A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover

Matt Kaufmann

Technical Report #19 May 1988

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

Acknowledgements

An early version of part of this system was written by J Moore, who I also thank for suggesting this project. Also, Bob Boyer and J Moore have been very helpful in answering questions about their theorem prover. I also thank David Goldschlag, Carl Pixley, Matt Wilding, and Bill Young for their helpful feedback in the development of this system. Finally, I truly appreciate the congenial and stimulating atmosphere that has been present during my employment at the Institute for Computing Science at the University of Texas and Computational Logic, Inc. I can't imagine a more pleasant work situation for me anywhere.

This report is a revision of ICSCA Technical Report 60 [1], which was supported in part at the University of Texas by ONR Contract N00014-81-K-0634; by the Defense Advanced Research Projects Agency, Arpa Order 5246, issued by the Space and Naval Warfare Systems Command under Contract N00039-85-K-0085; and by IBM grant award, "ICSCA--Research in Hardware Verification, Various Purposes." At Computational Logic, Inc., work was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

Preface

NOTE: Most users can get by with reading no further than Section 1 of this manual, at least until they desire to utilize the potential of this system more fully. That said .....

This manual accompanies a system for checking the provability of terms in the Boyer-Moore logic, as described in [2] and (more recently) updated in [3]. This system is loaded on top of the Boyer-Moore Theorem Prover, as explained below, and is integrated with that prover[1]. Thus, the user can give commands at a low level (such as deleting a hypothesis) or at a high level (such as calling the Boyer-Moore Theorem Prover). As with a variety of proof-checking systems, this system is goal-directed: a proof is completed when the main goal and all subgoals have been proved. A notion of *macro commands* lets the user create compound commands, in the spirit of the *tactics* and *tacticals* of LCF [4]. Upon completion of an interactive proof, the lemma with its proof may be stored as a Boyer-Moore *event* which can be added to the user's current library of events (i.e. definitions and lemmas). An on-line help facility is provided.

We assume a little familiarity with the Boyer-Moore Theorem Prover, especially, with *definition* events **(DEFN ....)** and *lemma* events **(PROVE-LEMMA ....)**. Instructions on how to install the system are provided in the file "READ_ME".

The manual is organized into sections as follows. Section 1 gives an introduction to the system, including a short annotated transcript of a sample session. Section 2 contains a reasonably careful explanation of the notion of a proof "state" and an introduction to the various commands and what they do. The third section is an explanation of the facility for interactive proofs of termination for **DEFN** events. The fourth section is a presentation of helpful tips. The fifth section contains a description of the *macro command* facility together with a more detailed description of the top-level loop. The sixth and final section lists some additional (rather specialized) features provided by the system for advanced users. This manual concludes with four appendices. The first appendix contains an annotated transcript of a sample session which is somewhat more realistic than the one presented in Section 1, so that beginning users may gain some additional feel for how the system might be used. The second appendix presents a soundness argument for this system. The third appendix describes an extended syntax for terms. The final appendix lists information printed by the "short" help facility.

---

[1] This system uses a slight extension of the syntax for terms, as described in Appendix 3.

1. Introduction to the system

This section is divided into four parts. The first part contains the essentials needed to get started with the system. In the second part, a few words are said about the organization of the system into goals, states, and the state stack. The third part describes the four types of commands (change, help, meta, and macro) that the user may give. The final part (which is perhaps the most important) gives a basic introduction to the use of the system by way of an example. Although more precise details are given in later sections, the four parts of this section should provide an adequate introduction for most users.

1.1 Getting started

The system is started up by loading the Boyer-Moore theorem prover (in whatever manner you normally do this) and then loading the file "proof-checker.lisp", i.e. submitting the following command to the Lisp prompt and hitting <return>:

```
(LOAD "proof-checker.lisp")
```

You may now do whatever you normally would do when using the Boyer-Moore system. But suppose that you now have a proposed theorem that you want to verify. Then you may submit that theorem as an argument to the Lisp function **VERIFY**; for example, to proof-check the associativity of **APPEND**, submit the form[2]

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

You will see the prompt "->: ", indicating that the system is ready to accept *proof commands*, i.e. commands which alter the *state* of the system. The idea then is to give various commands, some of which may introduce subgoals. Upon completion of the session, you may give an **EXIT** command, which may be used to cause the goal to be stored as a Boyer-Moore *event* in case the proof is complete; more on this later. However, whenever you leave the interactive proof-checker (by using an **EXIT** command), you may re-enter where you left off simply by submitting the form

```
(VERIFY)
```

to Lisp. By the way, if one *aborts* in the middle of executing a command, for example if one aborts in the middle of a **PROVE** command, the system is designed so that a quit (e.g. **:q** in Kyoto Common Lisp) should then bring one back to the "->:" prompt; if not for some reason, then **(VERIFY)** will take one back to this prompt. Aborting during the execution of any *change command* should result in no change to the global state of the system. The user can ascertain whether the command has completed by submitting the help command **COMMANDS**; we believe that it is not possible for a change command to complete only partially.

---

[2]The form should be submitted directly to Lisp, not read in from a file.

At any point during the proof, you may review the available commands by submitting **HELP** or **HELP-LONG**, which give lists of the main commands, or **(HELP <command$_1$> <command$_2$> ... <command$_n$>)**, (similarly for **HELP-LONG**), which give descriptions of the indicated commands. The principle here is that most users will be content with the help provided by **HELP** rather than **HELP-LONG**, but **HELP-LONG** may be used to get more precise specifications of the commands (which can be useful in obscure cases) and more details about unusual arguments[3] that are allowed for various commands. More precisely, the difference between **HELP** and **HELP-LONG** tends to be that **(HELP <command$_1$> <command$_2$> ... <command$_n$>)** prints out enough information for most purposes and generally gives examples, while the corresponding use of **HELP-LONG** rarely gives examples but instead gives more dry and often more complete specifications of the commands.

During the course of a session, the user will submit a number of commands which will alter the state of the system. Some of these commands will create new goals to be proved. The session is complete when all goals have been proved, in the sense that their conclusions have been reduced to **T** (*true*). (The notions of *goal* and *conclusion*, among others, are explained in the next subsection.) At that time the user may create an event by submitting an appropriate **EXIT** command. For example, the final subsection of Section 1 (below) will display an interactive session for proving the associativity of the **APPEND** function. Upon completion of that session, the user will type an **EXIT** command in order to "create" the following Boyer-Moore event:

```
(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND
             (REWRITE)
             (EQUAL (APPEND (APPEND X Y) Z)
                    (APPEND X (APPEND Y Z)))
             ((INSTRUCTIONS (INDUCT (APPEND X Y))
                            PROMOTE
                            (DIVE 1 1)
                            X UP X NX X TOP
                            (DIVE 1 2)
                            = TOP S S)))
```

The **INSTRUCTIONS** hint is a record of all the (successful) proof commands given during the session. This event may be submitted like any other Boyer-Moore event when running events in *batch mode*, i.e. when submitting events to Lisp rather than creating them through the interactive proof checker.[4]

---

[3]By *arguments* to a command we mean, for example, that the numbers 3 and 4 are arguments to **DIVE** in the instruction **(DIVE 3 4)**.

[4]Since the Lisp machine prints this event out in the Lisp window, those using a Lisp machine may want to have output "dribbling" into a "dribble file", where they can then grab this event and copy it into the list of events. See p. 443 of the Common Lisp manual [5] for a discussion of the Common Lisp function **DRIBBLE**. Those using an appropriate environment may instead prefer to run their Lisp under Emacs.

1.2 Organization: goals, states, and the state stack

This section gives an informal introduction to the organization of the system. Details are postponed until Section 2.

The history of an interactive session is stored as a *state stack*, which is a list of *proof states* (or, *"states"* for short). A *state* contains a collection of *goals*, where each *goal* has a list of *hypotheses* and a *conclusion*. Each of the goal's hypotheses can either be active or hidden; hidden hypotheses are generally ignored by proof commands unless (and until) they are made active (again). Dependencies are recorded between goals: the goals are stored in a directed acyclic graph, where an arc joins one goal to another if the former depends on the latter. At the start of an interactive session, only one state is on the state stack, namely the one corresponding to the user's input.

Let us consider an example. Suppose the user enters the system with the goal of proving the associativity of **`APPEND`**:

```
(verify (equal (append (append x y) z)
               (append x (append y z))))
```

Then the unique state on the state stack contains only one goal, namely the goal whose conclusion is the argument given to **verify** above and whose hypothesis list is empty. Now various commands may be given to create new states by modifying this goal, possibly introducing subgoals in the process. *The idea is that when invoking a proof command, the goal follows from its modified version together with the subgoals that are created.* Note that all goals are viewed as (implicitly) universally quantified; for example, the initial goal asserts the equality of the two **`APPEND`** expressions shown above for *all* values of **`x`**, **`y`**, and **`z`**.

Continuing with this example, notice that the initial goal follows by the Boyer-Moore induction principle from the following two subgoals, which one might call the "base step" and "induction step" respectively:

```
(IMPLIES (NOT (LISTP X))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))

(IMPLIES (AND (LISTP X)
              (EQUAL (APPEND (APPEND (CDR X) Y) Z)
                     (APPEND (CDR X) (APPEND Y Z))))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))
```

Now this induction is "dual to" the recursion in the definition of the function **`APPEND`**. Hence the following command may be invoked to generate these subgoals. (This command is in complete analogy to the giving of this as a hint to the Boyer-Moore **PROVE-LEMMA** command.)[5]

```
(induct (append x y))
```

At any rate, the original goal follows from the two subgoals above (when all goals are viewed as universally quantified). So, when the above command is executed, a new state is pushed onto the state stack, where the new state contains two new goals (one for each of the subgoals displayed above). Moreover, since the original goal follows from these two goals, its conclusion may be replaced by **T** (*true*) in the new state. That is, the property mentioned earlier does indeed hold (and we restate it now for emphasis):

(*) *When invoking a proof command, the goal follows from its modified version together with the subgoals that are created.*

Now in fact, a *state* also has a *current goal* as well as a pointer to the *current subterm* of the current goal's conclusion. Here is another example of creating a new state to be pushed onto the state stack. Suppose that the current goal has hypotheses as follows:

```
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))
```

Also suppose that the conclusion of the current goal is as follows, where the current subterm has been "highlighted" with asterisks:

```
(EQUAL (CONS (CAR X)
             (*** (APPEND (APPEND (CDR X) Y) Z)
                  ***))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))
```

Now the second hypothesis equates the current subterm with the term **(APPEND (CDR X) (APPEND Y Z))**, and we might wish to make a substitution of this new term for the current subterm. It turns out that the command = does just that. But more precisely, this = command pushes a new state on top of the state stack, where the new state is obtained from the old state by making the substitution indicated by the second hypothesis (for the current subterm in the conclusion of the current goal). Unlike the previous example, no new subgoals are generated, and the modified version of the current goal has a conclusion that is not yet **T** (*true*). But again, the key property (*) above is maintained, since the current version of the goal follows from the modified version (in fact it follows by making the reverse of the equality substitution that was used).

---

[5]The user is also allowed to give the command **INDUCT**, which lets the system choose an induction scheme according to Boyer and Moore's heuristics. However, like this example, **INDUCT** does *not* call the prover but instead creates appropriate subgoals. We'll see later that there are ways to call the Prover if one wants to, e.g. by instead submitting **(THEN INDUCT)**. But now we're getting carried away here....

Suppose instead that the second hypothesis had not been present in the example above. Then a different version of the = command would be appropriate here, namely

```
(= * (APPEND (CDR X) (APPEND Y Z)) 0)
```

(This and other commands are introduced in the example here and in the first appendix, and are explained by the help facility.) In this case, the substitution would still have been made but also a new subgoal would have been generated. This new subgoal would have had the same hypotheses as the current goal, but its conclusion would have been the equality of the current subterm with the term to be substituted for it. Again the key property (*) would be maintained: the current goal follows from the new version (obtained by substitution) together with the goal stating the equality of the relevant terms.

1.3 The four types of commands: change, help, meta, and macro

Recall that the purpose of an interactive proof session is ultimately to create a state in which every goal has conclusion equal to **T** (*true*). The commands that push new states on top of the state stack, such as the **INDUCT** and **=** commands described in Subsection 1.2 above, are called *change* commands. These are the only "official" commands, in the sense that each **PROVE-LEMMA** event created upon **EXIT** from the system will store these commands in the **INSTRUCTIONS** hint, as shown in Subsection 1.1 above.

However, there is a vast difference between the result of an interactive session and the session itself. That is, even though one's aim is to arrive at a sequence of change commands which result in all goals having conclusions of **T**, one would certainly like helpful support toward achieving that end. The other three types of commands provide such support.

The *help* commands display useful information while making no change whatsoever in the state of the system. Probably the most commonly used help command is **P**: print the current subterm. Some other commonly used help commands include **HYPS** (print the current goal's hypotheses), **SHOW-REWRITES** (show the rewrite rules which apply to the current subterm), **COMMANDS** (show the proof commands thus far in the current session), **GOALS** (print the names of the remaining goals to be proved), and **HELP** and **HELP-LONG** (print information about the commands); a complete list is given in the final appendix.

The *meta* commands allow one to manipulate the entire state stack to (potentially) create a new state stack. However, the new state stack will still correspond to a sequence of change commands.[6] Probably the most commonly used meta command is **UNDO**, which may be used to pop the state stack (i.e. revert to a previous state). In addition, the previous state stack is stored to that the meta command **RESTORE** may be used to undo the effect of an **UNDO** command. Most of the other meta commands are used to create *macro* commands, which are the remaining type of command.

The idea behind *macro* commands are that they enable the user to extend the system. For example, suppose that one wants a command which prints the value of an arbitrary Lisp form **`<exp>`**. Now the meta command **LISP** evaluates an arbitrary Lisp form, so one could simply submit the command **(LISP (PRINT <exp>))**. But perhaps the user requires this sort of thing frequently and is tired of typing **(LISP (PRINT ...))**. Then he can define a macro command which does just that. In fact, some macro commands are provided in the system that is initially loaded, including a macro command **PRINT**. Macro commands are discussed at length in Section 5, including a detailed description of the top-level evaluation mechanism and how to define macro commands. So we'll keep the discussion here short. Even at this stage though it is worth pointing out that the help facility does print information about the predefined macro commands, so the user can begin using them right away. It is also worth pointing out that, as the terminology implies, a macro command is actually expanded (textually) into its body, and then the resulting command is resubmitted. (The resulting command may however also be a macro command, which is in turn resubmitted, and so on.) So for example, in the example described above where we defined the macro command **(PRINT X)** to expand to **(LISP (PRINT X))**[7], if the user submits the macro command **(PRINT <exp>)** then the top-level interactive loop expands this command into the command **(LISP (PRINT X))**, which (being a meta command rather than a macro command) is then executed.

The full story of macro commands is presented in Section 5. That section is however *not* necessary reading for the user to be able to use macro commands.

---

[6]In fact, the command creating a state is one of the fields of the *state* record, so the commands actually exist in the state stack. Actually, a malicious user could create "invalid" state stacks using e.g. the **LISP** meta command; however, we claim that the non-malicious user would not get into this trouble. Moreover, even the malicious user cannot violate soundness, in the sense that we do not consider events to have been completely checked until they have been run back through the system. Since only *change* commands may be given as **`INSTRUCTION`** hints to the **PROVE-LEMMA** events, malicious **LISP** commands will be ignored when these events are run back through.

[7]In fact the **PRINT** command is a bit fancier than this.

1.4 An introduction by way of example

Here is a annotated display of a short interactive session corresponding to the **PROVE-LEMMA** event shown above. Comments will be enclosed in curly braces {} and italicized. Other than the initial **verify** command, user input is preceded on each line by the prompt "->: "; the rest is printed by the system. Some extra blank lines have been added for readability.

A slightly more realistic example appears in Appendix 1.

```
(verify (equal (append (append x y) z)
               (append x (append y z))))

->: p  {Print the current subterm}

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

->: (induct (append x y))  {Similar to the Boyer-Moore INDUCT hint}

Creating 2 new subgoals, (MAIN . 1) and (MAIN . 2).

The proof of the current goal, MAIN, has been completed.  However, the
following subgoals of MAIN remain to be proved: (MAIN . 1) and (MAIN . 2).
Now proving (MAIN . 1).
{Note:  The proof of this goal is "completed" because there's nothing left to do except to prove its subgoals.}

->: p  {Print the current subterm}

(IMPLIES (AND (LISTP X)
              (EQUAL (APPEND (APPEND (CDR X) Y) Z)
                     (APPEND (CDR X) (APPEND Y Z))))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))

->: hyps  {Print the current hypotheses}

*** Active top-level hypotheses:
There are no top-level hypotheses to display.

*** Active governors:   {Governors are discussed later}
There are no governors to display.

->: promote  {Turn the left side of the implication into top-level hypotheses}

->: hyps  {Print the current hypotheses}

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))

*** Active governors:
There are no governors to display.

->: p  {Print the current subterm}

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

->: (dive 1 1) *{Point to the first argument of the current subterm and then to that subterm's first argument}*

->: p *{Print the current subterm}*

```
(APPEND X Y)
```

->: pp-top *{Print the entire conclusion, highlighting the current subterm}*

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

->: x *{Expand the function call in the current subterm, namely in* **(APPEND X Y)***, and simplify the result.}*

->: p *{Print the current subterm -- Notice that the expansion using the* **X** *command simplified away the* **IF** *test in the body of the definition of* **APPEND***.}*

```
(CONS (CAR X) (APPEND (CDR X) Y))
```

->: pp-top *{Print the entire conclusion, highlighting the current subterm}*

```
(EQUAL (APPEND (*** (CONS (CAR X) (APPEND (CDR X) Y))
                    ***)
               Z)
       (APPEND X (APPEND Y Z)))
```

->: up *{Move up to the enclosing term}*

->: pp-top *{Print the entire conclusion, highlighting the current subterm}*

```
(EQUAL (*** (APPEND (CONS (CAR X) (APPEND (CDR X) Y))
                    Z)
            ***)
       (APPEND X (APPEND Y Z)))
```

->: x *{Expand the function call in the current subterm and simplify}*

->: pp-top *{Print the entire conclusion, highlighting the current subterm}*

```
(EQUAL (*** (CONS (CAR X)
                  (APPEND (APPEND (CDR X) Y) Z))
            ***)
       (APPEND X (APPEND Y Z)))
```

->: nx *{Move to the next argument in the current subterm}*

->: pp-top *{Print the entire conclusion, highlighting the current subterm}*

```
(EQUAL (CONS (CAR X)
             (APPEND (APPEND (CDR X) Y) Z))
       (*** (APPEND X (APPEND Y Z)) ***))
```

->: x *{Expand the function call in the current subterm and simplify}*

->: pp-top *{Print the entire conclusion, highlighting the current subterm}*

```
(EQUAL (CONS (CAR X)
             (APPEND (APPEND (CDR X) Y) Z))
       (*** (CONS (CAR X)
                  (APPEND (CDR X) (APPEND Y Z)))
            ***))
```

->: top *{Move to the top of the goal's conclusion}*

```
->: hyps
```
{Print the current hypotheses}

```
*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))

*** Active governors:
There are no governors to display.

->: p
```
{Print the current subterm}

```
(EQUAL (CONS (CAR X)
             (APPEND (APPEND (CDR X) Y) Z))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))

->: (dive 1 2)
```
{Point to the first argument of the current subterm and then to that subterm's second argument}

```
->: pp-top
```
{Print the entire conclusion, highlighting the current subterm}

```
(EQUAL (CONS (CAR X)
             (*** (APPEND (APPEND (CDR X) Y) Z)
                  ***))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))

->: hyps

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))

*** Active governors:
There are no governors to display.

->: =
```
{Make a substitution for the current subterm, using an equality among the current hypotheses and governors.}

```
->: p
```
{Print the current subterm}

```
(APPEND (CDR X) (APPEND Y Z))

->: top

->: p
```
{Print the current subterm}

```
(EQUAL (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z)))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))

->: s
```
{Simplify}

```
The current goal, (MAIN . 1), has been proved, and has no dependents.
Now proving (MAIN . 2).

