December 1969

USER'S GUIDE TO QA3.5 QUESTION-ANSWERING SYSTEM

by

Thomas D. Garvey
Robert E. Kling

Artificial Intelligence Group
Technical Note 15
SRI Project 7494

This research was sponsored by the Advanced Research Projects Agency and the Rome Air Development Center under Contract F30602-69-C-0056.

USER'S GUIDE TO QA3.5 QUESTION-ANSWERING SYSTEM

1. Introduction

QA3.5 is a question-answering system based on a first-order predicate calculus theorem prover using Robinson's resolution principle. The system is made up of about 240 LISP (written in BBN LISP on the SDS 940) functions, most of which run under QAS, the executive function.

The executive contains provisions for changing the strategies, tracing proofs, unwinding proofs (i.e., printing out only those steps that lead directly to a proof), stepping through proofs by hand, and many different operations on the axiom base.

Many facilities make it easier for the user to perform operations on his data base. This appendix is a compilation of documentation relating to the use of QA3.5. (Portions of it had been previously prepared by C. Green, R. Kling, A. Robinson, and R. Yates.)

The program has proved theorems in group theory, number theory, geometry, and algebra; it has solved "real-time" robot problems; it has been used to draw inferences from data bases containing several hundred axioms. It is hoped that this documentation will allow even more extensive use of QA3.5.

2. Internal Representations

In general, the resolution theorem prover attempts to prove the unsatisfiability of the conjunction of a set of clauses. These clauses originate from the axioms and the negation of the theorem entered by the user.

Data types are terms, literals, and clauses and are defined (and represented) as outlined in the following subsections.
a. **Terms**

A term is either a variable (individual symbol) or a function symbol followed by an ordered list of terms. The QA3.5-LISP representation of a term is the following type of S-expression.

(1) **Individual symbol**--represented by a LISP atom.

(2) **Function symbol followed by a list of arguments**--represented by a LISP list \((f \ t_1 \ t_2 \ldots \ t_n)\). The first element is the function symbol (a LISP atom). The remainder of the list is the list of terms \((t_1 \ t_2 \ldots \ t_n)\) representing the arguments.

(3) **Constants**--a constant is a function symbol followed by the null list of terms. If \(c\) is a constant symbol, as a term it would be represented as \((c)\).

**Example:** The term \(f(x, g(x, g(y, z)), c)\) would be represented as

\[
(f \ x \ (g \ x \ (g \ y \ z)) \ (c))
\]

b. **Atoms and Literals**

An **atom** is a predicate letter followed by list of terms. A **literal** is either an atom or a negation sign followed by an atom.

(1) **Predicate letter**--predicate letters are represented as positive integers.

(2) **Atoms**--an atom is represented as a list \((p \ t_1 \ldots \ t_n)\) where \(p\) is the predicate letter and where \((t_1 \ldots \ t_n)\) is a list of terms.

(3) **Literals**--a literal of the form \(-A\), where \(A\) is an atom and is represented as a list \((-p \ t_1 \ldots \ t_n)\), where \(-p\) is the negative of the number \(p\) representing the predicate letter of the atom.
c. **Clauses**

A clause is a disjunction of literals. It is represented as the list \( (\text{HDR} \; l_1 \; l_2 \ldots \; l_n) \), where \( l_1 \ldots l_n \) are the (representations of the) literals of the clause. HDR is a list containing information relevant to the clause. It has the form \( \text{HDR} = (L \; \text{level hist answer props}) \).

1. \( L \) is a list containing the T-support status of the clause plus information pertaining to what clauses have not yet been resolved with \( c \). So \( L = (T \; \text{supp units 2-clauses 3-clauses} \ldots) \).
   
   a. \( T \)-supp is \( \text{True}(T) \) if \( c \) has T-support and \( \text{False}(NIL) \) otherwise.
   
   b. \( \text{units} = (u_{n_1} \; . \; u_{n_2}) \) where:
      
      i. \( u_{n_1} = (l_i \ldots l_n) \) is a pointer into the literals of clause \( c \)
      
      ii. \( u_{n_2} = (u_j \ldots u_n \; \text{END}) \) is a pointer into the list of all unit clauses filed on the array CLAUSEARRAY (described below).
   
   c. \( j \)-clauses = \( (c_1 c_2 \ldots c_k \; \text{END}) \) is a pointer into the list on CLAUSEARRAY of all clauses of length \( j \). If \( j \)-clauses is NIL then, by default, clause \( c \) has not yet been resolved against any \( j \)-clauses.

2. **Level** is the level of the clause defined as 0 for original clauses (axioms and the negation of the theorem) and \( 1 + \max (\lambda(c_1), \lambda(c_2)) \), for \( c \) where \( c \) is a resolvent of \( c_1 \) and \( c_2 \) and where \( \lambda(c_1) \) is the level of clause \( c \).

