STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

June 1969

THE APPLICATION OF THEOREM PROVING TO QUESTION-ANSWERING SYSTEMS*

by

Cordell Green

Artificial Intelligence Group

Technical Note No. 8

SRI Project 7494

*Submitted to the Department of Electrical Engineering and the Committee on the Graduate Division of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

This paper shows how a question-answering system can use first-order logic as its language and an automatic theorem prover, based upon the resolution inference principle, as its deductive mechanism. The resolution proof procedure is extended to a constructive proof procedure. An answer construction algorithm is given whereby the system is able not only to produce yes or no answers but also to find or construct an object satisfying a specified condition. A working computer program, QA3, based on these ideas, is described. The performance of the program, illustrated by extended examples, compares favorably with several other question-answering programs. Methods are presented for solving state transformation problems. In addition to question answering, the program can do automatic programming (simple program writing, program verifying, and debugging), control and problem solving for a simple robot, pattern recognition (scene description), and puzzles.

FOREWORD

Portions of this work have already been presented in papers in the Proceedings of the ACM National Conference,[1] Machine Intelligence 4,[2] and the Proceedings of the International Joint Conference on Artificial Intelligence.[3]

The research reported here was supported at Stanford Research Institute under Contract AF 30(602)-4147 and Contract F30602-69-C-0056, Rome Air Development Center and the Advanced Research Projects Agency, Advanced Sensors Group; and also under Contract AF 19(628)-5919, Air Force Cambridge Research Laboratories.

ACKNOWLEDGMENTS

Professor John McCarthy has given me many ideas and suggestions. I owe a great debt to him for developing the "advice taker" approach to artificial intelligence.

The research itself was done in the Stanford Research Institute Artificial Intelligence Group under the patient guidance and supervision of Dr. Bertram Raphael.

Robert A. Yates has contributed both substantial programming effort to this research and many hours of valuable discussions.

Throughout my years at Stanford, Professor Thomas Cover has been both a close friend and counselor. He has provided the opportunity for me to do this interdepartmental research.

Professor Jerome Feldman has contributed a great deal of constructive advice on my writing and research.

Discussions with Dr. Larry Wos of Argonne National Laboratories and Dr. George Robinson of Stanford Linear Accelerator Center have been very valuable, allowing me to learn from their experience in theorem proving. I am very grateful to Robert Kowalski of Edinburgh University, who found errors in my mathematics and provided suggestions for improvement.

I would like to acknowledge valuable discussions with Claude Brice, Stephen Coles, Richard Duda, Claude Fennema, Peter Hart, Robert Kling, John Munson, Nils Nilsson, and Charles Rosen, all of SRI, and David Luckham and Zohar Manna of Stanford.

CONTENTS

I INTRODUCTION

A. A Guide to Reading this Thesis

This section provides a guide to reading this thesis.

The reader who wishes to begin with an example of question answering by theorem proving will find explanatory examples in Secs. III-B, V-A, and V-C. The remainder of this introductory section (Sec. I) provides background material and an overview of the research described in this thesis.

For the reader who is unfamiliar with automatic theorem proving by resolution, a review and summary is presented in Sec. II. A dialogue illustrating the basic question and answer process is provided in Secs. III-A and B. The mathematical basis for the type of answer construction used in this research is provided in Secs. III-C and III-D, but the reader unfamiliar with automatic theorem proving may want to skip the rather complicated answer construction algorithm and corresponding proof of correctness in Sec. III-D. Extended question-answering dialogues are presented in Sec. V, using a program, QA3, described in Sec. IV.

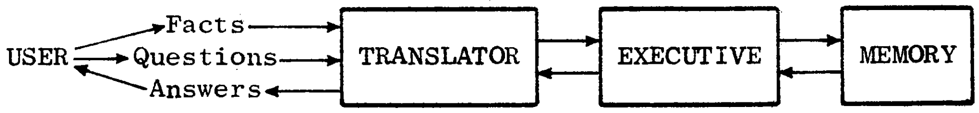
A very wide class of problems that can be posed as various kinds of state transformation problems do not on the surface fit into first-order logic. In Sec. VI a method is presented whereby state transformation problems can be solved by means of the answer mechanism in a first-order, resolution theorem prover. This method is illustrated by the well-known "Monkey and Bananas" and "Tower of Hanoi" puzzles. An application is presented in Sec. VII-A in a discussion of the Stanford Research Institute robot project.

Another promising application, discussed in Sec. VII-B, is automatic program writing, program debugging and verifying, and program simulation. The programming language used is "pure" LISP 1.5, a lambda-calculus-like language. LISP is the language in which the question-answering program itself is written. To raise the issue of purposeful self-modification, an oversimplified self-description of the program's own problem-solving strategy (in its own problem-solving language) is presented in Sec. VII-C. Finally, a simple scene description problem--describing a cube from a two-dimensional projection of its edges--is given in Sec. VII-D.

B. General Description of a Question-Answering System

The purpose of this section is to roughly define a question-answering system. A question-answering system may be broadly defined as a system that accepts information and uses this information to answer questions. Often the information, questions, and answers are presented in a form that is relatively easy for people to learn, such as some restricted class of typewritten English sentences. If the question-answering system, a computer program, produces reasonable responses, it may be attributed the human characteristic, "understanding."

The following diagram shows the essential components of a question-answering system.



Its operation is as follows: The user presents statements (facts and questions). A translator converts them into an internal form. Facts are stored in memory. (The store of facts is referred to as the data base.) Answers to questions are formed in two ways: (1) the explicit answer is found in memory, or (2) the answer is computed from the information stored in memory. The executive program controls the process of storing information, finding information, and computing answers. This first description of a question-answering system is greatly oversimplified, but will serve as a starting point for discussion. Elaboration of the components and processes will be provided in the next section.

Before discussing question-answering systems as such, we distinguish between a question-answering (QA) system and an information-retrieval (IR) system (or an information-storage-and-retrieval system). In an information-retrieval system, all the information that is available to the user is explicitly stored in memory. Such a system may be a document-retrieval system or a fact-retrieval system. Typically, the data base is quite large, and may be stored on magnetic tapes, discs, or other mass storage devices.

A question-answering system, on the other hand, does not explicitly store all information that is available to the user. Instead, a smaller data base of compactly coded facts is used. New information, not explicitly stored in the data base, but implied by the stored facts, is computed or deduced from this data base by an answer-computation mechanism.

The dividing line between a question-answering system and a fact-retrieval system is not always clear-cut. (For example, a fact-retrieval system may encode its facts to such an extent that a considerable computation process is necessary to recover the information.) Often one labels a system as a question-answering system when the human user believes that the system is making inferences during the answer-computation process.

C. Characteristics of Question-Answering Systems

The purpose of this section is to acquaint the reader with some of the significant characteristics of question-answering systems, as well as the terminology used to describe such systems.

This thesis discusses a particular kind of question-answering system. The most recent working version of a system of this kind is a computer program called QA3. QA3 will be characterized in the following discussion of characteristics of question-answering systems.

1. Methods for Computing Answers

The method of computing answers is one of the most distinguishing features of a question-answering system. There are many methods (and variations thereof) for finding answers not explicitly stored. They include the following:

(1) A different prewritten subprogram computes answers for each class of question. Such a system is Raphael's SIR.[4,5] The disadvantages are that (a) a new subprogram must be written for each class of questions, and (b) if questions require interactions between existing classes, either combinations involved must have been anticipated or else new subprograms must be written.

(2) Each question is automatically translated directly into a particular program for answering this question. Such a system is Kellog's.[6] This method requires that the automatic translator be general enough to produce a program for answering every desired question.

(3) All questions are presented to a subprogram that examines the question and <u>infers</u> the answer. The program discussed herein, QA3, is basically of this type, using a theorem prover as this subprogram. This method requires that the inference mechanism of the subprogram be general enough to infer an answer to any desired question.

A system can of course use mixtures of these methods. For example, QA3 uses Method 1 for certain arithmetic question answering.

2. Languages

Another important characteristic of question-answering systems is the set of languages used. At the outermost level is the <u>dialogue language</u> or languages. These may include (1) a language for presenting facts to the system (often one or more restricted natural languages), (2) a query language employed by the user, and (3) an answer language employed by the system. In the process of answering a question, some systems may require additional information from the user. This requires a query language employed by the system and an answer language employed by the user. QA3 uses first-order logic for all these purposes, although an English-to-logic translator of Coles[7,8] is linked to the system to provide a restricted English fact-input and query language. A system may also have a control language employed by the user to control the system. The control language of QA3 is described in Sec. IV-A.

A system may have other forms of input and output (e.g., graphical). Information conveyed in these other ways is not explicitly linguistic, but can be translated into a language (such as a picture-description language, discussed in Sec. VII-C). Section V-B describes an application of QA3 in which information is input to sensors and output to effectors.

Usually a question-answering system will have one or more internal languages. These languages are used either as intermediate steps in translation or as "working" languages in which the system calculates, infers, stores, retrieves, etc. The internal language of QA3 is the language of clauses (see Sec. II).

One frequently mentioned characterization of a language is the degree to which it is a formal language. A formal language (such as first-order logic) is syntactically well defined by a set of rules [such as a set of Backus Naur Form (BNF) productions].

3. Representation

The problem of representing data may be divided into three parts:

(1) Determining the relevant semantic content of the data. For example, we may decide that the semantics of the sentence, "John is the father of Bill," is expressed by the binary relation "is-the-father-of" applied to the objects named "John" and "Bill."

(2) Choosing a language in which to express this semantic content. For example, we may use the notation of mathematical logic and pick appropriate symbols--i.e., Father(John,Bill). (Forms of language were discussed above.)

(3) Choosing an internal representation of the language. For example, a binary relation may be expressed by a list of three elements in which the first element of the list is the name of the relation and the next two elements are two arguments of the relation--e.g., (Father John Bill).

In expressing the semantic content of, say, a sentence of English, we are deciding what information that sentence can provide for the question-answering system. For example, the style or tone of the sentence may carry considerable information about, say, the psychology of the creator of the sentence, but we may choose to ignore all such information and just take the explicit facts. More precisely, when we choose an internal representation, we restrict the set of statements that may be inferred or calculated from the representation of that sentence. Thus, one goal-directed criterion used to determine what is selected in specifying semantic content is: Will the system be able to correctly answer questions concerning the subject matter of that sentence?

The language mentioned in Item 2 above should be selected to represent, unambiguously and compactly, the semantic content of the data. A crucial factor in selecting the language is that one must be able to use this language--i.e., be able to construct an answer-computation program that can effectively produce correct answers from facts expressed in this language. For example, a theorem-proving program can answer questions from facts expressed in the language of logic.

Many considerations are important in selecting the internal representation--storage efficiency, ease of translation, usability by question-answering subprograms, etc. QA3 uses a list-structure representation of clauses. Meta-statements about statements, such as "This information is useful in answering a certain question," must also be expressed in some way. Typically, meta-level information is not necessarily in the form of explicit statements, but instead may be known to hold because of the position of the item in the memory--e.g., QA3 uses the convention that if an item is on a particular list, then it is relevant to answering a certain question.

4. Memory Organization

An issue that is very closely related to representation, and nearly inseparable from it, is that of memory organization. This refers to where and how the internal representations are stored. Important issues here include: What kind and how much indexing of the statements is to be done? How much common substructure is to be shared by items of data? What information should be explicitly stored? How will information be added or accessed? As one example of memory organization, consider a commonly used property-list technique. In the LISP programming system, statements may be placed on property lists of atomic symbols--e.g., on the property list of the atomic symbol "John" we place the value "Bill" under the attribute "Father." The atomic symbol provides an entry point--an index--to the information, "John is the father of Bill." The first argument "John" of the relation "father of" is not stored explicitly with the relation, but instead is implied by the fact that the attribute-value pair occurs on the property list of John.

Other candidates for storage methods include the many varieties of node-link list structures, hash coding techniques, arrays, and files of various sorts. Large, slower secondary storages present their own special organization problems. The memory organization of QA3 is described in Sec. IV-C.

The stored information, including the language, semantic content, internal representation, and memory organization is sometimes referred to as the system's model of its world. To fully characterize its "model," the question-answering routines must be included; in some systems some of the question-answering mechanisms themselves are explicitly stored in the model.

5. General vs. Special Purpose

It is important to emphasize the distinction between general vs. special-purpose question answering. If the class of questions asked of a system is small, completely specified in advance, and concerned with a particular subject area, such as the question-answering system of Green, Wolf, Chomsky, and Laughery[9] concerned with baseball, or the question-answering system of Lindsay[10] concerned with family relations, then we will call such a system "special purpose." Frequently the goal in designing a special-purpose system is to achieve good performance, measured in terms of running speed and memory utilization. In this case the best approach may be first to construct a special data base or memory that is optimized for that subject area and question class, and then to write special question-answering subroutines that are optimized for the particular data base and question class. On the other hand, a "general" question-answering system is one that allows the addition of widely varied subject areas, questions, and interactions between subject areas during the process of answering a question. QA3 is a general question-answering system.

