ICOT Technical Memorandum: TM-0086

TM-0086

THE BOYER-MOORE THEOREM PROVER IN PROLOG USER'S MANUAL (V3.6 Nov. 1984)

December, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

THE BOYER-MOORE THEOREM PROVER IN PROLOG

USER'S MANUAL

(V3.6 Nov. 1984)

TABLE OF CONTENTS

APPENDIX

INTRODUCTION

The Boyer Moore Theorem Prover (abbreviated as BMTP in the following) is one of the most powerful theorem prover developed by Boyer and Moore, for proving theorems about LISP programs.

For an introduction to BMTP, readers are recommended to consult [Boyer & Moore 79]. However, for the benefit of those who do not have access to a copy of this book, and for those who have some prior knowledge of BMTP, a summary of the formal system is included in Appendix I. of this manual.

This manual describes the BMTP reproduced in The Institute for New Generation Computer Technology for the DECsystem-20. The system is written in DECsystem-10 Prolog.

In this manual we shall assume the DECsystem-10 Prolog convention which is adopted in the manual [Bowen et. al. 83].

This work is based on the result of a subproject, Intelligent Programming System, for the Fifth Generation Computer Systems project.

1. How to Run BMTP

The BMTP offers the user an interactive theorem proving environment with tools for incrementally proving theorems, tracing proofs, and modifying parts of proof environments without having to start again from scratch.

(In the current version (v3.6 Nov. 1984), the system is equipped with no special editing facilities for proof environments yet.)

1.1. Getting Started

To run the BMTP, perform the monitor command:

```
@run bmtp
```

The system responds with a message of identification and the prompt "| ?- " as soon as it is ready to accept input, thus:

```
BMTP in Prolog (V3.6 Nov. 1984)
yes
| ?-
```

At this point the system is expecting input of a directive, i.e. a question or command. This state is in fact the Prolog interpreter top level.

1.2. Top-level Commands

The top-level commands of the system are listed as follows.

`shell(`*S*`:`*R*`/`*BTM*`).`	define shell
`definition(`*F*`=`*B*`).`	define function
`lemma(`*H*`=`*B*`).`	add lemma
`axiom(`*H*`=`*B*`).`	add axiom
`theorem(`*H*`=`*B*`).`	prove theorem
`abbrev(`*X*`=`*Y*`).`	define abbreviation

1.2.1. Defining Shells

`| ?- shell` C`(` A_1`:`TR_1`/`DV_1`, ... ,` A_n`:`TR_n`/`DV_n `)` `:`R `/`BTM`.`

The shell command introduces the system a new shell with constructor C, accessors A_1, ..., A_n, type restrictions TR_1, ..., TR_n (optional), default values DV_1, ..., DV_n (optional), recognizer R, and bottom object BTM (optional), according to the shell principle (see APPENDIX I.4).

Example.

```
| ?- add1( sub1:numberp/zero ):numberp/zero.
| ?- pack( unpack/zero ):litatom/nil.
| ?- shell cons( car/nil, cdr/nil ):listp.
```

1.2.2. Defining Functions

```
| ?- definition F( V1, ... , Vn ) = Body.
```

The definition command introduces the system a new function F, with formal parameters V_1, ..., V_n, and the definition body *Body*, according to the definition principle (see APPENDIX I.5).

Example.

```
| ?- definition reverse(x) =
|       if(listp(x),
|          append(reverse(cdr(x)),cons(car(x),nil)),
|          nil).
```

1.2.3. Adding Axioms and Lemmas

```
| ?- axiom Name( L1, ... , Ln ) = Body.

| ?- lemma Name( L1, ... , Ln ) = Body.
```

The axiom and lemma commands introduce the system a new axiom/lemma named *Name*, with labels *L1*, ..., *Ln*, and the body *Body*.

Example.

```
| ?- axiom numberp_apply(rewrite) =
|          numberp(apply(fn,x,y)).
| ?- lemma remainder_quotient_elim(elimination) =
|          implies(and(not(zerop(y)),numberp(x)),
|                   equal(plus(remainder(x,y),times(y,quotient(x,y))),
|                         x)).
```

1.2.4. Proving Theorems

```
| ?- theorem Name( L1, ... , Ln ) = Body.
```

The theorem command instructs the system to prove a theorem named *Name*, with labels *L1*, ..., *Ln*, and the body *Body*.

The proof proceeds interactively between the system and the user through the prover commands described in section 1.3, and the proof-step options described in section 1.4.

Example.

```
| ?- theorem reverse_reverse(rewrite) =
|            implies(plistp(x),equal(reverse(reverse(x)),x)).
```

1.2.5. Defining Abbreviations

```
| ?- abbrev LHS = RHS.
```

The abbreviation command instructs the system that any occurrence of the term *LHS* in the succeeding input terms (definition bodies, axiom/lemma and theorem bodies, and *RHS*'s in other abbreviations) should be interpreted as the term *RHS*.

Example.

```
| ?- abbrev cddr(x) = cdr(cdr(x)).
| ?- abbrev cdddr(x) = cdr(cddr(x)).
| ?- abbrev cdr3(x) = cddr(cdr(x)).
```

Note that both `cdddr(x)` and `cdr3(x)` are expanded into `cdr(cdr(cdr(x)))`.

Such abbreviations seem to be unnecessary if we had corresponding nonrecursive function definitions. In fact, they are logically equivalent in the system except that the former are completely expanded at any time a term mentioning the abbreviation is read-in to the system, although the latter at a proof time in the simplifier. Thus they are to be understood just as read-in macros.

The way to define more sophisticated abbreviations is described in chapter 3.

1.3. Prover Commands

In a process of a proof, the system can be instructed to switch several modes, to control the proof process and steps, and to display the proof environment.

These are called *Prover Commands* and listed as follows.

```
b.      break
c.      toggle continue mode
e.      toggle tracing (rewrite,expand)
g.      give up the proof (but store as a lemma)
h.      help
p.      display remaining conjectures
q.      process next conjecture
r.      repeat q.
t.      trace
v.      toggle VT100 mode
```

Command	Description
b.	This invokes "break" of the Prolog interpreter.
c.	See section 1.4.
e	See section 1.4.
g.	The command instructs the prover to give up the current proof; that is, all of the remaining conjectures (the descendant conjuncts of the initial body of a theorem) are cleared away from the system pools. But add as a lemma if so specified in the command that invoked the proof.
h.	The help message for prover commands is displayed.
p.	All of the remaining conjectures are displayed.

