE (CDR  X)) (CONS (CAR X) NIL)) NIL)) (PROVE.LEMMA
REVERSE.REVERSE  (REWRITE) (IMPLIES (PLISTP X) (EQUAL
(REVERSE (REVERSE X)) X))))
```

```
;Here is a very simple procedure for saving FOO on a TOPS-20
;disk file.

10 (SETQ REVCOMS '((VARS FOO)))

((VARS FOO))

11 MAKEFILE(REV)

<USERNAME>REV..1

^C

@DIR REV<cr>

;We note that the file REV..1 has been created.

   <USERNAME>
 REV..1

;Just to make sure, we now type the file on the terminal.

@TYPE REV..1<cr>

(FILECREATED " 9-Apr-79 06:52:29" <USERNAME>REV..1 648

     changes to:  REVCOMS FOO)


(PRETTYCOMPRINT REVCOMS)

(RPAQQ REVCOMS ((VARS FOO)))

(RPAQQ FOO ((DEFN APPEND (X Y)
                 (IF (LISTP X)
                     (CONS (CAR X)
                           (APPEND (CDR X)
                                   Y))
                     Y))
            (DEFN PLISTP (X)
                  (IF (LISTP X)
                      (PLISTP (CDR X))
                      (EQUAL X NIL)))
            (DEFN REVERSE (X)
                  (IF (LISTP X)
                      (APPEND (REVERSE (CDR X))
                              (CONS (CAR X)
                                    NIL))
                      NIL))
            (PROVE.LEMMA REVERSE.REVERSE (REWRITE)
                         (IMPLIES (PLISTP X)
                                  (EQUAL (REVERSE (REVERSE X))
```

```
                                                    X)))))
DECLARE: DONTCOPY
  (FILEMAP (NIL)))
STOP
```

B. Example 2

Having created the disk file REV..1, we can redo the list of events thus saved in another theorem proving session.

```
@THM<cr>
(<MOORE>THM.EXE.1 . <LISP>LISP.EXE.128)

2.INIT(T)

compiled on  7-Apr-79 22:54:15
FILE CREATED  7-Apr-79 22:54:12
CODE1COMS
FILE CREATED  7-Apr-79 18:09:07
DATA1COMS
compiled on  7-Apr-79 17:56:06
(GROUND.ZERO)

3.LOAD(REV)

FILE CREATED  9-Apr-79 06:52:29
REVCOMS
<USERNAME>REV..1

4.PP(FOO)

  ((DEFN APPEND (X Y)
        (IF (LISTP X)
            (CONS (CAR X)
                  (APPEND (CDR X)
                          Y))
            Y))
   (DEFN PLISTP (X)
        (IF (LISTP X)
            (PLISTP (CDR X))
            (EQUAL X NIL)))
   (DEFN REVERSE (X)
        (IF (LISTP X)
            (APPEND (REVERSE (CDR X))
                    (CONS (CAR X)
                          NIL))
            NIL))
   (PROVE.LEMMA REVERSE.REVERSE (REWRITE)
                (IMPLIES (PLISTP X)
```

```
                    (EQUAL (REVERSE (REVERSE X))
                           X))))
(FOO)

;Here is a method for getting proofs printed to a file.

5←(SETQ PROVE.FILE (OUTPUT (OUTFILE 'REV.PROOFS]

(PROVE.FILE reset)

<USERNAME>REV.PROOFS.1

6←(REDO.UNDONE.EVENTS FOO)

What should be done if a proof fails? ?<cr>

one of:
Quit
Continue
Add as axiom

What should be done if a proof fails? Quit<cr>

APPEND,PLISTP,REVERSE,REVERSE.REVERSE

Do you want to redo this event? Yes<cr>

,(APPEND PLISTP REVERSE PROVED)

7←(CLOSEF PROVE.FILE)
<USERNAME>REV.PROOFS.1

^C

;The file REV.PROOFS.1 now contains all of the theorem-prover
;output in response to the event list FOO.  Here is what
;the file looks like:

@TYPE REV.PROOFS.1<cr>

                    <LISP>LISP.EXE.128

                      <MOORE>CODE..3

                     <MOORE>CODE1..4

                      <MOORE>DATA..3

                     <MOORE>DATA1..3
```

Monday, April 9, 1979 6:55AM-PST^L

```
_DEFN(APPEND (X Y)
             (IF (LISTP X)
                 (CONS (CAR X) (APPEND (CDR X) Y))
                 Y))
```

The lemma CDR.LESSP establishes that (COUNT X) decreases according to the well-founded function LESSP in each recursive call. Hence, APPEND is accepted under the definitional principle. Note that:

```
(OR (LISTP (APPEND X Y))
    (EQUAL (APPEND X Y) Y))
```

is a theorem.

Load average during definition: 1.478143
Elapsed time: 4.332 seconds
CPU time (devoted to theorem proving): .781 seconds
GC time: .254 seconds
IO time: 0.0 seconds
CONSes consumed: 615

APPEND
^L

```
_DEFN(PLISTP (X)
             (IF (LISTP X)
                 (PLISTP (CDR X))
                 (EQUAL X NIL)))
```

The lemma CDR.LESSP informs us that (COUNT X) goes down according to the well-founded function LESSP in each recursive call. Hence, PLISTP is accepted under the definitional principle. Observe that:

```
(OR (FALSEP (PLISTP X))
    (TRUEP (PLISTP X)))
```

is a theorem.

Load average during definition: 1.478143
Elapsed time: 1.109 seconds
CPU time (devoted to theorem proving): .541 seconds
GC time: .175 seconds
IO time: 0.0 seconds
CONSes consumed: 436

```
PLISTP
^L

_DEFN(REVERSE (X)
              (IF (LISTP X)
                  (APPEND (REVERSE (CDR X))
                          (CONS (CAR X) NIL))
                  NIL))
```

The lemma CDR.LESSP establishes that (COUNT X) goes down according to the well-founded function LESSP in each recursive call. Hence, REVERSE is accepted under the principle of definition. Observe that:

```
      (OR (LITATOM (REVERSE X))
          (LISTP (REVERSE X)))
```

is a theorem.

```
Load average during definition:  1.478143
Elapsed time:  .999 seconds
CPU time (devoted to theorem proving):  .607 seconds
GC time:  .178 seconds
IO time:  0.0 seconds
CONSes consumed:  538

REVERSE
^L

_PROVE.LEMMA(REVERSE.REVERSE (REWRITE)
                     (IMPLIES (PLISTP X)
                              (EQUAL (REVERSE (REVERSE X)) X)))
```

Name the conjecture *1.

We will try to prove it by induction.

```
;We again omit most of the proof.
```

That finishes the proof of *1.1, which, consequently, also finishes the proof of *1. Q.E.D.

```
Load average during proof:  1.503348
Elapsed time:  22.516 seconds
CPU time (devoted to theorem proving):  6.309 seconds
GC time:  3.032 seconds
IO time:  2.49 seconds
CONSes consumed:  7781

PROVED
```

C. Example 3

Suppose that in Example 1, immediately after the proof of the REVERSE.REVERSE theorem, we had typed

```
9.MAKE.LIB(REV.LIB)
```

and the system returned

```
<USERNAME>REV.LIB.1
```

Then the entire state of the theorem-prover would have been written to the new file <USERNAME>REV.LIB.1. Using that file we could start another session from that state, without having to reprocess the definitions of APPEND, PLISTP or REVERSE, or prove REVERSE.REVERSE again:

```
@THM<cr>
(<MOORE>THM.EXE.1 . <LISP>LISP.EXE.128)

2.INIT(REV.LIB)

REVERSE.REVERSE

3.PPE(APPEND REVERSE.REVERSE)

(DEFN APPEND
      (X Y)
      (IF (LISTP X)
          (CONS (CAR X) (APPEND (CDR X) Y))
          Y))

(PROVE.LEMMA REVERSE.REVERSE (REWRITE)
             (IMPLIES (PLISTP X)
                      (EQUAL (REVERSE (REVERSE X))
                             X)))

NIL

;Note that APPEND and REVERSE.REVERSE are events in the
;current data base.  So are PLISTP and REVERSE.  In fact,
;the theorem-prover is in the same state it was in right
;after we proved REVERSE.REVERSE in Example 1.  For example,
;it now knows that (REVERSE (REVERSE X)) is X when (PLISTP X)
;is true and it will use it as a rewrite rule.
```

D. Example 4

We conclude with some light humor.