->: p
```
{Print the current subterm}

```
(IMPLIES (NOT (LISTP X))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))
```

```
->: s {Simplify}

The current goal, (MAIN . 2), has been proved, and has no dependents.

*!*!*!*!*!*!*  All other goals have also been proved!  *!*!*!*!*!*!*
You may wish to EXIT -- type (HELP EXIT) for details.

->: (exit associativity-of-append (rewrite)) {As described previously}

The indicated goal has been proved.  Here is the desired event:

(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND
             (REWRITE)
             (EQUAL (APPEND (APPEND X Y) Z)
                    (APPEND X (APPEND Y Z)))
             ((INSTRUCTIONS (INDUCT (APPEND X Y))
                            PROMOTE
                            (DIVE 1 1)
                            X UP X NX X TOP
                            (DIVE 1 2)
                            = TOP S S)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? Y {User response}

[ 0.2 0.0 0.0 ]
ASSOCIATIVITY-OF-APPEND
{The event has been stored in the Boyer-Moore database of events, i.e. it now shows up in the Lisp variable CHRONOLOGY.}
```

Remark. The triple of numbers above the event name "ASSOCIATIVITY-OF-APPEND" above is meaningless when printed as a response to the prompt at the end of an interactive session. However, when the event is submitted to Lisp (in what we refer to as *batch mode*), the numbers have the following meaning (which agrees with the usual meaning). The sum of the second and third numbers is the amount of time spent inside the Theorem Prover, and the third number alone is that portion of the sum which is spent inside the I/O routines. The first number is what is left of the total, i.e. the sum of the three numbers is the total time spent on the event. To summarize, we have three times:

[Miscellaneous_time Proof_time I/O_time(inside Prover)]

2. A more detailed description of the system

In this section we present detailed accountings of the notions of *goal*, *state*, and *state stack* that were introduced in Section 1, together with some related notions. At the same time we also give an overview of the *commands* that the user may give; more complete details of the commands may be found by using the help facility, i.e. by typing **(HELP <command$_1$> <command$_2$> ... <command$_n$>)**, or similarly with **HELP-LONG**. (Information is also included in the final appendix.) Finally, top-level matters such as creation of **PROVE-LEMMA** events and aborting and re-entering an interactive session are dealt with in detail in the final subsection.

2.1 Goals

A *goal* is a record with the fields discussed below. It is not important for the user to know the names of these fields (or of the fields for the *state* record to be presented next), but the concepts underlying them are helpful for using the system. It is also important to keep in mind that a goal is viewed as the universal closure [8] of the implication whose antecedent is the term formed by conjoining the goal's hypotheses and whose consequent is the goal's conclusion (or, of just the conclusion in case there are no hypotheses). Recall though the convention from [2] that a term **`<exp>`** may be used as a formula by identifying it with the negation of **`[EXP = F]`**.

`A` *goal* **`consists of:`**
`CONC, HYPS, DEPENDS-ON, and GOAL-NAME`

`CONC` is called the *conclusion* of the goal, and is a Boyer-Moore term.

`HYPS` is called the *hypotheses* of the goal, or sometimes the *top-level hypotheses*, and is a list of Boyer-Moore terms. The hypotheses of the current goal can be found by employing the help command **`HYPS`**. But actually --

`HYPS` is *really* a list of *pairs*, where the first element of the pair is a hypothesis and the second element is either the atom **`A`** or the atom **`H`**, indicating that the hypothesis is *active* or *hidden* (respectively). All of the *change* commands are set up so that they "ignore" hidden hypotheses. For example, the **S** (simplify) command uses only the active hypotheses in simplifying the current subterm. The change commands **HIDE-HYPS** and **SHOW-HYPS** are used to hide and activate hypotheses, so that one can hide hypotheses temporarily and then bring them back. There is also a **DROP** change command which eliminates hypotheses.[9]

There are several other change commands which modify the **`HYPS`**. Here are brief, simplified descriptions; use the help facility or see the final appendix for more details. The **CLAIM** command allows one to add additional hypotheses that follow from the existing (active) hypotheses. The command **PROMOTE** modifies a current goal with conclusion $\mathbf{(IMPLIES\ TERM_1\ TERM_2)}$ by replacing its conclusion with $\mathbf{TERM_2}$ while adding $\mathbf{TERM_1}$ to its hypotheses. (More accurately, as with all change commands it pushes a new state on the state stack which agrees with the old state except that the current goal has been modified approximately as

---

[8]The *universal closure* of a formula in first-order logic is obtained by prefixing it with a sequence of all quantifiers of the form **`(FORALL X)`** as **`X`** ranges over the free variables of the formula.

[9]Except one can always return to a previous state with the **UNDO** command; more on **UNDO** later.

indicated.) On the other hand, the command **DEMOTE** is (roughly) an inverse to **PROMOTE**. The **CONTRADICT** command exchanges a hypothesis and the conclusion while negating each of them; perhaps a better name would have been "contrapose". Finally, **USE-LEMMA** allows one to add hypotheses that are instances of a lemma from the chronology.

`DEPENDS-ON` is a list of Lisp objects (S-expressions): the names of the goals that the given goal depends on. (Here, goal X *depends on* goal Y if Y is related to X by the transitive closure of the (immediate) subgoal relation, i.e. Y is a subgoal of X or a subgoal of a subgoal of X or ....) The dependents of all the goals are shown with the help command **(GOALS ALL)**. Several commands modify the **`DEPENDS-ON`** field. For example, a command that creates a dependency is **PUSH**, which replaces the current subterm **`<exp>`** by **`T`** (*true*).[10] In this case, the current goal is further modified by adding the name of a new goal to its **`DEPENDS-ON`** field, where the new goal has the current goal's active hypotheses and governors as its (top-level) hypotheses and has **`<exp>`** as its conclusion. The **INDUCT** command creates subgoals as discussed in the demo in Section 1. Other change commands that create dependents are **GENERALIZE**, **SPLIT**, and sometimes **CLAIM**, **REWRITE**, **BASH**, and **=**.

`GOAL-NAME` is a Lisp object that we call the *name* of the goal. When the system generates subgoals of a given goal named **`<name>`** it does so by creating new names of the form **`(<name> . N)`**, where **`N`** is a positive integer. (Exceptions: the commands **PUSH** and **GENERALIZE** allow one to specify the new subgoal.)

2.2 States

Next, we consider the *state* record type.

```
A state consists of:
INSTRUCTION, CURRENT-TERM, GOVERNORS, CURRENT-ADDR-R, GOAL,
   OTHER-GOALS, CUMULATIVE-LEMMAS-USED, FREWRITE-DISABLED-RULES,
   and ABBREVIATIONS
```

`INSTRUCTION` is the change command that created the given state from the previous state. (However, **`INSTRUCTION`** is **`START`** for the initial state, i.e. the state created by the call of **VERIFY** on the term to be proved.)

[10] Exception: if **`<exp>`** is not known either to be boolean or to be in a position where only propositional equivalence needs to be maintained, then it is replaced by **`(IF <exp> <exp> T)`**.

The next three fields -- **CURRENT-TERM**, **GOVERNORS**, and **CURRENT-ADDR-R** -- are all based on the notion of a pointer to a subterm of the current goal's conclusion. This pointer can be thought of as a list of positive integers which gives directions for diving in to the conclusion. For example, the *address* would be **(2 3 1)** for the subterm **(PLUS X Y)** of:

```
(TIMES (ADD1 X)
       (IF (EQUAL X Y)
           Y
           (SUB1 (PLUS X Y))))
```

since: we are diving to the second argument of the **TIMES** term, then the third argument of that **IF** term, and then finally the first argument of the **SUB1** term. The **CURRENT-TERM**, usually called the *current subterm*, is in this case **(PLUS X Y)**, while the **CURRENT-ADDR-R**, i.e. current-address-reversed, is **(1 3 2)**. The **GOVERNORS** are, roughly speaking, the IF-tests accumulated on the way to diving down to the current subterm. More precisely, the **GOVERNORS** is a list of all terms which *govern* the current subterm, in the sense of the definition of *governs* on the top of page 45 of [2]. For example, the **GOVERNORS** of **(PLUS X Y)** in the term displayed above is the one-element list **((NOT (EQUAL X Y)))**. NOTE: the **GOVERNORS** are defined only with respect to the **IF**-structure of the term. So for example, there are no governors of **X** in the term **(IMPLIES Y X)**. (There are of course other ways of using the hypothesis, as discussed in the "Proving Implications" commentary in Subsection 4.6.)

Let us discuss the commands which are particularly related to these notions of **CURRENT-TERM**, **GOVERNORS**, and **CURRENT-ADDR-R**. The commands **P** and **PP** both print the current subterm, the difference being that **P** introduces some notational conventions for terms with top function symbols among **CAR**, **CDR**, **CONS**, **AND**, **OR**, **PLUS**, **TIMES**. So for example, the command **P** prints the term **(PLUS X (PLUS Y Z))** as **(PLUS X Y Z)**, while **PP** prints it as is. In other words, the command **P** prints the current subterm just as the Boyer-Moore Theorem Prover would print it, while **PP** simply prints the term according to its actual structure.[11] Why ever use **PP** rather than the (prettier) **P**? Because the command **DIVE** may be used to move to a subterm of the current subterm according to a specified list of addresses (as explained in the example in Section 1), and on a few occasions it thus helps to see the term displayed in the more "raw" form given by the **PP** command. (However, a macro command **DV** has been provided for use with the **P** command. For example, if the current term is shown by the **P** command as **(PLUS X Y Z)** but by the **PP** command as **(PLUS X (PLUS Y Z))**, then to move to the subterm **Z** one gives either **(DIVE 2 2)** or **(DV 3)**.) Other change commands besides **DIVE** which change the current subterm are **UP**, **TOP**, **NX**, and **BK**, all of which are of course explained by the help facility (and in the final appendix).

[11] An exception is that both commands print explicit value terms using the quote notation, which is explained in detail in [6]. So for example, **(LIST 'A 'B)** is printed as **'(A B)** with both commands.

We continue with our description of the fields of a **STATE**. **GOAL** is the current goal, in the sense of "goal" described in the previous subsections (i.e. a record consisting of a **CONC**, **HYPS**, **DEPENDS-ON**, and **GOAL-NAME**). The user may change to another goal by using the change command **CHANGE-GOAL**. Also, when a goal has been completed, i.e. its conclusion is **T** (*true*), the system automatically chooses an uncompleted goal as the current goal (unless of course there are no further goals to prove, in which case it informs the user of that situation).

OTHER-GOALS is a list of all the goals in the **STATE** other than the current **GOAL** together with information about the current subterm of each goal.[12]

CUMULATIVE-LEMMAS-USED is a list of atoms to be used when forming the dependencies for a **PROVE-LEMMA** event which results from an interactive session. So for example, any function symbol which has a call expanded by the **S** (simplify) command will be added to this field in the new state, as will any function symbol or name of lemma used in a call to the Theorem Prover by the **PROVE** or **BASH** command (or any of several other commands).

FREWRITE-DISABLED-RULES is a list of atoms which are names of function symbols or rewrite rules which are to be ignored by the so-called "fast rewriter", which is called by the **S** (simplify) and **X** (expand) commands. This field is updated when the **ENABLE** and **DISABLE** commands are executed.

Finally, **ABBREVIATIONS** is a list of abbreviations in the following sense. An *abbreviation* is a pair consisting of an atom and a term, where the atom begins with the '@' character. The user interface is set up so that both printing of terms and (generally) reading of terms in commands is done with respect to this list of abbreviations. Unabbreviating takes place from the outside in. So, for example, if the abbreviations consist of the pairs

```
((@W . (ADD1 X))
 (@V . (PLUS (ADD1 X) Y)))
```

then the term **(TIMES Z (PLUS (ADD1 X) Y))** would be printed as **(TIMES Z @V)** rather than as **(TIMES Z (PLUS @W Y))**. Notice that this is true whether that term is printed as part of the current term or as one of the hypotheses (*via* the **HYPS** command). The **ADD-ABBREVIATION** and

---

[12]More precisely, each member of the list **OTHER-GOALS** is a list of the form **(goal' current-term' governors' current-addr-r')**, where each member of this list is of the type that one would expect from its name. The idea here is that when **goal'** becomes the current goal, then the other three members of the list will become the current subterm, governors, and current-address-reversed (respectively).

REMOVE-ABBREVIATIONS commands modify the **`ABBREVIATIONS`** field of the state, or more precisely, they push a new state onto the state stack which obtained from the previous top state by changing the **`ABBREVIATIONS`** field appropriately. The current abbreviations can be viewed using the help command **SHOW-ABBREVIATIONS**.

2.3 State stacks and other related matters

The next topic for this section is that of the *state stack*. As we already discussed in Section 1, a stack of states is maintained -- in fact it is stored in the global Lisp variable **`STATE-STACK`** -- such that execution of a change command pushes a new state on top of this stack. More precisely, when the user submits a change command then one of two things can happen. First, the command might not be "allowed", either for syntactic or semantic reasons. For example, one might reference an undefined and undeclared function symbol in **ADD-ABBREVIATION**; or the **S** (simplify) command may fail to make any changes in the current subterm. In such cases, **`STATE-STACK`** is unchanged. However, if the command "succeeds", then the appropriate new state is pushed on top of the state stack.

UNDO is a meta command which can be used to pop states off the state stack. In this way, abortive "branches" in an interactive proof effort can be undone. The **UNDO** and **BOOKMARK** (see below) commands interact in that the argument to **UNDO** can be a bookmark. (As usual, we defer the details to the final appendix or to the user’s inquiry of the help facility.) Generally, though, the user will give **UNDO** a numeric argument which indicates the number of states to be popped. (The default of **`1`** is used when no arguments to **UNDO** are given.) The help command **COMMANDS** (or macro command **COMM**) may be used to list the **`INSTRUCTION`** fields of the states in **`STATE-STACK`**. They are listed in reverse order in order to aid in the use of the **UNDO** command.

On occasion, a user may wish to undo an **UNDO** command. The meta command **RESTORE** has been provided for this purpose. What **RESTORE** *really* does is swap the value of **`STATE-STACK`** with the value of another global Lisp variable, **`OLD-SS`**. Each time an **UNDO** command is executed, the variable **`OLD-SS`** is set to the existing value of **`STATE-STACK`** while **`STATE-STACK`** is in turn reset to be the popped state stack; and that is why **RESTORE** will undo an **UNDO**.

Two change commands push new states on the state stack in particularly trivial ways. The command **COMMENT** simply inserts comments: it creates a new state from the previous top state by simply inserting an appropriate comment into the **`INSTRUCTION`** field. The command **BOOKMARK** is similar in that it simply creates an instruction of the form **(BOOKMARK x)**; however, as mentioned above, bookmarks are understood by the meta command **UNDO**.

We omit mention of the other commands here, referring the reader once again to the help facility or the final appendix.

2.4 Top-level matters

In this final subsection of Section 2, we discuss carefully the relation between this system and the Boyer-Moore system. Let us begin by recalling some such matters which have been explained above. To enter the system, one submits

```
(verify <term>)
```

to Lisp. This puts the user into a read-eval-print loop signified by the prompt "->: ", where one can submit the various proof-checker commands. However, the user may for some reason wish to leave this loop and return to the top level of Lisp, either by aborting out or by giving the command **EXIT**. By submitting the form

```
(verify)
```

to Lisp, the user will return to the system's read-eval-print loop (and receive the prompt "->: "), where the global state (i.e. **`STATE-STACK`** and **`OLD-SS`**) is exactly as it was when the system was exited[13]. **(VERIFY)** also initializes certain parameters to match the current state of the Theorem Prover's database. So for example, one can **EXIT** the interactive environment, prove a rewrite rule using the regular Boyer-Moore **PROVE-LEMMA** mechanism, and then return to the existing interactive proof with **(VERIFY)**; the new rewrite rule will then be available for the rest of that session. We'll give an example of this activity below. (See also the discussion of **SAVE** and **RETRIEVE** in Section 4.)

We have already mentioned that a command of the form

```
(exit <event-name> <lemma-types>)
```

will let the user create a Boyer-Moore event by simply answering "Y" to the prompt (as illustrated by the example in the Subsection 1.4 above). Generally this form of the **EXIT** command will only be used once all the goals have been proved (i.e. their conclusions have all been reduced to **T** *(true)*). However, it is possible to create an event which records progress made during an interactive session. If there are remaining goals to prove, then upon execution of the **EXIT** command displayed above the system will first print a warning and

[13]unless, of course, changes were made by a malignant user with **SETQ** or **RPLACA** or such evilness in the top level of Lisp!

then will print an appropriate event. This event will have a conclusion of **T** (*true*), and thus the event has no logical content. It does have operational content, however: the Lisp function **RE-ENTER** may be used to get back in to the environment recorded by the exit process above.

For example, suppose that one is proving commutativity of times and has given commands which result in a state where a goal still remains to be proved. Here is the information which might be printed in such a situation, upon invocation of the **VIEW** macro command:

```
*** Active top-level hypotheses:
H1.  (NOT (ZEROP X))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (PLUS Y (TIMES Y (SUB1 X)))
       (TIMES Y X))
```

At this point, one might choose to exit the interactive loop:

```
->: (exit times-comm-progress nil)

WARNING:  The appropriate goal has not been proved.
The following event notes progress made during this (fresh) session.