6. Level of Difficulty of Answering

The major consideration here is the average amount of computation necessary to answer a question. One obvious measure of difficulty is the average distance of the answer from the question, measured, for example, in terms of the number of fixed-size steps of inference from the facts. Another way of viewing this is the degree of decoding necessary to recover an implicit answer. This aspect of a question may be termed the average depth of questions.

Another factor contributing to the search effort is the number of different questions that are answerable. To increase the number of answerable questions (without increasing the depth of questions), one may increase the size of the data base of else expand the capabilities of the answer-computation mechanism or both.

Systems having broad, possibly interrelated data bases whose answer-computation mechanism is not capable of great depth tend to be called question-answering systems. Systems having less-interrelated data bases whose answer-computation mechanism is capable of more depth tend to be called problem-solving systems. QA3 seems to be on the boundary line between the two kinds of systems.

7. Consistency of Data Base

As the amount of stored information increases, one problem can be the consistency of this information. Systems with informal inference rules, such as Colby's,[11] are still effective with inconsistent data bases. In formal logic systems, such as QA3, inconsistency can lead to incorrect answers, so that new information must be checked for consistency before acceptance.

8. Modifiability

A very interesting feature is the degree to which new information modifies the system. As new information is entered, the performance of the system is altered, and we can refer to this as a modification of the program even though only the data is altered. In more sophisticated systems new information can have an effect on how questions are answered. Consider the following increasingly sophisticated ways in which new information can modify a program's performance:

(1) A new fact provides the answer to a new question.

(2) A new fact provides the information needed to get the answers to a new class of questions.

(3) A new fact provides a new procedure for answering a new class of questions.

(4) A new fact modifies the representation of information.

(5) A new fact modifies the question-answering strategies of the program.

New information in the form of reprogramming can, of course, provide all such modifications to the system. The more interesting case is when information in the dialogue language can effect such changes as a major modification of question-answering strategies. A system possessing a high degree of modifiability through a formal dialogue language has been termed an advice taker[12,13] by McCarthy.

Another source of information besides the user is the system itself. New information may be generated by the system through question-answering routines, sensors, internal monitoring of performance statistics, etc. Such information may also be stored and be usable to improve performance.

QA3 has the abilities described in Items 1 and 2 above. The control language allows some modification (see Sec. IV) of the question-answering procedures and strategies. The program-writing and self-description capability allows for theoretical self-modification, but in practice this problem lies beyond the problem-solving capacity of QA3. Representation of information can only be modified by the user's editing the data base.

9. Control of Interaction

This leads us to the modes of control in complex information processing systems. The control is not always so clearly resident in the human "user." There exist programs[11] that are also question-asking systems that interrogate the "user" and store (possibly after significant processing) the answers. One of the ultimate goals of research in machine intelligence is to create an independent system. In QA3 control is clearly with the user, although in some applications (see Sec. VII-A) QA3 requests information from the user and from other programs.

D. Previous Work in Question Answering

A great deal of work has been done on the many aspects of question answering and several reviews of the subject have appeared. Rather than repeat a review of the past and present state of the art, I shall mention several of these papers. Aspects of question answering are discussed under many titles, including computational linguistics, structural linguistics, semantics, psycholinguistics, (natural) language processing, mechanical translation, verbal understanding, word concepts, semantic memory, belief systems, and semantic interpretation.

Two excellent reviews of question answering have been written by Simmons. His reviews discuss both natural language processing and question-answering procedures. The first survey[14] covers early work until 1965, including fifteen experimental English language question-answering systems. The second paper[15] surveys systems from 1965 up to 1969. In addition, Raphael's SIR[5] dissertation provides an early discussion of question answering and understanding. Wood's "Semantics for a Question-Answering System"[16] discusses several systems, as well as the representation of English sentences by mathematical logic. A paper by Bobrow, Fraser, and Quillian[17] provides a review of relevant recent linguistic literature.

E. Summary of Problem, Solution, and Contribution

1. The Problem

The problem investigated in this research effort is primarily that of calculating an answer to a question stated in mathematical logic, given facts stated in logic. The principal subproblems focused on are:

(1) How does one represent statements, questions, and answers--for a reasonably wide range of subjects--in mathematical logic (in particular, first-order predicate calculus)?

(2) How does one compute an answer to a question stated in logic, given a set of facts stated in logic?

(3) How does one develop such a working system--i.e., embed such a "logic machine" in a larger question-answering or information-processing system?

Involved in these subproblems are problems of information storage and retrieval, memory organization, measurement of relevance, generality of inference systems, and the many other problems of heuristic programming.

We refer to this "logic machine," which is capable of question answering in logic, as the Question-Answering System, abbreviated QAS. It is assumed that in a given application QAS may be used in conjunction with language translators such as English-to-logic and logic-to-English translators. Indeed, as mentioned earlier, a working version of QAS, called QA3, has been coupled to an English-to-logic translator by Coles. The translation problems are <u>not</u> the subject of this paper. (One view of question answering holds that once there exists a suitable underlying logical question-answering system, then a solution to the translation problem will be simpler. The translation target language--logic--is well defined, the semantics of the target language is well defined, and the logic problem solver is available to provide necessary assistance in the translation process. If one knows how the semantics of a given subject is to be expressed in logic, it is then easier to develop an English-to-logic translator.)

2. <u>The Solution</u>

This section presents a summary of the solutions offered to the three subproblems listed above: representation, answer computation, and development of a question-answering system.

The problem of representation was solved by encoding facts and questions in terms of statements of first-order logic. The particular technique of encoding is illustrated in detail for several common question-answering and problem-solving subjects. These subjects include simple games and puzzles, many "common sense" topics (classification systems such as family relationships, structures of objects, part-whole relationships, set-theoretic relationships, etc.), picture descriptions, state transformation processes, programming languages, induction, and theorem proving itself.

Our solution to the problem of computing answers to questions follows from our representation of facts as axioms, and questions as conjectures to be established as theorems. The question-answering process is a modification of the process of proving such theorems. The theorem-proving process is based on Robinson's "Resolution" techniques[18,19,20]. These techniques are extended to include "constructive" proofs. An algorithm for generating "constructive" answers is developed, and the answers provided by the algorithm are proved correct. Also, proof strategies and heuristics suitable for question answering are developed. The system can answer questions in each of the subject areas discussed above.

The solution to the third problem, system development, consists of the design and implementation of QA3, a system of programs written in the LISP language for the SDS 940 computer. The system has a control language, storage and retrieval capabilities, significant problem-solving capabilities, an interface with a natural language translator, an interface with libraries of programs in LISP and FORTRAN, and an interface with sensors and effectors (for the robot application, described in Sec. VII-A). In terms of the previous characterizations of question-answering systems, the implemented system is a general, formal question-answering system. Its dialogue language is first-order logic, and its internal language consists of clauses. Its answer-computation mechanism is an extended resolution theorem prover. Interactions between subject areas are allowed. The answers it generates are always logically correct consequences of its data base (which therefore should only contain consistent information). It can handle difficult problems if compared to existing general question-answering programs, but only easy problems compared to existing specialized programs (chess programs, for example). To some extent, rules for answering questions can be given in dialogue. Some modification and guidance of the question-answering strategy is possible through the special control language.

3. Contribution to Information Processing

The purpose of this section is to outline the contribution of this work to information processing.

The notion of using logic to describe the world has been pursued by philosophers, logicians, and mathematicians for centuries. The particular representations and axiomatizations given here are somewhat original, but a greater contribution lies in showing how such axiomatizations can be _used_ in problem solving and question answering.

This work represents one of the first developments of the theory and application of a formal, complete, first-order logic proof procedure to question answering. In particular, it applies the resolution proof procedure to question answering, thus showing in detail how perhaps the best of the known theorem-proving methods can be applied to question answering. It extends the resolution procedure, in theory and practice, to constructive proofs and to methods for solving state-transformation problems. The representation selected for state-transformation problems provides a machine-usable first-order logic basis for McCarthy's situational logic.[12,13] It extends the resolution procedure to interface a "pure" theorem-proving program with other problem-solving subprograms. Many of the above results have been thought feasible or plausible by some logicians for many years. However, this work represents concrete, implemented, proven solutions, rather than feasibility or plausibility discussion; thus it makes many previous ideas more precise.

The feasibility of constructive proof procedures by Herbrand methods has been known to logicians essentially since the 1930's. McCarthy saw this potential in the resolution procedure. Robinson[21] carried the development of related ideas nearer to realizability. My work probably represents the first implement development of such constructive resolution proof procedures. Independently, Waldinger and Lee[22] developed and implemented another successful approach. Slightly later, I believe, Sussman,[23] and then Burstall,[24] developed related systems. Sussman's system seems to have a sophisticated heuristic theorem prover. Darlington has been successfully exploring logical question answering by related approaches for several years.[25,26,27] Darlington[25] developed possibly the closest forerunner of this work; he used a method related to resolution although the method was logically incomplete and did not include constructive proofs. Other related work is discussed in Section VIII-D and E.

In addition to its contributions to theory, this work has resulted in a working question-answering system that in certain respects can do what no previous such system could do. This system has contributed to several research projects at SRI. In applications other than those mentioned herein, Raphael and Coles[28] have begun to study medical question answering in a project for the National Library of Medicine, supported by the National Institutes of Health. This application has required extensions of QA3 to deal efficiently with finite sets, and a two-way communication facility. Kling[29] has used and modified QA3 in a research project concerning the use of analogy to discover difficult proofs. The SRI automaton (robot)[30] uses QA3 as one of its problem-solving mechanisms. This application is discussed further in Sec. VII-A.

Another contribution of this work is that it shows how one formal problem-solving mechanism can be used for seemingly diverse problems. It emphasizes the strong unity underlying the many aspects of machine intelligence. I believe that from this and similar work emphasizing generality, we will approach more purposefully self-modifying and and independent "learning" machines.

It is hoped that formal techniques such as those developed here may be of general value to the field of artificial intelligence. The use of a formal framework can lead to insights and generalizations that are difficult to develop while working with an ad hoc system. A common, well-defined framework facilitates communication between researchers, and helps to unify and relate diverse results that are difficult to compare.

The theorem proving by resolution solution to the formal question-answering problem works. We will show that it is adequate for many question-answering and problem-solving tasks. Its performance compares favorably to SIR, DEDUCOM,[31] and other previous question-answering systems. Its principal limitation is that it cannot solve very difficult or highly specialized problems. A more detailed discussion of the advantages and disadvantages of this approach, as well as a comparison to other systems, is given in Sec. VIII-D.

II REVIEW OF AUTOMATIC THEOREM PROVING

The purpose of this section is to provide a brief review of logic and automatic theorem proving by resolution. An introduction to theorem proving by resolution can be found in "A Review of Automatic Theorem Proving" by J. A. Robinson.[19] Cooper, in Ref. 32, provides an introduction to pre-resolution automatic theorem proving. J. A. Robinson[18] presents a recent and broad treatment of theorem proving in "The Present State of Mechanical Theorem Proving," and also provides an excellent bibliography of relevant work.

Progress in automatic theorem proving is exemplified by two of the most powerful theorem-proving systems--that of Wos, Robinson, et al.,[33,34,35] and that of Guard et al.[36] The program of Wos, Robinson, et al. is a highly developed "pure" resolution theorem prover (with special treatment of equality). Guard's system (quite closely related to resolution) is a highly interactive man/machine system that has already proved a lemma leading to a previously unproven mathematical result.

The branch of formal logic referred to as first-order logic deals with well-defined strings of symbols called well-formed formulas (wffs). Well-formed formulas (also called statements) are composed of constants (I will often use a,b,c,d,e, other lower-case letters, or numbers to represent constants), variables (usually s,t,u,v,w,x,y,z), function letters (usually f,g,h,j, or other lower-case letters), predicate letters (usually P,Q,R,A,B,C, or other upper-case letters), connectives, and quantifiers. A term is either a constant, a variable, or a function (formed by applying a function letter to other terms)--e.g., f(b,y) is a term. The word "function" is often conveniently misused to refer to either a function letter, a term composed of a function letter applied to its arguments, or else a function (the mapping itself). A function of n variables is called an n-ary or n-place function. A constant is often considered to be a special case of a function--namely, a function of no variables. An atomic formula is obtained by applying a predicate letter to terms--e.g., P(x,a) is an atomic formula. A predicate letter of n arguments is an n-ary or n-place predicate letter. A proposition is an atomic predicate of no arguments. A well-formed formula is either an atomic formula, a formula obtained by applying connectives to other wffs, or a formula obtained by applying a quantifier to another wff. We will use the connectives $\sim$, $\supset$, $\vee$, $\wedge$, and $\equiv$, meaning, respectively, NOT, IMPLIES, OR, AND, and EQUIVALENCE. The quantifiers are the universal quantifier $\forall$ and the existential quantifier $\exists$. The string of quantifiers $(\forall x_1)(\forall x_2)\ldots(\forall x_n)$ is sometimes abbreviated as $(\forall x_1,x_2,\ldots,x_n)$. If a wff contains a variable that is not bound by either a universal or existential quantifier, then that variable is said to be a free variable. Wffs containing no free variables are closed wffs.