`q.` The next conjecture is taken from the appropriate pool depending on the proof stage, and processed.

`r.` This repeats the above "q" command until either there remains no conjecture in any of the pools, or the proof eventually ends to success (with a signal "`Q.E.D.`") or failure.

`t.` This invokes "trace" of the Prolog interpreter.

`v.` This enables/disables using the VT100 video options for pretty displaying various informations that the system produces.

1.4. Proof-step Options

In a step of a proof, the system may stop with the following prompt:

```
Option (h for help):
```

At this point the system is expecting input of an option which changes one of proof mode switches, controlls proof steps, or displays the proof environment.

These are called *Proof-step Options* and listed as follows.

```
a       abort
b       break
c       toggle continue mode
e       toggle tracing (rewrite,expand)
g       give up the proof (but store as a lemma)
h       help
p       display remaining conjectures
t       trace
v       toggle VT100 mode
@       accept command
```

Option	Description
a	This invokes "abort" of the Prolog interpreter.
b	This invokes "break" of the Prolog interpreter.
c	This disables the "Option" prompts in the succeeding proof steps.
e	This enables/disables tracing in the simplifier. If this switch is on, all tentative rewriting (applying rewrite lemmas and expanding function definitions) are indicated to the display, nothing otherwise.
g	See section 1.3.

h The help message for proof step options is displayed.

p See section 1.3.

t This invokes "trace" of the Prolog interpreter.

v See section 1.3.

e The command accepts a goal (any Prolog directive) and execute it.

In no option (followed by carriage-return) is specified, the interrupted proof step restarts without changing any of the current proof modes.

2. Using Files

The text of a problem is normally created in a file or a number of files using one of the standard text editors. The BMTP can then be instructed to read-in problems from these files; this is called consulting the file (as in the DECsystem-10 Prolog).

2.1. Problem Files

A problem file is made up of a sequence of top-level commands that is shell definitions, function definitions, axioms, lemmas, theorems to be proved, abbreviation definitions, single letter prover commands, and commands for saving/restoring environments that are descrived further in the followings.

Each command in a problem file is preceded by a # sign, and called a *#-command.*

Example.

```
#in -basic.

#definition append(x,y) =
        if(listp(x),cons(car(x),append(cdr(x),y)),y).

#definition reverse(x) =
        if(listp(x),
           append(reverse(cdr(x)),cons(car(x),nil)),nil).

#definition plistp(x) =
        if(listp(x),plistp(cdr(x)),equal(x,nil)).

#theorem reverse reverse(rewrite) =
        implies(plistp(x),equal(reverse(reverse(x)),x)).

#out revrev.
```

Remember that their order is very impotant.

2.1.1. Reading-in Problem Files

To input a problem from a file *File*, just type the file name inside list brackets (followed by full stop and carriage-return), thus:

| ?- [*File*].

This instructs the Prolog interpreter to read-in (consult) the problem. This is just the same way to consult a Prolog program file.

2.1.2. Additional Prover Commands

To process problems read-in from files, following three additional prover commands are available.

```
x.      display remaining #-commands
y.      process next #-command
z.      repeat y.
```

Command	Description
x.	This displays remaining #-commands.
y.	The next #-command is taken and processed.
z.	This repeats the above "y" command until either there remains no #-commands, or a theorem command is invoked.

2.2. Proof Environment

The system remembers various assertions while processing top-level commands, prover commands, and proof-step options. They make up *Proof Environment.*

Example.

```
bit_place(listp,16).
constructor([add1,A]).
constructor_bottom([add1,A],zero).
constructor_TRs([add1,A],[numberp]).
bottom_object(zero).
accessor([sub1,A]).
type_of_A([numberp,[sub1,A]]).
recognizer([numberp,A]).
recognizer_constructor([numberp,[add1,A]]).
recognizer_bottom([numberp,zero]).
rewrite_lemma([count,[cons,A,B]],
        [add1,[plus,[count,A],[count,B]]],
        t,no,axiom,count(cons)).
induction_lemma([lessp,[count,[cdr,A]],[count,A]],
        [[[listp,A]]],axiom,lessp(cdr)).
elimination_lemma(axiom,elim([car,cdr]),
        [implies,[listp,A],[equal,[cons,[car,A],[cdr,A]],A]]).
definition([fix,A],[if,[numberp,A],A,zero]).
type_prescription([count,A],tp(4,[])).
nonrecursive([zerop,A]).
recursive([plus,A,B]).
measured_subset([plus,A,B],[A]).
induction_template([plus,A,B],template([A,B]/[A],
        [([[[numberp,A]],[[not,[equal,A,zero]]]]->
        [[[sub1,A],B]/[[sub1,A]]])])).
```

2.2.1. Saving Environments

```
| ?- out File.
```

The "out" command writes out a proof environment to the "*File*.env". The file can be read with an "in" command.

2.2.2. Restoring Environments

```
| ?- in File.
```

The "in" command reads in a proof environment from the "*File*.env". The file is possibly created by some "out" command.

If a filename is preceded by a minus sign, as in:

```
| ?- -File.
```

then that file is reconsulted (in the same way as for Prolog programs).

If a saving operation is performed during a proof process, i.e. when there remains some conjectures in the system pools, the process status is also saved into the output file, and can be restored later or in another session of the prover by performing an "in" command to the file.

3. Meta-logical Constructs

The system offers a special feature as to axiom, lemma and theorem commands.

The body supplied to the above three commands is allowed to include Prolog native codes that is to be interpreted during a proof step.

Some more sophisticated way of abbreviations than those introduced in section 1.2.2 are implemented by using this feature.

3.1. Embedding Prolog Codes

As an extention of a term specification for a body of an axiom, lemma, or theorem, an *embedded Prolog code* is introduced.

An embedded Prolog code is a Prolog predicate (a conjunction or disjunction of predicates) delimited by brackets "{" and "}", and is allowed to occur as an argument of the term that is supplied as a body of an axiom, lemma, and theorem command.

(In the current version (v3.6 Nov. 1984), embedded Prolog codes are allowed to occur only in the hypothesis part of "`implies`" terms for lemmas and axioms labeled "`rewrite`".)

Example.

```
| ?- lemma foo(rewrite) =
|            implies(and(listp(x),
|                          {write('foo('),write(x),write(')')})
|                     equal(foo(x),zero)).
```

The rewrite lemma "foo" is applied in a proof of the theorem that mentions the term "`foo(`X`)`" under the assumption that X is "`listp`", rewriting it to "zero" with the side effect of writing "`foo(`X`)`" while establishing the lemma hypothesis.