(PROVE-LEMMA TIMES-COMM-PROGRESS NIL T
             ((START-GOAL (EQUAL (TIMES X Y) (TIMES Y X)))
              (INSTRUCTIONS (INDUCT 1)
                            PROVE PROMOTE
                            (DIVE 1)
                            X
                            (DIVE 2)
                            =
                            (HIDE-HYPS 2)
                            TOP)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? Y

[ 0.0 0.0 0.0 ]
TIMES-COMM-PROGRESS
```

Now one might prove a lemma which can be used to finish the proof of the goal displayed above:

```
(prove-lemma times-add1 (rewrite)
  (equal (times x (add1 y))
         (plus x (times x y))))
```

Finally, one can re-enter the previously recorded interactive environment. This is done by supplying **RE-ENTER** with the name of the interactive event displayed above:

```
(re-enter times-comm-progress)
```

At this point the user will see the prompt "->: " and can proceed with the interactive proof. It turns out that the single command **PROVE** finishes the proof. After execution of this **PROVE** command the user can exit:

```
->: (exit times-comm (rewrite))

The indicated goal has been proved.  Here is the desired event:

(PROVE-LEMMA TIMES-COMM
             (REWRITE)
             (EQUAL (TIMES X Y) (TIMES Y X))
             ((PREVIOUS-EVENT TIMES-COMM-PROGRESS)
              (INSTRUCTIONS PROVE)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? Y

[ 0.1 0.0 0.0 ]
TIMES-COMM
```

The user doesn't have to understand the meaning of the hints above, since the system prints this event and the user need only put it in his event file (or retrieve it using the Theorem Prover's Lisp function **PPE**, e.g. **(PPE 'TIMES-COMM)**). Actually, though, the hints mean what they say: the **PREVIOUS EVENT** from which the proof was continued is called **TIMES-COMM-PROGRESS**, and the only instruction needed to complete the proof was the single instruction **PROVE**.

Now in fact, the user had at least one other option in the way the proof was managed above. Instead of leaving the system the first time with the command **(EXIT TIMES-COMM-PROGRESS NIL)**, the user could have simply typed **EXIT**. Then no event would be stored in the Theorem Prover's database. The user could then submit the lemma **TIMES-ADD1** as shown above. Finally, the re-entry mechanism could have been used that was shown earlier: one simply executes **(VERIFY)**. Then one would re-enter the interactive system at the same state in which it was left (but with the new lemma **TIMES-ADD1** appropriately available), and could give the final **PROVE** command and then submit **(EXIT TIMES-COMM (REWRITE))** to conclude the proof. There is a small danger with this approach. Imagine re-playing the events created in this manner. Since the lemma **TIMES-ADD1** would precede **TIMES-COMM**, the **TIMES-ADD1** rewrite rule would be available for calls to the Theorem Prover that take place during the replay of the proof of **TIMES-COMM**. This approach could (in rare cases) cause the replay to fail, since calls to the prover which precede the place where re-entry took place were not originally made with **TIMES-ADD1** present. Such failures are likely to be rare, and the user can of course check the replayability by answering "R" to the exit prompt instead of "Y". Thus, it should not often be necessary to use the premature exit feature described above with the example **TIMES-COMM-PROGRESS**.

3. Interactive definitions

The first two sections above show how to use the interactive system to create **PROVE-LEMMA** events. In this section we explain how to use the system to do interactive proofs of termination for **DEFN** events.

The basic function that one invokes to do interactive proofs of termination is the function **VERIFY-DEFN**. The syntax is exactly the same as the syntax for **DEFN**. However, rather than calling on the Theorem Prover to prove termination, instead the user finds himself looking at the familiar "->:" prompt, indicating that he has entered the proof-checker. The goal's conclusion is simply the conjunction of one or more terms which must be proved to establish termination (and there are no hypotheses).

At this point the user simply gives whatever commands he wishes, just as if he were doing a proof begun with an invocation of **VERIFY**. The difference here is in how one exits. To exit such a proof, simply type **(EXIT T)**. This will cause the appropriate **DEFN** event to be printed on the screen, just as **(EXIT <event-name> <lemma-types>)** causes an appropriate **PROVE-LEMMA** event to be printed on the screen.

Here is an example. Suppose one has around the definition of a function **`ODDS`** which returns every other element of a list, starting with its first element:

```
(defn odds (x)
  (if (listp x)
      (if (listp (cdr x))
          (cons (car x) (odds (cddr x)))
        (list (car x)))
    nil))
```

Now the usual mergesort function is defined by recursively mergesorting two halves of a given list and then merging the results. Here is a definition of the auxiliary function **`MERGE`**.

```
(defn merge (x y)
  (if (not (listp x))
      y
    (if (not (listp y))
        x
      (if (lessp (car x) (car y))
          (cons (car x)
                (merge (cdr x) y))
        (cons (car y)
              (merge x (cdr y))))))
  ((lessp (plus (count x) (count y)))))
```

Finally, we would like to define **`MERGESORT`** as indicated above.

```
(defn mergesort (x)
  (if (listp x)
      (if (listp (cdr x))
          (merge (mergesort (odds x))
                 (mergesort (odds (cdr x))))
        x)
    nil)
  ((lessp (length x))))
```

However, this definition is rejected. Now with some thought the user can find appropriate rewrite rules to prove in order to get this definition accepted. However, the alternate approach that we are introducing here goes as follows. First, one submits the following to Lisp. Notice that it's just like the corresponding **DEFN** event except that we replace **DEFN** by **VERIFY-DEFN**:

```
(verify-defn mergesort (x)
  (if (listp x)
      (if (listp (cdr x))
          (merge (mergesort (odds x))
                 (mergesort (odds (cdr x))))
        x)
    nil)
  ((lessp (length x))))
```

At this point the user will see the usual prompt "->: " which indicates that one is talking to the interactive environment. The user can, of course, submit the (macro) command **VIEW** to see the current (in this case, the initial) conclusion and hypotheses.

```
->: view


*** Active top-level hypotheses:
There are no top-level hypotheses to display.

*** Active governors:
There are no governors to display.

The current subterm is:
(AND (IMPLIES (AND (LISTP X) (LISTP (CDR X)))
              (LESSP (LENGTH (ODDS (CDR X)))
                     (LENGTH X)))
     (IMPLIES (AND (LISTP X) (LISTP (CDR X)))
              (LESSP (LENGTH (ODDS X)) (LENGTH X))))
->:
```

The **SPLIT** command will separate the conjuncts into two separate goals. The user can then provide other appropriate proof commands in order to complete the session. Finally, one is at a state where all goals have been proved, and it is time to **EXIT**. Such an exit is indicated by the command **`(EXIT T)`**.

```
->: p

T
->: (exit t)


The indicated goal has been proved.  Here is the desired event:
(DEFN MERGESORT
      (X)
      (IF (LISTP X)
          (IF (LISTP (CDR X))
              (MERGE (MERGESORT (ODDS X))
                     (MERGESORT (ODDS (CDR X))))
              X)
          NIL)
      ((LESSP (LENGTH X)))
      (INSTRUCTIONS SPLIT
                    (CLAIM (LESSP (LENGTH (CDR X)) (LENGTH X)))
                    (CLAIM (NOT (LESSP (LENGTH (CDR X))
                                       (LENGTH (ODDS (CDR X)))))
                           0)
                    BASH
                    (CONTRADICT 4)
                    (GENERALIZE (((CDR X) Z)))
                    DROP PROVE PROVE))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? y

     Linear arithmetic, the lemmas SUB1-ADD1, CDR-LESSP, CDR-LESSEQP, CDR-CONS,
CAR-CDR-ELIM, and CDR-NLISTP, and the definitions of IMPLIES, AND, LESSP,
LENGTH, CDR, NOT, ODDS, ADD1, EQUAL, SUB1, and LISTP can be used to show that
the measure (LENGTH X) decreases according to the well-founded relation LESSP
in each recursive call.  Hence, MERGESORT is accepted under the definitional
principle.  Note that (OR (LITATOM (MERGESORT X)) (LISTP (MERGESORT X))) is a
theorem.



[ 0.2 0.0 0.1 ]
MERGESORT
```

As with an interactive **PROVE-LEMMA** event, the time triple just above is meaningless in this context, but has the usual meaning when the event is run in "batch mode".

There are some variations to the use of **VERIFY-DEFN** illustrated above. First, the relation-measure hint is optional; however, if it is omitted the the Theorem Prover will use its own heuristics to discover a relation and a measure, and only the first one it finds will be tried. (This is similar to what happens upon execution of an ordinary **DEFN** event, except that the Prover tries *all* "reasonable" relation-measure combinations when there are no such hints.) What's more, only a *single* relation-measure hint is allowed for interactive **DEFN** events. Finally, one can exit a session that began with a call of **VERIFY-DEFN** in such a manner that an interactive **PROVE-LEMMA** event is created instead. One simply exits in the usual fashion rather than with **(EXIT T)**, in which case the system behaves just as though one had begun the session with a call of **VERIFY** on the original conclusion instead. For example, in the **MERGESORT** example above, suppose that one wished to create a lemma whose statement was exactly the (conclusion of the) goal that was created upon execution of the **VERIFY-DEFN** call in that example. Then one exits in the usual way rather than with **(EXIT T)**. Here is an example where one exits before the proof is complete.

```
->: (exit mergesort-help nil)


This session began as a proof of termination for the function MERGESORT, but
you are making a lemma out of the goal which guarantees termination.  This is
allowed, but is this really what you want to do?  If so, confirm by answering
YES ; otherwise, answer NO:  yes

**NOTE** -- Now transforming this interactive session into one for PROVE-LEMMA
rather than for DEFN.

WARNING:  The appropriate goal has not been proved.
The following event notes progress made during this (fresh) session.

(PROVE-LEMMA BOO NIL T
             ((START-GOAL (AND (IMPLIES (AND (LISTP X) (LISTP (CDR X)))
                                        (LESSP (LENGTH (ODDS (CDR X)))
                                               (LENGTH X)))
                               (IMPLIES (AND (LISTP X) (LISTP (CDR X)))
                                        (LESSP (LENGTH (ODDS X))
                                               (LENGTH X)))))
              (INSTRUCTIONS SPLIT
                            (CLAIM (LESSP (LENGTH (CDR X)) (LENGTH X)))
                            (CLAIM (NOT (LESSP (LENGTH (CDR X))
                                               (LENGTH (ODDS (CDR X)))))
                                   0)
                            BASH
                            (CONTRADICT 4)
                            (GENERALIZE (((CDR X) Z)))
                            DROP PROVE)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? n
NIL
```

```
;;; Now let's see what happens when we go back into the interactive session.  Notice
;;; that when we exit we get a PROVE-LEMMA event, i.e. the transformation from
;;; DEFN to PROVE-LEMMA event is permanent.
>(verify)

->: (exit boo (rewrite))


The indicated goal has been proved.  Here is the desired event:
(PROVE-LEMMA BOO
             (REWRITE)
             (AND (IMPLIES (AND (LISTP X) (LISTP (CDR X)))
                           (LESSP (LENGTH (ODDS (CDR X)))
                                  (LENGTH X)))
                  (IMPLIES (AND (LISTP X) (LISTP (CDR X)))
                           (LESSP (LENGTH (ODDS X)) (LENGTH X))))
             ((INSTRUCTIONS SPLIT
                            (CLAIM (LESSP (LENGTH (CDR X)) (LENGTH X)))
                            (CLAIM (NOT (LESSP (LENGTH (CDR X))
                                               (LENGTH (ODDS (CDR X)))))
                                   0)
                            BASH
                            (CONTRADICT 4)
                            (GENERALIZE (((CDR X) Z)))
                            DROP PROVE PROVE)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? y

[ 0.1 0.0 0.0 ]
BOO
;;; We may now submit the definition of MERGESORT in the usual manner, i.e. for automatic
;;; proof of termination (not that we recommend this approach instead of using (EXIT T)).

>(DEFN MERGESORT
      (X)
      (IF (LISTP X)
          (IF (LISTP (CDR X))
              (MERGE (MERGESORT (ODDS X))
                     (MERGESORT (ODDS (CDR X))))
              X)
          NIL)
      ((LESSP (LENGTH X))))

     Linear arithmetic, the lemmas SUB1-ADD1 and BOO, and the definitions of
LESSP and LENGTH can be used to show that the measure (LENGTH X) decreases
according to the well-founded relation LESSP in each recursive call.  Hence,
MERGESORT is accepted under the definitional principle.  Note that:
      (OR (LITATOM (MERGESORT X))
          (LISTP (MERGESORT X)))
is a theorem.




[ 0.3 1.3 0.1 ]
MERGESORT
```

One very subtle point: For technical reasons, in the rare case that the function symbol being defined actually occurs in the initial goal upon execution of **VERIFY-DEFN**, the system will refuse to allow the user to convert this to a **PROVE-LEMMA** event.

Another useful tip: the macro command **DEFN** is a way to get a good start on the proof of the initial goal created by **VERIFY-DEFN**. This command is of course documented in the help facility.

Finally, an implementation detail which was illustrated above: Once the user has converted a session to an interactive session for **PROVE-LEMMA** rather than **DEFN**, by submitting a two-argument **EXIT** command as illustrated above, there is no going back: the change is permanent.

4. Helpful tips

When I sit down to use the system and am ready to prove a lemma, I generally see first if it can be proved automatically (possibly with appropriate hints). If the first few seconds either give virtually no output or give output that's discouraging, I'm likely to decide then to use the interactive system by submitting **(VERIFY <lemma to be proved>)**. Sometimes I'll start with an **INDUCT** command and then try **PROVE** on each of the goals thus created. Often the base case(s) will present no problem (using **PROVE** or even **S**). Then sometimes the inductive step(s) can be proved by opening up various function calls using **X**, **X-DUMB**, or **S-PROP** and then messing around. Occasionally this approach reveals that one or more rewrite rules are really called for, in which case I may exit the interactive system, prove some rewrite rules, and then go back in with **(VERIFY)**. Occasionally the proof can actually now be done automatically in the presence of these rules.

Some people may wish to use the interactive system only as a tool for finding automatic proofs. The strategy outlined above can be used to test induction strategies. That is, suppose one thinks that a theorem should be true by induction according to **`(foo x y z)`**. One might invoke **VERIFY** and then give the command **(INDUCT (FOO X Y Z))**. Better yet, one might use the macro command **PG** (which is mnemonic for "print goals") by instead submitting the command **(PG (INDUCT (FOO X Y Z)))**, which causes all the new goals to be printed out. One can then look at all the new goals and decide if they are all true. Once convinced that they are all true, the user might then try simplifying or proving each of them in turn -- or one might choose to focus on a particularly worrisome inductive step.

Here are some other random observations.

4.1 The Syntax of Commands

There is a general rule which can help the user remember whether a built-in command (rather than a macro command, though ideally this should hold true for those as well) expects an arbitrary number of arguments or a single list of arguments: commands generally expect a fixed number of arguments *unless* all arguments are of the same "type". For example, we have

```
(DIVE 2 3 1)
```

rather than **(DIVE (2 3 1))**, and

```
(PROVE (DISABLE TIMES APPEND) (INDUCT (PLUS X Y)))
```

rather than **(PROVE ((DISABLE TIMES APPEND) (INDUCT (PLUS X Y))))**, since **PROVE** expects a list all of whose members are hints. Compare though with

```
(GENERALIZE (((PLUS X Y) A)) MAIN-SUBGOAL)
```

which has two arguments, namely a list of term-variable pairs and a new goal name. Now in fact the new goal name is optional; however, for consistency one still would use **(GENERALIZE (((PLUS X Y) A)))** rather than **(GENERALIZE ((PLUS X Y) A))**.

Of course, the help facility is the final authority on the syntax of commands. However, this subsection is intended to save the user some time in some cases. One final note on this subject: the meta command **BIND** is an exception to the rule, since it takes an arbitrary number of arguments but the first is of a different type. We did this to make **BIND** have a syntax similar to Lisp's **`LET`**.

4.2 Brief Introduction to Several Useful Macro Commands

NOTE: More detailed descriptions of these commands may of course be obtained by using the help facility. We comment here on those macro commands which are mentioned at the end of the message printed upon invocation of the command **HELP**.

The command **VIEW** (equivalently, **TH**) is very handy, as it shows the hypotheses, governors, and current subterm after clearing the screen. The command **VIEW-TOP** is similar: it shows the entire conclusion instead of just the current subterm, and it highlights the current subterm.

The command **DV** is similar to **DIVE**, the difference being that **DV** is sensitive to abbreviations. This command is discussed further in Subsection 2.2, and is of course documented by the help facility. Suffice it to say here that for example, if the current subterm is **`(AND X (AND Y Z))`**, which is printed as **`(AND X Y Z)`** by the **P** command, then the command **(DV 3)** corresponds to the command **(DIVE 2 2)**; both put us at the subterm **`Z`**.

The command **PLAY** may be used to play a sequence of commands. The command **REPLAY** is similar except that it first undoes back to the start of the current session and then plays the undone commands.

The command **HYP** is explained in Section 5. Roughly, it is used to apply a given command to a given hypothesis (rather than to the current subterm).

The commands **PROVE+**, **BASH+**, **CLAIM+**, and **=+** are just like their counterparts without the "+"; the difference is that the **ENABLE**, **DISABLE**, **ENABLE-THEORY**, and **DISABLE-THEORY** commands during the current interactive session so far are taken into account in forming appropriate hints to the Prover. More information about these may be found in Subsection 4.3 below (and by using the help facility).

The command **S*** is described in Subsection 4.4 as a robust sort of simplifier.

The command **DEFN** is often useful upon entering an interactive session by way of an invocation of **VERIFY-DEFN**.

The command **PRINT-GOAL** prints the current goal (or any other goal if its name is supplied as an argument), using the *original* hypotheses and conclusion.

The command **ELIM** is used to perform elimination. For example, if one has a goal in which **`(SUB1 X)`** occurs, one may wish to use the **ELIM** to eliminate **`X`** in favor of **`(ADD1 Z)`** (for some variable **`Z`**).

The command **PRINT** may be used to pretty-print the value of an arbitrary Lisp form.

The command **SAVE** saves the current state, to be recovered by invoking **(RETRIEVE)** from Lisp.

The command **PG** is used to print out the newly created goals after invocation of some given command.

The command **TAUT** checks to see whether the current goal is essentially a tautology. (The primitive command **SPLIT** can also be used for this purpose, though unlike **TAUT** it does not expand **`ZEROP`** and **`NLISTP`**.)

The command **THEN** applies a given command and then applies another command to each of the new subgoals. For example, the command **(THEN REWRITE S)** applies the command **REWRITE** to the current goal and then applied the **S** command to each of the newly created subgoals. Another common use of **THEN** might be for calling the Theorem Prover on each subgoal created by **REWRITE** (or any other command that creates subgoals). The syntax would be **(THEN REWRITE PROVE)**. But in fact, **THEN** is allowed one argument with the second defaulting to **PROVE+** (where **PROVE+** is described in Subsection 4.3 below), and hence **(THEN REWRITE)** is acceptable syntax.

The command **WRAP** is used to put bookmarks around (complicated) invocations of macro commands. It is explained more in Section 5.

The command **UNDO+** is used to undo back through bookmarks.

4.3 Calling the Theorem Prover

The change commands which may call the Theorem Prover are **CLAIM**, **PROVE**, **BASH**, and **=** (though **BASH** calls only that part of the Prover which is called by the **DEFN** command; in particular, it doesn't call induction). In each of these cases, the Prover is called with a context that ignores the interactive session, i.e. the same context that existed at the start of the session. In particular, and **ENABLE** and **DISABLE** commands given during that session are irrelevant. Often, though, the user would expect that such commands would in fact be respected by the Prover. For that purpose we have provided the macro commands **CLAIM+**, **PROVE+**, **BASH+** and **=+**, which do respect the context provided during the interactive session. The user may wish to use these on any non-trivial calls to the Prover. More on these commands may be found in Section 5.

4.4 Simplification: S and S-PROP

In this subsection we comment briefly on the simplification commands available in this system. More detailed descriptions can of course be found using the help facility. These commands are used for simplifying the current subterm.

Let us begin by giving a quick summary of the different variants of the **S** command. The **S** and **(S ALL)** commands simplify the current subterm by expanding all nonrecursive function symbols and doing unconditional rewriting, i.e. rewriting when there are no hypotheses to relieve (except ones that are sufficiently trivial relative to the current active hypotheses). The difference between them is that **S** only uses *primitive* rules, i.e. ones whose function symbols all come from **(BOOT-STRAP)** or from an invocation of the Theorem Prover's shell principle. **(S LEMMAS)** is like **(S ALL)**, except that it does not open up nonrecursive function symbols, which makes it quite useful for the application of rewrite rules which contain nonrecursive function symbols. (Recall that the Theorem Prover will usually not apply rewrite rules whose left-hand sides contain function symbols unless these symbols are all disabled.) **S** may also be given a numeric argument, which is like **(S LEMMAS)** except that the argument specifies the backchaining depth (which is changed by 1 each time one dives in to rewrite a hypotheses of a rewrite rule).

On the other hand, **S-PROP** is merely a request to perform propositional simplification. Unlike **S**, **S-PROP** never expands nonrecursive function symbols and never applies rewrite rules. Also unlike the **S** command (with or without arguments), the **S-PROP** command normalizes expressions in the sense that **`IF`** is pushed to the outside. To be precise, after applying **S-PROP** it will be the case that for every subterm of the current subterm which has the form **`(IF x y z)`**, there is no occurrence of the function symbol **`IF`** in the first argument **`x`** of that subterm. This movement of **`IF`**-expression to the outside is often quite helpful. For example, consider the term:

```
(EQUAL (FIX X) (IF (ZEROP X) 0 X))
```

To prove this term, one might give the command **S**; but this only results in the term

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

which however can then be proved with **S-PROP**. If instead **S-PROP** is done first, one obtains the simpler term

```
(IF (ZEROP X)
    (EQUAL (FIX X) 0)
    (EQUAL (FIX X) X))
```

which in fact can be proved using **S**.[14]

There are times when one wants to expand all calls of one or more function symbols without having to move to the individual relevant subterms first. In this case the command **S-PROP** with arguments (as explained by the **HELP-LONG** command) is useful. Finally, use **(S-PROP NIL)** when *all* you want to do is to normalize **`IF`**-expressions, in the sense described above.

4.5 Case Splitting

A common strategy in all sorts of proof methodologies is the splitting of a goal into cases. One can do this with **(CLAIM <term> 0)**, where **`<term>`** is the term on which one want to case (according to whether or not it is **F**) and the argument **`0`** indicates that the theorem-prover should not attempt to prove **`<term>`**. The current goal will then have **`<term>`** as an additional hypothesis, but a new goal will be created which is identical to the current goal except that instead **`(NOT <term>)`** is added as a hypothesis. Moreover, the current goal now depends on the new goal (which in turn has no dependents). (Incidentally, the macro

[14]In fact, other commands such as **PROVE** and the macro command **S*** can be used to prove the original goal; this example was chosen merely to illustrate the differences between **S** and **S-PROP**.

command **CLAIM0** is useful when the claimed **`<term>`** follows from the current hypotheses, as explained below.)

Another way to split into cases is to use the **SPLIT** command. This command takes the current goal and replaces it by cases based on its propositional structure; see the help facility for details. Note that it may be a good idea to do **(S-PROP NIL)** just before **SPLIT** in order to bring the propositional structure to the top first.

4.6 Proving Implications

There are at least two basic approaches to proving terms of the form **`(IMPLIES <hyp> <conc>)`**. The preferability of one or the other approach may well be mainly a matter of taste. One approach is to begin with the command **PROMOTE**, which adds **`<hyp>`** to the hypotheses (after flattening its **`AND`** structure, if it is an **`AND`** expression). The other approach is to invoke **S** or **S-PROP** or **S***, which will create a (possibly rather large) **`IF`** tree and then **DIVE**, which when given no arguments will put you at a non-**`T`** branch of the **`IF`**-tree (if there is one). Often the two approaches are equivalent except that the second approach makes the assumptions governors rather than top-level hypotheses. I suppose that the first approach has the advantage of keeping the hypotheses out of the way (at the top level) and eliminating the need for **DIVE**, while the second approach has the advantage of allowing you to do simplification in the assumptions without doing the contortions involving **DEMOTE** that are illustrated in Subsection 4.11 below. Usually I prefer the former approach, but again, that's probably just a matter of taste.

However, if there are more than a couple of cases implicit (or explicit) in the implication, I recommend the use of **SPLIT** (which is described briefly in the paragraph before last). Or, one could use **BASH**, which invokes the Theorem Prover's rewriter and built-in linear arithmetic.

4.7 Substitution

A lot of the work in an interactive proof may involve substituting equals for equals. Often this is accomplished by simplification or by expanding function calls (**X**, **X-DUMB**, **S**, **S-PROP**). However, when the underlying equality seems fairly straightforward but requires some proof, it may be worthwhile using the **=** command. For example, suppose that the current term has the form **`(IF <test> <branch1> <branch2>)`**, where the user sees that **`<test>`** equals **`T`**. Although one might be tempted to expand the outermost function call of **`<test>`** and to do various other low-level operations, there may be an easier way: simply submit the command **(DIVE 1)** (or equivalently, just **1**) in order to make **`<test>`** the current term, and then submit the command **(= T)**. If the Theorem Prover is able to prove this equality, then the substitution will be made.[15] Then of course one can submit **UP** followed by **S** or **S-PROP** to replace **`(IF T <branch1> <branch2>)`** with simply **`<branch1>`** (or a simplified version thereof).

4.8 Managing Goals

Suppose that one is trying to prove a rather complicated goal. It may be that this amounts to proving that various subterms of the conclusion are equal to **`T`**. Perhaps one or two of these subterms seem(s) difficult to simplify. Then it may be helpful for bookkeeping purposes to use the **PUSH** command. Roughly speaking, the **PUSH** command replaces the current subterm by **`T`** and creates a new goal to prove the current subterm (under the current active hypotheses and governors). In this manner one can prove the goal that was under consideration without getting lost in details, and then go prove the goals that were **PUSH**ed in the process. Of course, one may further **PUSH** subgoals of the new goals, and so on.

A similar modularization is accomplished by using versions of **CLAIM**, **CLAIM0** (see "Bringing in Useful Information" below) and **=** which allow one to postpone proof attempts; see the help facility. The **REWRITE** command is also useful for proof control, and in particular for controlling backward chaining via application of conditional rewrite rules.

When a "significant" goal is proved by pushing difficult subgoals, you may wish to insert a **BOOKMARK** instruction to make it easy to undo back to the point at which the difficult goal's proof was begun (in case the proofs of the messy subgoals get gnarled up). One can also insert comments using the **COMMENT** command.

4.9 Exiting Temporarily

Suppose that you wish to exit an interactive session without creating an event but with a record of what you have done. One way to do this is to type the form **(PRINT-INSTRS)** either to Lisp or to the "->: " prompt; this will give you a list of all change commands stored during the current session. Alternatively, one can submit the command **(EXIT <any name> NIL)** and then answer **`N`** to the query about adding an event. One can then grab this "event" from a buffer[16], and then run it another time followed by **(UBT)** (so that you don't actually create an event) and then **(VERIFY)** (in order to resume the proof).[17] Alternatively, you can just enter the

---

[15] If not, then one could use the **PUSH** command described in the following paragraph, or else one could use a different form of the **=** command, namely **(= * T 0)** -- see the help facility for details.

[16] at least, if you're running the system under an editor such as EMACS or if you're "forking" or "dribbling" the output to an edit buffer

[17] An obscure but sometimes helpful fact is that the global Lisp variable **`EV0`** stores the most recent **`PROVE-LEMMA`** expression printed in response to the user's answer to the query caused by the **EXIT** command.

interactive system later with **(VERIFY <term>)** and submit the macro command **(PLAY (<command$_1$> ... <command$_k$>))**, where these **<command$_i$>** are the change commands that were put in the **INSTRUCTIONS** hint (equivalently, these are the commands printed by **(PRINT-INSTRS)**.)

What should you do if you exit the interactive system, prove some rewrite lemmas, and then go back in where you were? (By the way, this is often an excellent approach when you are confronted with a goal which follows from a relatively simple and general fact.) The issue here is that if the interactive session had involved calls to the theorem-prover, those calls may not work later in the batch re-play of events because of the presence of these rewrite rules. As mentioned in Section 2, one can create events noting the progress of an interactive session. While that solves this problem, it probably isn't necessary in a majority of cases: usually the added rewrite rules won't cause a problem. If you suspect that there isn't going to be a problem but want to make sure, the command **REPLAY** will play back all the instructions and let you know if one or more of them didn't work. Alternatively, you can finish the interactive proof and then answer **R** ("replay") to the query regarding adding the event.

Sometimes one wants to quit in the middle of an interactive proof, then perform one or more other interactive proofs, and then finally return to the one that was suspended. The macro command **SAVE** saves the current state (i.e. the current *state stack*) so that when the user later submits the command **(RETRIEVE)** to Lisp, he re-enters that suspended proof. Thus, **(RETRIEVE)** is just like **(VERIFY)**, except that one re-enters that suspended interactive proof rather than merely the most recent interactive proof. More generally, the user can name these saved proofs. So for example, suppose that one is trying to prove a certain lemma, which (say) he intends to name **LEMMA-A**. Perhaps in the course of interactively proving this lemma, he sees that he wants to prove a certain rewrite rule; let us suppose that he intends to call this lemma **LEMMA-A-1**. Then he can submit the command **(SAVE LEMMA-A)**[18] in the interactive proof of **LEMMA-A**, and then begin an interactive proof of **LEMMA-A-1** by submitting **(VERIFY <term>)** to Lisp. In the course of this proof he may see that he wants to prove a subsidiary lemma, call it **LEMMA-A-1-1**. So he might want to submit **(SAVE LEMMA-A-1)** to the "->:" prompt and then call **VERIFY** on the new term to be proved. When that proof is completed, he could then submit **(RETRIEVE LEMMA-A-1)** to Lisp, and then finish the proof of **LEMMA-A-1**. Finally, he could then submit **(RETRIEVE LEMMA-A)** to Lisp, and finish the proof of **LEMMA-A**. For what it's worth, **SAVE** is equivalent to **(SAVE TEMP-SS)** and **(RETRIEVE)** to **(RETRIEVE TEMP-SS)**.

---

[18]Or indeed, any command of the fom **(SAVE <name>)** -- though one may as well use the name that one intends to assign to the lemma when it is proved.

4.10 Bringing in Useful Information

Suppose that you are working on a goal and you need a certain fact. How can you make that fact explicit? There are a few ways. One is to **CLAIM** it, i.e. submit **(CLAIM <fact>)**. This will invoke the theorem-prover. If the proof fails then you may wish to submit the macro command **(CLAIM0 <fact>)**, which will create a new subgoal to prove **<fact>** from the currently active hypotheses. (If you for some reason also want the negation of the current conclusion to be a hypothesis of the subgoal, use **(CLAIM <fact> 0)** instead.)

If the fact is an instance of a lemma in the chronology, you may use the command **USE-LEMMA** or the macro command **USE**. These commands are of course described in the help facility.

4.11 Simplifying Hypotheses

Suppose that during the course of an interactive proof, one has a situation in which one wants to simplify (or make any sort of substitution in) a hypothesis. Now this system is designed for working on the *conclusion* of the current goal, and especially on the current subterm of that conclusion; so what can one do? One approach is to use **(CLAIM0 <new_version>)** to add the "new version" of the hypothesis to the hypothesis list, and then prove the correctness of this operation later (by proving the resulting subgoal). Here we describe another approach.

A simple approach is to shift a hypothesis from the hypothesis list into the conclusion, by making it the antecedent for an implication concluding with the current conclusion; one then may manipulate the hypothesis, and then move it back. (In fact there is a macro command **HYP** which uses **CONTRADICT** for this purpose; it is described in Section 5. However, let us proceed with a manual approach based on **DEMOTE**.) Here's an example. (We'll denote the current subterm as "`<some_term>`" since it's not important what it actually is.)

```
->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)
H2.  (EQUAL (PLUS 0 X) Y)

*** Active governors:
There are no governors to display.

The current subterm is:
<some_term>

->: (demote 2)
```

```
->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (EQUAL (PLUS 0 X) Y)
         <some_term>)

->: 1

->: p
(EQUAL (PLUS 0 X) Y)

->: s

->: p
(EQUAL X Y)

->: top

->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (EQUAL X Y) <some_term>)

->: promote

->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)
H2.  (EQUAL Y Z)

*** Active governors:
There are no governors to display.

The current subterm is:
<some_term>
```

So now the hypothesis **`(EQUAL (PLUS 0 X) Y)`** has been replaced by the simpler hypothesis **`(EQUAL X Y)`**, and we can proceed with the proof however we choose to.

4.12 Pesky Abbreviations

As things stand currently, the functions **`LEQ`**, **`GREATERP`**, and **`GEQ`** are immediately expanded upon translation. This means that they will never be seen except as user input. In fact, they won't even appear in events which have been created by **EXIT**, even if you originally entered **VERIFY** with a term having such function symbols in it. Now of course you can edit this event before making it part of your events file so that the original term is the body of the event, and the proof should replay in batch mode just fine, but admittedly that's a nuisance. Of course, you can just leave the event so that your pretty **`(LEQ A B)`** has been replaced by **`(IF (LESSP B A) F T)`** in the body of your lemma, and that will work too. Sorry about the pair of undesirable choices.

Perhaps the best choice is to avoid these symbols completely, as is quite a common practice for many serious users of the Theorem Prover. So for example, to say that **`x`** $\geq$ **`y`** we say **`(NOT (LESSP X Y))`** rather than **`(GEQ X Y)`**.

5. Macro commands and the top level loop

The system described above may well be adequate for many users' needs. However, the system is in fact extensible by way of a facility called *macro commands*. The idea behind these commands is that a single macro command ultimately generates zero or more "real" commands, i.e. change, help, and meta commands. Several macro commands are currently provided by the system and have been described above, for example **VIEW**, **CLAIM0**, **DV**, **PLAY**, and **PROVE+**. However, a mechanism exists for adding additional macro commands (and even for redefining the given ones). The soundness of the system is not affected by macro commands, in the following sense: macro commands are ignored by **PROVE-LEMMA**. That is, a user may give macro commands freely during an interactive session, but in order to be secure in the correctness of the proofs constructed, the resulting events should be run through once more. The **`INSTRUCTIONS`** hints to **PROVE-LEMMA** events should contain no macro commands, since these are all "expanded out" during the interactive proof session. But even if a malicious user tries to sneak some such commands in to the **`INSTRUCTIONS`**, no harm will arise: the system will simply complain and fail in the proof, since only *change* commands are recognized by **PROVE-LEMMA**.

This section is organized into four subsections, the first two of which should suffice for most any user. We begin by giving an introduction to the *use* of macro commands by describing informally how to use some of the predefined macro commands. The second section is an introduction, by way of example, to the writing of simple macro commands. The third subsection gives a detailed definition of the execution of the top-level loop. We conclude with examples which more thoroughly illustrate the *writing* of macro commands.

5.1 How to use macro commands: some examples

Suppose that you wish to call the Theorem Prover with the **PROVE** command, but you have already disabled or enabled various functions and rewrite rules during the current interactive session (with the **DISABLE** and **ENABLE** commands). Perhaps you want to call the Prover with the corresponding "environment". For example, perhaps you have begun the current session with the commands **(DISABLE TIMES-COMMUTATIVITY)** and **(ENABLE APPEND)**, and you are ready to call the Prover. But suppose for some reason that you want **`TIMES-COMMUTATIVITY`** disabled and **`APPEND`** enabled (even though they are presumably the other way around, say, in the global Theorem Prover state). Now one way to do this, as explained by invoking **(HELP PROVE)**, is to submit the command

```
(PROVE (DISABLE TIMES-COMMUTATIVITY) (ENABLE APPEND)) .
```

However, there is a better way. The macro command **PROVE+** has been defined to accomplish the same effect. Here is the documentation provided by submitting **(HELP PROVE+)**:

```
->: (help prove+)

Macro Command  [Use HELP-LONG to see the definition]
(PROVE+ &REST HINTS)
Call the theorem prover with hints that match the local
disable-enable environment in the current interactive
session, as updated (optionally) by the HINTS argument.
Notice that this command has the same syntax as PROVE,
e.g. (PROVE (DISABLE PLUS TIMES)).  This command
"succeeds" if and only if the corresponding PROVE
command "succeeds", i.e. the proof completes.
```

Thus, in the example above, the command **PROVE+** will actually generate the command displayed above, namely **(PROVE (DISABLE TIMES-COMMUTATIVITY) (ENABLE APPEND))**. This **PROVE** command is then the one that is actually executed by the system: if the proof is successful, a new state will be pushed onto the state stack and its **`INSTRUCTION`** field will be the particular **PROVE** command shown above, while if the proof is not successful, there will be no change in the state stack.

The information provided above by **(HELP PROVE+)** also mentions an optional argument **`HINTS`**, which may be provided using essentially the same syntax as for the **PROVE** command. Consider then the example above but where we instead give the command

```
(PROVE+ (DISABLE PLUS DIFFERENCE)
        (USE (PLUS (X A))))
```

In this case, the command generated would be

```
(PROVE (USE (PLUS (X A)))
       (DISABLE TIMES-COMMUTATIVITY PLUS DIFFERENCE)
       (ENABLE APPEND))
```

and this would be the one that is actually executed.

Let us consider another example. Suppose that hypothesis number 2 is of the form **(IF T X Y)**, which one would like to simplify to **X**. Now the command **S** only works on the conclusion; hence one approach is to move that hypothesis into the conclusion somehow, then simplify it there, and then move it back into its former position. The following sequence of commands accomplishes this task. (We provide some comments in italics.)

(CONTRADICT 2) *{Swap* **<hyp>** *hypothesis number 2, with the conclusion}*
(DIVE 1) *{Dive to the first argument of* **(NOT <hyp>)}**
S *{Simplify}*
TOP *{Move to the top of the conclusion}*
(CONTRADICT 2) *{Swap back the second hypothesis with the conclusion}*

As this sequence of commands is really independent of the contents of the second hypothesis and the conclusion, one can imagine a macro command that generates this sequence of commands.[19] Such a command is in fact a predefined macro command called **HYP**. Here is its description, as provided by the help facility:

```
->: (help hyp)

Macro Command  [Use HELP-LONG to see the definition]
(HYP N INSTR)
Applies instruction INSTR to hypothesis number N.  If
this "fails", then there is no change made to the state
stack, the command "fails", and a message to that
effect is printed; otherwise it "succeeds".
```

We'll talk about "success" and "failure" in Subsection 5.3; roughly, the submission of any command (whether change, meta, help, or macro) either "succeeds" or "fails". We always put these notions of "success" and "failure" in double-quotes, in order to emphasize that they are technical terms. In the case of change commands, "success" means creation of a new state to push on top of the state stack. For the other commands, the **HELP-LONG** facility gives the definitions of "success". At any rate, the sequence of commands given above (for simplifying hypothesis number 2) is in fact generated by the invocation of: **(HYP 2 S)**.

Now suppose that one gives the command **(HYP 2 S)** in the example above, but then decides that one wants to undo this command. The command **UNDO** is inadequate, since in fact five actual change commands have been given, as can be seen by giving the command **COMMANDS** or **COMM**. Thus the command **(UNDO 5)** would have the desired effect. However, it's not always convenient to look through the commands, and in fact doing so doesn't *always* make it clear just how far back one wants to undo. For this purpose it is handy to use the predefined macro command **WRAP** to put bookmarks around the use of a macro command when that command generates several change commands. Here is the information provided by the help facility:

---

[19] Actually, a macro command generates a single command, but there are several *meta* commands to use as sequencers. For example, the above sequence could be put in the single command **(DO-ALL (CONTRADICT 2) (DIVE 1) S TOP (CONTRADICT 2))**.