An example of a well-formed formula is

$$(\forall x)(\exists y)[P(x,a) \supset \sim R(x,f(b,y))] \quad .$$

The terms it contains are x, y, a, b, and f(b,y). The terms x and y are variables bound by the quantifiers. By definition, a and b are constants. The atomic formulas it contains are P(x,a) and R(x,f(b,y)). We may read the statement as "For every x there exists a y such that if P(x,a), then it is not the case that R(x,f(b,y))."

By presenting the formula above or by stating the formula P(x,a) as in the last sentence, one typically means to assert that it is "true" or that it "holds" in some sense. The precise sense of "truth" (or lack of such precision) is usually evident from the context.

In first-order logic variables may occur only as term variables, never as predicate or statement variables. Thus the statement $(\forall x)P(x)$ is a legal first-order logic construction, whereas the formulas $(\forall P)(P(x))$ and $(\forall s)s$ are not legal. These constructions are higher-order logic.

Other notations that are related include the descriptive operator ι and the notation for a set $\{x:P(x)\}$. The term $\iota x.P(x)$ means "the unique x" such that P(x) holds, and if there is not a unique x such that P(x) holds, then the term $\iota x.P(x)$ is typically taken as undefined or equal to some special value--say, 0. In Sec. III-C and D we shall introduce a method for finding some x such that P(x) holds; this is close to $\iota x.P(x)$, but not necessarily restricted to a unique x. The set notation $\{x:P(x)\}$ means the set of all x such that P(x) holds.

Two aspects of logic are the <u>syntactic</u> notions and the <u>semantic</u> notions. A wff is a syntactic or linguistic entity. Legal wffs are completely specified by a set of grammar rules. One usually intends a wff to have some "meaning" or semantics. The notion of semantics and its correspondence to syntax can be made quite rigorous. The semantics of a statement is specified by an <u>interpretation</u>. An interpretation consists of (1) a non-empty set of objects called the <u>domain</u> (or universe), (2) an assignment of an object in the domain to each constant, (3) an assignment of an n-ary function on the domain to each n-ary function letter, and (4) an assignment of an n-ary relation (set of ordered n-tuples) on the domain to each n-ary predicate letter. A variable then ranges over the elements of the domain.

A closed wff (no free variables) is then true or false with respect to this interpretation. We shall consider only closed wffs. An interpretation that makes a wff true is said to <u>satisfy</u> the wff, or equivalently, the interpretation is said to be a <u>model</u> of the wff. A wff is <u>satisfiable</u> if and only if there exists an interpretation that satisfies the wff. A wff is <u>logically valid</u> if and only if it is satisfied by all possible interpretations, or equivalently if the negation of the wff is <u>unsatisfiable</u>.

The <u>propositional calculus</u> (or boolean logic) does not allow quantifiers or variables. An atomic formula is considered as the smallest undecomposable element. In propositional calculus, a logically valid wff is a <u>tautology</u>. For example, the propositional statement $P \vee \sim P$ is a tautology. In propositional calculus, an unsatisfiable wff is said to be a <u>contradiction</u>, or <u>truth-functionally unsatisfiable</u>. The method of <u>truth tables</u> may be used to indicate that a propositional wff is a tautology or truth-functionally unsatisfiable. We can consider a predicate calculus formula to be a propositional formula by considering an atomic formula to be a proposition. For example, the formula $P(x) \vee \sim P(x)$ is obviously truth-functionally unsatisfiable.

The theorems of a logical system are usually intended to be the valid wffs. However, since it is not practical in general to enumerate and test all possible interpretations, formal syntactic procedures called proof procedures must be used to establish theorems. If every theorem produced by a proof procedure is indeed valid, the procedure is called sound. If every valid formula can be demonstrated to be a theorem, the procedure is complete. In the desirable case that a proof procedure is both sound and complete, the theorems of the procedure coincide with the valid wffs. A decision procedure is a procedure that can decide in a finite number of steps whether or not any given wff is valid.

Unfortunately, it is known that there are proof procedures for first-order logic, but there is no decision procedure for first-order logic. This means that there is no guarantee that a proof procedure will converge to a proof in a finite number of steps when attempting to prove a non-theorem.

As a practical matter, however, this lack of a decision procedure does not limit the applicability of logic as much as it may at first appear. Because of the time and space constraints on practical computation, the heuristic power of a proof procedure--i.e., its ability to prove useful theorems efficiently--is more important than its theoretical limitations. This issue is discussed fully in an interesting paper by Robinson[37] (see also Ref. 18). A decision procedure that requires enormous amounts of time or intermediate storage is indistinguishable, in practice, from a proof procedure that never terminates for some wffs.

In recent years, much work has been done on the development of proof procedures suitable for implementation on a digital computer. The most effective of these seem to be those that use the Robinson resolution principle in conjunction with Herbrand's "semantic tableau" methods of theorem proving.

A wff Q is a logical consequence of (follows from, semantically) a set of axioms (premises) B if and only if every model of B is a model of Q. [The corresponding syntactic notion is that a conjecture Q is a theorem if it can be proved (by a proof procedure) from a set of axioms B.] It can be easily shown that Q is a logical consequence of B if and only if $B \supset Q$ is logically valid, or, equivalently, if the statement $\sim[B \supset Q]$ (logically equivalent to $B \wedge \sim Q$) is unsatisfiable. The basic approach of Herbrand proof procedures is to use syntactic rules of inference in an effort to determine that the negation of the wff to be proved ($B \wedge \sim Q$) is unsatisfiable. From a set of formulas, the rules of inference produce new formulas, preserving unsatisfiability, until an explicitly unsatisfiable formula--a contradiction--is produced. The resolution procedure is such a Herbrand type of procedure.

The resolution procedure finds proofs by refutation. To prove a theorem Q by refutation, one assumes that the theorem is not a logical consequence of the axioms B, and then derives a contradiction. The resolution procedure is a refutation algorithm that deduces from $B \wedge \sim Q$ an explicit contradiction. The search for a contradiction is an attempt to construct a model that satisfies $B \wedge \sim Q$. It has been shown that the resolution procedure deduces a contradiction if and only if $B \wedge \sim Q$ is unsatisfiable ($B \supset Q$ is logically valid); thus, resolution is a sound and complete proof procedure. To prove that a statement Q does not follow from a set of axioms B, one assumes it does and attempts to derive a contradiction from $B \wedge Q$. No decision procedure exists for the first-order logic, so in general, for a given B and a given Q, one cannot guarantee that the proof procedure will terminate in either the attempted proof of Q or the attempted disproof of Q from B.

In most automatic theorem proving, statements are converted into a standard quantifier-free form. First, a wff C is converted algorithmically into a prenex conjunctive normal form C', in which all the quantifiers occur in one quantifier prefix at the beginning of C'. The rest of C', called the matrix, is an "and" of "or's" of atomic formulas. Each existentially quantified variable can be replaced by a Skolem function applied to those universally quantified variables within whose scope the existential quantifier lies. The Skolem functions are formed from new function letters. For example, in the statement $(\forall x)(\exists y)P(x,y)$ the existentially quantified variable y is replaced by the Skolem function f(x), and the quantifier ($\exists$x) is dropped to yield the new statement ($\forall$x)P(x,f(x)). The function f(x) may be thought of as denoting the y that is asserted to exist. The dependence of y on x is reflected by the fact that the Skolem function depends on x. The next step in the conversion process is to drop the universal quantifiers, leaving it understood that all variables are universally quantified. The final quantifier-free form of the statement is satisfiable if and only if the original statement is satisfiable. An equivalent notion is that the original formula and the final formula are interprovable; one is a theorem if and only if the other is a theorem. The proof of this, along with a detailed discussion of the conversion algorithm, is given by Davis.[38]

In the resulting quantifier-free conjunctive normal form formula, each conjunct is called a clause. Each clause is a disjunction of literals; a literal is either an atomic formula or the negation of an atomic formula. As an example, the wff $(\forall x)(\exists y)[P(x) \supset R(y)]$ is converted to the clause

$$\sim P(x) \vee R(f(x))$$

where f denotes the Skolem function replacing y. A conjunction of several clauses may be referred to as a set of clauses. A clause may be referred to as a set of literals, and may be represented as a set--i.e., $\{\sim P(x), R(f(x))\}$.

The resolution proof procedure uses statements in the standard clause form. First, the formula $B \wedge \sim Q$ (B is a set of axioms, $\sim Q$ is the negation of the theorem) is represented as a set of clauses. Then new clauses--resolvents--are deduced from the starting clauses by the resolution rule of inference. The main theorem of resolution states that if a resolvent is not satisfiable, then neither of its antecedents are satisfiable, and that the empty formula is not satisfiable. The goal of the procedure is to deduce the empty clause, an explicit contradiction that is not satisfiable. This demonstrates that all its antecedents, including the starting wff, are not satisfiable.

The rule of resolution is best illustrated first in its propositional form: if $p \vee \alpha$ and $\sim p \vee \beta$ are two wffs in which p is any proposition and α and β are any wffs, one may deduce the wff $\alpha \vee \beta$. More concisely, $(p \vee \alpha) \wedge (\sim p \vee \beta) \supset (\alpha \vee \beta)$.

The exact statement of the resolution rule requires that we introduce the notion of a substitution. A substitution gives a set of terms that are to be substituted for a set of variables. A substitution σ may be written as a set, $\sigma = \{t_1/x_1, t_2/x_2, \ldots, t_n/x_n\}$, meaning that term t_1 is to be substituted for x_1, t_2 for x_2, etc. If L is a formula then $L\sigma$ denotes the formula resulting from performing the substitution σ on the formula L.

Two formulas L_1 and L_2 are said to unify if there exists a substitution σ such that $L_1\sigma = L_2\sigma$. If $L' = L\sigma$, for any σ, then L' is said to be an instance of L. The substitution σ is said to be the most general unifier of two formulas L_1 and L_2 if $L_1\sigma = L_2\sigma$ and, for any other unifier λ of L_1 and L_2, $L_1\lambda = L_2\lambda$ is an instance of $L_1\sigma = L_2\sigma$. Robinson has shown that if two formulas unify, there exists a most general unifier of the two formulas.

The heart of the resolution process is the unification algorithm that determines whether or not two formulas unify, and, if they do, finds the substitution set σ that is the most general unifier of the two formulas. This algorithm guarantees that in one sense each resolution inference step is as general as possible, since every less general unification is implied.

The exact statement of the resolution rule of inference begins as follows. Let L_1 be any atomic formula. Let $\sim L_2$ be the negation of an atomic formula consisting of the same predicate symbol letter of L_1, but in general with different arguments. Using the set notation to represent clauses, the resolution rule of inference is: Given two clauses $\{L_1,\alpha\}$ and $\{\sim L_2,\beta\}$ where α and β are disjunctions of literals and L_1 and L_2 are atomic formulas, and if L_1 and L_2 have the most general unifier σ, infer by resolution the resolvent $\{\alpha,\beta\}\sigma$.

Example:

$$P(x,f(y)) \vee Q(x) \vee R(f(a),y)$$

and

$$\sim P(f(f(a)),z) \vee R(z,w)$$

imply, by resolution,

$$Q(f(f(a))) \vee R(f(a),y) \vee R(f(y),w)$$

where the substitution $\sigma = \{f(f(a))/x,\ f(y)/z\}$ applied to the two literals $P(x,f(y))$ and $\sim P(f(f(a)),z)$ yields the two literals $P(f(f(a)),f(y))$ and $\sim P(f(f(a)),f(y))$ so that the two clauses resolve.

The complete statement of the resolution rule is in Refs. 19 and 20. There are several variations of the resolution principle. The theorem prover in QA3 uses a variation (not that given by Robinson in Ref. 20) of resolution that employs another rule of inference, factoring. Given a clause $C = \{L_1 \vee L_2 \vee \beta\}$, where L_1 and L_2 are literals and β is a disjunction of literals, if L_1 and L_2 unify with the most general unifier σ (thus $L_1\sigma = L_2\sigma$), infer the factor $C' = (L_1\sigma \vee \beta\sigma)$.