3.2. Extended Abbreviations

The formal definition of numbers, "cons"ed pairs (or lists), and packed strings by the shell principle forces the system and its users to handle the strict representations such as:

```
zero, add1(zero), add1(add1(zero)), ...
cons(X,nil), cons(X,cons(Y,nil)), ...
pack(cons(65,nil)), pack(cons(65,cons(66,nil))), ...
```

(with appologies for using abbreviations 65 and 66
for sixfty five and sixty six times application of
"add1" on "zero" respectively, which would
consume too much space were they written down!)

instead of:

```
0, 1, 2, ...
[X], [X,Y], ...
"A", "AB", ...
```

which are far more convenient to users.

The latter representations abbreviating the former above are incorporated into the system effectively without losing soundness of proof procedures by using the embedded Prolog codes in rewrite lemmas (axioms) described in the previous section.

Moreover, this method keeps the system from full expansion of large explicit values such as "999999", "[a,b,...,z]", and ""ab...z"", that would happen were it equipped with a simple minded abbreviation expanding procedure, which can be delayed until the time it is really needed.

In the following subsections, some standard methods of the extended abbreviations for the three basic shell objects are introduced.

3.2.1. Abbreviations for numberp objects

For the "numberp" abbreviations 0, 1, 2, ... to work appropriately, the following axioms are added to the system.

```
axiom zero_0(rewrite) =
        equal(0,zero).

axiom sub1_0(rewrite) =
        equal(sub1(0),zero).

axiom sub1_add1_integer(rewrite) =
        implies({integer(x),x>0,x1 is x-1},
                equal(sub1(x),x1)).

axiom add1_sub1_integer(rewrite) =
        implies({integer(x),x>0},
                equal(add1(sub1(x)),x)).

axiom numberp_integer(rewrite) =
        implies({integer(x)},
                numberp(x)).
```

Note that the variable "x" ("x1") is shared between the embedded Prolog codes and the logical terms of the theorey of the system; which is instantiated to the same thing and at the same time in the prover's context.

3.2.2. Abbreviations for listp objects

For the "listp" abbreviations list(*A*), list(*A*,*B*), list(*A*,*B*,*C*), ... to work appropriately, the following axioms are added to the system.

```
axiom car_list(rewrite) =
        implies({x=[list,y|_]},
                equal(car(x),y)).

axiom cdr_list(rewrite) =
        implies({x=[list,_,L1|L],y=[list,L1|L]},
                equal(cdr(x),y)).

axiom list_cons(rewrite) =
        implies({x=[list,X],y=[cons,X,[`,nil]]},
                equal(x,y)).

axiom listp_list(rewrite) =
        implies({x=[list|_]},
                listp(x)).
```

Note that the variable "x" ("y") behaves in the way described in the "numberp" abbreviations; the variable "L" ("L1" or "_") is, however, the usual Prolog variable (shared only among the predicates between "{" and "}").

Note also that a term in the prover's world is represented by the Prolog list structure inside the brackets "{" and "}". Thus the last axiom above actually says that:

```
implies(equal(x,list(a1)),
        listp(x))
implies(equal(x,list(a1,a2)),
        listp(x))
...
implies(equal(x,list(a1,a2,...,an)),
        listp(x))
```

for all integer *n*.

3.2.3. Further Abbreviations

For the abbreviations such as:

`[] for "nil",
`[X,0] for "cons(X,cons(zero,nil))",
`[x,1] for "cons(pack(cons(120,nil)),cons(add1(zero),nil))", etc.
(120 is the ascii code for the lower-case letter "x")

to work appropriately, the following axioms are added to the system.

```
axiom backq_nil(rewrite) =
        equal(`[],nil).

axiom car_backq_list(rewrite) =
        implies({x=[A|_],y=A},
                equal(car(`x),`y)).

axiom cdr_backq_list(rewrite) =
        implies({x=[_|A],y=A},
                equal(cdr(`x),`y)).

axiom listp_backq_list(rewrite) =
        implies({x=[_|_]},
                listp(`x)).

axiom backq_integer(rewrite) =
        implies({integer(x)},
                equal(`x,x)).

axiom numberp_backq_integer(rewrite) =
        implies({integer(x)},
                numberp(`x)).

axiom backq_pack(rewrite) =
        implies({atom(x),name(x,N),y=[list|N]},
                equal(unpack(`x),y)).
```

REFERENCES

[Boyer & Moore 79] Robert S. Boyer and J Strother Moore
A Computational Logic. Academic Press, New York, 1979.

[Boyer & Moore 81] Robert S. Boyer and J Strother Moore
Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, R.S. Boyer and J S. Moore, Eds. Academic Press, London, 1981, pp.103-184.

[Boyer & Moore 84] Robert S. Boyer and J Strother Moore
A Mechanical Proof of the Unsolvability of the Halting Problem. *Journal of the ACM*, Vol.31, No.3, July 1984, pp.441-458.

[Bowen et. al. 83] D.L. Bowen, L. Byrd, F.C.N. Pereira, L.M. Pereira, and D.H.D. Warren
DECsystem-10 PROLOG USER'S MANUAL.
Department of Artificial Intelligence, University of Edinburgh, 1981.

APPENDIX

I. Summary of The Formal System

I.1. Syntax

<identifier> ::= *<lowercase-letter>* [*<lowercase-letter>* | *<digit>* | "_"]*
| "'" [*<any character>*]* "'"

<integer> ::= *<digit>* [*<digit>*]*

<variable> ::= *<identifier>*

<function symbol> ::= *<identifier>*

<term> ::= *<variable>* | *<function symbol>* | *<integer>*
| *<function symbol>* "(" *<term>* [*<term>*]* ")"

<constructor> ::= *<function symbol>*

<accessor> ::= *<function symbol>*

<recognizer> ::= *<function symbol>*

<type restriction> ::= *<function symbol>*

<default value> ::= *<identifier>*

<bottom object> ::= *<function symbol>* | *<integer>*

<function> ::= *<function symbol>*

<formal parameter> ::= *<variable>*

<definition body> ::= *<term>*

<name> ::= *<identifier>*

<label> ::= "rewrite"
| "induction"
| "elimination"
| "generalize"