3. **Hist** has several forms, depending on the type of clause.
   
   a. If \( c \) is an axiom, then
      
      \( \text{HIST} = (\text{AXIOM} \; (p_1 \ldots p_n) \; (f_1 \ldots f_k) \; s) \)
      
      where \( s \) is the original WFF from which the clause \( c \) was derived
      
      \( (p_1 \ldots p_n) \) is a list of the distinct predicate letters of \( s \).
(f₁ ... fₙ) is a list of the distinct function symbols in s.

(b) If c is the negation of the question being asked, then HIST = (NTH).

(c) If c is the resolvent of clauses c₁ and c₂ on literals l₁ of c₁ and l₂ of c₂, then HIST = (RES c₁ l₁ c₂ l₂).

(d) If c is a factor of clause c₁ on literals l₁ and l₂, then HIST = (FAC c₁ l₁ l₂).

(4) Answerc is the answer-clause being built up by the theorem prover.

(5) Props is the property list associated with each clause. It has the form

   ((prop₁ . val₁)
    (prop₂ . val₂) ...). This is used to store various properties (such as ROLE, KILL, etc. described below) with each clause.

3. Memory

For greater efficiency, the clause storage in QA3.5 is separated into two stages: CLAUSEARRAY and MEMARRAY.

a. CLAUSEARRAY

CLAUSEARRAY is an array containing the clauses that the theorem prover is currently using. Its structure is as follows:

![CLAUSEARRAY Diagram]

TA-7494-49
(1) Clauses are filed on CLAUSEARRAY by length. Clauses of length n (including the header) will go on the list of n-clauses, which is the n_th element of CLAUSEARRAY. Hence, unit-clauses have length 2 by this convention and go on the list in the second element of CLAUSEARRAY.

(2) CMAX is an integer that references the first unused location on CLAUSEARRAY. Thus CMAX is greater than the length of any clause on the array. The first element of CLAUSEARRAY contains this maximum CMAX.

(3) Each list (NIL c₁ ... cₓ END) begins with a NIL and ends with the atom END. The remaining elements are clauses of the same length as each other.

   Note: CLAUSEARRAY is a global variable bound by the function START () and should not in general be tampered with. START () should not be called unless a fresh symbolic version of QA3.5 has been loaded.

b. MEMARRAY

MEMARRAY is an array containing all the clauses entered as axioms by the user. Every predicate letter currently in use appears on this array: The position on the array is the internal numerical representation given the predicate letter. Following the predicate letter is a pointer to an array (whose structure is the same as that of CLAUSEARRAY) of all the clauses in memory containing that predicate letter.

Axioms appear in MEMARRAY with the T-support marker set to T, as a bookkeeping device. When the axiom is fetched and used during a proof, this T-support is set to NIL as required, and the axiom together
with a pointer to its position in the subarray of MEMARRAY are stored on the list MCLAUSES so that the T-support marker can be restored after the proof.

MEMARRAY can be set to NIL either by executing the command RESET under the function QAS or else the top-level command (RSETMEMARRAY) - a function of no arguments.

4. Input

Inputs to QA3.5 are axioms and the theorem to be proved. These must be well-formed formulas (WFF's) in the first-order predicate logic. WFF's are entered in prefix form with the following QA3.5 operators allowed:

<table>
<thead>
<tr>
<th>Logical operator</th>
<th>QA3.5 operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>\neg</td>
<td>NOT</td>
</tr>
<tr>
<td>\land</td>
<td>AND</td>
</tr>
<tr>
<td>\lor</td>
<td>OR</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>IF,IMP</td>
</tr>
<tr>
<td>\leftrightarrow</td>
<td>IFF,EQV</td>
</tr>
<tr>
<td>\forall</td>
<td>FA</td>
</tr>
<tr>
<td>\exists</td>
<td>EX</td>
</tr>
</tbody>
</table>
NOT is followed by one argument: the expression that is to be negated; AND (OR) is followed by any number of arguments, which are the expressions that are conjoined (disjoined).

IF and IMP are followed by two arguments: the antecedent and the consequent. Likewise, IFF and EQV must get two arguments: the two expressions that are equivalent.

FA and EX must also be followed by two arguments, the first of which is a list of quantified variables, the second is the statement over which the quantification takes place. Any input statement that violates these rules is not a WFF and will not be accepted by the system.

As an example, the logical expression

\((\forall x,y \ (F(x) \land P(y)) = (\exists z \ (\neg Q(x,y) \lor P(z))))\)

would be typed to QAS as

\((\text{FA } (X \ Y) \ (\text{IF } (\text{AND } (F \ X)(P \ Y)))(\text{EX } (Z)(\text{OR } (\text{NOT}(Q \ X \ Y))(P \ Z))))\).

When a WFF is entered, the system does a certain amount of translation (prenexing) and generates the HDR.