```
->: (help wrap)

Macro Command  [Use HELP-LONG to see the definition]
(WRAP NAME INSTR)
The main difference between (WRAP instr) and instr is
that in case of "success", the former will insert two
bookmarks:  (BOOKMARK (BEGIN name)) before the first
new primitive command put on the state stack, and
(BOOKMARK (END name)) after the last.  "Success" is the
same for both, however.  One other difference is that
if instr does not "succeed", then all state is restored
to what it was before running this command (in the
sense of PROTECT).  Use WRAP+ instead if you don't want
this restoration to take place in case of failure.
```

If one submits the command **(WRAP (HYP 2 S))** in place of **(HYP 2 S)** in the example above, one obtains the same five change commands together with **(BOOKMARK (BEGIN HYP))** preceding these five and **(BOOKMARK (END HYP))** following these five. Now if one wants to undo these commands, one can submit either **(UNDO 7)** or, and here's the main point of using **WRAP**, submit the command **(UNDO (BEGIN HYP))**. One final note regarding this use of **UNDO**, however: when using this (bookmark) form of **UNDO**, the undoing only goes back *up to* (and not including) the given bookmark; hence the atomic command **UNDO** still needs to be given. But have no fear -- there is also a predefined macro command **UNDO+** which undoes up to *and including* a specified bookmark!

The examples above are intended to give enough of an introduction to the use of macro commands so that the user can use the predefined macro commands with ease. One other useful thing to know is that the global Lisp variable **`BASIC-MACRO-COMMAND-NAMES`** stores a list of names of particularly useful macro commands, essentially the ones discussed in Section 4 above. (The user can of course use Lisp to set this variable to anything he chooses.) This variable is used by the help facility: unlike **HELP-LONG**, the output from **HELP** does not mention any macro commands other than these. Several of these are very useful right from the start, for example **VIEW** (which has been discussed already in this manual).

5.2 Writing macro commands: an introduction by way of examples

Let us begin with a very simple example. Suppose that you get tired of typing **(S LEMMAS)**, as I did. The following definition takes care of this annoyance:

```
(DEFINSTR SL () (S LEMMAS)
  (|Same| |as| (S LEMMAS) |.|))
```

This is a definition of the macro command **SL**, using the **`DEFINSTR`** facility. The second argument to **`DEFINSTR`**, namely **`()`** (i.e. **`NIL`**), is the list of formal arguments for the command (in this case, none). Next comes the body of the definition, in this case **`(S LEMMAS)`**. Finally, the documentation appears as a list of Lisp objects, with vertical bars enclosing those objects which contain lower case characters or punctuation marks.

When this instruction is submitted, the effect is as though the instruction **SL** is simply replaced by the instruction **(S LEMMAS)**, which is then executed.

Let us consider a slightly more interesting example.

```
(DEFINSTR INDUCT-PROMOTE (&OPTIONAL ARG)
  (THEN (IF (QUOTE ARG) (INDUCT ARG) INDUCT)
        (SUCCEED (REPEAT PROMOTE)))
  (|Runs| |the| INDUCT |command| |,| |and| |if| |that| |"succeeds"|
          |then| |this| |command| |"succeeds"| |too| |,| |and| |PROMOTEs|
          |all| |hypotheses| |on| |all| |of| |the| |subgoals| |generated| |.|))
```

As suggested by the documentation, the command **INDUCT-PROMOTE** is intended to be just like the **INDUCT** command except that hypotheses are promoted in the resulting subgoals. This time there is one argument, named **ARG** in fact, but it is optional (as indicated by the keyword **&OPTIONAL**, which indicates that all subsequent arguments in the list are optional). The way optional arguments work is that, in the Lisp tradition, they are assigned the value **NIL** when they are omitted. Thus for example, the command **(INDUCT-PROMOTE)** (or, equivalently, **INDUCT-PROMOTE**) is equivalent to the command **(INDUCT-PROMOTE NIL)**. Now let us analyze the body of this **DEFINSTR** definition. Here are the descriptions of the auxiliary commands, as provided by the **HELP** facility.

```
->: (help then)

Macro Command  [Use HELP-LONG to see the definition]
(THEN INSTR &OPTIONAL COMPLETION)
If the given instruction INSTR "succeeds", then the instruction COMPLETION is
run on each of the new subgoals of the original goal (if any).  If no
COMPLETION is given, then PROVE+ is used as the COMPLETION.  This command
"succeeds" if INSTR "succeeds" and each resulting invocation of COMPLETION
"succeeds".
->: (help if)

Macro Command  [Use HELP-LONG to see the definition]
(IF TEST INSTR1 INSTR2)
Evaluate the TEST in Lisp.  Then execute INSTR1 if the result is not  NIL  and
INSTR2 otherwise.
     OBSCURE NOTES:   {We omit the rest of this explanation of IF.}
->: (help succeed)

     (SUCCEED instr):  This instruction is only of interest to writers of
macro commands -- execute (HELP-LONG SUCCEED) if you really care.
->: (help repeat)

Macro Command  [Use HELP-LONG to see the definition]
(REPEAT INSTR)
Runs INSTR, stopping the first time it "fails".
->:
```

Suppose now that the user types in **(INDUCT-PROMOTE (FOO X Y))**. Then one imagines that the argument **(FOO X Y)** is textually substituted for the argument **ARG** into the body of the definition of **INDUCT-PROMOTE**. (The mechanism is not exactly this, but is very close; see the discussion of **DEFINE-MACRO-COMMAND** below.) With this evaluation mechanism in mind, let us explain the body of the definition of **INDUCT-PROMOTE**. The code **(IF 'ARG (INDUCT ARG) INDUCT)** may be understood as follows. The command **IF**, which is documented above, evaluates its first argument in Lisp. The reason for the **QUOTE** is that if it is not there, then the actual argument supplied to the call of **IF** will be evaluated in Lisp, and hence will cause an error if it is an unbound symbol, say. Since **(QUOTE NIL)** is **NIL**, we can use **(QUOTE ARG)** to test whether or not **ARG** is **NIL**. The evaluator returns the command **(INDUCT ARG)** or simply **INDUCT**, according to whether or not this argument is **NIL**, which is appropriate since it is indeed considered to be **NIL** if it is not supplied at all, as in the command **(INDUCT-PROMOTE)** or in the equivalent command **INDUCT-PROMOTE**. Next, the call of the macro command **THEN** in the body of the definition of **INDUCT-PROMOTE** will guarantee that the instruction **(SUCCEED (REPEAT PROMOTE))** will be run on each of the subgoals; exactly how this is accomplished here isn't particularly important, but the reader is invited to invoke **(HELP-LONG THEN)** to see the definition of **THEN**. Now the call of **SUCCEED** can actually be omitted; it's there only to guarantee that the "success" of the entire command is independent of the "success" of the invocations of **(REPEAT PROMOTE)**. As for **(REPEAT PROMOTE)**, the definition of **REPEAT** shows that this simply causes **PROMOTE** to be repeated until it no longer "succeeds", i.e. until the conclusion is no longer of the form **(IMPLIES x y)**.

Let us now consider the other mechanism for defining macro commands, which is the **DEFINE-MACRO-COMMAND** facility. The aforementioned **DEFINSTR** facility should be adequate for most users, but for complicated macro command definitions the **DEFINE-MACRO-COMMAND** facility is perhaps more appropriate. Also, this facility allows for the definition of macro commands which accept an arbitrary number of arguments. For this discussion we assume that the reader is familiar with the notion of *macros* in Lisp; at least, such familiarity might well be helpful for understanding what follows, especially with respect to backquote and comma notation.

We begin by revisiting the example **SL** above. In terms of the **DEFINE-MACRO-COMMAND** facility, the definition of **SL** could be made as follows:

```
(DEFINE-MACRO-COMMAND SL () '(S LEMMAS)
       (|Same| |as| (S LEMMAS) |.|))
```

Notice the single quote, i.e. notice that the body is **(QUOTE (S LEMMAS))** this time rather than simply **(S LEMMAS)** as was the case when using **DEFINSTR**. The macro-command evaluation mechanism proceeds as follows in this case. The **SL** command causes a call of its body, namely of **(QUOTE (S LEMMAS))**, in Lisp. The result of evaluating this call to Lisp is returned as the new command to be evaluated, which in this case is the command **(S LEMMAS)**.

Now let us consider the second example considered above, namely **INDUCT-PROMOTE**, but with **DEFINE-MACRO-COMMAND** used rather than **DEFINSTR**.

```
(DEFINE-MACRO-COMMAND INDUCT-PROMOTE (&OPTIONAL ARG)
  '(THEN ,(IF ARG (LIST 'INDUCT ARG) 'INDUCT)
         (SUCCEED (REPEAT PROMOTE)))
  (|Runs| |the| INDUCT |command| |,| |and| |if| |that| |"succeeds"|
          |then| |this| |command| |"succeeds"| |too| |,| |and| |PROMOTEs|
          |all| |hypotheses| |on| |all| |of| |the| |subgoals| |generated| |.|))
```

Notice that if one considers this as a normal Lisp definition, and if one considers the instruction **(INDUCT-PROMOTE (FOO X Y))** as generating a call of this Lisp function on the argument **(FOO X Y)**, then what is returned is the command

```
(THEN (INDUCT (FOO X Y))
      (SUCCEED (REPEAT PROMOTE)))
```

which is just what was returned in the **DEFINSTR** case when the argument **(FOO X Y)** was textually substituted for **ARG** into the body of the definition.[20] In fact, **DEFINSTR** is actually implemented essentially by turning it into a **DEFINE-MACRO-COMMAND** form with the appropriate backquotes and commas.

The syntax for **DEFINE-MACRO-COMMAND** is that of an arbitrary Lisp **DEFUN** form, except of course for the (optional) documentation argument for the former, which comes at the end. In particular, both **&OPTIONAL** and **&REST** forms are allowed.

5.3 The top-level loop

Roughly speaking, the top-level loop has the following operational-style semantics. First, the instruction is checked to see if it is a call of a macro command, and if so, it is repeatedly expanded until the result is no longer a call of a macro command. Now the resulting instruction is executed. If the resulting command is a meta command then its execution may in fact result in the execution of any number of other commands (of any kinds); this sequencing is controlled by notions of "success" and "failure". Let us now be more precise.

It is convenient to refer to the main Lisp procedures by their names. The key ones here are called **INTERACTIVE-SINGLE-STEP** and **READ-INSTR**. Here are their specifications.[21] The discussion below can also be viewed as documentation for the code.

---

[20] In fact, the **DEFINE-MACRO-COMMAND** form above really defines a Lisp function ***PC-MACRO*-INDUCT-PROMOTE** with exactly this definition, which is called on the (unevaluated) arguments of the user's command. What is returned is supposed to be a new command, such as the **THEN** form displayed above, which is then considered as though it were the form originally submitted.

[21] We omit details concerning the "identification" of atomic instructions with instructions that are lists of length 1, e.g. <cmd> vs. (<cmd>).

READ-INSTR takes a (potential) instruction, which might be a call of a macro command, and returns either a call of a primitive (non-macro) command or else returns **NIL** (indicating that the instruction did not expand into a call of a primitive command). In the latter case, explanatory comments are printed to the screen.

More precisely, **READ-INSTR** is defined as follows. First, for each argument **<arg$_i$>** of the potential instruction **(<cmd> <arg$_1$> ... <arg$_n$>)**, if that argument has the form **(! x)** then replace it by the result of evaluating **x** in Lisp. More precisely, there is a global variable called **COMMAND-BINDINGS** which contains a list of pairs each of the form **<variable, S-expression>**, and the form **x** is evaluated in this environment.[22] Even in the case that **READ-INSTR** is applied recursively (for macro commands, as explained below), arguments of the form **(! x)** are always evaluated in the sense above. In a nutshell: **READ-INSTR** causes evaluation of the "(! x)" arguments of the given instruction as explained above, and then returns an instruction as follows:

- If the instruction is a positive integer **N**, return **(DIVE N)**.
- If the instruction is a call of a macro command but the arity of the macro command does not match the number of arguments, cause an error.
- If the instruction is a call of a macro command in which the arity *does* match the number of arguments, then evaluate the call in the sense described in Subsection 5.4 below. Then, recursively apply **READ-INSTR** to the result.
- If the instruction is a call of a primitive command (either a change, meta, help or exit command), then return the instruction.
- Otherwise, return **NIL** (and print to the screen that the given instruction is not a valid instruction).

INTERACTIVE-SINGLE-STEP is evaluated for side-effect, especially to the **STATE-STACK**. It takes an arbitrary S-expression and immediately applies **READ-INSTR** to that form. If the result of **READ-INSTR** is **NIL**, then nothing further happens (and **READ-INSTR** is responsible for printing helpful information to the screen). Otherwise, **READ-INSTR** returns a form which is a call of a legitimate command (and not a macro command). What happens next depends on the kind of command.

- If the instruction is a call of a change command, then the current state is fed to an appropriate function, named **CHANGE-STATE-WITH-<command_name>**. That function either returns a new state, which is then pushed on top of the **STATE-STACK**, or else returns **NIL** (in which case the state stack is not changed).
- If the instruction is a call of a help command, then an appropriate function (named **PRINT-HELP-WITH-<command_name>**) is called to print information to the screen.
- If the instruction is a call of a meta command, then the function **CHANGE-STATE-STACK-WITH-<command_name>** is called. That function may itself call **INTERACTIVE-SINGLE-STEP**. For example, the meta command **DO-ALL** invokes the

[22]The meta command **BIND** modifies this environment.

function **CHANGE-STATE-STACK-WITH-DO-ALL**, which in turn sequentially calls **INTERACTIVE-SINGLE-STEP** on each instruction in its list of arguments.

- If the instruction is **EXIT**, the top-level loop is exited.
- If the instruction is **(EXIT <name> <lemma-types>)** or **(EXIT T)**, the top-level loop is exited in the manner explained in the example in Sections 1 and 3, i.e. with a query to the user regarding creation of an event.
- Otherwise, the instruction is not legitimate, and there is no change in the **STATE-STACK**.

An important consideration was left out of the specifications above. Each call of **INTERACTIVE-SINGLE-STEP** returns a value which we think of as denoting "success" or "failure", according to whether that value is non-**NIL** or is **NIL** (respectively). For change commands, "success" occurs (i.e. a non-**NIL** value is returned) if and only if a new state is indeed pushed on to the state stack. (The user can tell if this is the case by checking, using the instruction **COMM** or **COMMANDS**, whether or not the new instruction was stored.) There is no specification of "success" and "failure" for help commands. For calls of meta and macro commands, the "success" or "failure" depends on the particular command (as specified by the **HELP-LONG** facility).[23] Various meta and macro commands cause multiple calls to **INTERACTIVE-SINGLE-STEP** which are however guarded by the "success" or "failure" of subcalls. Another important use of the "success" notion is made by the meta command **PROTECT**, where **(PROTECT <instruction>)** runs the given instruction, but reverts the system to the global state existing at call time in case that instruction fails (by restoring the values of the Lisp variables **STATE-STACK** and **OLD-SS**).

5.4 Defining macro commands

In this section we formally present the syntax for defining macro commands. In fact there are two such definition mechanisms, both illustrated in Subsection 5.2, and we explain the evaluation of macro commands (which was postponed to now in the discussion of **READ-INSTR** above) here as well. The ideas are presented with some examples.

The syntaxes for defining macro commands are:

```
(DEFINSTR <command_name> <formals> <body> <documentation>) and

(DEFINE-MACRO-COMMAND <command_name> <formals> <body> <documentation>)
```

where **<documentation>** is optional. The arguments to **DEFINSTR** and to **DEFINE-MACRO-COMMAND** may be described briefly as follows:

---

[23]As "success" and "failure" are considered "advanced" notions, the long help must generally be used to get such information about meta commands.

- **<command_name>** is the name that one wishes to assign to the command (e.g. **PROVE+**). It may not be the name of a primitive (i.e. change, meta, help, or **EXIT**) command.
- **<formals>** is the list of arguments to the macro command. For **DEFINSTR**, this argument should be a list of distinct symbols, one of which may be the symbol **&OPTIONAL**. For **DEFINE-MACRO-COMMAND**, this argument should be of the same form as the arguments to the Lisp function **DEFUN**.
- **<body>** is an arbitrary S-expression. However, only certain forms will make sense here; what they should look like will be apparent below when we describe the evaluation mechanism.
- **<documentation>** is a list of atoms to be printed by the help facility (both **HELP** and **HELP-LONG**). This printing is done with code written by Boyer and Moore, and certain conventions apply. The atom **CR** denotes a carriage return (with line feed), while **|#|** denotes a tab. Lower-case words and punctuation need to be enclosed in vertical bars. Finally, this printing routine is intelligent about line length and punctuation. The examples below will make all of this clearer.

The most recent definition of **<command_name>** may be removed by invoking the following.

```
(UNDEFINSTR <command_name>)
```

Let us consider an example which illustrates the evaluation mechanism. We begin with the definition of a simple macro command which changes goals except when its argument is already the current goal. Explanation follows.

```
(DEFINSTR MAYBE-CHANGE-GOAL (NAME)
  (IF (EQ (QUOTE NAME) (GOAL-NAME))
      SKIP
      (CHANGE-GOAL NAME))
  (|Change| |to| |the| |goal| |with| |the| |indicated|
          |name| |,| |unless| |we| |are| |already|
          |at| |that| |goal| |.|))
```

Here, **IF** and **SKIP** are themselves macro commands, where **IF** has been discussed above and **SKIP** does nothing but always "succeeds".

Suppose that we are at the prompt "->: ", and we submit the instruction **(MAYBE-CHANGE-GOAL (MAIN . 1))**. Let us trace through the calls of **INTERACTIVE-SINGLE-STEP** and **READ-INSTR**, assuming that the current goal's name is not **(MAIN . 1)**. Notice that here **(GOAL-NAME)** is the call of a Lisp function **GOAL-NAME** which returns the **GOAL-NAME** of the current **GOAL** of the **STATE** on top of the state stack (cf. Section 2). (All corresponding functions returning the fields of the current state and goal are also defined at the top of the file "macro-commands-aux.lisp", as are other functions called in the macro command definitions, all of which are in turn found in the file "macro-commands.lisp".) Also important is the definition of the macro command **IF**.[24] (We omit the documentation here, as it's shown above.)

[24]The reason for **(DO-ALL (! INSTR))** rather than simply **(! INSTR)** is the semantics of **BIND**. The extension of the environment which is given in the first argument of **BIND** is only available *inside* calls of its arguments.