<embedded Prolog code>
::= "{" *<any Prolog predicate(s)>* "}"

I.2. The Theory of if and equal

axiom $\mathtt{t} \neq \mathtt{f}$
axiom $X = Y \rightarrow \mathtt{equal}(X,Y) = \mathtt{t}$
axiom $X \neq Y \rightarrow \mathtt{equal}(X,Y) = \mathtt{f}$
axiom $X = \mathtt{f} \rightarrow \mathtt{if}(X,Y,Z) = Z$
axiom $X \neq \mathtt{f} \rightarrow \mathtt{if}(X,Y,Z) = Y$

I.3. The Logical Functions

definition $\mathtt{not}(P) = \mathtt{if}(P,\mathtt{f},\mathtt{t})$
definition $\mathtt{and}(P,Q) = \mathtt{if}(P,\mathtt{if}(Q,\mathtt{t},\mathtt{f}),\mathtt{f})$
definition $\mathtt{or}(P,Q) = \mathtt{if}(P,\mathtt{t},\mathtt{if}(Q,\mathtt{t},\mathtt{f}))$
definition $\mathtt{implies}(P,Q) = \mathtt{if}(P,\mathtt{if}(Q,\mathtt{t},\mathtt{f}),\mathtt{t})$

As for "and" and "or", arity is extended to more than two as follows.

definition $\mathtt{and}(P_1,P_2,\ldots,P_n) = \mathtt{and}(P_1,\mathtt{and}(P_2,\ldots,P_n))$
definition $\mathtt{or}(P_1,P_2,\ldots,P_n) = \mathtt{or}(P_1,\mathtt{and}(P_2,\ldots,P_n))$

The axioms and function definitions above are all "wired-in" to the system.

I.4. The Shell Principle

To add the shell constructor C of n arguments $(X_1,\ldots, X_n)$
with (optionally, bottom object BTM),
recognizer R,
accessors $A_1,\ldots, A_n$,
type restrictions $TR_1,\ldots, TR_n$, and
default values $DV_1,\ldots, DV_n$,

where

(a) C is a new function symbol of n arguments, (BTM is a new function symbol of no arguments, if a bottom object is supplied), R, $A_1,\ldots, A_n$ are new function symbols of one argument, and all of the above function symbols are distinct;
(b) each TR_i is a function symbol that is either a previously introduced shell recognizer, or R; and
(c) if no bottom object is supplied, the DV_i are bottom objects of previously introduced shells, and for each i,

$$\texttt{implies(equal(}X_i\texttt{,}DV_i\texttt{),}TR_i(X_i)\texttt{)}$$

is a theorem; if a bottom object is supplied, each DV_i is either BTM or a bottom object of some previously introduced shell, and for each i,

$$\texttt{implies(and(equal(}X_i\texttt{,}DV_i\texttt{),}R(BTM)\texttt{)),}\ TR_i(X_i)\texttt{)}$$

is a theorem,

means to extend the theory by adding the following axioms (using t for $R(BTM)$ and f for all terms of the form equal(X,BTM) if no bottom object is supplied, and using t for each $TR_i(X)$ if the TR_i is not specified):

(1) axioms "wired-in" to the prover:

$$R(C(X_1,\ldots,X_n)),$$

$$R(BTM),$$

```
or(equal(R(X),t),equal(R(X),f)),

not(equal(C(X_1,...,X_n),BTM)),

not(R(t)),

not(R(f));
```

(2) axioms "wired-in" to the prover (for each i from 1 to n):

```
TR_i(A_i(X));
```

(3) axioms "wired-in" to the prover (for each recognizer R' of a shell class previously added to the theory):

```
implies(R(X),not(R',X));
```

(4) axioms used as rewrite lemmas (for each i from 1 to n):

```
equal(AC_i(C(X_1,...,X_n)),
      if(TR_i(X_i),X_i,DV_i)),

implies(not(R(X)),
        equal(A_i(X),DV_i)),

implies(not(TR_i(X_i)),
        equal(C(X_1,...,X_i,...,X_n),
              C(X_1,...,DV_i,...,X_n))),

equal(A_i(BTM),DV_i),

equal(C(A_1(X),...,A_n(X)),
      if(and(R(X),
             not(equal(X,BTM))),
         X,
         C(DV_1,...,DV_n))),
```

```
equal(equal(C(X1,...,Xn),
            C(Y1,...,Yn)),
      and(if(TR1(X1),
             if(TR1(Y1),
                equal(X1,Y1),
                equal(X1,DV1)),
             if(TR1(Y1),
                equal(DV1,Y1),
                t)),
          ...
          if(TRn(Xn),
             if(TRn(Yn),
                equal(Xn,Yn),
                equal(Xn,DVn)),
             if(TRn(Yn),
                equal(DVn,Yn),
                t))));
```

(6) axioms used as rewrite lemmas (assuming that "count" is the basic measure function, and "zero" is the bottom object of the basic "add1" shell):

```
equal(count(C(X1,...,Xn)),
      add1(plus(if(TR1(X),count(X1),zero),
                ...
                if(TRn(X),count(Xn),zero)))),

equal(count(BTM),zero);
```

(7) axioms used as induction lemmas (for each i from 1 to n):

```
implies(and(R(X),
            not(equal(X,BTM))),
        lessp(count(Ai(X)),count(X)));
```

(8) an axiom used as an elimination lemma:

```
implies(and(R(X),
            not(equal(X,BTM))),
        equal(C(A1(X),...,An(X)),
              X)).
```

I.5. The Definition Principle

To define F of $X_1, \ldots, X_n$ to be *Body*,

where

(a) F is a new function symbol of n arguments;
(b) $X_1, \ldots, X_n$ are distinct variables;
(c) *Body* is a term and mentions no symbol as a variable other than $X_1, \ldots, X_n$; and
(d) there is a well-founded relation denoted by a function symbol R and a function symbol M of n arguments, such that for each occurence of a subterm of the form $F(Y_1, \ldots, Y_n)$ in *Body* and the *F-free* terms $T_1, \ldots, T_k$ *governing* it, it is a theorem that:

$$\texttt{implies(and}(T_1, \ldots, T_k),\ R(M(Y_1, \ldots, Y_n), M(X_1, \ldots, X_n)))$$

means to add as an axiom the defining equation:

$$F(X_1, \ldots, X_n) = \mathit{Body}.$$

Terminology

A term is *F-free* if the symbol F does not occur in the term as a function symbol.