5. QAS Commands

Typing QAS(FILE) to LISP will cause the QAS executive to take control. If FILE is NIL (or missing), QAS will expect commands to be entered from the teletype; otherwise, QAS will get the appropriate file and read commands from it.

QAS allows the user to specify many different commands from several basic types.

The commands (with their arguments) under their general groupings are listed below.
a. **Axiom Entering**

S WFF—if WFF is well-formed, QAS enters it into MEMARRAY as an axiom; otherwise, an error message is generated.

SS WFF ROLE—Operates like S, except that it assigns a ROLE to the property list of the axiom; if ROLE is NIL, then the axiom is parsed according to the following format:

<table>
<thead>
<tr>
<th>INPUT FORM</th>
<th>ROLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[x_1;...;x_n]</td>
<td>FACT</td>
</tr>
<tr>
<td>p[x_1;...;x_n;SI]</td>
<td>INITIAL-STATE</td>
</tr>
<tr>
<td>\forall [x_1;...;x_n] \ wff_1 \Rightarrow p[x_j;...;x_n]</td>
<td>(SUFFCOND P)</td>
</tr>
<tr>
<td>\forall [x_1;...;x_n] \ wff_1 \Rightarrow \exists y \ p[x_j;...;y;...;x_n]</td>
<td>(EXISTENCE P)</td>
</tr>
<tr>
<td>\forall [x_1;...;x_n] \ p[x_1;...;x_j] \Rightarrow wff_1</td>
<td>(CONSEQUENT P)</td>
</tr>
<tr>
<td>\forall [x_1;...;x_n] \ p[x_1;...;x_j] \Rightarrow q[x_{j+1};...;x_n]</td>
<td>(CONSEQUENT Q P)</td>
</tr>
<tr>
<td>\forall [x_1;...;x_n] \ p[x_1;...;x_n] \Rightarrow q[x_1;...;x_n]</td>
<td>(SUBSET P Q)</td>
</tr>
<tr>
<td>wff_1 \Rightarrow p[x_1;...;x_j] is split into (a) \forall [x_1;...;x_n] \ p[x_1;...;x_j] \Rightarrow wff_1</td>
<td>(DEFINITION P)</td>
</tr>
<tr>
<td>and (2) \forall [x_1;...;x_n] \ wff_1 \Rightarrow p[x_1;...;x_j]</td>
<td>(NEOCOND P)</td>
</tr>
</tbody>
</table>

any other WFF

The ROLE is printed out by LISTSENT (LIST within QAS) and UNWIND, as well as entered on the proof tree (see below).

AX AXNAME—Takes the name of an axiom previously put on AXIOMLIST (by AXIOMS), fetches the axiom, prints it on the teletype, and enters it into MEMARRAY; if the AXNAME is not on the list, then a message to that effect is typed out.

AXL AXNAMELIST—Similar to AX, but takes a list of axiom names and performs the steps for AX for each name. (These last two commands form one of the two types of axiom-naming available in the system.)
b. **Theorem Entering**

**Q WFF**—This command enters a WFF into CLAUSEARRAY and attempts to prove the WFF from axioms in MEMARRAY. If there are any existentially qualified variables in WFF, an attempt is made to generate an answer clause during a proof. This returns YES if the WFF is satisfied by the axioms, NO if the WFF is not satisfied by the axioms, NO PROOF FOUND if various bounds are exceeded in the course of the proof. If a proof is found and an answer clause is generated, the answer is printed.

**TQ WFF**—Exactly like Q WFF, except that it prints out the tracing of the proof as it goes; this does not print out generated clauses that are subsumed, but it can be made to do so by setting TR2 to T.

**AQ WFF**—Like Q WFF, except that it finds all answers to an existentially quantified WFF.

**PROVE THNAME**—Finds the WFF corresponding to THNAME on AXIOMLIST, prints it, and attempts to prove it in a fashion exactly similar to Q.

**NPROVE THNAME**—Exactly like PROVE, but negates the theorem before attempting the proof.

**TPROVE THNAME**—Like PROVE, but the proof is traced.

**TNPROVE THNAME**—Like NPROVE, with proof-tracing.

**APROVE THNAME**—Like PROVE, but attempts to find all answers to the question.

**ANPROVE THNAME**—Like APROVE, but the theorem is negated before the proof is attempted.

c. **Clause Deletion**

**RESET**—Clears MEMARRAY of all clauses.

**FORGET P N**—Deletes the Nth axiom stored under predicate P. If N is ALL, all axioms stored under P are deleted.
FORGETC P N--Deletes the Nth clause stored under predicate P.

d. Clause Listing

LIST P--Lists all axioms in MEMARRAY stored under the predicate P.

LISTC P--Lists all clauses in MEMARRAY stored under the predicate P.