The resolution rule tells us how to derive a new clause from a specified pair of clauses containing a specified literal, but does not tell us how to choose which clauses to resolve. A mechanical attempt to resolve all possible pairs of clauses generally results in the generation of an unmanageably large number of irrelevant clauses. Therefore, various heuristic search principles have been developed to guide and control the selection of clauses for resolution. Among the most important of these are the set of support,[33] unit preference,[34] and subsumption[20] strategies. All these strategies preserve completeness of the theorem prover.

The statement of a theorem to be proved usually consists of a set of premises (axioms) and a conclusion. The set-of-support strategy consists of designating the conclusion, and perhaps a small number of the most relevant axioms, as "having the support property"--i.e., lying in the set of support for the theorem. Thereafter, only those pairs of clauses containing at least one member with support are considered for resolution, and every resolvent is automatically attributed the support property. This strategy is aimed at avoiding the deduction of consequences for some of the premises that are independent of (and irrelevant to) the particular conclusion desired. The extended set-of-support[35] strategy is like the set-of-support strategy, but non-set-of-support clauses are allowed to resolve together or be factored, if the resultant clause is less than a given level. The intent of this strategy is to allow a potential "lemma" to be produced by, say, resolving two axioms. If the lemma is used several times in the proof, less search is required.

The unit-preference strategy essentially orders the clauses to be resolved by their length--i.e., by the number of literals they contain. Contradictions become apparent only when two unit (one-literal) clauses resolve together to produce the empty clause. Therefore, one might hope to discover a contradiction in the least time by working first with the shortest clauses. This strategy says to first produce the shortest resolvent possible in which at least one of the "parent" clauses is a unit. If no such resolutions are possible, attempt to produce the shortest possible resolvent or factor next.

Occasionally any strategy like the unit-preference strategy may cause one to continue to resolve sequences of unit resolutions to the neglect of longer but perhaps more fruitful clauses. This difficulty can be overcome by placing a bound on computation that will determine when the unit-preference strategy should be abandoned in favor of a broader search. One such bound sets a maximum on the number of levels--i.e., intermediate steps, between a deduced clause and the original theorem. Of course, these bounds cause loss of completeness.

In the course of a resolution proof, several clauses may be introduced that carry equivalent information and therefore lead to distracting, extraneous steps. In particular, if C is any clause, and if $C_0 = C\sigma$ is obtainable as an instance of C by some substitution σ, and if clause

$D = C_0 \vee \alpha$, where α is any formula, then C subsumes D in the sense that the set of clauses $\{C,D\}$ is satisfiable if and only if C alone is satisfiable. Therefore, we delete from our proof any clause that is subsumed by another clause in the proof.

The proof procedure implemented as part of QA3 is a resolution procedure using some form of each of the above search strategies, as well as extensions thereto.

As an example of a proof using resolution, set-of-support strategy, and unit-preference strategy, let the axioms be

Axiom 1	$P(a)$
Axiom 2	$(\exists y)Q(y)$
Axiom 3	$P(a) \supset R(a)$
Axiom 4	$(\forall x)[P(x) \wedge R(x) \supset Q(g(x))]$

where a is a constant, g is a function letter, and P, Q, and R are predicate letters. The axioms are converted to the following corresponding clauses:

Clause 1	$P(a)$	from Axiom 1
Clause 2	$Q(b)$	from Axiom 2
Clause 3	$\sim P(a) \vee R(a)$	from Axiom 3
Clause 4	$\sim P(x) \vee \sim R(x) \vee Q(g(x))$	from Axiom 4.

The constant "b" in Clause 2 is the Skolem function of no arguments generated by the elimination of $(\exists y)$ in Axiom 2. The theorem to be proved from these axioms is

$$(\exists x)Q(g(x)) \quad .$$

The clause representing the negation of the theorem is

Clause 5	$\sim Q(g(x))$	from negation of theorem.

We show that this set of clauses is unsatisfiable. From the negation of the theorem, suppose Clause 5 is selected (as is typical) as the only clause in the set of support. Following the unit-preference strategy, the first inference attempted is to resolve Clause 5 with Clause 1, a unit clause, which fails. Similarly, Clause 5 does not resolve with Clause 2. Then Clause 5 fails to resolve with Clause 3, a two-clause (clause of length 2). Finally, Clause 5 resolves with Clause 4, producing

<u>Clause 6</u> $\sim P(x) \vee \sim R(x)$ from 4 and 5;

then Clause 6 resolves with the unit Clause 1, yielding

<u>Clause 7</u> $\sim R(a)$ from 1 and 6;

then

<u>Clause 8</u> $\sim P(a)$ from 3 and 7;

then

<u>Clause 9</u> contradiction from 1 and 8,

completing the proof. (The QA3 theorem-proving program is more clever than the strategy outlined above. For example, it would never even attempt to resolve Clause 5 with Clause 1, since they share no common predicate letter. The details of the real strategy used are given in Sec. IV-C).

Observe that there is an alternate proof if the unit-preference strategy is not used. The axioms and the negation of the theorem are the same as before. First, Clause 6 can be produced from 4 and 5 as before.

<u>Clause 6</u> $\sim P(x) \vee \sim R(x)$ from 4 and 5.

Then Clause 6 and Clause 3 resolve to produce

<u>Clause 7′</u> $\sim P(a) \vee \sim P(x)$ from 3 and 6.

By the other rule of inference, factoring, we have

<u>Clause $8'$</u> $\sim P(a)$ from $7'$.

Finally,

<u>Clause 9</u> contradiction from 1 and $8'$,

completing the proof.

As shown by the first proof of the above theorem, a proof is sometimes possible without factoring, but, in general, factoring is necessary for completeness.

III THE THEOREM-PROVING APPROACH TO QUESTION ANSWERING

A. Introduction to the Formal Approach

The use of a theorem prover as a question answerer can be explained very simply. The question answerer's knowledge of the world is expressed as a set of axioms, and the questions asked it are presented as theorems to be proved. The process of proving the theorem is the process of deducing the answer to the question. For example, the fact "George is at home" is presented as the axiom, AT(George,home). The question "Is George at home?" is presented as the conjectured theorem, AT(George,home). If this theorem is proved true, the answer is yes. (In this simple example the theorem is obviously true since the axiom is the theorem.) The theorem prover can also be used to find or construct an object satisfying some specified conditions. For example, the question "Where is George?" requires finding the place x satisfying AT(George,x). The theorem prover is embedded in a system that controls the theorem prover, manages the data base, and interacts with the user. These ideas are explained in more detail later in Sec. IV.

Even though it might be clear that theorem proving can be used for question answering, why would one want to use these very formal methods? One answer is that one is seeking generality. Theorem proving may be a good approach to the achievement of generality for several reasons:

(1) The language is well defined, unambiguous, and rather general, so that one can hope to describe many desired subjects, questions, or answers.

(2) The proof procedure used allows all possible interactions among the axioms and is logically "complete"--i.e., if a theorem is a logical consequence of the axioms, then this procedure will find a proof, given enough time and space. This completeness property is important, since several general question-answering programs have resulted in incomplete deductive systems, even in the practical sense of being unable to answer some simple types of questions that are short, reasonable deductions from

the stored facts--e.g., the author's QA1,[1] Raphael's SIR[4] and Slagle's DEDUCOM.[31]

(3) The theorem prover is subject-independent, so to describe a new subject or modify a previous description of a subject, only the axioms need to be changed, and it is not necessary to make any changes in the program.

(4) Theorem provers are becoming more efficient. Even though the theorem-proving method used is theoretically complete, in practice its ability to find proofs is limited by the availability of computer time and storage space. However, the kind of theorem proving--resolution--used by the program described herein has been developed to the point of having several good heuristics. Further improvements in theorem proving are ahead, and, hopefully, the improvements will carry over into corresponding improvements in question answering. It should be possible to communicate precisely new theorem-proving results to other researchers, and it is relatively easy to communicate precisely particular formalizations or axiomatizations of subjects.

B. An Explanatory Dialogue

The explanation of question answering given in this section will be illustrated primarily by the techniques used in a working question-answering program called QA3 (see Sec. IV) and is on the SDS 940 computer, which has a time-sharing system. The user works at a teletype, entering statements and questions, and receiving replies. The notation we present in this thesis is slightly different from the actual computer input and output, as the character set available on the teletype does not contain the symbols we use here. QA3 is an outgrowth of QA2[1] (see Appendix A), but is somewhat more sophisticated and practical, and is now being used for several applications.

Facts are presented as statements of first-order logic. The statement is preceded by STATEMENT to indicate to the program that it is a statement. These statements (axioms) are automatically converted to clauses and stored in the memory of the computer. The memory is a list structure indexed by the predicate letters, function symbols, and constant symbols occurring in each clause. A statement can be a very specific fact such as

STATEMENT: COLOR(book,red)

corresponding to the common attribute-object-value triple. A statement can also be a more general description of relations, such as:

STATEMENT: $(\forall x)(\forall A)(\forall B)[A \subseteq B \wedge x\epsilon A \supset x\epsilon B]$

meaning that if A is a subset of B and if x is an element of A, then x is an element of B.

Questions are also presented as statements of first-order logic. QUESTION is typed before the question. This question becomes a conjecture and QA3 attempts to prove the conjecture in order to answer YES. If the conjecture is not proved, QA3 attempts to prove the negation of this question in order to answer NO. The theorem prover attempts a proof by refutation. During the process of searching for a proof, clauses that may be relevant to a proof are extracted from memory and utilized as axioms. If the question is neither proved nor disproved, then a NO PROOF FOUND answer is returned. ANSWER indicates an answer.

We now present a very simple dialogue with QA3. The dialogue illustrates a "yes" answer, a "no" answer, and an "or" answer. Questions 4, 7, and 8 below illustrate questions whose answer is a term generated by the proof procedure. These kinds of answers will be called "constructive" answers.

(1) The first fact is "Smith is a man."

STATEMENT: MAN(Smith)

OK

The OK response from QA3 indicates that the statement is accepted, converted to a clause, and stored in memory.

(2) We ask the first question, "Is Smith a man?"

QUESTION: MAN(Smith)

ANSWER: YES

(3) We now state that "Man is an animal," or, more precisely, "If x is a man then x is an animal."

STATEMENT: $(\forall x)[\mathrm{MAN}(x) \supset \mathrm{ANIMAL}(x)]$

OK

(4) We now ask "Who is an animal?" This question can be restated as "Find some y that is an animal" or "Does there exist a y such that y is an animal? If so, exhibit such a y."

QUESTION: $(\exists y)\mathrm{ANIMAL}(y)$

ANSWER: YES, y = Smith

The YES answer indicates that the conjecture $(\exists y)\mathrm{ANIMAL}(y)$ has been proved (from Statements 1 and 3 above). "y = Smith" indicates that "Smith" is an instance of y satisfying ANIMAL(y)--i.e., ANIMAL(Smith) is a theorem.

(5) Fact: Every robot is a machine.

STATEMENT: $(\forall x)[\mathrm{ROBOT}(x) \supset \mathrm{MACHINE}(x)]$

OK

(6) Fact: Rob is a robot.

STATEMENT: $\mathrm{ROBOT(Rob)}$

OK

(7) Fact: No machine is an animal.

STATEMENT: $(\forall x)[\mathrm{MACHINE}(x) \supset \sim\mathrm{ANIMAL}(x)]$

OK

(8) The question "Is everything an animal?" is answered NO. A counterexample is exhibited--namely, Rob the robot.

QUESTION: $(\forall x)\mathrm{ANIMAL}(x)$

ANSWER: NO, $x = \mathrm{Rob}$

The answer indicates that $\sim\mathrm{ANIMAL(Rob)}$ is a theorem. Note that a NO answer produces a counterexample for the universally quantified variable x. This is a dual of the construction of a satisfying instance for an existentially quantified variable in a question answered YES.

(9) Fact: Either Smith is at work or Jones is at work.

STATEMENT: $\mathrm{AT(Smith,work)} \vee \mathrm{AT(Jones,work)}$

OK

(10) "Is anyone at work? If so, who?"

QUESTION: $(\exists x)\mathrm{AT}(x,\mathrm{work})$

ANSWER: YES, $x = \mathrm{Smith}$
<u>or</u> $x = \mathrm{Jones}$

From the previous statement it is possible to prove that someone is at work, although it is not possible to specify a unique individual.