A term T *governs* an occurrence of a term S in a term B either if B contains a subterm of the form $\texttt{if}(T, P, Q)$ and the occurrence of S is in P, or if B contains a subterm of the form $\texttt{if}(T', P, Q)$ where T is $\texttt{not}(T')$, and the occurrence of S is in Q.

Example.

$$\begin{array}{l}\texttt{if}(P,\\ \quad \texttt{if}(\texttt{if}(Q, \texttt{f}, S),\\ \quad\quad S,\\ \quad\quad R),\\ \quad T)\end{array}$$

P and $\texttt{not}(Q)$ govern the first occurrence of S, P and $\texttt{if}(Q, \texttt{f}, S)$ govern the second occurrence of S.

I.6. The Basic Environment

The basic environment comprises various informations created from three shells "add1", "pack", and "cons", two nonrecursive functions "zerop" and "fix", and three recursive functions "plus", "lessp", and "count".
These are defined in the file "basic." that would produce the basic environment file "basic.env" when processed by the system.

I.6.1 The Basic File

```
#shell add1( sub1:numberp/zero ):numberp/zero.

#shell pack( unpack/zero ):litatom/nil.

#shell cons( car/nil, cdr/nil ):listp.

#definition zerop(x) =
      or(equal(x,zero), not(numberp(x))).

#definition fix(x) =
      if(numberp(x),x,zero).

#definition plus(x,y) =
      if(zerop(x),
         fix(y),
         add1(plus(sub1(x),y))).

#definition lessp(x,y) =
      if(zerop(y),
        f,
        if(zerop(x),
           t,
           lessp(sub1(x),sub1(y)))).
```

```
#definition count(x) =
      if(numberp(x),
         if(equal(x,zero),
            zero,
            add1(count(sub1(x)))),
         if(litatom(x),
            if(equal(x,nil),
               zero,
               add1(count(unpack(x)))),
            if(listp(x),
               add1(plus(count(car(x)),
                         count(cdr(x)))),
               zero))).

#out basic.
```

I.6.2. The Basic Environment File

```
bit_place(listp,16).
bit_place(litatom,8).
bit_place(numberp,4).
bit_place([t],2).
bit_place([f],1).

constructor([add1,A]).
constructor([pack,A]).
constructor([cons,A,B]).

constructor_bottom([add1,A],zero).
constructor_bottom([pack,A],nil).

constructor_TRs([add1,A],[numberp]).
constructor_TRs([pack,A],[t]).
constructor_TRs([cons,A,B],[t,t]).

bottom_object(zero).
bottom_object(nil).

accessor([sub1,A]).
accessor([unpack,A]).
```

```
accessor([car,A]).
accessor([cdr,A]).

type_of_A([numberp,[sub1,A]]).

recognizer([numberp,A]).
recognizer([litatom,A]).
recognizer([listp,A]).

recognizer_constructor([numberp,[add1,A]]).
recognizer_constructor([litatom,[pack,A]]).
recognizer_constructor([listp,[cons,A,B]]).

recognizer_bottom([numberp,zero]).
recognizer_bottom([litatom,nil]).

rewrite_lemma([count,[cons,A,B]],
        [add1,[plus,[count,A],[count,B]]],t,no,axiom,count(cons)).
rewrite_lemma([cons,[car,A],[cdr,A]],
        [if,[listp,A],A,[cons,nil,nil]],t,no,axiom,[cons,[car,cdr]]).
rewrite_lemma([equal,[cons,A,B],[cons,C,D]],
        [and,[equal,A,C],[equal,B,D]],t,no,axiom,equal(cons)).
rewrite_lemma([cdr,A],nil,[[[not,[listp,A]]]],no,axiom,[cdr,not,listp]).
rewrite_lemma([car,A],nil,[[[not,[listp,A]]]],no,axiom,[car,not,listp]).
rewrite_lemma([cdr,[cons,A,B]],B,t,no,axiom,[cdr,cons]).
rewrite_lemma([car,[cons,A,B]],A,t,no,axiom,[car,cons]).
rewrite_lemma([count,nil],zero,t,no,axiom,count(nil)).
rewrite_lemma([count,[pack,A]],[add1,[count,A]],t,no,axiom,count(pack)).
rewrite_lemma([pack,[unpack,A]],
        [if,[and,[litatom,A],[not,[equal,A,nil]]],A,[pack,zero]],
        t,no,axiom,[pack,[unpack]]).
rewrite_lemma([equal,[pack,A],[pack,B]],[equal,A,B],
        t,no,axiom,equal(pack)).
rewrite_lemma([unpack,nil],zero,t,no,axiom,[unpack,nil]).
rewrite_lemma([unpack,A],zero,[[[not,[litatom,A]]]],
        no,axiom,[unpack,not,litatom]).
rewrite_lemma([unpack,[pack,A]],A,t,no,axiom,[unpack,pack]).
rewrite_lemma([count,zero],zero,t,no,axiom,count(zero)).
rewrite_lemma([count,[add1,A]],
        [add1,[if,[numberp,A],[count,A],zero]],t,no,axiom,count(add1)).
rewrite_lemma([add1,[sub1,A]],
```

```
        [if,[and,[numberp,A],[not,[equal,A,zero]]],A,[add1,zero]],
        t,no,axiom,[add1,[sub1]]).
rewrite_lemma([equal,[add1,A],[add1,B]],
        [if,[numberp,A],[if,[numberp,B],[equal,A,B],[equal,A,zero]],
        [if,[numberp,B],[equal,zero,B],t]],t,no,axiom,equal(add1)).
rewrite_lemma([sub1,zero],zero,t,no,axiom,[sub1,zero]).
rewrite_lemma([add1,A],[add1,zero],[[[not,[numberp,A]]]],
        no,axiom,type_restriction(sub1)).
rewrite_lemma([sub1,A],zero,[[[not,[numberp,A]]]],
        no,axiom,[sub1,not,numberp]).
rewrite_lemma([sub1,[add1,A]],[if,[numberp,A],A,zero],
        t,no,axiom,[sub1,add1]).

induction_lemma([lessp,[count,[cdr,A]],[count,A]],
        [[[listp,A]]],axiom,lessp(cdr)).
induction_lemma([lessp,[count,[car,A]],[count,A]],
        [[[listp,A]]],axiom,lessp(car)).
induction_lemma([lessp,[count,[unpack,A]],[count,A]],
        [[[litatom,A]],[[not,[equal,A,nil]]]],axiom,lessp(unpack)).
induction_lemma([lessp,[count,[sub1,A]],[count,A]],
        [[[numberp,A]],[[not,[equal,A,zero]]]],axiom,lessp(sub1)).

elimination_lemma(axiom,elim([car,cdr]),
        [implies,[listp,A],[equal,[cons,[car,A],[cdr,A]],A]]).
elimination_lemma(axiom,elim([unpack]),
        [implies,[and,[litatom,A],[not,[equal,A,nil]]],
        [equal,[pack,[unpack,A]],A]]).
elimination_lemma(axiom,elim([sub1]),
        [implies,[and,[numberp,A],[not,[equal,A,zero]]],
        [equal,[add1,[sub1,A]],A]]).

definition([zerop,A],[or,[equal,A,zero],[not,[numberp,A]]]).
definition([fix,A],[if,[numberp,A],A,zero]).
definition([plus,A,B],[if,[zerop,A],[fix,B],[add1,[plus,[sub1,A],B]]]).
definition([lessp,A,B],
        [if,[zerop,B],f,[if,[zerop,A],t,[lessp,[sub1,A],[sub1,B]]]]).
definition([count,A],
        [if,[numberp,A],
            [if,[equal,A,zero],zero,[add1,[count,[sub1,A]]]],
        [if,[litatom,A],
```

```
            [if,[equal,A,nil],zero,[add1,[count,[unpack,A]]]],
        [if,[listp,A],
            [add1,[plus,[count,[car,A]],[count,[cdr,A]]],zero]]]).

type_prescription([count,A],tp(4,[])).
type_prescription([lessp,A,B],tp(3,[])).
type_prescription([plus,A,B],tp(4,[])).
type_prescription([fix,A],tp(4,[])).
type_prescription([zerop,A],tp(3,[])).
type_prescription([not,A],tp(3,[])).
type_prescription([and|A],tp(3,[])).
type_prescription([or|A],tp(3,[])).
type_prescription([implies,A,B],tp(3,[])).

nonrecursive([zerop,A]).
nonrecursive([fix,A]).

recursive([plus,A,B]).
recursive([lessp,A,B]).
recursive([count,A]).

measured_subset([plus,A,B],[A]).
measured_subset([lessp,A,B],[A]).
measured_subset([lessp,A,B],[B]).
measured_subset([count,A],[A]).

induction_template([plus,A,B],template([A,B]/[A],
        [([[[numberp,A]],[[not,[equal,A,zero]]]]->
        [[[sub1,A],B]/[[sub1,A]]])])).
induction_template([lessp,A,B],template([A,B]/[A],
        [([[[numberp,A]],[[not,[equal,A,zero]]]]->
        [[[sub1,A],[sub1,B]]/[[sub1,A]]])])).
induction_template([lessp,A,B],template([A,B]/[B],
        [([[[numberp,B]],[[not,[equal,B,zero]]]]->
        [[[sub1,A],[sub1,B]]/[[sub1,B]]])])).
induction_template([count,A],template([A]/[A],
        [([[[litatom,A]],[[not,[equal,A,nil]]]]->
        [[unpack,A]]/[[unpack,A]]),
        ([[[listp,A]]]->[[[car,A]]/[[car,A]],[[cdr,A]]/[[cdr,A]]]),
        ([[[numberp,A]],[[not,[equal,A,zero]]]]->
        [[[sub1,A]]/[[sub1,A]]])])).
```