LISTN NAME--Prints the clause named NAME (this is used with axiom naming).

e. Input and Output to Permanent Storage

AXIOMS P
AXN1 AX1
AXN2 AX2

AXNM AXM--Builds up AXIOMLIST (which is an association-list). If P is T, AXIOMLIST is first set to NIL and then pairs of the form (AXNI • AXI) are inserted into it. If P is NIL, the pairs are placed at the front of the old AXIOMLIST. AXIOMLIST is a global variable.

WRITE file--Writes the commands or S-expressions following file and up to but not including STOP into file. If file is a legitimate external file name, then that file is used; otherwise, the file is on the property list of atom file.

CREATE file--Puts statements in MEMARRAY into file in the format (S AX1 S AX2 ...); file is added to the QAS file directory.

RUN file--Reads commands from file and executes them.

QAS--Sets file to NIL and begins reading from the teletype.

EDIT file--Allows editing of file using the LISP editor.

FILES--Lists the current file directory.

TREE treename--Saves the current proof tree on the atom treename.
f. **Status**

**STATUS**--Prints out the setting of various indicators in the system.

**Q4STATUS**--Prints out some of the more esoteric system variables.

g. **Proof Output**

**UNWIND**--Prints out the proof, along with a set of statistics indicating the amount of work that was done in the course of the proof.

h. **Strategies**

**STRATEGY strat model**--Allows the user to choose a search strategy. If **AF** is specified, the ancestry-filter strategy will be used (and no model should be typed in). If **strat** is **MODEL**, then **model** must be specified; **model** can be specified as **ANL** (all negative literals), **APL** (all positive literals), or a list of positive and negative literals. **Strat** can be a list of **AF** and **MODEL** in order to use a combination of the strategies. **TSUPP** is always used despite what other strategies are specified. If **strat** is **NIL**, then the strategy used will be **TSUPP** and unit-preference.

i. **Miscellaneous**

**COMMENT**--Allows the user to type in comments in the proof.

**EXIT**--Returns control to **EVALQUOTE**, and leaves **QAS**.

**CONTINUE**--Used when the proof terminates because **MAXLEV** was reached and sets **MINLEV** to the current value of **MAXLEV**, and **MAXLEV** to **MAXLEV** + 2; then continues the proof.

6. **Other Capabilities**

**QA3.5** has several facilities that are handled by using the **E** command to **QAS** rather than by direct commands; some of the features are listed below and discussed in subsequent subsections.

(1) **Property Lists** may be associated with any clause. A set of functions for manipulating these lists has been written.
(2) **Axiom and Clause Naming**—A name can be associated with each axiom, and a mnemonically similar name associated with each of its clauses.

(3) **Predicate and Function Evaluation**—The user can associate with logical predicates and function a LISP function that will "evaluate" a literal and return T, NIL, a literal, or a value. The evaluation often dramatically improves the search time for a proof.

(4) **Syntactic Symmetries**—Two forms have been added to the unification and subsumption tests to enable resolution between literals that have permuted arguments or a list of identical (unordered) arguments.

(5) **Human-Directed Proof Guidance**—Several functions have been written to enable a user to "step" through a proof by attempting specified resolutions, as well as "killing" undesired clauses.

(6) **Statistics**—Certain interesting memory statistics can be printed out.

(7) **MEMARRAY I/O**—New functions have been written to simplify memory listings and to dump out and read in the complete MEMARRAY.

(8) **Clause Size**—The maximum length of clauses handled by the system is now a variable.

(9) **Memory Map Functions**—Simple functions have been written to apply an arbitrary LISP form to each clause in MEMARRAY, a subarray, or CLAUSEARRAY, and return a value.

(10) **Factoring**—A factor-checking function and answer-factoring feature have been written.

(11) **Stopping a Proof**—A proof can be stopped and resumed later.
a. **Property Lists**

A clause (MEMARRAY axiom or QA3 resolvent) can have a property list. Six standard items are stored by position or computed directly and referenced by the following flags: literals of a clause (LITS), T-support (TSUPP), level (LEVEL), history (HIST), length of a clause (LENGTH), and answer clause (ANSWERC). Other items are stored with an associated flag on the property list.

\[
\text{CLAUSE-FORM} \leftarrow (((\text{TSUPP LEVEL HIST ANSWERC PROPERTY-LIST}) \text{lit}_1 \ldots \text{lit}_k)\\
\text{HEADER} = \text{CAR}(C) \quad \text{LITERALS} = \text{CDR}(C)
\]

\[
\text{PROPERTY-LIST} \leftarrow ((\text{prop}_1, \text{val}_1)(\text{prop}_2, \text{val}_2)\ldots)
\]

Use:

1. \( \text{ptc}[c; \text{prop}; \text{val}] \) places the value \( \text{val} \) under the flag \( \text{prop} \) on the property list (in the header) of \( c \).

\( \text{gtc}[c; \text{prop}] \) retrieves the value of \( \text{prop} \).