Statements, questions, and answers can be more complex so that their corresponding English form is not so simple. Statements and questions can have many quantifiers and can contain functions. The answer can also contain functions. Consider the question "Is it true that for all x there exists a y such that P(x,y) is true?" where P is some predicate letter. Suppose QA3 is given the statement,

(11) STATEMENT: $(\forall z)P(z,f(z))$

where f is some function. We ask the question

(12) QUESTION: $(\forall x)(\exists y)P(x,y)$

ANSWER: YES, $y = f(x)$

Notice that the instance of y found to answer the question is a function of x, indicating the dependence of y on x. Suppose that instead of Statement 11 above, QA3 has other statements about P. An answer to Question 12 might be

ANSWER: NO, $x = a$

where "a" is some instance of x that is a counterexample.

A term in the answer can be either a constant, a function, a variable, or some combination thereof. If the answer is a constant or a known function, then the meaning of the answer is clear. However, the answer may be a Skolem function generated by dropping existential quantifiers. In this case, the answer is an object asserted to exist by the existential quantifier that generated the Skolem function. To know the meaning of this Skolem function, the system must exhibit the original input statement that caused the production of the Skolem function. Free variables in clauses correspond to universally quantified variables, so if the answer is a free variable, then any term satisfies the formula and thus answers the question.

Two more types of answers are NO PROOF FOUND and INSUFFICIENT INFORMATION. Suppose the theorem prover fails to prove some conjecture and also fails to disprove the conjecture. If the theorem prover runs out of time or space during either the attempted "yes" proof or the attempted "no" proof, then there is the possibility that some proof is possible if more time or space is available. The answer in this case is NO PROOF FOUND.

Now suppose both proof attempts fail without exceeding any time or space limitations. The theorem-proving strategy is complete so that if no time or space limitation halts the search for a proof and the conjecture is a logical consequence of the axioms, then a proof will be found. So we know that neither a "yes" nor a "no" answer is possible from the given statements. The answer returned is INSUFFICIENT INFORMATION. For example, suppose QA3 has no statements containing the predicate letter "R".

QUESTION: $(\exists x)R(x)$

The negated question is the clause $\{\sim R(x)\}$, and no other clauses in the memory of QA3 can resolve with it. Thus the system will respond

ANSWER: INSUFFICIENT INFORMATION .

C. Constructing Answers

The Resolution method of proving theorems allows us to produce correct constructive answers. This means that if, for example, $(\exists x)P(x)$ is a theorem, then the proof procedure can find terms $t_1, t_2, \ldots, t_n$ such that $P(t_1) \vee P(t_2) \ldots \vee P(t_n)$ is a theorem.

First, we will present some examples of answer construction. After these examples we will show how a proof by resolution can be used to generate an answer.

Examples of answer construction will be explained by means of the ANSWER predicate used by QA3 to keep track of instantiations. Consider the question

QUESTION: $(\exists y)\text{ANIMAL}(y)$

which is negated to produce the clause

$$\{\sim\text{ANIMAL}(y)\} \quad .$$

The special literal, $\underline{\text{ANSWER}}(y)$, is added to this clause to give

$$\{\sim\text{ANIMAL}(y) \vee \underline{\text{ANSWER}}(y)\} \quad .$$

The proof process begins with this clause. When the literal ANIMAL(x) is resolved against the literal ~ANIMAL(y), the term y is instantiated to yield the term x. In the new clause resulting from this resolution, the argument of ANSWER is then x. In the next resolution the argument of ANSWER becomes Smith. We list the complete proof of the clause $\{\underline{\text{ANSWER}}(\text{Smith})\}$.

(1)	$\{\sim\text{ANIMAL}(y) \vee \underline{\text{ANSWER}}(y)\}$	Modified negation of the question.
(2)	$\{\sim\text{MAN}(x) \vee \text{ANIMAL}(x)\}$	Axiom fetched from memory.
(3)	$\{\sim\text{MAN}(x) \vee \underline{\text{ANSWER}}(x)\}$	From resolving 1 and 2.
(4)	$\{\text{MAN}(\text{Smith})\}$	Axiom fetched from memory.
(5)	$\{\underline{\text{ANSWER}}(\text{Smith})\}$	"Contradiction" from 3 and 4 for y = Smith.

The first clause can be interpreted as "For every y, either y is not an animal or else y is an answer." The second clause means "For all x, x is an animal or x is not a man." From these two statements, we deduce the third clause, "For all x, either x is not a man or x is an answer." Clause 4 states that Smith is a man, and we deduce that Smith is an answer. The argument of the ANSWER predicate is the instance of y (namely, Smith) that answers the question. QA3 returns

ANSWER: YES, y = Smith .

This answer means, as will be explained later, that

ANIMAL(Smith)

is a theorem.

The ANSWER literal is added to each clause in the negation of the question. The arguments of ANSWER are the existentially quantified variables in the question. When a new clause is created, each ANSWER literal in the new clause is instantiated in the same manner as any other literal from the parent clause. However, the ANSWER literal is treated specially; it is considered to be invisible to resolution in the sense that no literal is resolved against it and it does not contribute to the length (size) of the clause containing it. We call a clause containing only ANSWER literals an "answer clause." The search for an answer (proof) successfully terminates when an answer clause is generated. The addition of the ANSWER predicate to the clauses representing the negation of the theorem does not affect the completeness of this modified proof procedure. The theorem prover generates the same clauses, except for the ANSWER predicate, as the conventional theorem prover. Thus in this system an answer clause is equivalent to the empty clause that establishes a contradiction in a conventional system.

An answer clause specifies the sets of values that the existentially quantified variables in the question may take in order to preserve the provability of the question. The precise meaning of the answer will be specified in terms of a question Q that is proved from a set of axioms $B = \{B_1, B_2, \ldots, B_b\}$.

As an example illustrating some difficulties with Skolem functions, let the axioms B consist of a single statement,

$$\text{STATEMENT:} \quad (\forall z)(\exists w)P(z,w) \quad .$$

Suppose this is converted to the clause

$$\{P(z,f(z))\}$$

where $f(z)$ is the Skolem function due to the elimination of the quantifier $(\exists w)$. We ask the question Q,

$$\text{QUESTION:} \quad (\forall y)(\exists x)P(y,x) \quad .$$

The negation of the question is $\sim Q$,

$$(\exists y)(\forall x)\sim P(y,x) \quad .$$

The clause representing $\sim Q$ is $\{\sim P(b,x)\}$, where b is the constant (function of no variables) introduced by the elimination of $(\exists y)$. Adding the answer literal, the initial clause in the proof is

$$\{\sim P(b,x) \vee \underline{\text{ANSWER}}(x)\} \quad .$$

The proof, obtained by resolving these two clauses, yields the answer clause

$$\{\underline{\text{ANSWER}}(f(b))\} \quad .$$

The Skolem function b is replaced by y, and the answer printed out is

$$\text{ANSWER:} \quad \text{YES, } x = f(y) \quad . \qquad (1)$$

At present in QA3 the Skolem function $f(y)$ is left in the answer. To help see the meaning of some Skolem function in the answer, the user can ask the system to display the original statement that, when converted to clauses, caused the generation of the Skolem function.

As an illustration, consider the following interpretation of the statement and question of this example. Let $P(u,v)$ be true if u is a person at work and v is this person's desk. Then the statement $(\forall z)(\exists w)P(z,w)$ asserts that every person at work has a desk, but the statement does not name the desk. The Skolem function $f(z)$ is created internally by the program during the process of converting the statement $(\forall z)(\exists w)P(z,w)$ into the clause $\{P(z,f(z))\}$. The function $f(z)$ may be thought of as the program's internal name for z's desk. [The term f(z) could perhaps be written more meaningfully in terms of the descriptive operator ι as "$\iota w.P(z,w)$"--i.e., "the w such that $P(z,w)$," although w is not necessarily unique.]

The question $(\forall y)(\exists x)P(y,x)$ asks if for every person y there exists a corresponding desk. The denial of the question, $(\exists y)(\forall x){\sim}P(y,x)$, postulates that there exists a person such that for all x, it is not the case that x is his desk. The Skolem function of no arguments, b, is also created internally by the program as it generates the clause $\{{\sim}P(b,x)\}$. The function b is thus the program's internal name for the hypothetical person who has no desk.

The one-step proof merely finds that b does have a desk--namely, f(b). The user of the system does not normally see the internal clause representations unless he specifically requests such information. If the term f(b) that appears in the answer clause were given to the user as the answer--e.g., YES, $x = f(b)$--the symbols f and b would be meaningless to him. But the program remembers that b corresponds to y, so b is replaced by y, yielding a slightly more meaningful answer, YES, $x = f(y)$. The user then knows that y is the same y he used in the question. The significance of the Skolem function f is slightly more difficult to express. The program must tell the user where f came from. This is done by returning the original statement $(\forall z)P(z,f(z))$ to the user [alternatively, the descriptive operator could be used to specify that f(z) is "$\iota w.P(z,w)$"]. As a rule, the user remembers, or has before his eyes, the question, but the specific form of the statements (axioms) is forgotten. In this very simple example the meaning of f is specified completely in terms of the question predicate P, but in general the meanings of Skolem functions will be expressed in terms of other predicates, constants, etc.

The exact meaning of the answer $x = f(y)$ is that the statement

$$(\forall y)P(y,f(y))$$

follows from the axioms. For this example, this statement _is_ an axiom clause, so it obviously follows from the axiom clauses. In general the precise meaning of an answer may not be so obvious.

The statement above is called the "answer statement." In the next section, we will show in general how to construct an answer statement. The answer statement will be a wff in prenex form, that (1) has _only_ universal quantifiers, (2) contains no Skolem functions from the negation of the theorem, (3) is a logical consequence of the axiom clauses, and (4) provides an exact meaning for the answer.

D. The Answer Statement

We will now show how to construct an "answer statement," and then we will prove that the answer statement is a logical consequence of the axiom clauses. On some questions the user may require that an answer statement be exhibited, in order to better understand the meaning of a complicated answer.

Consider a proof of question Q from the set of axioms $B = \{B_1, B_2, \ldots, B_b\}$. B logically implies Q if and only if $B \wedge \sim Q$ is unsatisfiable. The statement $B \wedge \sim Q$ can be written in prenex form $PM(Y,X)$, where P is the quantifier prefix, $M(Y,X)$ is the matrix, $Y = \{y_1, y_2, \ldots, y_u\}$ is the set of existentially quantified variables in P, and $X = \{x_1, x_2, \ldots, x_e\}$ is the set of universally quantified variables in P.

Eliminating the quantifier prefix P by introducing Skolem functions to replace existential quantifiers and dropping the universal quantifiers produces the formula $M(U,X)$. Here U is the set of terms $\{u_1, u_2, \ldots, u_u\}$, such that for each existentially quantified variable y_i in P, u_i is the corresponding Skolem function of all the universally quantified variables in P preceding y_i. Let $M(U,X)$ be called S. The statement $B \wedge \sim Q$ is unsatisfiable if and only if the corresponding statement S is unsatisfiable. Associated with S is a Herbrand Universe of terms H that includes X, the set of free variables of S. If $\varphi = \{t_1/x_1, t_2/x_2, \ldots, t_n/x_n\}$ represents a substitution of terms $t_1, t_2, \ldots, t_n$ from H for the variables $x_1, x_2, \ldots, x_n$, then the formula $S\varphi$ denotes the instance of S over H formed by substituting the terms $t_1, t_2, \ldots, t_n$ from H for the corresponding variables $x_1, x_2, \ldots, x_n$ in S.

Let S_i represent a variant of S--i.e., a copy of S with the free variables renamed. Let the free variables be renamed in such a way that no two variants S_i and S_j have variables in common. By the Skolem-Löwenheim-Gödel theorem,[19] S is unsatisfiable if and only if there exists an instance of a finite conjunction of variants of S that is truth-functionally unsatisfiable. A resolution theorem prover proves S unsatisfiable by finding such a finite conjunction.

Suppose the proof of Q from B finds the conjunction $S_1 \wedge S_2 \wedge \ldots \wedge S_k$ and the substitution θ such that

$$(S_1 \wedge S_2 \wedge \ldots \wedge S_k)\theta$$

is truth-functionally unsatisfiable. Let F_0 denote the formula

$$(S_1 \wedge S_2 \wedge \ldots \wedge S_k)\theta \quad .$$

Let L be the conjunction of variants of M(Y,X),

$$L = M(Y_1,X_1) \wedge M(Y_2,X_2) \wedge \ldots \wedge M(Y_k,X_k)$$

and let λ be the substitution of Skolem functions for variables such that

$$\begin{aligned} L\lambda &= M(U_1,X_1) \wedge M(U_2,X_2) \wedge \ldots \wedge M(U_k,X_k) \\ &= S_1 \wedge S_2 \wedge \ldots \wedge S_k \quad . \end{aligned}$$

Thus, $L\lambda\theta = F_0$.