II. Example: The Theorem reverse-reverse

```
@prove
BMTP in Prolog V3.6 (Nov. 1984)
yes
| ?- [revrev].

revrev consulted   198 words      0.26 sec.

yes
| ?- z.

basic.env reconsulted   4006 words      1.62 sec.

defun reconsulted   6577 words      3.70 sec.

Definition append(x,y)
    =   if(listp(x),cons(car(x),append(cdr(x),y)),y)
is accepted.
Its type is OR(listp,type_of(y)).

Definition reverse(x)
    =   if(listp(x),append(reverse(cdr(x)),cons(car(x),nil)),nil)
is accepted.
Its type is OR(listp,litatom).

Definition plistp(x)
    =   if(listp(x),plistp(cdr(x)),equal(x,nil))
is accepted.
Its type is OR(t,f).

revrev.env told.

defun.dmy reconsulted   -1644 words      0.46 sec.

Theorem reverse_reverse (rewrite)
        implies(plistp(x),equal(reverse(reverse(x)),x))
Option (h for help):
Proving...
Poured a conjecture into the simplification-pool.

yes
| ?- p.
```

```
Conjecture waiting simplification
           theorem reverse_reverse

        implies(plistp(x),equal(reverse(reverse(x)),x))

yes
| ?- q.

Conjecture theorem reverse_reverse

        implies(plistp(x),equal(reverse(reverse(x)),x))

Simplifying...

Poured a conjecture into the heuristics-pool.

yes
| ?- q.

Conjecture theorem reverse_reverse

        implies(plistp(x),equal(reverse(reverse(x)),x))

Trying heuristic rewrite...

Poured a conjecture into the induction-pool.

yes
| ?- q.

Conjecture theorem reverse_reverse

        implies(plistp(x),equal(reverse(reverse(x)),x))

Trying induction...

The scheme:

        and(implies(not(listp(x)),P(x)),
            implies(and(listp(x),P(cdr(x))),P(x)))

which accounts for

        reverse(x)

is subsumed by the scheme:

        and(implies(not(listp(x)),P(x)),
            implies(and(listp(x),P(cdr(x))),P(x)))

which accounts for

        plistp(x)

getting a new scheme:

        and(implies(not(listp(x)),P(x)),
```

```
            implies(and(listp(x),P(cdr(x))),P(x)))

which accounts for

        (plistp(x),reverse(x))

Option (h for help):

We will induct according to the following scheme

        and(implies(not(listp(x)),P(x)),
            implies(and(listp(x),P(cdr(x))),P(x)))

which accounts for

        (plistp(x),reverse(x))

Base Case

Poured a conjecture into the simplification-pool.

Induction Step

Poured 2 conjectures into the simplification-pool.

yes
| ?- p.

Conjecture waiting simplification
           base case
           theorem reverse_reverse

        implies(and(not(listp(x)),plistp(x)),
                equal(reverse(reverse(x)),x))

Conjecture waiting simplification
           induction step-1
           theorem reverse_reverse

        implies(and(listp(x),not(plistp(cdr(x))),plistp(x)),
                equal(reverse(reverse(x)),x))

Conjecture waiting simplification
           induction step-2
           theorem reverse_reverse

        implies(
           and(listp(x),
               equal(reverse(reverse(cdr(x))),cdr(x)),
               plistp(x)),
           equal(reverse(reverse(x)),x))

yes
| ?- r.

Conjecture base case
```

```
           theorem reverse_reverse

        implies(and(not(listp(x)),plistp(x)),
                equal(reverse(reverse(x)),x))

Simplifying...

Option (h for help): c

Under the assumption

        and(not(listp(x)),not(equal(reverse(reverse(x)),x)))

expanding plistp(x)

we rewrite the literal

        not(plistp(x))
to
        not(equal(x,nil))

Under the assumption

        equal(x,nil)

expanding reverse(x)
expanding reverse(nil)

the literal

        equal(reverse(reverse(x)),x)

is true.

The conjecture is true.

Conjecture induction step-1
           theorem reverse_reverse

        implies(and(listp(x),not(plistp(cdr(x))),plistp(x)),
                equal(reverse(reverse(x)),x))

Simplifying...

Under the assumption

        and(not(plistp(cdr(x))),
            listp(x),
            not(equal(reverse(reverse(x)),x)))

expanding plistp(x)

the literal

        not(plistp(x))

is true.
```

```
The conjecture is true.

Conjecture induction step-2
           theorem reverse_reverse

         implies(
            and(listp(x),
                equal(reverse(reverse(cdr(x))),cdr(x)),
                plistp(x)),
            equal(reverse(reverse(x)),x))

Simplifying...

Under the assumption

         and(equal(reverse(reverse(cdr(x))),cdr(x)),
             listp(x),
             not(equal(reverse(reverse(x)),x)))

expanding plistp(x)

we rewrite the literal

         not(plistp(x))
to
         not(plistp(cdr(x)))

Under the assumption

         and(plistp(cdr(x)),
             equal(reverse(reverse(cdr(x))),cdr(x)),
             listp(x))

expanding reverse(x)

we rewrite the literal

         equal(reverse(reverse(x)),x)
to
         equal(reverse(append(reverse(cdr(x)),cons(car(x),nil))),x)

Poured a conjecture into the simplification-pool.

Conjecture simplified
           induction step-2
           theorem reverse_reverse

         implies(
            and(listp(x),
                equal(reverse(reverse(cdr(x))),cdr(x)),
                plistp(cdr(x))),
            equal(reverse(append(reverse(cdr(x)),cons(car(x),nil))),x))

Simplifying...
```

```
Poured a conjecture into the heuristics-pool.

Conjecture simplified
           induction step-2
           theorem reverse_reverse

        implies(
           and(listp(x),
               equal(reverse(reverse(cdr(x))),cdr(x)),
               plistp(cdr(x))),
           equal(reverse(append(reverse(cdr(x)),cons(car(x),nil))),x))

Trying heuristic rewrite...

We now replace x by cons(a,b)

to eliminate (car(x),cdr(x))

Poured a conjecture into the simplification-pool.

Conjecture destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           and(equal(reverse(reverse(b)),b),plistp(b)),
           equal(reverse(append(reverse(b),cons(a,nil))),cons(a,b)))

Simplifying...

Poured a conjecture into the heuristics-pool.

Conjecture destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           and(equal(reverse(reverse(b)),b),plistp(b)),
           equal(reverse(append(reverse(b),cons(a,nil))),cons(a,b)))

Trying heuristic rewrite...

We now use the above equality hypothesis by cross-fertilizing

        reverse(reverse(b))
for
        b

and throwing away the equality.

Poured a conjecture into the simplification-pool.

Conjecture used an equality
```

```
            destructor(s) replaced by [a,b]
            simplified
            induction step-2
            theorem reverse_reverse

         implies(
            plistp(b),
            equal(reverse(append(reverse(b),cons(a,nil))),
                  cons(a,reverse(reverse(b)))))

Simplifying...

Poured a conjecture into the heuristics-pool.

Conjecture used an equality
            destructor(s) replaced by [a,b]
            simplified
            induction step-2
            theorem reverse_reverse

         implies(
            plistp(b),
            equal(reverse(append(reverse(b),cons(a,nil))),
                  cons(a,reverse(reverse(b)))))

Trying heuristic rewrite...

We generalize the conjecture by replacing

         reverse(b)
by
         c

Poured a conjecture into the simplification-pool.

Conjecture generalized with [c]
            used an equality
            destructor(s) replaced by [a,b]
            simplified
            induction step-2
            theorem reverse_reverse

         implies(
            plistp(b),
            equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c))))

Simplifying...

Poured a conjecture into the heuristics-pool.

Conjecture generalized with [c]