\( \text{ftc}[c; \text{prop}] \) deletes the pair \( (\text{prop}, \text{val}) \). Only flagged items can be deleted.

The preceding functions call \( \text{ptc}[c; \text{prop}; \text{val}] \), \( \text{gtc}[c; \text{prop}; \text{val}] \), and \( \text{ftc}[c; \text{prop}; \text{val}] \) to deal with flagged properties.

2. A clause \( c \) in the preceding function is represented explicitly. If clause naming is in effect (see below), one can call \( \text{eval}[\text{ptc}\{\text{name} ; \text{prop} ; \text{val}\}] \).

If a user breaks into a proof and wishes to mark a clause numbered \( N \) in the search, he can call \( (\text{PTC}(\text{CAR}(\text{LASSOC N TRLIST})) \text{prop} \text{val})) \).
(3) If one does not have clause naming, the following functions perform the same effect:

\[
\text{putc}[\text{predletter}; n; \text{prop}; v1] \\
\text{getc}[\text{predletter}; n; \text{prop}] \\
\text{fgtc}[\text{predletter}; n; \text{prop}]
\]

where \( n \) is the number of the desired clause as listed (LSTC or listclauses[predletter]) under the predicate letter. Each of these uses the function nthclause[predletter;n] which returns the \( n \)th clause listed.

(4) Many of the features described below use property lists—including axiom naming, parsing, and proof guidance.

b. **Axiom and Clause Naming**

A user may specify a name for an axiom if it is of the form AXN, or he may have the system generate an axiom name of similar form for an axiom input via ENTER. Also, each clause in the prenex version of the axiom can be assigned a name of the form AXN-J. Thus AX10, an axiom yielding three clauses, can generate AX10-1, AX10-2, and AX10-3. Each clause name is stored under the clause property list (see below) under the flag NAME. Each WFF or clause is the value of the appropriate atom. In case of the clause, the atom value, e.g., CAR(AX10-2), accesses (HDR lit \_1 lit \_2 ... lit \_j)—a list of axiom names is stored under the atom AXNAMELIST.

Use:

(1) For axiom naming, AXNAMING = T. For clause naming, CLNAMING = T.

(2) ENTER a WFF from the QAS or LISP executive as in the past. All the work is done within ENTER,
and a list of clause names followed by the axiom names is printed out as a side effect (but not returned as a value).

(3) The QAS FORGET functions work as before, but do not delete axioms or clauses from the system, since they are pointed to by top-level atoms. To delete an axiom or clause from the system (but not from MEMARRAY) call:

(a) delax[axname]
(b) delax1[(AX1 AX2 ... AXN)] or delax1[j;k] to delete AX_j, AX_{j+1}, ..., AX_k. (Note that delax1[10;13] and delax1[(AX10 AX11 AX12 AX13)] have the same effect.) These functions reset all the clause names (e.g., AX10-3) and axiom names to NOBIND and delete the axiom names from AXNAMELIST.

(4) AXCTR is a top-level atom, which serves as a counter for axiom names.

(5) If an axiom is deleted from MEMARRAY and re-entered, a new set of names and new clauses are generated. A function ENTERAX[axname] that will simply reinsert the old clauses in MEMARRAY is contemplated but unimplemented.

Note: Since a WFF and its descendent clauses are bound to atoms, their structure can be easily modified with the atom editor EDITV.

c. Predicate Evaluation

A user can associate a LISP function with any predicate to evaluate literals during search. MAKECLAUSE can be flagged to call evc[clause], which checks each literal of a newly constructed resolvent and allows options to evaluate the literal if its sign is positive, negative, or both, or if it is a ground literal, or not yet fully instantiated. If function predfn is associated with predicate pred,
predfn[lit] is evaluated as follows:

\[
\begin{align*}
\text{predfn}[\text{lit}] &= \text{T} \quad \text{delete clause} \\
\text{NIL} &\quad \text{delete literal} \\
\text{lit} &\quad \text{no change} \\
\text{lit}_1 &\quad \text{substitute lit}_1 \text{ for lit within the clause.}
\end{align*}
\]

If \textit{pred} has been specified for both positive and negated literals and \textit{predfn[lit]} returns \text{T} or \text{NIL}, \textit{predfn[\neg \text{lit}]} will return \text{\neg predfn[lit]}. Furthermore, even if \textit{pred} is specified only for evaluation on negated literals, \textit{predfn[lit]} should be written for positive literals and the system will negate the result.

Use:

1. Set \textit{EVCHECK} \text{=} \text{T}. For each function \textit{predfn} associated with a predicate \textit{pred}, execute \textit{evalpred[pred;sign;predfn;groundp]}.

   If sign = \text{POS} only positive literals are evaluated
   = \text{NEG} only negated literals are evaluated
   = \text{BOTH} literals of both signs are evaluated.