```
(DEFINSTR IF (TEST INSTR1 INSTR2)
  (BIND ((INSTR
           (IF TEST
               (QUOTE INSTR1)
               (QUOTE INSTR2))))
        (DO-ALL (! INSTR))))
```

Now we are ready for the trace. Since anyone reading this far had better be familiar with Lisp, we make no apology for presenting the trace in a traditional Lisp format. Comments are inserted in curly braces in italics, *{<comment>}*. Note: the function **INTERACTIVE-SINGLE-STEP** is evaluated essentially only for side effect, e.g. for pushing a new state onto the state stack in the case of change instructions and for printing out information in the case of help instructions. On the other hand, recall that **READ-INSTR** returns the result of continually expanding the instruction until it is no longer a call of a macro command, where "expanding the instruction" includes the notion of first evaluating every argument of the form **(! x)**.

```
->: (maybe-change-goal (main . 1))

  1> (INTERACTIVE-SINGLE-STEP (MAYBE-CHANGE-GOAL (MAIN . 1)))
     {Now INTERACTIVE-SINGLE-STEP calls READ-INSTR on its argument, thus expanding away the top-level calls
     of macro commands}
    2> (READ-INSTR (MAYBE-CHANGE-GOAL (MAIN . 1)))
       {Now expand the call of MAYBE-CHANGE-GOAL}
    <2 (READ-INSTR
          (BIND ((INSTR (IF (EQ '(MAIN . 1) (GOAL-NAME)) 'SKIP
                             '(CHANGE-GOAL (MAIN . 1)))))
                (DO-ALL (! INSTR))))
       {The BIND form is returned by READ-INSTR because BIND is not a macro command.
       Now the meta command BIND is handled by repeatedly calling INTERACTIVE-SINGLE-STEP on each
       of its arguments except its first. Its first argument modifies the COMMAND-BINDINGS to provide
       an appropriate environment for calls to Lisp.}
    2> (INTERACTIVE-SINGLE-STEP (DO-ALL (! INSTR)))
       {As explained above, this call to INTERACTIVE-SINGLE-STEP is taking place in an environment where
       the variable COMMAND-BINDINGS is appropriately bound. In fact, the BIND form above
       guarantees that in calls to Lisp via the ! mechanism, INSTR will be bound to (CHANGE-GOAL (MAIN . 1))}
      3> (READ-INSTR (DO-ALL (! INSTR)))

      <3 (READ-INSTR (DO-ALL (CHANGE-GOAL (MAIN . 1))))

      3> (INTERACTIVE-SINGLE-STEP (CHANGE-GOAL (MAIN . 1)))
         {This call to INTERACTIVE-SINGLE-STEP results from the way DO-ALL is processed.}
        4> (READ-INSTR (CHANGE-GOAL (MAIN . 1)))

        <4 (READ-INSTR (CHANGE-GOAL (MAIN . 1)))

  (CHANGE-GOAL (MAIN . 1))   {Output from the system -- not part of the trace}
Now proving (MAIN . 1).      {Output from the system -- not part of the trace}
      <3 (INTERACTIVE-SINGLE-STEP
             <current state stack>)
         {output suppressed -- it's not important anyhow}
    <2 (INTERACTIVE-SINGLE-STEP T)

  <1 (INTERACTIVE-SINGLE-STEP T)
->:
```

Let us conclude by analyzing the definition of the macro command **HYP** which was discussed earlier in this section. In fact **HYP** is defined in terms of a macro command **HYP0**; here are the definitions.

```
(DEFINSTR HYP (N INSTR)
  (ORELSE
    (HYP0 N INSTR)
    (PROG2 (PRINT "HYP failed")
           FAIL)))

(DEFINSTR HYP0 (N INSTR)
  (PROTECT
    (DO-STRICT
      ((CONTRADICT-DUMB N)
       (DIVE 1)
       (DO-ALL (! (SUBST (QUOTE (DO-ALL TOP 1))
                         (QUOTE TOP)
                         (QUOTE INSTR))))
       TOP
       (CONTRADICT? N)))))
```

The reader is invited to use the help facility (especially **HELP-LONG**) to learn about the macro commands called inside the definitions of **HYP** and **HYP0**. But here is a brief explanation of these definitions. First, **(HYP N INSTR)** calls for the execution of **(HYP0 N INSTR)**. **ORELSE** is defined so that if this call of **HYP0** "succeeds", then so does the call of **HYP** and nothing further happens. Otherwise, the macro command **PRINT** is called to let us know that the call to **HYP** in fact failed (and the entire call "fails" because of the definition of **PROG2**). Next consider **HYP0**. **HYP0** has a call of **PROTECT** wrapped around its body, so that if it does not "succeed" then no change is made to the global state. Inside that, a call of **DO-STRICT** is made to sequence the commands so that once a command "fails", no further execution is made of commands in the sequence and the entire sequence is deemed to have "failed". Then five commands are given in the list fed to **DO-STRICT**, and behave as explained earlier in this section. The first and last of the five commands are appropriate versions of **CONTRADICT**, and the call of **DO-ALL** is there to create an instruction which is the same as the given instruction, except that the atom **TOP** is replaced everywhere by **(DO-ALL TOP 1)**. The idea here is that if the conclusion is of the form **`(NOT HYP)`**, where **`HYP`** was a hypothesis before invocation of a **CONTRADICT** command, and if one then wants to run **`INSTR`** on **`HYP`**, one wants any occurrence of **TOP** to lift one only to the top of **`HYP`**, not to the top of **`(NOT HYP)`**. Such mundane considerations often arise when writing macro commands, which is why this entire section has been considered optional (as explained in the first sentence of the report).

6. Additional features for advanced users

In this section we document some features of the system that are available for advanced users.

6.1 Timing calls to the theorem prover

We have provided a mechanism, written primarily by J Moore, for allowing users to time their calls to the prover. (In fact this feature can be used independently of the prover.) The function **`SET-TIME-LIMIT`** takes a single positive integer argument and sets a time limit of roughly that many seconds. For example, if one submits the form **`(SET-TIME-LIMIT 5)`** to Lisp, then calls to the prover will abort after approximately 5 seconds of proof time. By "calls to the prover" here we mean calls either to the main prover (as by the interactive **PROVE** command or by **PROVE-LEMMA**) or calls to the theorem prover's rewriter/simplifier only (as by the interactive **BASH** command or by **DEFN**).[25] The user can check the current setting of the time limit by invoking the Lisp function **`time-limit`**. The user can also run a list of events in the same manner as with **`DO-EVENTS`** (see [3]) but where one proceed even upon failure, and running out of time causes such an individual event's failure.

The time limit may be set to **`NIL`**, which is how it is initialized. Such a setting indicates that there is not any time limit set.

Time limits are especially useful for the interactive system in conjunction with the macro command **THEN**, since the command **(THEN <instr>)** calls the theorem prover on each new subgoal generated by **<instr>**. Suppose that one wishes to use **THEN**, with the proviso that not more than a small amount of time be spent on the proof of any particular subgoal (and that way each one gets a chance!). In fact, the macro command **TIME** has been provided with the system. Its first argument is the instruction that one wants to run, and its second argument is the number of seconds that one wants to for the time limit on each call to the theorem prover. So for example, one might wish to submit a command like **(THEN <instr> (TIME PROVE+ 3))**, which allows only about 3 seconds of proof time for a call to the prover on each new subgoal generated by the execution of **<instr>**. (See the help facility for more details, such as when one needs to reset the time afterwards.)

6.2 Theories

The current version of the interactive system also includes a facility for defining *theories*. There is a mechanism for *defining* theories as well as a hint mechanism for *using* theories, along with interactive commands for enabling and disabling theories.

---

[25] Actually, the time is checked inside of a modified version of the theorem prover's function **`SIMPLIFY-CLAUSE1`**.

Defining theories. To *define* a theory, one simply uses the **DEFTHEORY** event mechanism. The syntax for this event is:

```
(DEFTHEORY <theory_name> (<name_1> ... <name_n>))
```

where each **<name$_i$>** is a name which is legal as an argument to the **TOGGLE** events, i.e. to the **DISABLE** and **ENABLE**) events -- let's call such a name *disablable* -- or is the name of an existing **DEFTHEORY** event. The resulting *theory* is considered to be the union of the theories named by all of the **<name$_i$>**, where for any name that is not the name of a theory, we consider such a name to correspond to the "theory" containing itself together with all its disablable satellites.[26]

Using theories by way of hints. One may give *hints* regarding theories in **PROVE-LEMMA** events, as well as in any position where theorem prover hints are expected in an interactive command. The idea is to give lists of theories for which one wants all of their members enabled (or disabled, as the case may be). The atom **T** is used in place of theory names when wants to set up an environment where everything is enabled (or disabled, as the case may be). Here is the syntax: further explanation will follow below.

```
(ENABLE-THEORY <theory_name_1> ... <theory_name_n>)
(DISABLE-THEORY <theory_name_1> ... <theory_name_n>)
(ENABLE-THEORY T)
(DISABLE-THEORY T)
```

One may have zero or one **ENABLE-THEORY** hints and zero or one **DISABLE-THEORY** hints, and one may not have both **(ENABLE-THEORY T)** and **(DISABLE-THEORY T)**. There is one special case: one may use **GROUND-ZERO** as a theory name, i.e. **GROUND-ZERO** is considered as a theory whose members are all of the disablable citizens introduced in t[GROUND-ZERO].

Formally, when the prover is called with some hint, it checks whether or not the name of a given function or rewrite rule is disabled according to the following algorithm.

- If the given name is included in an **ENABLE** hint, consider it enabled.
- Otherwise, if the given name is included in a **DISABLE** hint, consider it disabled.
- Otherwise, if the given name belongs to some theory named in an **ENABLE-THEORY** hint, consider it enabled.
- Otherwise, if the given name belongs to some theory named in an **DISABLE-THEORY** hint, consider it disabled.
- Otherwise, if there is an **(ENABLE-THEORY T)** hint (respectively, a **(DISABLE-THEORY T)** hint), consider it enabled (respectively, disabled).
- Otherwise, simply check whether it is enabled or disabled according to the global database.

[26] The notion of *satellite* is discussed in [3].

Thus, for example, one may wish to give a hint which enables several relevant theories but disables certain events (possibly from those theories); then the latter take precedence over the former hints for the individual events in question, which is what one would hope to be the case.

Interactive theory commands. The interactive commands **ENABLE-THEORY** and **DISABLE-THEORY** may be used to enable or disable all the names belonging to the specified theories (as explained by the help facility). The macro commands **PROVE+**, **CLAIM+**, **BASH+**, and **=+** all take into account these commands.[27]

There are no global **ENABLE-THEORY** or **DISABLE-THEORY** events.

6.3 Turning off printing

The global Lisp variable **`BLURB-FLG`** is initialized to **`T`**. If the user sets it to **`NIL`**, either by submitting **(SETQ BLURB-FLG NIL)** or by invoking the macro command **Q**, then much of the printing will be suppressed. Printing by the theorem prover will still take place, though, unless one sets the theorem-prover's global variable **`IO-FN`** to **`NO-IO`**, e.g. **(SETQ IO-FN 'NO-IO)**. Incidentally, this also happens upon invokation of the macro command **Q**. If one ever aborts during the use of **Q**, the printing can all be turned back on by invoking the macro command **NOISE**.

6.4 Shelving lemmas

The **SHOW-REWRITE** and **REWRITE** commands are set up so that they give equal treatment to enabled and disabled lemmas. This is intentional: the user may well wish to prove some rewrite rules which would be bad from the point of view of the theorem prover but quite appropriate for the interactive use of the **REWRITE** command, perhaps because they contain nonrecursive function symbols on the left side of rewrite rules. Thus one may want a notion of disabling which is appropriate for the **SHOW-REWRITE** and **REWRITE** commands. We refer to this notion as *shelving* lemmas. The syntax for shelving lemmas is

```
(SHELVE <name_1> ... <name_n>)  .
```

The effect of submitting this command (either to Lisp or to the interactive prompt) is to make the given rewrite rules invisible from the point of view of the **SHOW-REWRITE** and **REWRITE** commands, except that **REWRITE** will still work if one gives the *name* of the given rewrite rule as an argument. To remove the names above from the list of shelved lemmas (which is stored in the Lisp variable **`SHELVED-LEMMAS`**), simply

[27] though on rare occasions the hint they generate may have a form that slightly surprises the user

submit the form **`(UNSHELVE <name`$_1$`> ... <name`$_n$`>)`**. One can simply eliminate this feature by submitting the form **`(TOGGLE-SHELVED-LEMMAS)`** to Lisp; then submitting this form again later will bring back in all the lemmas that were shelved before this "toggle".

6.5 Printing old macro command definitions

The system is set up so that when one redefines an existing macro command, the old definition is printed out. This can be useful in case one didn't mean to obliterate the old definition -- at least one can see the old definition (and perhaps resubmit it). But it's not hard to imagine that in a large proof, there may be a lot of such redefinition going on, where the user would rather not be bothered with seeing the old definitions. In that case, simply **(SETQ PRINT-OLD-MACRO-DEF-FLG NIL)**. Setting this flag back to **`T`** will restore the printing of old macro command definitions.

6.6 Changing the prompt

The user can change the prompt from "->: " to anything else. The initial prompt may be reinstated by submitting **`(SETQ *PC-PROMPT* (QUOTE |->: |))`**; it may be changed by replacing the prompt just shown in the **`SETQ`** form with any other desired prompt.

6.7 Compiling macro commands

The instructions for compiling the system will result in an environment in which all of the predefined macro commands are included in an appropriately compiled form. However, macro commands which are subsequently defined or redefined by the user will exist in a form which may be made somewhat more efficient by appropriate compilation. By submitting the form **`(COMPILE-MACRO-COMMANDS)`** to Lisp, the user will cause all the appropriate compilation to take place (and no "unnecessary" recompilation will take place). Alternatively, the user may submit **`(COMPILE-MACRO-COMMANDS <name`$_1$`> ... <name`$_n$`>)`** and then only the macro commands among the given names which are not already compiled will be recompiled.

6.8 Excess instructions

Suppose you complete an interactive proof. If you then attempt to run extra commands, you will simply be told that all goals have already been proved, and the command will *not* generate a new state. However, there are three exceptions to this rule: the command will generate a new state if it is a **BOOKMARK**, a **COMMENT**, or a **CHANGE-GOAL** command. The former two have been allowed because they often provide useful information in the **`INSTRUCTIONS`** hint of a **`PROVE-LEMMA`** event; for example, the bookmark may have resulted from a call of the macro command **WRAP**. The **CHANGE-GOAL** command has been allowed simply for the convenience of the user.

If there are extra instructions (besides these three types) at the end of an **INSTRUCTIONS** hint, they will simply be ignored; that is, they will not cause the **PROVE-LEMMA** event to fail.

6.9 Three useful global variables

For those who are interested, we document here some global variables which may be of interest to the user. The global variable **RESTART-STACK-DEPTH** is rather arbitrarily initialized at 8; the user may reset this. It's the number of times that one can cause an error and then abort back into the interactive system, before finally the system kicks the user back into Lisp. If the user sets the **RESTART-STACK-DEPTH** to 0, then the feature is eliminated: the system will not be reentered when the user aborts (quits). A special convention is that if **RESTART-STACK-DEPTH** is **NIL**, then there is no limit at all (though the particular Lisp implementation may cause a stack overflow at some point).

The global variable **UNDONE-INSTRS** stores the list of instructions printed by the command **UNDONE-INSTRUCTIONS**; submit **(HELP UNDONE-INSTRUCTIONS)** for a description of this command.

Finally, as mentioned already, **EV0** holds the most recent event printed out by the interactive system.

6.10 The macro command USER

It is possible to "customize" one's environment as follows. Each time an instruction is submitted by the user to the interactive system, the system checks to see if the macro-command **USER** is currently defined. If so, it syntactically replaces each such instruction **<ins>** by **(USER <ins>)**. For example, the following version of **USER** causes the system to print 'hi' before running the user's instruction:

```
(definstr user (x) (do-all (print "hi") x))
```

Appendix A
Transcript of Sample Session

Here's a sample session showing how I actually use this system. I'll use it to verify the commutativity of **TIMES**. This is a good example because a novice user of the Boyer-Moore Theorem Prover is likely to find it to be a bit awkward to prove this lemma. The interactive mode helps one discover the facts needed to prove this lemma (after which one could, in fact, get the proof through without the interactive enhancement).

```
(verify (equal (times x y) (times y x)))

->: induct

Inducting according to the scheme:

      (AND (IMPLIES (ZEROP X) (p X Y))
           (IMPLIES (AND (NOT (ZEROP X)) (p (SUB1 X) Y))
                    (p X Y)))

Creating 2 new subgoals, (MAIN . 1) and (MAIN . 2).

The proof of the current goal, MAIN, has been completed.  However, the
following subgoals of MAIN remain to be proved: (MAIN . 1) and
(MAIN . 2).
Now proving (MAIN . 1).

->: view {A "macro command" which prints the hypotheses and the current subterm}

*** Active top-level hypotheses:
There are no top-level hypotheses to display.

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (ZEROP X)
         (EQUAL (TIMES X Y) (TIMES Y X)))

->: prove {We call the Boyer-Moore theorem prover, since this looks easy.}

***** Now entering the theorem prover *****:

This formula simplifies, expanding the functions ZEROP, EQUAL, and
TIMES, to two new conjectures:

{And so on -- I'll omit the output of the theorem-prover.  It does an
induction and then two sub-inductions, but it only takes a few seconds.}
....
     That finishes the proof of *1.1, which, consequently, finishes the
proof of *1.  Q.E.D.


The current goal, (MAIN . 1), has been proved, and has no dependents.
Now proving (MAIN . 2).
```

```
->: view

*** Active top-level hypotheses:
There are no top-level hypotheses to display.

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (AND (NOT (ZEROP X))
              (EQUAL (TIMES (SUB1 X) Y)
                     (TIMES Y (SUB1 X))))
         (EQUAL (TIMES X Y) (TIMES Y X)))

->: promote

->: view

*** Active top-level hypotheses:
H1.  (NOT (ZEROP X))
H2.  (EQUAL (TIMES (SUB1 X) Y)
            (TIMES Y (SUB1 X)))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (TIMES X Y) (TIMES Y X))

->: 1  {Move to the first argument of EQUAL}

->: x  {Recall that X simplifies}

->: p
(PLUS Y (TIMES (SUB1 X) Y))

->: 2

->: p
(TIMES (SUB1 X) Y)

->: =  {Apply the inductive hypothesis}

->: p
(TIMES Y (SUB1 X))

->: top

->: p
(EQUAL (PLUS Y (TIMES Y (SUB1 X)))
       (TIMES Y X))



->: hyps

*** Active top-level hypotheses:
H1.  (NOT (ZEROP X))
H2.  (EQUAL (TIMES (SUB1 X) Y)
            (TIMES Y (SUB1 X)))

*** Active governors:
There are no governors to display.

->: (hide-hyps 2)
```

```
->: prove


***** Now entering the theorem prover *****:

This formula can be simplified, using the abbreviations ZEROP, NOT, and
IMPLIES, to:

      (IMPLIES (AND (NOT (EQUAL X 0)) (NUMBERP X))
               (EQUAL (PLUS Y (TIMES Y (SUB1 X)))
                      (TIMES Y X))).
```

{I'll omit the rest of the theorem-prover's output. It does a straightforward proof by induction.}

```
The current goal, (MAIN . 2), has been proved, and has no dependents.