```
4. DEFN(PP (X) (PP X))

WARNING:  The admissibility of PP has not been established.
We will assume the function to be well-defined.  This may
render the theory inconsistent.  An induction principle for
this function has been assumed, corresponding to the obvious
subgoal induction for the function.


Note that F is a theorem.

;Pretty smart theorem-prover, eh?  Do you see a proof of F?

*************************************************************
***                                                       ***
***                    F A I L E D !                      ***
***                                                       ***
*************************************************************

Load average during proof:  1.399432
Elapsed time:  4.016 seconds
CPU time (devoted to theorem proving):  .238 seconds
GC time:  .315 seconds
IO time:  0.0 seconds
CONSes consumed:  110

(
"

*************************************************************
***                                                       ***
***                    F A I L E D !                      ***
***                                                       ***
*************************************************************
")

;Here's a hint.

5. PROVE.LEMMA(PP.NONSENSE (REWRITE) (IMPLIES (NOT (PP X)) (PP X]

     Name the conjecture *1.

     We will try to prove it by induction.  Two inductions
are suggested by terms in the conjecture.  However, they
merge into one likely candidate induction.  We will induct
according to the following scheme (AND).  This scheme is
```

```
justified by the assumption that PP is total.  This
produces:

      (TRUE).

     That finishes the proof of *1.  Q.E.D.

Load average during proof:  2.536146
Elapsed time:  12.445 seconds
CPU time (devoted to theorem proving):  .571 seconds
GC time:  .576 seconds
IO time:  0.0 seconds
CONSes consumed:  348


PROVED

^C
@LOGOUT

 System shutdown scheduled for Wed 11-Apr-79 23:00:00,
 Up again at Thu 12-Apr-79 04:00:00
Logout Job 18, User USERNAME, Account ACCOUNT, TTY 250, at
  9-Apr-79 07:24:08 Used 0:0:5 in 0:2:50
```

REFERENCES

1. Robert S. Boyer and J Strother Moore, A Computational Logic, to appear, approximately in October 1979, in the ACM Monograph Series (Academic Press, New York, 1979).

2. Digital Equipment Corporation, "DECSYSTEM-20 Users's Guide," Order Nos. AD-4179B-T1, AA-4179B-TM, First Printing (Updated), Maynard, Massachusetts (January 1978).

3. Warren Teitelman, "INTERLISP Reference Manual," Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California, 94304 (1978).

DISTRIBUTION LIST

The below listing is the official distribution list for the technical reports for Contract N00014-75-C-0816.

Defense Documentation Center — 12 copies
Cameron Station
Alexandria, VA. 22314

Office of Naval Research — 2 copies
Information Systems Program
Code 437
Arlington, VA. 22217

Office of Naval Research — 1 copy
Branch Office, Boston
495 Summer Street
Boston, MASS. 02210

Office of Naval Research — 1 copy
Branch Office, Chicago
536 South Clark Street
Chicago, ILL. 60605

Office of Naval Research — 1 copy
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA. 91106

New York Area Office — 1 copy
715 Broadway - 5th Floor
New York, N.Y. 10003

Assistant Chief for Technology — 1 copy
Office of Naval Research
Code 200
Arlington, VA. 22217

Naval Research Laboratory — 6 copies
Technical Information Div.,
Code 2627
Washington, D.C. 20375

Dr. A. L. Slafkosky — 1 copy
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380

Office of Naval Research — 1 copy
Code 455
Arlington, VA. 22217

Office of Naval Research — 1 copy
Code 458
Arlington, VA. 22217

Naval Elec. Laboratory Center — 1 copy
Advanced Software Tech. Div.
Code 5200
San Diego, CA. 92152

Mr. E. H. Gleissner — 1 copy
Naval Ship Res. & Dev. Center
Computation & Math. Dept.
Bethesda, MD. 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, D.C. 20350

Officer-in-Charge — 1 copy
Naval Surface Weapons Center
Dahlgren Laboratory
Dahlgren, VA. 22448
Attn: Code KP