   If groundp = \text{T} the literal is evaluated only if it is in ground state
   = \text{NIL} evaluated regardless of instantiation.

2. \text{Evalpred places the flag (EVALPOS}p\text{(predfn,groundp))}
   and/or the flag (EVALNEG}p\text{(predfn,groundp)) on the property list of the atom pred. Executing
   \textit{evalpred[pred;sign;NIL]} will cause evaluation to cease for the case specified by \textit{sign}.

Example: \textit{evalpred[GREATER;BOTH;GREATERP;T]} will cause the LISP function \textit{greaterp}_1[lit] to be associated with the predicate "greater." If \textit{(GREATER J K)} is fully instantiated (since groundp\text{=}\text{T}), \textit{greaterp}_1[(\pm P J K)] would be evaluated. Note that the LISP function \textit{greaterp}[x;y] cannot be
used, since it requires two arguments. One could define
\texttt{greaterp}_1[\texttt{lit}] := \texttt{greater}[\texttt{cadadr}[\texttt{lit}]; \texttt{cadadr}[\texttt{lit}]]. Executing
\texttt{evalpred}[\texttt{GREATER; POS; NIL}] causes only literals of the form \texttt{(-P J K)} to
be evaluated by \texttt{greaterp}_1[\texttt{lit}]. \texttt{evalpred}[\texttt{GREATER; BOTH; NIL}] ceases all
evaluation by \texttt{greaterp}_1[\texttt{lit}].

d. Syntactic Symmetries and Generalized (Set) Unification

Occasionally a user encounters predicates that obey syntactic
symmetries. In the past, the only way to represent these symmetries
was to write a special axiom, e.g., \( \forall x \forall y \text{ line}[x;y] \Rightarrow \text{line}[y;x] \). The
functional evaluation described in the preceding section has been used
to implement a symmetric match function (\textsc{Permatchf}) associated with the
function \( p[x_1; \ldots; x_n] \), for arbitrary \( n \). Any two equal, but unordered,
sets will unify if they are represented as the arguments of the function
\( \&[x_1; \ldots; x_n] \).

Examples:

\[ \forall abc \text{ point}[a] \land \text{point}[b] \land \text{point}[c] \land k\text{-distinct}[[a;b;c]] \]
\[ \Rightarrow \text{plane}[[a;b;c]] \]

\[ \forall abc \text{ line}[p[a;b]] \land \text{point}[c] \land \text{not}[on[c;p(a b)]] \Rightarrow \text{triangle}[p[a;b;c]] \]

The preceding axioms exemplify a single use of these devices.

Now consider the following clauses:

\begin{align*}
\text{C1.} & \quad \text{line}[p[A;B]] \\
\text{C2.} & \quad \text{line}[p[B;A]] \triangledown \text{triangle}[p[B;A;x]] \\
\text{C3.} & \quad \neg \text{triangle}[p[C;B;A]]
\end{align*}

A and B are constants; \( x \) is a variable.

Note: \texttt{line}[A;B] and \texttt{line}[B;A] will not unify, but

\[ \texttt{R1} = \texttt{C1} \times \texttt{C2} = \texttt{triangle}[p[B;A;x]] \]
\[ \texttt{R2} = \texttt{R1} \times \texttt{C3} = \texttt{NIL} \text{ with } \theta = A/A; B/B; C/x \]
Any two terms $p[a_1;...;a_n]$ and $p[b_1;...;b_n]$ unify only if $n = m$ and there exists a cyclic permutation of b's that unifies with the a's.

Use:

1. For syntactic symmetries, set $P \leftarrow T$. For unordered sets set $L \leftarrow T$.

2. $P$ is associated with `permatch[]`. $L$ is associated with `tsubsume[]`. These function names are stored on the property lists of $P$ and $L$ under the flag `EVALFN`.

3. Write axioms with the appropriate function segments.

**e. Human-Directed Proof Guidance**

Occasionally a user will want to step through a proof to see whether a given set of axioms is adequate or see the form of a possible proof. The user can now specify two clauses with two of their literals and attempt a resolution. During an automatic research a user can stop the system, denote certain resolvents or axioms as KILLED, and when the system is spurting automatically it will behave as if those clauses no longer existed.

Use:

1. To stop a search in progress, hit a $^c_k$. To prevent any search, but also enable QUESTION to put the prefix version of the question on `CLAUSELIST`, execute: `BREAKIN(FN1(BEFORE UNITRETEST)T)`. Set `TRL-T` at least. (TR2-T is optional.)

2. `THVR(C1 L1 C2 L2)` attempts to resolve $C1$ and $C2$ on $L1$ and $L2$, which must be unifiable. If $L1$ and $L2$ do not unify, an error message is returned. All the arguments of `THVR` are atomic. `TRYR(AX11-1 3 53 2)` attempts to resolve the clause named `AX11-1` on its third literal with
the clause numbered 53 in the proof search, on
its second literal. Some extra unit resolvents
might be added automatically by UNITSECTION.