            used an equality
            destructor(s) replaced by [a,b]
            simplified
```

```
            induction step-2
            theorem reverse_reverse

         implies(
            plistp(b),
            equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c))))

Trying heuristic rewrite...

We eliminate the irrelevant term(s)

         plistp(b)

Poured a conjecture into the simplification-pool.

Conjecture irrelevance(s) eliminated
            generalized with [c]
            used an equality
            destructor(s) replaced by [a,b]
            simplified
            induction step-2
            theorem reverse_reverse

         equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c)))

Simplifying...

Poured a conjecture into the heuristics-pool.

Conjecture irrelevance(s) eliminated
            generalized with [c]
            used an equality
            destructor(s) replaced by [a,b]
            simplified
            induction step-2
            theorem reverse_reverse

         equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c)))

Trying heuristic rewrite...

Poured a conjecture into the induction-pool.

Conjecture irrelevance(s) eliminated
            generalized with [c]
            used an equality
            destructor(s) replaced by [a,b]
            simplified
            induction step-2
            theorem reverse_reverse

         equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c)))

Trying induction...
```

```
The scheme:

        and(implies(not(listp(c)),P(c)),
            implies(and(listp(c),P(cdr(c))),P(c)))

which accounts for

        append(c,cons(a,nil))

is subsumed by the scheme:

        and(implies(not(listp(c)),P(c)),
            implies(and(listp(c),P(cdr(c))),P(c)))

which accounts for

        reverse(c)

getting a new scheme:

        and(implies(not(listp(c)),P(c)),
            implies(and(listp(c),P(cdr(c))),P(c)))

which accounts for

        (reverse(c),append(c,cons(a,nil)))

We will induct according to the following scheme

        and(implies(not(listp(c)),P(c)),
            implies(and(listp(c),P(cdr(c))),P(c)))

which accounts for

        (reverse(c),append(c,cons(a,nil)))

Base Case

Poured a conjecture into the simplification-pool.

Induction Step

Poured a conjecture into the simplification-pool.

Conjecture base case
           irrelevance(s) eliminated
           generalized with [c]
           used an equality
           destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           not(listp(c)),
           equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c))))
```

```
Simplifying...

Under the assumption

        not(listp(c))

expanding append(c,cons(a,nil))
expanding reverse(cons(a,nil))
   applying the axiom: [cdr,cons]
   expanding reverse(nil)
   applying the axiom: [car,cons]
   expanding append(nil,cons(a,nil))
expanding reverse(c)

the literal

        equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c)))

is true.

The conjecture is true.

Conjecture induction step
           irrelevance(s) eliminated
           generalized with [c]
           used an equality
           destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           and(
              listp(c),
              equal(reverse(append(cdr(c),cons(a,nil))),
                    cons(a,reverse(cdr(c))))),
           equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c))))

Simplifying...

Under the assumption

        and(
           equal(reverse(append(cdr(c),cons(a,nil))),
                 cons(a,reverse(cdr(c)))),
           listp(c))

expanding append(c,cons(a,nil))
expanding reverse(cons(car(c),append(cdr(c),cons(a,nil))))
   applying the axiom: [cdr,cons]
   applying the axiom: [car,cons]
expanding reverse(c)

we rewrite the literal
```

```
        equal(reverse(append(c,cons(a,nil))),cons(a,reverse(c)))
to
        equal(
           append(reverse(append(cdr(c),cons(a,nil))),
                  cons(car(c),nil)),
           cons(a,append(reverse(cdr(c)),cons(car(c),nil))))

Poured a conjecture into the simplification-pool.

Conjecture simplified
           induction step
           irrelevance(s) eliminated
           generalized with [c]
           used an equality
           destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           and(
              listp(c),
              equal(reverse(append(cdr(c),cons(a,nil))),
                    cons(a,reverse(cdr(c)))),
           equal(
              append(reverse(append(cdr(c),cons(a,nil))),
                     cons(car(c),nil)),
              cons(a,append(reverse(cdr(c)),cons(car(c),nil)))))

Simplifying...

Poured a conjecture into the heuristics-pool.

Conjecture simplified
           induction step
           irrelevance(s) eliminated
           generalized with [c]
           used an equality
           destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           and(
              listp(c),
              equal(reverse(append(cdr(c),cons(a,nil))),
                    cons(a,reverse(cdr(c)))),
           equal(
              append(reverse(append(cdr(c),cons(a,nil))),
                     cons(car(c),nil)),
```

```
            cons(a,append(reverse(cdr(c)),cons(car(c),nil)))))

Trying heuristic rewrite...

We now replace c by cons(d,e)

to eliminate (car(c),cdr(c))

Poured a conjecture into the simplification-pool.

Conjecture destructor(s) replaced by [d,e]
           simplified
           induction step
           irrelevance(s) eliminated
           generalized with [c]
           used an equality
           destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           equal(reverse(append(e,cons(a,nil))),cons(a,reverse(e))),
           equal(append(reverse(append(e,cons(a,nil))),cons(d,nil)),
                 cons(a,append(reverse(e),cons(d,nil)))))

Simplifying...

Poured a conjecture into the heuristics-pool.

Conjecture destructor(s) replaced by [d,e]
           simplified
           induction step
           irrelevance(s) eliminated
           generalized with [c]
           used an equality
           destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        implies(
           equal(reverse(append(e,cons(a,nil))),cons(a,reverse(e))),
           equal(append(reverse(append(e,cons(a,nil))),cons(d,nil)),
                 cons(a,append(reverse(e),cons(d,nil)))))

Trying heuristic rewrite...

We now use the above equality hypothesis by cross-fertilizing

        cons(a,reverse(e))
for
        reverse(append(e,cons(a,nil)))
```

```
and throwing away the equality.

Poured a conjecture into the simplification-pool.

Conjecture used an equality
           destructor(s) replaced by [d,e]
           simplified
           induction step
           irrelevance(s) eliminated
           generalized with [c]
           used an equality
           destructor(s) replaced by [a,b]
           simplified
           induction step-2
           theorem reverse_reverse

        equal(append(cons(a,reverse(e)),cons(d,nil)),
              cons(a,append(reverse(e),cons(d,nil))))

Simplifying...

expanding append(cons(a,reverse(e)),cons(d,nil))
   applying the axiom: [car,cons]
   applying the axiom: [cdr,cons]

the literal

        equal(append(cons(a,reverse(e)),cons(d,nil)),
              cons(a,append(reverse(e),cons(d,nil))))

is true.

The conjecture is true.

Q.E.D.

Theorem reverse_reverse (rewrite)

        implies(plistp(x),equal(reverse(reverse(x)),x))

is added.

yes
| ?- halt.

[ Prolog execution halted ]

EXIT
@
```