*!*!*!*!*!*!*  All goals have been proved!  *!*!*!*!*!*!*
You may wish to EXIT -- type (HELP EXIT) for details.
```

Appendix B
Soundness

In this appendix we argue that if the interactive system certifies a term to be a theorem of the Boyer-Moore logic, i.e. the appropriate interactive **`PROVE-LEMMA`** event completes successfully, then that term is indeed a theorem of the logic (relative to the existing list of events, of course). We say "argue" rather than "prove" because we sluff over some important issues:

- correctness of Boyer and Moore's code for their Theorem Prover;
- correctness of the code for the interactive system;
- details involving preservation of the invariant (*) from Section 1 (reviewed below)

However, we do feel that there is value in at least *stating* the theorem below and in presenting an outline that could in principle be fleshed out to a rigorous proof. First we need a definition.[28]

DEFINITION. A *valid state stack* is a state stack which can be produced from an interactive session which begins with a call of the form **(VERIFY <term>)** and then results from the execution of a sequence of change commands.

And now, one more definition.

DEFINITION. Let's say that a goal is *provable* if its hypotheses (hidden and active) provably imply its current conclusion, i.e. the appropriate implication is provable. (If there are no hypotheses then we simply mean that the conclusion is provable.)

We wish to prove the following theorem:

THEOREM. Suppose that **`G`** is a goal of the top state in a valid state stack, and suppose that **`G`** and all of its dependents have conclusions of **`T`**. Then the original version of **`G`** is provable.

In particular, if a sequence of change commands results in a state in which every goal has a conclusion of **`T`**, then the original term given to **VERIFY** is provable.

---

[28]Again, we beg the reader to forgive us. The following definition is clearly much more operational than might be desired. However, we believe it to be a routine (though very tedious) exercise to give a purely logical specification of each command's action so that this definition can itself be purely logical.

To prove this theorem we begin by recalling property (*) from Section 1. As we mentioned above, we will omit the proof of this property, which we believe to be without deep content but rather to depend on careful tedious thinking about the code.

(*) *When invoking a proof command, the goal follows from its modified version together with the subgoals that are created.*

Here, a goal is said to "follow from" others if the provability of the others implies the provability of the goal. Now property (*) trivially implies the following key lemma, which says (roughly) that each state-changing operation preserves validity in reverse.

LEMMA. Fix a valid state stack. Let **`G`** be a goal in a given state **`s`** of that state stack, let **`s'`** be the next state in the stack (thus created from **`s`** by a change command), and let **`G'`** be the version of **`G`** in **`s'`**. Suppose that all subgoals of **`G'`** in **`s'`** are provable in their original forms and that **`G'`** is provable. Then **`G`** is provable.

We may now complete the proof of the theorem, which we do by contradiction. Fix a valid state stack for which the theorem fails. By acyclicity of the goal graph (at the top state in the stack), there is a goal **`G`** with (final) conclusion **`T`** such that all original versions of its (final) subgoals are provable yet its original version is not provable. Now the set of subgoals of any previous version of **`G`** is contained in the set of subgoals of the final version of **`G`**, and hence the lemma above implies (by a trivial induction argument) that every previous version of **`G`** is provable. In particular, the original version of **`G`** is provable, which contradicts our choice of **`G`**.

Appendix C

Extension of the syntax for terms: `COND`, `CASE`, and `LET`

J Moore has graciously consented to the inclusion of code he has written which allows for an extension to the ordinary syntax for terms in the Boyer-Moore logic. In this appendix we describe the three new "macros": **`COND`**, **`CASE`**, and **`LET`**. These are "macros" or "abbreviations" in about the same sense in which **`LIST`** is a macro. For example, the term **`(LIST x y)`** is really an abbreviation for the term **`(CONS x (CONS y 'NIL))`**. Each of these three symbols is intended to be analogous to the corresponding symbols in Lisp.

Here we print essentially the messages which are printed when one misuses these symbols, followed by illustrative examples of correct usage.

```
The COND expression must have at least one argument, all must be
proper lists of length two, the last must start with the symbol T, and
none but the last may start with that symbol.

Example:
(COND ((EQUAL X 1) 2)
      ((LESSP X 7) 3)
      (T 4))

is equivalent to
(IF (EQUAL X 1)
    2
    (IF (LESSP X 7) 3 4))

The CASE expression must have at least two arguments, all but the
first must be proper lists of length two, the last must start with the
symbol OTHERWISE, and none but the last may start with that symbol.

Example:
(CASE X
      (A (PLUS U V))
      ((FOO ARG) B)
      (OTHERWISE 4))

is equivalent to {notice that the "keys" A and (FOO ARG) are quoted}
(IF (EQUAL X 'A)
    (PLUS U V)
    (IF (EQUAL X '(FOO ARG)) B 4))

LET must be given exactly two arguments.  The first argument of a
LET must be a proper list of doublets, i.e., lists of the form
(var term).

Example:
(LET ((X A) (Y (PLUS B 3)))
     (LESSP Y (TIMES X Y)))

is equivalent to
(LESSP (PLUS B 3)
       (TIMES A (PLUS B 3)))
```

The user may notice that the terms are sometimes printed using the **`COND`** and **`CASE`** abbreviations even when the user typed in the same terms using **IF**. However, the system will essentially never use **LET**, though it is a useful abbreviation, especially in complicated definitions.

Appendix D
Command Summary

Here is a list of the messages printed in response to commands of the form **(HELP <command>)**. For more details (which probably will not be necessary too often), the user is advised to use **(HELP-LONG <command>)**. This appendix shows the commands given by invoking HELP (with no arguments), which includes some basic macro commands. We list below descriptions of all commands which are printed by invoking **HELP**, followed by descriptions of all macro commands which are printed by invoking **HELP**.

```
----------------------------------------------------

----------------------------------------------------
***BUILT-IN COMMANDS*** (SHORT) HELP:
----------------------------------------------------

----------------------------------------------------
     (ADD-ABBREVIATION <var> <exp>):  Add the abbreviation which displays
<exp> as <var>.
     NOTE:  The variable <var> should start with the character .
EXAMPLE: (ADD-ABBREVIATION @V (TIMES X Y)) causes future occurrences of
(TIMES X Y) to be printed as @V.  Moreover, user input can henceforth contain
the symbol @V as an abbreviation for (TIMES X Y).
----------------------------------------------------
     BASH:  Attempt to simplify the current goal using the theorem prover's
simplifier (which includes the rewriter).  Any new subgoals are printed out as
they are created.
     (BASH hint1 ... hintk):  as above, where each hint is as in PROVE-LEMMA
and the theorem prover is to use these hints as it does with PROVE-LEMMA.
EXAMPLE:
    (BASH (DISABLE TIMES APPEND) (INDUCT (PLUS X Y))).
----------------------------------------------------
     (BIND bindings instr1 instr2 ... ):  This instruction is only of interest
to writers of macro commands -- execute (HELP-LONG BIND) if you really care.
----------------------------------------------------
     BK:  Move backward one argument in the enclosing term.
EXAMPLE:  If the conclusion is
    (G (H X (FOO Y)) Z)
and the current subterm is (FOO Y), then after executing BK, the current
subterm will be X.
----------------------------------------------------
     (BOOKMARK x):  Makes no change in the state except to insert this
instruction, for the purpose of possible UNDOing later.
EXAMPLE:  The command (BOOKMARK HOWDY) creates an instruction such that if
later one submits (UNDO HOWDY), then the state stack will be unwound back to
(but not including) this bookmark.
----------------------------------------------------
     (CHANGE-GOAL <goal-name>):  Change to the goal with the name <goal-name>,
i.e. make it the current goal.
EXAMPLE: (CHANGE-GOAL (MAIN . 1))

     CHANGE-GOAL:  Change to the first unproved goal (in the list shown by the
command GOALS).
```

```
---------------------------------------------------
     (CLAIM exp):  attempt to prove  exp  from the currently active hypotheses
(using the theorem prover), and if this can be done, then add  exp  as a
hypothesis.
EXAMPLE: (CLAIM (NUMBERP (PLUS X Y))) will simply add (NUMBERP (PLUS X Y)) as
a hypothesis in the current goal.  However, if one submits the command
(CLAIM (NUMBERP X)) and if this cannot be proved by the prover (from the
existing active hypotheses), then there is no change.

     (CLAIM exp 0):  Add  exp  as a hypothesis, and create a new subgoal with
the negation of  exp  added as a hypothesis.
EXAMPLE:  (CLAIM (NUMBERP X) 0) causes a case split on (NUMBERP X), in the
sense that (NUMBERP X) is added as a hypothesis of the current goal, and a new
subgoal is created which is identical to the current goal except that
(NOT (NUMBERP X)) is added as a hypothesis.

     OTHER FORMS (see HELP-LONG):
(CLAIM exp (hint1 hint2 ... hintK))  {to give hints to the prover}
(CLAIM exp TAUT)  {to replace the prover call with a tautology check}

---------------------------------------------------
     COMMANDS:  Show all of the previous commands, in reverse order.
EXAMPLE:  Here is some possible output upon execution of COMMANDS:

The commands thus far (in reverse order, i.e. last one first) have been:
1. (CLAIM (LESSP A B))
2. (REWRITE LESSP-TRANS)
3. PROMOTE
4. START

Thus, there have been three commands so far, with the CLAIM command being the
most recent.

     (COMMANDS n):  Show the last n previous commands, in reverse order.
---------------------------------------------------
     (COMMENT ....):  A no-op, except that the instruction field of the new
state contains the given instruction.
EXAMPLE:  (COMMENT HI THERE HUMAN) makes this instruction the new instruction,
as shown e.g. by executing COMMANDS.

     OTHER FORMS (see HELP-LONG):
COMMENT makes a comment which displays the current subterm.
---------------------------------------------------
     (CONTRADICT n):  Negate the current conclusion and make it the n-th
hypothesis, while negating the current n-th hypothesis and making it the
current conclusion.
EXAMPLE:  If the active hypotheses are
     H1. (LESSP X Y)
     H2. (NOT (ZEROP Z))
and the conclusion and current subterm are both
    (LESSP (TIMES X Z) (TIMES Y Z)),
then after executing (CONTRADICT 2), the hypotheses are
     H1. (LESSP X Y)
     H2. (NOT (LESSP (TIMES X Z) (TIMES Y Z)))
and the conclusion is
    (ZEROP Z).
```

```
---------------------------------------------------
     (DEMOTE n1 n2 ... nk):  Replace the current conclusion with the goal of
proving that the conjunction of the indicated hypotheses implies the current
conclusion, and drop these hypotheses.  Note that this command may only be
used when at the top of the conclusion.

     DEMOTE:  As above, but demote all the active hypotheses.
EXAMPLE:  If the active hypotheses are
     H1. (LESSP X Y)
     H2. (NOT (ZEROP Z))
and the conclusion and current subterm are both
    (LESSP (TIMES X Z) (TIMES Y Z)),
then after executing DEMOTE, there will be no active hypotheses and the
conclusion will be

      (IMPLIES (AND (LESSP X Y) (NOT (ZEROP Z)))
               (LESSP (TIMES X Z) (TIMES Y Z)))

---------------------------------------------------
     (DISABLE name1 ... nameK):  Disable the given fast rewrite rules and
function definitions for the fast rewriter.  (These terms are explained with
(HELP-LONG S).)
EXAMPLE USE: (DISABLE APPEND ZEROP).

     DISABLE:  Disable all fast rewrite rules and function definitions.
---------------------------------------------------
     (DISABLE-THEORY <name1> ... <nameK>):  Disable the rules and function
definitions from the given theories for the fast rewriter.  (These terms are
explained with (HELP-LONG S).)
EXAMPLE USE: (DISABLE-THEORY LIST-FACTS ARITHMETIC).

     NOTE:  to disable all fast rewrite rules and function definitions, use
DISABLE.
---------------------------------------------------
     (DIVE N):  For N>0, move to the Nth argument.
EXAMPLE:  If the current subterm is
    (TIMES (PLUS A B) C),
then after (DIVE 1) it is
     (PLUS A B).

     DIVE:  Dive down to the leftmost unproved branch in the IF-structure
(from the top of the conclusion, unless the current term is an IF term).
EXAMPLE:  If the conclusion is
    (IF A (IF B T (G X)) Y),
then DIVE puts the current subterm at (G X).

     NOTE:  N is an abbreviation for (DIVE N), so we could have typed 1
instead of (DIVE 1) in the first example above.

     OTHER FORMS (see HELP-LONG):  (DIVE N1 ... Nk).
---------------------------------------------------
     (DO-ALL instr1 instr2 ...):  Run all of the given instructions.
EXAMPLE: (DO-ALL UNDO DIVE (PLAY S S-PROP)).
---------------------------------------------------
     DROP:  Drop all the top-level hypotheses.

     (DROP n1 ... nk):  Drop the top-level hypotheses with the indicated
indices.
EXAMPLE: (DROP 2 5).

     (DROP HIDDEN):  Drop the hidden top-level hypotheses.
---------------------------------------------------
     (ENABLE <name1> ... <nameK>):  Enable the given fast rewrite rules and
function definitions for the fast rewriter.  (These terms are explained with
(HELP-LONG S).)
EXAMPLE USE: (ENABLE APPEND ZEROP).

     ENABLE:  Enable all fast rewrite rules and function definitions.
```

(ENABLE-THEORY <name1> ... <nameK>): Enable the rules and function definitions from the given theories for the fast rewriter. (These terms are explained with (HELP-LONG S).)
EXAMPLE USE: (ENABLE-THEORY LIST-FACTS ARITHMETIC).

NOTE: to enable all fast rewrite rules and function definitions, use ENABLE.

= : Make a substitution for the current subterm, using an equality which is explicitly among the current active hypotheses and governors.
EXAMPLE: If the current subterm is (PLUS X Y) and the equality (EQUAL (TIMES U V) (PLUS X Y)) is an active hypothesis, then after executing the = command, the current subterm will be (TIMES U V) (unless perhaps some other hypothesis equates (PLUS X Y) with something).

(= Exp): Replace the current subterm by Exp, if they can be proved equal by the theorem prover under the current active hypotheses and governors.
EXAMPLE: (= T) will replace the current subterm with T if the prover can prove their equality.

(= Exp1 Exp2): Replace Exp1 by Exp2 in the current subterm, if they can be proved equal by the theorem prover under the current active hypotheses and governors.
EXAMPLE: If the current subterm is (LESSP (PLUS X Y) Z), then after executing the command (= (PLUS X Y) (PLUS Y X)), the new current subterm will be (LESSP (PLUS Y X) Z).

OTHER FORMS: In the following one may use * in place of Exp1 if it’s the current subterm.
(= Exp1 Exp2 (hint1 hint2 ... hintK)): for giving hints to the prover.
(= Exp1 Exp2 0): As above, but create a new goal to prove (EQUAL Exp1 Exp2) from the currently active hypotheses and governors if and only if neither this equation nor its commuted version are among the currently active hypotheses or governors.

(EXIT name lemma-types): Exit the proof checker, printing out the appropriate PROVE-LEMMA event with the given name and types (e.g. (REWRITE) or NIL).
EXAMPLE: (EXIT TIMES-ASSOCIATIVITY (REWRITE)).

(EXIT T): Exit the proof checker upon completion of an interactive verification of termination of a definition. Thus, this version should be used when the session started with VERIFY-DEFN rather than VERIFY.

EXIT: Quit the proof checker without making an event. In this case, (VERIFY) may be executed to get back in at the same state, as long as there hasn’t been an intervening use of the proof-checker.

(GENERALIZE ((term1 V1) ... (termn Vn))): Replace each of the given terms by the indicated new variable (which must not occur anywhere in the current goal). A question mark (?) may be used in place of any or all variables Vi.
EXAMPLE: (GENERALIZE (((PLUS X Y) ?))).

OTHER FORMS (see HELP-LONG):
(GENERALIZE ((term1 V1) ... (termn Vn)) new-goal-name).

GOALS: Print the name of each unproved goal, current goal first.

OTHER FORMS (see HELP-LONG): (GOALS TREE), (GOALS ALL).

HELP: Print names of selected instructions, including a selected set of predefined macro commands. NOTE: To get the names of all instructions and all macro commands, use HELP-LONG.

(HELP instr1 ...): Print short help on instr1 ..., generally with examples. NOTE: Use HELP-LONG to get more complete information, but without examples.

HELP-LONG: Print names of all instructions, including all macro commands. For a shortened list appropriate for people new to this system, use HELP instead.

(HELP-LONG instr1 ...): Print help on instr1 .... NOTE: Use HELP to get less detailed information, but with examples, which is perhaps more appropriate for people who are just beginning to use this system.

HIDE-GOVS: Hide all of the governors.

(HIDE-GOVS ...): Hide the indicated governors, referenced by number. To obtain these numbers, type HYPS or (HYPS NIL ALL).

HIDE-HYPS: Hide all of the hypotheses.

(HIDE-HYPS ...): Hide the indicated hypotheses, referenced by number. To obtain these numbers, type HYPS or (HYPS ALL NIL).
EXAMPLE: (HIDE-HYPS 2 4).

HYPS: Print the current active hypotheses, both top-level and governors.

(HYPS ALL ALL) As above, but show the hidden hypotheses and governors as well.
EXAMPLE: If the current hypotheses are (EQUAL X1 Y) and (EQUAL X2 Y) (in that order), where the first of these is hidden but the second is active, and there are no governors, then (HYPS ALL ALL) will display:

```
*** Top-level hypotheses (with "*" when hidden):
*H1.  (EQUAL X1 Y)
H2.  (EQUAL X2 Y)

*** Governors (with "*" when hidden):
There are no governors to display.
```

while HYPS will simply display:

```
*** Active top-level hypotheses:
H2.  (EQUAL X2 Y)

*** Active governors:
There are no governors to display.
```

OTHER FORM (see HELP-LONG): (HYPS (H1 H2 ... Hk) (G1 G2 ... Gn))

(INDUCT (g v1 ... vn)): Induct as in the corresponding INDUCT hint given to the theorem prover, creating new subgoals for the base and induction steps.
EXAMPLE: If the current conclusion is of the form (P X) and there are no hypotheses, then after (INDUCT (TIMES X Y)), there are two new subgoals with conclusions

```
    (IMPLIES (ZEROP X) (P X))  and
    (IMPLIES (AND (NOT (ZEROP X)) (P (SUB1 X))) (P X))
```

INDUCT: Induct according to a scheme chosen by the theorem prover's heuristics, creating subgoals for the base and induction steps.

(LISP form): Evaluate the given Lisp form.
EXAMPLE: (LISP (PROGN (PRINT (+ 2 5)) (TERPRI NIL))) will cause the numeral 7 to be printed, followed by a newline.

(NEGATE instr): This instruction is only of interest to writers of macro commands -- execute (HELP-LONG NEGATE) if you really care.

NX: Move forward one argument in the enclosing term.
EXAMPLE: If the conclusion is

```
    (G (H X (FOO Y)) Z)
```

and the current subterm is X, then after executing NX, the current subterm will be (FOO Y).