Before constructing the answer statement, observe that the Skolem functions of F_0 can be removed as follows. Consider the set $U = \{u_1, u_2, \ldots, u_u\}$ of Skolem-function terms in S. Find in F_0 one instance--say, u_1'--of a term in U. Select a symbol, z_1, that does not occur in F_0. Replace every occurrence of u_1' in F_0 by z_1, producing statement F_1. Now again apply this procedure to F_1, substituting a new variable throughout F_1 for each occurrence of some remaining instance of a Skolem-function term in F_1, yielding F_2. This process can be continued until no further instances of terms from U are left in F_n, for some n.

The statement F_i for $0 \leq i \leq n$ is also truth-functionally unsatisfiable for the following reasons. Consider any two occurrences of atomic formulas--say m_a and m_b--in F_0. If m_a and m_b in F_0 are identical, then the corresponding two transformed atomic formulas m_{a1} and m_{b1} in F_2 are identical. If m_a and m_b are not identical, then m_{a1} and m_{b1} are not identical. Thus, F_1 must have the same truth table, hence truth value, as F_0. This property holds at each step in the construction, so $F_0, F_1, \ldots, F_n$ must each be truth-functionally unsatisfiable.

This term-replacement operation can be carried out directly on the substitutions--i.e., for each statement F_i, $0 \leq i \leq n$, there exists a substitution σ_i such that $F_i = L\sigma_i$. We prove this by showing how such a σ_i is constructed. Let $\sigma_0 = \lambda\theta = \{t_1/v_1, t_2/v_2, \ldots, t_p/v_p\}$. By definition, $F_0 = L\sigma_0$. Let t_j' denote the term formed by replacing every occurrence of u_1' in t_j by z_1. The substitution $\sigma_1 = \{t_1'/v_1, t_2'/v_2, \ldots, t_p'/v_p\}$ applied to L yields F_1--i.e., $F_1 = L\sigma_1$. Similarly one constructs σ_i and shows, by induction, $F_i = L\sigma_i$, for $0 \leq i \leq n$.

Now let us examine some of the internal structure of F_0. Assume that S = M(U,X) is formed as follows. The axioms may be represented as $P_B B(Y_B, X_B)$, where P_B is the quantifier prefix, Y_B is the set of universally-quantified variables, and X_B is the set of existentially-quantified variables. These axioms are converted to a set of clauses denoted by $B(Y_B, U_B)$, where U_B is the set of Skolem-function terms created by eliminating X_B.

The question may be represented as $P_Q Q(Y_Q, X_Q)$, where P_Q is the quantifier prefix, Y_Q is the set of universally-quantified variables, and X_Q is the set of existentially-quantified variables. Assume that the variables of the question are distinct from the variables of the axioms. The negation of the question is converted into a set of clauses denoted by $\sim Q(U_Q, X_Q)$, where U_Q is the set of Skolem-function terms created by eliminating Y_Q. The function symbols in U_Q are distinct from the function symbols in U_B. Thus, $M(U,X) = [B(Y_B,U_B) \wedge \sim Q(U_Q,X_Q)]$. Now let $L_B = [B(Y_{B1},X_{B1}) \wedge B(Y_{B2},X_{B2}) \wedge \ldots \wedge B(Y_{Bk},X_{Bk})]$ and let $\sim L_Q = [\sim Q(Y_{Q1},X_{Q1}) \wedge \sim Q(Y_{Q2},X_{Q2}) \wedge \ldots \wedge \sim Q(Y_{Qk},X_{Qk})]$. Thus, $L = L_B \wedge \sim L_Q$.

Observe that one can construct a sequence of formulas $F_0',F_1',\ldots,F_m'$ (similar to the sequence $F_0,F_1,\ldots,F_n$) in which the only terms replaced by variables are those terms that are instances of terms in U_Q. This construction process terminates when, for some m, the set of clauses F_m' contains no further instances of terms in U_Q. By the same argument given earlier, each formula F_i' is truth-functionally unsatisfiable. Similarly, one can construct from $\lambda\theta$ a sequence of substitutions $\sigma_0',\sigma_1',\ldots,\sigma_m'$ such that $L\sigma_i' = F_i'$. Let $\sigma = \sigma_m'$.

To construct the answer statement, substitute σ into L_Q, forming

$$L_Q\sigma = [Q(Y_{Q1},X_{Q1})\sigma \vee Q(Y_{Q2},X_{Q2})\sigma \vee \ldots \vee Q(Y_{Qk},X_{Qk})\sigma] \quad .$$

Since σ replaces the elements of Y_{Qj} by variables, let the set of variables Z_{Qj} denote $Y_{Qj}\sigma$. Thus,

$$L_Q\sigma = [Q(Z_{Q1},X_{Q1}\sigma) \vee Q(Z_{Q2},X_{Q2}\sigma) \vee \ldots \vee Q(Z_{Qk},X_{Qk}\sigma)] \quad .$$

Now, let Z be the set of all variables occurring in $L_Q\sigma$. The <u>answer statement</u> is defined to be $(\forall Z)L_Q\sigma$. In its expanded form the answer statement is

$$(\forall Z)[Q(Z_{Q1},X_{Q1}\sigma) \vee Q(Z_{Q2},X_{Q2}\sigma) \vee \ldots \vee Q(Z_{Qk},X_{Qk}\sigma)] \quad . \tag{2}$$

We now prove that the answer statement is a logical consequence of the axioms in their clausal form. Suppose not; then $B(U_B,X_B) \wedge \sim L_Q\sigma$ is satisfiable; thus, $B(U_B,X_B) \wedge (\exists Z)\sim L_Q\sigma$ is satisfiable, implying that the conjunction of its instances $L_B\lambda \wedge (\exists Z)\sim L_Q\sigma$ is satisfiable. Now drop the existential quantifiers $(\exists Z)$. Letting the elements of Z in $\sim L_Q\sigma$ denote a set of constant symbols or Skolem functions of no arguments, the resulting formula $L_B\lambda \wedge \sim L_Q\sigma$ is also satisfiable.

Note that $L_B\sigma$ is an instance of $L_B\lambda$. To see this, let λ_B be the restriction of λ to variables in L_B. Thus, $L_B\lambda = L_B\lambda_B$. Suppose $\theta = \{r_1/w_1, r_2/w_2, \ldots, r_p/w_p\}$. Recall that σ is formed from $\lambda\theta$ by replacing in $\lambda\theta$ each occurrence of each instance--say, u_q'--of a "question" Skolem term by an appropriate variable. The "axiom" Skolem functions are distinct from the question Skolem functions, and occur only in λ_B. Thus no such u_q' is an instance of an axiom Skolem term. Therefore each occurrence of each such u_q' in $\lambda_B\theta$ must arise from an occurrence of u_q' in some r_j in θ. Thus, $L_B\sigma = L_B\lambda_B\varphi$, where the substitution $\varphi = \{r_1'/w_1, r_2'/w_2, \ldots, r_p'/w_p\}$ is formed from θ by replacing each such u_q' in each r_j by an appropriate variable. Since $L_B\lambda = L_B\lambda_B$, $L_B\lambda\varphi = L_B\sigma$. Since the only free variables of $L_B\lambda \wedge {\sim}L_Q\sigma$ occur in $L_B\lambda$, $[L_B\lambda \wedge {\sim}L_Q\sigma]\varphi = L_B\lambda\varphi \wedge {\sim}L_Q\sigma$.

The formula $L_B\lambda\varphi \wedge {\sim}L_Q\sigma$ logically implies all of its instances, in particular the instance $L_B\lambda\varphi \wedge {\sim}L_Q\sigma$. Thus, if $L_B\lambda \wedge {\sim}L_Q\sigma$ is satisfiable, its instance $L_B\lambda\varphi \wedge {\sim}L_Q\sigma$ is satisfiable. Since $[L_B\lambda\varphi \wedge {\sim}L_Q\sigma] = [L_B\sigma \wedge {\sim}L_Q\sigma] = [L_B \wedge {\sim}L_Q]\sigma = L\sigma = F_m'$ for some m, F_m' must be satisfiable. This contradicts our earlier result that F_m' is truth-functionally unsatisfiable, and thus proves that the answer statement is a logical consequence of the axiom clauses.

We make one further refinement of the answer statement (2). It is unnecessary to include the j^{th} disjunct if $X_{Qj}\sigma = X_{Qj}$--i.e., if σ does not instantiate X_{Qj}. Without loss of generality, we can assume that for $r \le k$, the last $k - r$ disjuncts are not instantiated--i.e.,

$$X_{Qr+1}\sigma = X_{Qr+1},\ X_{Qr+2}\sigma = X_{Qr+2},\ \ldots,\ X_{Qk}\sigma = X_{Qk} \quad .$$

Then the stronger answer statement

$$(\forall Z)[Q(Z_{Q1},X_{Q1}\sigma) \vee Q(Z_{Q2},X_{Q2}\sigma) \vee \ldots \vee Q(Z_{Qr},X_{Qr}\sigma)] \tag{3}$$

is logically equivalent to (2). [Since the matrix of (3) is a subdisjunct of (2), (3) implies (2). If $j \le r$, the j^{th} disjunct of (2) implies the j^{th} disjunct of (3). If $r < j \le k$, the j^{th} disjunct of (2) implies all of its instances, in particular all disjuncts of (3).]

The ANSWER predicate provides a simple means of finding the instances of Q in (3). Before the proof attempt begins, the literal ANSWER(X_Q) is added to each clause in $\sim Q(U_Q, X_Q)$. The normal resolution proof procedure then has the effect of creating new variants of X_Q as needed. The j^{th} variant, ANSWER(X_{Qj}), thus receives the instantiations of $\sim Q(U_{Qj}, X_{Qj})$. When a proof is found, the answer clause will be

$$\{\underline{\text{ANSWER}}(X_{Q1}\theta) \vee \underline{\text{ANSWER}}(X_{Q2}\theta) \vee \ldots \vee \underline{\text{ANSWER}}(X_{Qr}\theta)\} \quad .$$

Variables are then substituted for the appropriate Skolem functions to yield

$$\{\text{ANSWER}(X_{Q1}\sigma) \vee \text{ANSWER}(X_{Q2}\sigma) \vee \ldots \vee \text{ANSWER}(X_{Qr}\sigma)\} \quad .$$

Let $X_{Qj} = \{x_{j1}, x_{j2}, \ldots, x_{jm}\}$. Let σ restricted to X_{Qj} be $\{t_{j1}/x_{j1}, t_{j2}/x_{j2}, \ldots, t_{jm}/x_{jm}\}$. The answer terms printed out by QA3 are

$$\begin{array}{ll} & [x_{11} = t_{11} \text{ and } x_{12} = t_{12} \text{ and } \ldots \text{ and } x_{1m} = t_{1m}] \\ \text{or} & \\ & [x_{21} = t_{21} \text{ and } x_{22} = t_{22} \text{ and } \ldots \text{ and } x_{2m} = t_{2m}] \\ \text{or} & \\ & \cdot \\ & \cdot \\ & \cdot \\ \text{or} & \\ & [x_{r1} = t_{r1} \text{ and } x_{r2} = t_{r2} \text{ and } \ldots \text{ and } x_{rm} = t_{rm}] \quad . \end{array} \qquad (4)$$

According to (3), all the free variables in the set Z that appear in the answer are universally quantified. Thus, any two occurrences of some free variable in two terms must take on the same value in any interpretation of the answer.

In the example given above whose answer (1) had the single answer term f(y), the complete answer statement is

$$(\forall y)P(y, f(y)) \quad .$$

In Sec. VI-A we present more examples.

The answer statement proved can sometimes be simplified. For example, consider

QUESTION: $(\exists x)P(x)$

ANSWER: YES, $x = a$
or $x = b$,

meaning that the answer statement proved is

$$[P(a) \vee P(b)] \quad .$$

Suppose it is possible to prove $\sim P(b)$ from other axioms. Then a simpler answer is provable--namely,

ANSWER: YES, $x = a$.

On some problems an "or" answer is not allowed. One example is in the program-writing problem. To prevent "or" answers, the theorem prover is not allowed to create any clauses having two or more answer literals that do not unify.

IV QA3, A QUESTION-ANSWERING PROGRAM

In this section we describe the principal features of the QA3 program. QA3 is a system of programs written in the LISP language on the SDS 940 computer. The design goal of the system is the embedding of theorem-proving programs in a usable question-answering system. There is a "standard" proof strategy available that is designed for quick answering of easy questions. The strategy is flexible so that the program can be fitted to various applications. The user can observe and modify the proof process in an interactive mode. The system has two levels of memory, the first being a large data base of information that the user can easily modify. The second level is an active set of clauses; during a proof search, clauses are selected from the data base and added to an active set of clauses that the theorem prover considers.