III. Commands Summary

III.1. Top-level Commands

`shell(`*S*`:`*R*`/`*BTM*`).`	define shell
`definition(`*F*`=`*B*`).`	define function
`lemma(`*H*`=`*B*`).`	add lemma
`axiom(`*H*`=`*B*`).`	add axiom
`theorem(`*H*`=`*B*`).`	prove theorem
`abbrev(`*X*`=`*Y*`).`	define abbreviation

III.2. Environmental Commands

`[`*F*`].`	consult problem file
`in(`*F*`).`	input environment file
`out(`*F*`).`	output environment file

III.3. Prover Commands

`b.`	break
`c.`	toggle continue mode
`e.`	toggle tracing (rewrite,expand)
`g.`	give up the proof (but store as a lemma)
`h.`	help
`p.`	display remaining conjectures
`q.`	process next conjecture
`r.`	repeat q.
`t.`	trace
`v.`	toggle VT100 mode
`x.`	display remaining #-commands
`y.`	process next #-command
`z.`	repeat y.

III.4. Proof-step Options

`a`	abort
`b`	break
`c`	toggle continue mode
`e`	toggle tracing (rewrite,expand)
`g`	give up the proof (but store as a lemma)
`h`	help
`p`	display remaining conjectures
`t`	trace
`v`	toggle VT100 mode
`@`	accept command