(3) KILL(N) sets a flag (KILL T) on the property
list of the clause numbered n in the search.
Any clause with a KILL flag will never be
resolved.

UNKILL(N) reverses the effect of KILL(N);
The clause "reappears."

(4) A user who, at the outset of a proof, wants to
add certain clauses from memory to CLAUSEARRAY,
should execute either an H^c during search or
BREAKIN(QUESTION(BEFORE COND 4)T) prior to
search. Within the BREAK execute: (PREF pred n len s)
to add the n\textsuperscript{th} clause of length len filed under
predicate pred to CLAUSEARRAY and if s = T, the
clause will get T-support.

pref[ON;3;2;T] will place the third 2-clause
filed under predicate ON on CLAUSEARRAY with
T-support. pref1[pred((n_1;len_1;s)(n_2;len_2;s_2)\ldots)]
in an n-lambda which calls pref[n_1;len_1;s],
fref[n_2;len_2;s], etc.

Caution: A user guiding a proof who does the complete proof by hand
and reaches a contradiction, should execute (UNWIND PROOF) or (TREESAVE
TREENAME) immediately. Returning to FN1 to allow the system to stop on
its own accord will cause a deep error that will send the user back to
the LISP executive and, as a result, erase CLAUSEARRAY.

f. Statistics

Occasionally a user wants a more refined description of a data
base than the gross number of axioms. Executing MEMSTAT[ ] enables the
user to obtain the following printout for each predicate in MEMARRAY:

\[ \text{PREDICATE P1} \]
\[ \text{THERE ARE } X_1 \text{ clauses of length 1} \]
\[ \text{THERE ARE } X_2 \text{ clauses of length 2} \]
\[ \vdots \]
\[ \text{THERE ARE } X_n \text{ clauses of length } n \]

IT REFERENCES THE PREDICATES (P3 P4 P17 P21).

The following printout is global:

\[ \text{MEMORY STATISTICS} \]
\[ \text{DATE} \]
\[ \vdots \]

There are \( N \) clauses in all

Use:

\[ \text{Execute MEMSTART} \]

\[ \text{g. MEMARRAY I/O} \]

For a large data base, the use of LISTSENT(PRED) to obtain a complete listing of WFF's is all too time-consuming. Moreover, since WFF's are cross-referenced under each predicate in the WFF, each WFF may appear four or five times. BRIEFLIST([p_1 p_2 p_3 \ldots p_n]) prints out a "minimal list" of all the WFF's stored under the predicate names \( p_1, p_2, \ldots, p_n \). The role of each WFF is printed out along with a number.

When axioms yielding several clauses each appear and the number of WFF's begins to exceed 25, using the standard ENTER (or run[file]) can be quite time-consuming.
DUMPOUT[filename] copies the complete MEMARRAY onto the file specified along with the settings of SKOLEMS and AXCTR to reset the skolem function generator and the axiom-naming generator. LOAD[filename] calls FILLIN, which clears the current MEMARRAY and fills it with the dumpout file.

h. Clause Size

The current system handles clauses of length ≤ 9. If one wishes to handle clauses of substantially shorter length—e.g., ≤5—and wants to save space, or if one wants to occasionally handle larger clauses and know when one is entered, then the CLAUSESIZE facility is appropriate. No user changes are necessary, although several QA3.5 functions have been modified. A top-level parameter CLAUSESIZE is set to the desired clause size. Subarrays of MEMARRAY and CLAUSEARRAY are generated to handle clauses of appropriate length. If a WFF is entered that generates a clause of greater length than CLAUSESIZE, the entry is aborted and a message with length information is printed out to the user.

i. Memory Map Functions

Two simple functions have been written to apply an arbitrary function to each clause—e.g., \((\text{HDR lit}_1 \ldots \text{lit}_j)\) in MEMARRAY, CLAUSEARRAY, or any axiom-styled subarray. Mapmem[fun] applies fun to each clause in MEMARRAY. maparc[array;fun] applies fun to each clause stored in array, which is assumed to be of the form:

```
ARRAY
1  CMAX
2  UNITS
3  2-CLAUSES
   NIL c_1 c_2 END
```

TA-7494-48
Use:

(1) delprop[prop]:= mapmemc[function \(\lambda[c]\) ftc[c;prop]]
    deletes prop from each clause in MEMARRAY.

(2) names[array;x]:= prog[[x];mapcar[array;function[\(\lambda[c;y]\);
    setq[y;gtc[c;NAME]];
    cond[(membly[x] \rightarrow NIL; T \rightarrow setq[x;cons[y;x]]return[x]])],
    Names[array] returns a nonredundant list of the
    names of all the clauses is stored in array.

j. Factoring

The value of an existential variable is stored in clausal form
under the flag ANSWERC on a resolvent header. When this clause has
length \(>1\), factoring may give the most specific answer from a possible
set.