A. QA3 Control Language

This section describes the control language that can be used in dialogues with QA3. The user can converse in this language, which is described below, with the top-level LISP program in the QA3 system. The principal commands are QUESTION and STATEMENT. described in the previous section. These commands are abbreviated Q and S, respectively. In the following discussion, a "meta-level" word surrounded by the brackets, $\langle\ \rangle$, names a type of entity--e.g., ⟨wff⟩ stands for "any well-formed formula."

1. Statements

A statement is entered in one of the following formats:

(1) S⟨wff⟩

(2) S⟨name⟩⟨wff⟩

where the letter S signifies that the wff is to be converted to clauses and then both wff and clauses are added to the system's data base. In Case 1 the statement is given an internally generated name of the form AX100. In Case 2 the user supplies the name of the axiom. The clauses are also named internally. If the axiom named AX17 is converted to three clauses, the clauses are named AX17-1, AX17-2, and AX17-3. The naming is optional. If the statement is accepted, the system responds with the names of the statement and clauses.

A wff is formed as in ordinary first-order predicate calculus (see Sec. II). An atomic formula is represented in LISP in prefix form--e.g., the atomic formula P(f(x),a) is presented to QA3 as (P(F X)A). Wff's are formed by using quantifiers and connectives as prefixes. The symbols used by QA3 to represent first-order logic symbols are:

QA3 Symbol	Logic Symbol	Meaning	Example
FA	$\forall$	"for all" - universal quantifier	(FA(X)(P X))
EX	$\exists$	"there exists" - existential quantifier	(EX(X)(P X))
IF,IMP	$\supset, \Rightarrow$	"implies" - implication	(IF(P A)(Q A))
AND	$\wedge$,&	"AND" - conjunction	(AND(P A)(P B))
OR	$\vee$	"OR" - disjunction	(OR(P A)(P B))
NOT	$\sim$,–	"not" - negation	(NOT(P A))
IFF,EQV	$\equiv, \Leftrightarrow$	"if and only if" - equivalence	(IFF(P A)(Q A))

An example of a wff is a predicate calculus statement such as

(IN JOHN BOY)

or

((FA(X Y Z)(IF(AND(IN X Y)(INCLUDE Y Z))(IN X Z))) .

The first states that John is a boy, or, more precisely, that John is an element of the set named Boy.

The second is equivalent to the predicate calculus statement:

$$(\forall x)(\forall y)(\forall z)[x \epsilon y \wedge y \subseteq z \supset x \epsilon z] \quad .$$

2. Questions

A question is entered in a similar fashion:

Q⟨wff⟩

where Q signifies that the wff that follows is to be treated as a question to the system. When a question is received, the negation of the question is put into conjunctive normal form and passed on to a subexecutive program that attempts to answer the question based on the current information in the data base. (Sec. III shows how various questions may be posed as wff's.)

3. Proofs

(1) UNWIND

After a question has been successfully answered, the UNWIND command will print the proof of the answer given to the question.

(2) CONTINUE

If the system was unsuccessful in answering a question, the CONTINUE command will cause the system to continue searching for proof with the level bound raised. Level bound is the maximum depth of the search tree, measured by the number of steps of resolution or factoring required. The initial value of the level bound is set by the user.

(3) STATUS

STATUS lists the relevant parameters of the system such as level bound, term depth bound, etc., along with their current values.

4. Editing the Data Base

(1) LIST⟨pℓ⟩

The command LIST⟨pℓ⟩ will list all of the input statements in the data base that contain the predicate letter ⟨pℓ⟩.

(2) LISTC⟨pℓ⟩

The command LIST⟨pℓ⟩ will list all of the clauses in the data base that contain the predicate letter ⟨pℓ⟩

(3) FORGET⟨pℓ⟩⟨n⟩

The command FORGET⟨pℓ⟩⟨n⟩, where ⟨n⟩ is an integer, will cause the $\langle n\rangle^{th}$ statement in the list generated by

LIST⟨pℓ⟩ to be deleted.

(4) FORGETC⟨pℓ⟩⟨n⟩

The command FORGETC⟨pℓ⟩⟨n⟩, where ⟨n⟩ is an integer, will cause the $\langle n\rangle^{th}$ clause in the list generated by the command LISTC⟨pℓ⟩ to be deleted.

(5) WRITE⟨file⟩

The command WRITE⟨file⟩, where ⟨file⟩ is the name of a file (tape, disc, drum, or core), creates a file of that name. The file contains the commands entered after the WRITE⟨file⟩ command. The command STOP terminates the file.

(6) RUN⟨file⟩

The command RUN⟨file⟩ causes each of the commands in the file named ⟨file⟩ to be executed.

In addition to the editing commands listed here, there are other QA3 commands, special LISP functions, and LISP system functions for editing. These facilities allow list-structure editing, QA3 file editing, accessing statements and clauses by their names, data-base transferring (to be used to transfer a data base or a subset thereof to the new version on the occasions when QA3 is revised), etc.

B. Control of the Search Process

The "standard" strategy described in Sec. IV-C, below, is satisfactory for many question-answering applications, as illustrated in Sec. V. However, for applications involving difficult problem solving or for applications requiring a flexible question-answering or theorem-proving research tool, the system must be extended to allow new search strategies.

In this section we describe the extensions to the system that have been useful. A few of these facilities here are available within the QA3 command structure, but most are in the form of special LISP functions available to the user.

The first five features of the system, listed below, are simple controls on what is basically the normal strategy of QA3. These are controlled by simple program switches or high-level commands. The remaining features constitute means of exerting greater degrees of control, and generally require the user to modify parts of the QA3 program. These features are as follows:

(1) The user can request a search for just a "yes" answer, instead of both "yes" and "no."

(2) The CONTINUE command allows the program to keep trying, by increasing its effort if no proof is found within present limits. This lets QA3 search for a more difficult proof.

(3) The user can request that a proof be printed out when it is found. Included with the printout of the proof are statistics on the search: the number of clauses retained out of the number of clauses generated, the number of clauses subsumed out of the number attempted, the number of successful resolutions out of the number attempted, the number of successful factors generated out of the number attempted, and the proof time. These automatic statistics help the user to quickly determine the effect of a particular heuristic or modification of the strategy.

(4) The user can request that the course of the search be exhibited as it is in progress, by printing out each new clause as it is generated or selected from memory, along with specified information about the clause, such as level, corresponding answer clause, etc.

(5) The bounds on level and maximum term depth can be set by the user.

(6) A standard breadth-first strategy is available that first creates all possible resolvents and factors of Level 1, then 2, etc. Also, the program can optionally use different effort bounds such as the sum of the length plus the level of the candidate clauses, rather than just a level bound on the candidate clauses.

(7) Meta-statements about statements can be used to control the strategy. The statements about clauses are kept on a special form of a property list of each clause. Properties of a clause include the support property, level, history (its parent clause or clauses), its answer clause (if it has one), and its name. The property list also includes bookkeeping information from which the strategy program computes how to avoid equivalent proofs in selecting the next candidates for resolution and factoring. The user can add, fetch, and delete his own properties from clauses (such as some particular method of measuring the value of a clause), and then utilize such information to guide the proof. Axiom clauses in memory can have "permanent" properties stored with them. Clauses generated during a proof can have computed properties, based on, say, some evaluation function, parent clauses, etc. After each attempted resolution or factoring, the strategy programs consider a new candidate clause or pair of candidate clauses. The new candidates are selected by the "standard" strategy described in the next section. However, the user can create new acceptance tests for clauses based on the property lists of the clauses, as well as the clauses themselves. The strategy can then be put into a search mode where it examines all clauses until suitable candidates are found, based on the user's new acceptance tests.

(8) The <u>predicate evaluation</u> mechanism has the ability to use LISP to evaluate atomic formulas or terms within atomic formulas. For example, when i and j are numbers, the predicate $i < j$ can be evaluated by executing the LISP function LESSP with arguments i and j. This mechanism has an effect equivalent to generating, whenever needed, such axioms as ~LESSP(3,2) or LESSP(2,3). This mechanism also allows one form of transfer of control out of the theorem prover to peripheral devices or systems. This feature has been useful for handling arithmetic calculations, finite-set operations, a limited kind of equality, symbolic vector calculations, and special data representations.

(9) A limited form of equality is available during the unification process. This allows two terms to unify that would not unify under the standard unification algorithm. As an example, the commutative function (PLUS A B) can be allowed to unify with (PLUS B A). This feature provides a fast, built-in extension of the matching capabilities of the theorem prover. The user can provide his own special matching functions in LISP.

(10) A built-in polynomial clause evaluation facility allows the user to simply specify a new evaluation function to use on clauses in order to select the next candidates for resolution or factorization. This allows the user to experiment with simple search heuristics or a particularly suitable strategy to guide search for some class of problems, such as the hill-climbing strategy described in Sec. VII-D.

(11) The user can guide the search completely or partially by hand. At each step the user indicates the name of the next two candidates for resolution or factorization. Each newly created clause is assigned a name or number as it is created. The automatic and manual modes can be mixed; as the user is watching the progress of a proof, he may interrupt it for a while to guide it by hand.

C. Strategy

The standard theorem-proving strategy used in QA3 is similar to the unit-preference strategy, using an extended set of support and subsumption.

The principal modification for the purpose of the question-answering system is to have two sets of clauses during an attempted proof. The first set, called "Memory," contains all the statements (axioms) given the system. The second set, called "Clauselist," is the active set of clauses containing only the axioms being used in the current proof attempt and the new clauses being generated. Clauselist is intended to contain only the clauses most relevant to the question. (Neither Clauselist nor Memory are really lists, but rather indexed sets.)

There is a high cost, in computer time and space, for each clause actively associated with the theorem prover. The cost is due to the search time spent when the clause is considered as a candidate for resolution, factoring, or subsumption, and the extra space necessary for bookkeeping on the clause. Since most clauses in Memory are irrelevant to the current proof, it is undesirable to have them in Clauselist, unnecessarily consuming this time and space. So the basic strategy is to work only on the clauses in Clauselist, periodically transferring new, possibly relevant clauses from Memory into Clauselist. If a clause that cannot lead to a proof is brought into Clauselist, this clause can generate many unusable clauses. To help avoid this problem the strategy is reluctant to enter a non-unit clause into Clauselist.

Since the proof strategy of the program is modified frequently, the following is merely an approximate overview of its operation.

(1) First, let Clauselist be the set of clauses representing the negation of the question to be proved. All clauses representing this negated sentence are given T-support. (Note that a theorem of the predicate calculus--e.g., $(\forall x)[P(x) \vee \sim P(x)]$--may be provable without reference to facts in memory.)

(2) If no proof is found, the theorem prover then addresses Memory for a limited number of additional clauses that will resolve with clauses in Clauselist having T-support. (Suitable memory organization and use of the subsumption test can be used to increase the efficiency of the search.)

(3) If no proof is found with the new clauses, return to Step 2.

A modified unit-preference strategy is followed on Clauselist, using a bound on level. As this strategy is being carried out, clauses from Memory that resolve with clauses in Clauselist (a rough measure of relevance) are added to Clauselist. This strategy is carried out on Clauselist until no more resolutions are possible for a given level bound.

Finally, the bound is reached. Clauselist, with all of its bookkeeping, is temporarily saved. If the theorem prover was attempting a "yes" answer, it now attempts a "no" answer. If attempting a "no" answer, it also saves the "no" Clauselist, and returns a NO PROOF FOUND answer. The user may then continue the search by typing CONTINUE. If the bound is not reached in either the yes or no case, the INSUFFICIENT INFORMATION answer is returned. The strategy has the following additional features:

(1) After a newly created unit fails to resolve with any units in Clauselist, it is checked against the units in Memory for a contradiction. This helps to quickly find short proofs.

(2) Frequently in question-answering applications a proof consists of a chain of applications of "two-clauses"--clauses of length two. Semantically this usually means that set membership of some element is being found by chaining through successive supersets or subsets. To speed up this process, a special fast section is included that resolves units in Clauselist with two-clauses in Memory. Our experience so far is that this heuristic is worthwhile.

(3) Each new clause generated is checked to see if it is subsumed by a shorter clause in Clauselist. All longer clauses in Clauselist are checked to see if they are subsumed by the new clause. The longer subsumed clauses are deleted.

(4) Hart's theorem (1965) shows how binary resolution can generate redundant equivalent proofs. Equivalent proofs are avoided in the unit section by a bookkeeping device that prevents redundant resolutions. Wos terms this property "Singly-connected." We do not have a similar algorithm for the non-unit section.

(5) An extended set of support is used that allows pairs of clauses in Clauselist but not in the set of support to resolve with one another up to a level of 2.

(6) The sets, Memory and Clauselist, are indexed to facilitate search. The clauses in Memory are indexed by predicate letters and, under each predicate letter, by length. The clauses in Clauselist are indexed by length.