```
---------------------------------------------------
     P:  Prettyprint current subterm in the usual manner.  So for example,
(AND x y z) is printed rather than (AND x (AND y z)); compare with PP.
---------------------------------------------------
     PP:  Prettyprint current subterm without any special use of
pretty-printing abbreviations.  So for example, (AND x y z) is printed as
(AND x (AND y z)); compare with P.
---------------------------------------------------
     PP-TOP:  Prettyprint entire term , highlighting current subterm.
Printing is with respect to the same conventions as for the command PP.
EXAMPLE:  If the conclusion is (EQUAL (PLUS X X 0) (TIMES X 2)) and the
current subterm is (PLUS X X 0), then PP-TOP will cause the printing of:
      (EQUAL (*** (PLUS X (PLUS X 0)) ***)
             (TIMES X 2))

---------------------------------------------------
     (PROG2 instr1 instr2):  This instruction is only of interest to writers
of macro commands -- execute (HELP-LONG PROG2) if you really care.
---------------------------------------------------
     PROMOTE:  Replace (IMPLIES hyps exp) with simply  exp, adding  hyps  to
the list of hypotheses (after flattening its AND structure, if it is an AND
expression).
EXAMPLE:  If the conclusion is (IMPLIES (AND X Y) Z), then after execution of
PROMOTE, the conclusion will be Z and the terms X and Y will be hypotheses.
---------------------------------------------------
     (PROTECT instr):  Treats instr as an atomic entity, in the sense that if
it is a macro command, then either instr "succeeds" or else the global state
reverts to what it was before invocation of the PROTECT command.
EXAMPLE: (PROTECT (DO-ALL DIVE PUSH TOP S)) will either cause execution of all
instructions DIVE, PUSH, TOP, and S (in that order), or else will be a no-op
on the global state (in case any of these four instructions does not create a
new state).
---------------------------------------------------
     PROVE:  Attempt to prove the current goal.
     (PROVE hint1 ... hintk):  as above, where each hint is as in PROVE-LEMMA
and the theorem prover is to use these hints as it does with PROVE-LEMMA.
EXAMPLE:
    (PROVE (DISABLE TIMES APPEND) (INDUCT (PLUS X Y))).
---------------------------------------------------
     PUSH:  Replace the current subterm with T (assuming it’s boolean), and
create a new goal to prove that term under the active hypotheses and governors.
EXAMPLE:  Suppose that we have the single active hypothesis and single active
governor
     H1. (LESSP X Y)
     G1. (NOT (ZEROP Z)),
and conclusion
    (IF (ZEROP Z) T (LESSP (TIMES X Z) (TIMES Y Z)))
with current subterm
    (LESSP (TIMES X Z) (TIMES Y Z)).
Then after the execution of PUSH, the current subterm is T, the new conclusion
is
    (IF (ZEROP Z) T T),
and the new subgoal has hypotheses
     H1. (LESSP X Y)
     H2. (NOT (ZEROP Z))
and conclusion
    (LESSP (TIMES X Z) (TIMES Y Z)).

     OTHER FORMS (see HELP-LONG): (PUSH name).
---------------------------------------------------
     REMOVE-ABBREVIATIONS:  Remove all abbreviations.

     (REMOVE-ABBREVIATIONS var1 var2 ... varK):  Remove the indicated
abbreviations.
EXAMPLE: (REMOVE-ABBREVIATIONS @X @Y) will cause future proof states to be
unaware of @X and @Y as abbreviations (unless they are restored by future
invocations of ADD-ABBREVIATION).
```

RESTORE: Restores the state stack to its value just before the immediately preceding RESTORE or UNDO, if any.
EXAMPLE: After successfully executing (UNDO 3), the execution of RESTORE will put the state-stack back to its value before that UNDOing, provided that there is no intervening invocation of UNDO or RESTORE.

(REWRITE N): Replace the current subterm with a new term by using the Nth rewrite rule, as displayed by the command SHOW-REWRITES.
EXAMPLE: Suppose that the current subterm is (REVERSE (REVERSE X)). The command SHOW-REWRITES might result in the following being printed to the screen:

```
     1. REVERSE-NLISTP
     New term: NIL
     Hypotheses: ((NOT (LISTP (REVERSE X))))
     2. REVERSE-REVERSE
     New term: X
     Hypotheses: ((PLISTP X))
```

Then (REWRITE 2) results in a new current subterm of X together with a new subgoal of proving (PLISTP X) under the current goal's active hypotheses. (Note, however, that the new goal would not be created if (PLISTP X) is among the currently active hypotheses or governors.)
OTHER FORMS (see HELP-LONG):
One may specify the name of the rewrite rule instead of the number. One may also specify a substitution for the free variables. Also, REWRITE (with no arguments) may be used to apply the first hypothesis-free rewrite rule (or, just the first rule if all have hypotheses to relieve), as displayed by SHOW-REWRITES. Use (HELP-LONG REWRITE) to get the details.

S: Simplify the current subterm, expanding nonrecursive function calls and doing various nice things. (For details, use (HELP-LONG S).)
EXAMPLE: (ZEROP X) simplifies to

```
    (IF (EQUAL X 0) T (IF (NUMBERP X) F T))
```

if there are no active hypotheses or governors, but in the presence of the hypothesis (NOT (EQUAL X 0)) it simplifies to

```
    (IF (NUMBERP X) F T).
```

OTHER FORMS (see HELP-LONG): (S ALL), (S LEMMAS), (S n), (S (<lemma-1> ... <lemma-N>)).

S-PROP: Expand all calls of IMPLIES, AND, OR, and NOT in the current subterm, and then push all calls of IF to the top of the current subterm.

OTHER FORMS (see HELP-LONG): (S-PROP NIL), (S-PROP X1 ... XN).

SHOW-ABBREVIATIONS: Show all abbreviations.
EXAMPLE OUTPUT:

```
@Y corresponds to
     (TIMES B C)

@X corresponds to
     (PLUS A @Y)
       i.e. to
     (PLUS A (TIMES B C))
```

OTHER FORM (see HELP-LONG): (SHOW-ABBREVIATIONS var1 var2 ... varK).

SHOW-GOVS: Activate all of the governors.

(SHOW-GOVS ...): Activate the indicated governors, referenced by number. To obtain these numbers, type (HYPS NIL ALL).
EXAMPLE: (SHOW-GOVS 2 4).

SHOW-HYPS: Activate all of the top-level hypotheses.

(SHOW-HYPS ...): Activate the indicated top-level hypotheses, referenced by number. To obtain these numbers, type (HYPS ALL NIL).
EXAMPLE: (SHOW-HYPS 2 4).

SHOW-INDUCTIONS: Show all the heuristically-chosen induction schemes.

SHOW-REWRITES: Show all the rewrite rules which apply to the current subterm, including the name, resulting subterm, and hypotheses whose relieving will require the generation of new subgoals.
EXAMPLE: Suppose that the current subterm is (LESSP A (PLUS A 3)) and that there is a rewrite rule for transitivity of LESSP. Then SHOW-REWRITES might display the following:

```
1. LESSP-TRANS
New term: T
Hypotheses: ((LESSP A $Y) (LESSP $Y (PLUS A 3)))
```

The dollar sign indicates a free variable, i.e. a variable which occurs in the hypotheses of the rewrite rule (or, in rare cases, the right side of the conclusion of the rule) but does not occur in the left side of the conclusion of the rule. Submit (HELP REWRITE) to see the role of free variables in applying the REWRITE command.

OTHER FORM (see HELP-LONG): (SHOW-REWRITES <argument>).

SPLIT: Replace the current goal by the cases generated from its propositional structure. This command can only be used at the top of the conclusion. Roughly speaking, this command does OR-splitting in the hypotheses and AND-splitting in the conclusion.
EXAMPLE: Suppose that there is a hypothesis of the form (IMPLIES (AND P Q) R). Then a subgoal would be generated replacing this hypothesis by (NOT P), and similarly there would be one for (NOT Q) and for R. Of course, there could be more subgoals generated if P, Q, or R had additional propositional structure or if other hypotheses or the conclusion generated further cases.

(SUBV (v1 t1) ... (vn tn)): Do parallel substitution of each term ti for the respective variable vi into the current subterm, if justified by the current active hypotheses and governors.
EXAMPLE: If the equalities (EQUAL X (PLUS A B)) and (EQUAL (TIMES C D) Y) are among the active hypotheses, and if the current subterm is
    (DIFFERENCE X Y),
then after executing
    (SUBV (X (PLUS A B)) (Y (TIMES C D))),
the current subterm will be
    (DIFFERENCE (PLUS A B) (TIMES C D)).

SUBV: As above, but every such substitution will be performed.
EXAMPLE: In the example above, SUBV would have had the same effect.

(SUCCEED instr): This instruction is only of interest to writers of macro commands -- execute (HELP-LONG SUCCEED) if you really care.

TOP: Move to the top of the current conclusion.
EXAMPLE: If the conclusion is
    (G (H X (FOO Y)) Z)
and the current subterm is (FOO Y), then after executing TOP, the current subterm will be
    (G (H X (FOO Y)) Z).

(UNDO N): Undo N instructions.
EXAMPLE: (UNDO 3) undoes the last three instructions.

UNDO: Undo one instruction.

OTHER FORM (see HELP-LONG): (UNDO x), where x is a bookmark.

UNDONE-INSTRUCTIONS: Print the list of instructions which were given to, but not run by, the last invocation of any command which keeps track of such matters, such as PROVE-LEMMA.

```
---------------------------------------------------
     UP:  Move up to enclosing subterm.
EXAMPLE:  If the conclusion is
    (G (H X (FOO Y)) Z)
and the current subterm is
    (FOO Y),
then after executing UP, the current subterm will be
    (H X (FOO Y)).
---------------------------------------------------
     (USE-LEMMA name ((var1 term1) ... (varK termK))):  Add the given lemma or
axiom from the chronology to the list of hypotheses, after instantiating it
according to the given substitution.
EXAMPLE:  Invocation of (USE-LEMMA TIMES-0 ((X A))) will add the hypothesis
(EQUAL (TIMES A 0) 0) if the lemma TIMES-0 (from the chronology) is
(EQUAL (TIMES X 0) 0).

     USE-LEMMA:  Same as (USE-LEMMA ()).
---------------------------------------------------
     X:  Expand function call at current subterm, and simplify.
EXAMPLE:  If the current subterm is
    (APPEND (CONS A X) Y),
then after executing X the current subterm will be
    (CONS A (APPEND X Y)).

     OTHER FORMS (see HELP-LONG):  (X ALL), (X LEMMAS), (X n).
---------------------------------------------------
     X-DUMB:  Expand function call at current subterm without simplifying the
result.
EXAMPLE:  If the current subterm is
    (APPEND (CONS A X) Y),
then after executing X the current subterm will be

      (IF (LISTP (CONS A X))
          (CONS (CAR (CONS A X))
                (APPEND (CDR (CONS A X)) Y))
          Y)
---------------------------------------------------

---------------------------------------------------
Some ***MACRO-COMMANDS*** (SHORT) HELP:
---------------------------------------------------

---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(BASH+ &REST HINTS)
Call the theorem prover in the sense of the BASH command, but with hints that
match the local disable-enable environment in the current interactive session,
as updated (optionally) by the HINTS argument.  Notice that this command has
the same syntax as BASH, e.g. (BASH (DISABLE PLUS TIMES)).  This command
"succeeds" if and only if the corresponding BASH command "succeeds", i.e. the
proof completes.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(CLAIM+ EXP &OPTIONAL HINTS)
Same as using the command CLAIM with one or two arguments (where the second
argument is not an atom), except that the if the theorem prover is called,
then it is called with hints that match the local disable-enable environment
in the current interactive session, as updated (optionally) by the HINTS
argument.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(CLAIM0 EXP)
(CLAIM0 <exp>) is similar to (CLAIM <exp> 0), in that it adds <exp> to the
current active hypotheses.  However, the new subgoal has <exp> as its
conclusion and the currently active hypotheses as its active hypotheses.
```

```
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(DEFN &OPTIONAL FLG)
DEFN is good to use just after entering an interactive termination proof with
VERIFY-DEFN.  The effect is to try to prove each of the generated goals
(without induction) so that when this command completes, one is left with each
of the UNSIMPLIFIED goals that were not successfully proved.  However, if one
submits (DEFN T) then one is left with the SIMPLIFIED goals.
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(DV &REST ADDR)
Same as DIVE, except that DV is preferred when the address list refers to the
"nice" print representation of the term, i.e. that given by the command P
(rather than PP, say).
     For example, to dive to the second argument of (LIST X Y Z) you can
either (DV 2) or (DIVE 2 1), since (LIST X Y Z) abbreviates
(CONS X (CONS Y Z)).
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(ELIM VAR TERM DESTRUCTORS &OPTIONAL COMPLETION)
Replace VAR by TERM and then eliminate the DESTRUCTORS by generalization.  If
the equality of VAR with TERM creates a subgoal, then this subgoal will try to
be established by the fast rewriter (i.e. with the S command) or else a new
subgoal will remain to be proved.  If the optional argument COMPLETION is
supplied, then that command will be run on the subgoal rather than ESTABLISH
(which causes the above simplification).  This command "succeeds" as long as
the generalization "succeeds".  Use ELIM0 instead if you don't want bookmarks.
     Sample use:  (ELIM X (ADD1 (SUB1 X)) ((SUB1 X))) replaces X by (ADD1 Z)
for some variable Z.
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(=+ OLD NEW &OPTIONAL HINTS)
Same as using the command  =  with two or three arguments (where the third
argument is not an atom), except that the theorem prover is called with hints
that match the local disable-enable environment in the current interactive
session, as updated (optionally) by the HINTS argument.
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(HYP N INSTR)
Applies instruction INSTR to hypothesis number N.  If this "fails", then there
is no change made to the state stack, the command "fails", and a message to
that effect is printed; otherwise it "succeeds".
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(PG INS)
Runs the given instruction, "succeeding" if and only if the given instruction
"succeeds", and then prints out all the new goals that are created in the
course of execution of this instruction.  However, if the instruction is
RESTORE, PLAY, or REPLAY, then this printing of goals is suppressed.
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(PLAY INSTRS &OPTIONAL PROVE-LEMMA-FLG)
(PLAY (instr1 instr2 ...) &OPTIONAL PROVE-LEMMA-FLG):  Play the given
instructions from the current state, stopping if any fails and otherwise
noting if all instructions run successfully (in which case the PLAY
"succeeds").  NOTE:  If not all of the instructions "succeed", then the
command (UNDONE-INSTRUCTIONS) will print a list of all the instructions which
did not run (starting with the one that "failed").
     If the optional argument is anything except  NIL, then only CHANGE
commands (i.e. primitive commands as opposed to HELP, META, or MACRO commands)
will be allowed, which is the same restriction applied to the INSTRUCTIONS
hint for PROVE-LEMMA.
     After invoking PLAY, the RESTORE command will put one back to the state
existing just when the PLAY command was invoked.
--------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(PRINT X)
Prettyprints its argument, and always "succeeds".
```

```
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(PRINT-GOAL &OPTIONAL GIVEN-NAME)
(PRINT-GOAL <name>):  Print the ORIGINAL hypotheses and conclusion of the goal
named <name>.
PRINT-GOAL:  Print the ORIGINAL hypotheses and conclusion of the current goal.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(PROVE+ &REST HINTS)
Call the theorem prover with hints that match the local disable-enable
environment in the current interactive session, as updated (optionally) by the
HINTS argument.  That is, call the theorem prover in an environment which is
as though the current interactive session's modifications to the
enable-disable environment were made to the global data-base.  Notice that
this command has the same syntax as PROVE, e.g. (PROVE+ (DISABLE PLUS TIMES)).
This command "succeeds" if and only if the corresponding PROVE command
"succeeds", i.e. the proof completes.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(REPLAY)
REPLAY:  Undo back to the starting point of the current session and replay the
instructions.  This command "succeeds" if the proof state (i.e. the
STATE-STACK) gets back to where it is currently.  NOTE:  If not all of the
instructions "succeed", then the command (UNDONE-INSTRUCTIONS) will print a
list of all the instructions which did not run (starting with the one that
"failed").
     NOTE that this instruction saves the current STATE-STACK, so that RESTORE
may be used to get back to where one was before executing this command.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(RETRIEVE &OPTIONAL VAR)
(RETRIEVE var):  Exit the interactive environment with a suggestion that one
submit this command to Lisp.  Such a submission will result in a restoration
to the state stored the last time that (SAVE var) was executed in an
interactive session.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(S*)
Alternately applies S and S-PROP until either the goal is proved or there is
no change.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(SAVE &OPTIONAL VAR)
(SAVE var):  Saves the current state stack in the given variable; the default
for simply SAVE is TEMP-SS.  Submit the command (RETRIEVE VAR) (or, simply
(RETRIEVE) for the case TEMP-SS) to Lisp to get back to this state.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(TAUT)
TAUT:  Replace the current conclusion by TRUE if it follows tautologically
from the current hypotheses (upon expansion of functions from the list (AND OR
NOT IMPLIES FIX ZEROP IFF NLISTP)).
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(THEN INSTR &OPTIONAL COMPLETION)
If the given instruction INSTR "succeeds", then the instruction COMPLETION is
run on each of the new subgoals of the original goal (if any).  If no
COMPLETION is given, then PROVE+ is used as the COMPLETION.  This command
"succeeds" if INSTR "succeeds" and each resulting invocation of COMPLETION
"succeeds".
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(UNDO+ X)
(UNDO+ x) undoes back to and including the most recent instruction
(BOOKMARK x), if there is one, and "succeeds" in case it finds such an
instruction.  Notice that in this case, (UNDO x) only goes back to, but not
including, such a bookmark.
```

```
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(VIEW)
Print the HYPS and the current subterm.
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(VIEW-TOP)
Print the HYPS and the conclusion, highlighting the current subterm.
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(WRAP NAME INSTR)
The main difference between (WRAP instr) and instr is that in case of
"success", the former will insert two bookmarks:  (BOOKMARK (BEGIN name))
before the first new primitive command put on the state stack, and
(BOOKMARK (END name)) after the last.  "Success" is the same for both, however.
One other difference is that if instr does not "succeed", then all state is
restored to what it was before running this command (in the sense of PROTECT).
Use WRAP+ instead if you don’t want this restoration to take place in case of
failure.
----------------------------------------------------
```

References

1. Matt Kaufmann, ‘‘A User’s Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover’’, Tech. report 60, Institute for Computing Science, University of Texas at Austin, August 1987.

2. Robert S. Boyer and J Strother Moore, *A Computational Logic,* Academic Press, New York, 1979.

3. Robert S. Boyer and J Strother Moore, ‘‘The User’s Manual for A Computational Logic’’, Tech. report 18, Computational Logic, Inc., February 1988.

4. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF,* Springer-Verlag, New York, 1979.

5. Guy L. Steele Jr., *Common Lisp: The Language*, Digital Press, 1984.

6. R.S. Boyer and J S. Moore, *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures,* Academic Press, 1981, pp. 103-185.

Table of Contents