Use:

Set FACTORANSWER \(-\) T.

Given two clauses \(C_1\), \(C_2\), and \(\mathcal{A}_1\) and \(\mathcal{A}_2\) of \(C_1\) and \(C_2\), respec-
tively, a function checkres[\(C_1;C_2\)] determines whether the resolvent
of \(C_1\) and \(C_2\) should be computed. Likewise, a function factorcheck[C]
may be written by a user to check for necessary conditioning for factoring
before factoring is attempted. The function name is properly embedded
within FACTORSECTION but each user must specify his own function
definition.

Use:

Set FACTORCHECK to T.

k. Stopping a Proof

It is possible for a user to stop a proof and then continue
later from the same place. To do this, the user interrupts with \(H^C\).
Then set STOP to T and type OK. The proof will continue until it
reaches a convenient place to stop.

To restart, execute QUESTION with no arguments.
7. **Variables**

QAS references several global variables. Most of these control options; some are set by functions in the system, while others must be set explicitly by the user, via the SET function.

These variables are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABMAXLEV</td>
<td>n</td>
<td>Specifies the absolute maximum level bound</td>
</tr>
<tr>
<td>ANCESTORTEST</td>
<td>T</td>
<td>Ancestry-filter strategy is in effect</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No ancestry-filter</td>
</tr>
<tr>
<td>ANSSUBSM</td>
<td>T</td>
<td>Prevents a clause from being entered on CLAUSEARRAY when its answer clause is subsumed by another</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No check</td>
</tr>
<tr>
<td>ANSWERTEST</td>
<td>T</td>
<td>Causes a trace of all universally quantified variables in the negation of the theorem, after the proof is finished</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No action</td>
</tr>
<tr>
<td>ANSWERTEST1</td>
<td>T</td>
<td>Causes the answer clause to be printed during a trace</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No action</td>
</tr>
<tr>
<td>ANSWERTEST2</td>
<td>T</td>
<td>Causes the answer clause to be printed during an unwind</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No action</td>
</tr>
<tr>
<td>AXNAMING</td>
<td>T</td>
<td>Axiom naming is in use</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>CLNAMING</td>
<td>T</td>
<td>Clause naming in effect</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>ENTERUPT</td>
<td>T</td>
<td>Allows user to intercede in clause entering</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>EVCHECK</td>
<td>T</td>
<td>Initiates predicate evaluation</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>FACTORANSWER</td>
<td>T</td>
<td>Causes an attempt to factor the answer clause</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>Variable</td>
<td>Value</td>
<td>Result</td>
</tr>
<tr>
<td>------------------</td>
<td>-------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FACTORCHECK</td>
<td>T</td>
<td>Allows user to define a predicate to control factoring (in FACTORSECTION)</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>GREENGROW</td>
<td>T</td>
<td>Allows user to continue a proof</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No attempt made to continue</td>
</tr>
<tr>
<td>KEEPCLAUSESWITCH</td>
<td>T</td>
<td>Allows user to decide which clauses to keep</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>KEEPANSUNITONLY</td>
<td>T</td>
<td>Retains only unit answer clauses</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>MAXDEPTH</td>
<td>n</td>
<td>Controls depth of function nesting</td>
</tr>
<tr>
<td>MAXLEV</td>
<td>n</td>
<td>Controls level of proof tree</td>
</tr>
<tr>
<td>MODELTEST</td>
<td>T</td>
<td>Resolution is with respect to a MODEL</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No effect</td>
</tr>
<tr>
<td>NOONLY</td>
<td>T</td>
<td>Causes QAS to attempt only NO answers</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>Attempts both YES and NO answers</td>
</tr>
<tr>
<td>PROOFTIME</td>
<td>T</td>
<td>Causes a time message to be printed at end of UNWIND</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No time message</td>
</tr>
<tr>
<td>SKIP</td>
<td>T</td>
<td>Used to continue a proof</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>No continuation</td>
</tr>
<tr>
<td>TRSW</td>
<td>T</td>
<td>No effect</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>TRLIST is built up during proof, but no printing (tracing) is done</td>
</tr>
<tr>
<td>TR1</td>
<td>T</td>
<td>Builds TRLIST (list used in tracing) and traces proof</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>TRLIST is not built and no tracing is done</td>
</tr>
<tr>
<td>TR2</td>
<td>T</td>
<td>Causes clauses subsumed by other clauses to be printed during tracing</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>when TR1 = T Clauses subsumed are not printed</td>
</tr>
<tr>
<td>YESONLY</td>
<td>T</td>
<td>QAS attempts only YES answers</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>Attempt both YES and NO answers</td>
</tr>
</tbody>
</table>