In searching Memory for relevant clauses to add to Clauselist, clauses already in Clauselist are not considered. The clauses of each length in Clauselist are kept on a sub-list, with new clauses being added at the end of the list. Pointers, or place-keepers, are kept for these lists, and are used to prevent reconsidering resolving two clauses and also to prevent generating equivalent proofs in the unit section.

The strategy is "complete" in the sense that it will eventually find any proof that exists within the degree and space bound.

D. Special Uses of the Theorem Prover

The "theorem prover" refers to a collection of LISP functions used during the theorem-proving process--e.g., RESOLVE, FACTOR, PROVE, PRENEX, CHECKSUBSUMPTION, etc.

The management of the data in the data base, Memory, is aided by the theorem prover. The S command normally causes new clauses to be stored in Memory. However, a statement is stored in Memory only if it is neither a tautology nor a contradiction. A new clause is not stored in Memory if there already exists in Memory another clause of equal length or shorter length that subsumes the new clause. Two other acceptance tests are possible although they are not now implemented. A statement given the system can be checked for consistency with the current data base by attempting to prove the negation of the statement. If the statement is proved inconsistent, it would not be stored. As another possible test, the theorem prover could attempt to prove a new statement in only one or two steps. If the proof is sufficiently easy, the new statement could be considered redundant and could be rejected.

The theorem prover can also be used to simplify the answer, as described in Sec. III-D.

V QUESTION-ANSWERING EXAMPLES

This section presents listings of two dialogues with QA2 (a predecessor of QA3) and one dialogue with QA3. The first dialogue is with QA2 and includes a few question answered by Raphael's SIR plus a few more questions that are more difficult. The input and output format is that used when working with the system.

A. First Dialogue

```
S    (IN JOHN BOY)
     OK
```

The statement (indicated by "S") that John is contained in the set of boys is accepted and the response is "OK."

```
Q    (IN JOHN BOY)
     YES
```

The question (indicated by "Q") "Is John in the set of boys?" is answered "Yes." This is an example of a simple yes or "no proof found" answer.

```
Q    (EX(X)(IN JOHN X)
     YES WHEN X = BOY
```

The question asked is "Does there exist an x such that John is in the set x?" Note that the program reports what assignment is made to x to complete its proof.

```
S    (FA(X)(IF(IN X BOY)(IN X PERSON)))
     OK
```

This says that every boy is a person, or $(\forall x)[x \in BOY \supset x \in PERSON]$

```
Q    EX(X)(IN X PERSON))
     YES WHEN X = JOHN
```

The question asked is "Does there exist a member of the set of humans?" The theorem prover must have used two statements: John is a boy, and every boy is a person.

```
UNWIND

SUMMARY
1    IN(JOHN,BOY)                           AXIOM
2    -IN(X,PERSON)                          NEG OF THM
3    -IN(X,BOY)  IN(X,PERSON)               AXIOM
4    -IN(X,BOY)                             FROM 2,3
(CONTRADICTION FROM CLAUSES 1 AND 4)
(5 CLAUSES GENERATED)
```

The command UNWIND caused the proof to be printed out. Each numbered line corresponds to one clause. A clause may come from three sources:

AXIOM	-	retrieved from memory
NEG OF THM	-	the negation of the question
FROM N,M	-	the result of resolving together clauses N and M.

The number of clauses generated represents the size of the proof tree upon generating the empty clause; this is a measure of the amount of effort involved in completing the proof.

```
S    (FA (X) (IF (IN X PERSON)  (IN X HUMAN)))
     OK
```

It unquestioningly believes that all persons are human.

```
Q    (EX (X) (IN X HUMAN))
     YES WHEN X = JOHN

S    (FA (X) (IF (IN X HUMAN) (HP X ARM 2)))
     OK

Q    (HP JOHN ARM 2)
     YES
```

(HP JOHN ARM 2) means that John Has-as-Parts two elements of the set of all arms.

```
S    (FA (Y) (IF (IN Y ARM) (HP Y HAND 1)))
     OK
Q    (EX (X) (HP JOHN HAND X))
     NO PROOF FOUND
```

The crucial axiom, given next, was missing

```
S    (FA (X Y Z M N) (IF (AND (HP X Y M)
        (FA (U) (IF (IN U Y) (HP U Z N)))) (HP X Z (TIMES M N))))
     OK
Q    (EX (N) (HP JOHN HAND N))
     YES WHEN N = TIMES (2,1)
```

TIMES (2,1) represents the product of 2 and 1 (=2).

```
UNWIND
SUMMARY
1    IN(JOHN,BOY)                                        AXIOM
2    -HP(JOHN,HAND,N)                                    NEG OF THM
3    IN(SK8(N,M,Z,Y,X) ,Y)    -HP(X,Y,M)
      HP(X,Z,TIMES(M,N))                                 AXIOM
4    -HP(JOHN,Y,M)   IN(SK8(N,M,HAND,Y,JOHN),Y)          FROM 2,3
5    -IN(Y,ARM)   HP(Y,HAND,1)                           AXIOM
6    -HP(JOHN,ARM,M)   HP(SK8(N,M,HAND,ARM,JOHN),HAND,1) FROM 4,5
7    -HP(SK8(N,M,Z,Y,X),Z,N)   -HP(X,Y,M)
      HP(X,Z,TIMES(M,N))                                 AXIOM
8    -HP(JOHN,Y,M)   -HP(SK8(N,M,HAND,Y,JOHN),HAND,N)    FROM 2,7
9    -HP(JOHN,ARM,M)                                     FROM 6,8
10   -IN(X,HUMAN)   HP(X,ARM,2)                          AXIOM
11   -IN(JOHN,HUMAN)                                     FROM 9,10
12   -IN(X,PERSON)   IN(X,HUMAN)                         AXIOM
13   -IN(JOHN,PERSON)                                    FROM 11,12
14   -IN(X,BOY)   IN(X,PERSON)                           AXIOM
15   -IN(JOHN,BOY)                                       FROM 13,14
(CONTRADICTION FROM CLAUSES 1 AND 15)
(27 CLAUSES GENERATED)
```

This required an 8-step proof. SK8 is the name generated by the program for a Skolem function used to eliminate an existential quantifier.

```
S    (OR (AT JOHN HOME) (AT JOHN SRI) (AT JOHN AIRPORT))
     OK
Q    (EX (X) (AT JOHN X))
     YES   WHEN X = SRI
        OR   WHEN X = AIRPORT
        OR   WHEN X = HOME
```

Note that the output may be a logical "OR" of several possibilities. The precise form of allowed outputs is described in Sec. III-D.

```
S   (FA (X) (EQUALS X X))
    OK

S   (FA (X Y) (IF (EQUALS X Y) (EQUALS Y X)))
    OK

S   (FA (X Y Z) (IF (AND (EQUALS X Y) (EQUALS Y Z)) (EQUALS X Z)))
    OK

S   (FA (X Y Z) (IF (AND (AT X Y) (AT X Z)) (EQUALS Y Z)))
    OK

S   (NOT (EQUALS SRI AIRPORT))
    OK

S   (NOT (EQUALS AIRPORT HOME))
    OK

Q   (EX (X) (IF (NOT (AT JOHN AIRPORT)) (AT JOHN X)))
      YES   WHEN   X = HOME

      OR  WHEN   X = SRI

S   (IF (AT JOHN AIRPORT) (WITH JOHN BILL))
    OK

S   (FA (X Y Z) (IF (AND (AT X Y) (WITH Z X)) (AT Z Y)))
    OK

Q   (EX (X) (IF (AT JOHN AIRPORT) (AT BILL X)))
    NO PROOF FOUND

S   (FA (X Y) (IF (WITH X Y) (WITH Y X)))
    OK

Q   (EX (X) (IF (AT JOHN AIRPORT) (AT BILL X)))
      YES   WHEN   X = AIRPORT

Q   (EX (X) (IF (NOT (WITH BILL JOHN)) (AT JOHN X)))
      YES   WHEN   X = SRI

      OR  WHEN   X = AIRPORT

      OR  WHEN   X = HOME

S   (AT JOHN SRI)
    OK

Q   (NOT (AT JOHN AIRPORT))
      YES

S   (FA (X Y) (IFF (DISJOINT X Y) (FA (U)
      (IF (IN U X) (NOT (IN U Y))))))
    OK
```

```
Q    (FA (X Y) (IF (DISJOINT X Y) (DISJOINT Y X)))
     YES

S    (DISJOINT BOY GIRL)
     OK

S    (IN JOHN BOY)
     OK

Q    (NOT (IN JOHN GIRL))
     YES

S    (IN JUDY GIRL)
     OK

S    (FA (X Y Z) (IF (AND (IN X Y) (INCLUDE Y Z)) (IN X Z)))
     OK

S    (INCLUDE BOY PERSON)
     OK

Q    (EX (X) (IN X PERSON))
     YES   WHEN   X = JOHN

S    (INCLUDE GIRL PERSON)
     OK

Q    (EX (X) (AND (NOT (IN X BOY)) (IN X PERSON)))
     YES   WHEN   X = JUDY

UNWIND

SUMMARY

1    DISJOINT(BOY,GIRL)                                    AXIOM
2    INCLUDE(GIRL,PERSON)                                  AXIOM
3    IN(JUDY,GIRL)                                         AXIOM
4    IN(X,BOY) -IN(X,PERSON)                               NEG OF THM
5    -INCLUDE(Y,Z) -IN(X,Y)
      IN(X,Z)                                              AXIOM
6    IN(X,BOY) -IN(X,Y)
      -INCLUDE(Y,PERSON)                                   FROM 4,5
7    -INCLUDE(GIRL,PERSON)  IN(JUDY,BOY)                   FROM 3,6
8    IN(JUDY,BOY)                                          FROM 2,7
9    -DISJOINT(X,Y) -IN(U,X)
      -IN(U,Y)                                             AXIOM
10   -IN(JUDY,Y)  -DISJOINT(BOY,Y)                         FROM 8,9
11   -IN(JUDY,GIRL)                                        FROM 1,10
(CONTRADICTION FROM CLAUSES 11 AND 3)

(92 CLAUSES GENERATED)
```

B. Examples from SIR

This dialogue with QA2 is drawn entirely from questions answered by SIR. It is not edited, and illustrates how the user corrects errors, lists axioms, and changes axioms by using the control language.

```
S   (FA (X) (IF (IN X KEYPUNCH-OPERATOR) (IN X GIRL)))
    OK

LIST IN
 LISTING OF PREDICATE IN
1     (FA (X) (IF (IN X KEYPUNCH-OPERATOR) (IN X GIRL)))
S   (FA (Y) (IF (IN Y GIRL) (IN Y PERSON)))
    OK
Q   (IN KEYPUNCH-OPERATOR PERSON)
    NO PROOF FOUND
Q   (FA (X) (IF (IN X KEYPUNCH-OPERATOR) (IN X PERSON)))
    YES
Q   (FA (X) (IF (IN X PERSON) (IN X PERSON)))
    YES
Q   (FA (X) (IF (IN X PERSON) (IN X GIRL)))
    NO PROOF FOUND
Q   (FA (X) (IF (IN X MONKEY) (IN X KEYPUNCH-OPERATOR)))
    NO PROOF FOUND
S   (IN MAX COMPUTER)
    OK
LIST IN
 LISTING OF PREDICATE IN
 1 (IN MAX COMPUTER)
 2 (FA (Y) (IF (IN Y GIRL) (IN Y PERSON)))
 3 (FA (X) (IF (IN X KEYPUNCH-OPERATOR) (IN X GIRL)))
FORGET IN 1
 DONE
S   (IN MAX IBM-7094)
    OK
S   (FA (X) (IF (IN X IBM-7094) (IN X COMPUTER)))
    OK
Q   (IN MAX COMPUTER)
    YES
UNWIND
SUMMARY
1 IN(MAX, IBM-7094)                              AXIOM
2 -IN(MAX,COMPUTER)                              NEG OF THM
3 -IN(X,IBM-7094)  IN(X,COMPUTER)                AXIOM
4 -IN(MAX, IBM-7094)                             FROM 2,3
```

```
(CONTRADICTION FROM CLAUSES 1 AND 4)

(5 CLAUSES GENERATED)


S    (IN BOY STANFORD-STUDENT)
     OK

S    (FA (Z) (IF (IN Z STANFORD-STUDENT) (IN Z BRIGHT-PERSON)))
     OK

LIST IN
 LISTING OF PREDICATE IN
 1   (FA (Z) (IF (IN Z STANFORD-STUDENT) (IN Z BRIGHT-PERSON)))
 2   (IN BOY STANFORD-STUDENT)
 3   (FA (X) (IF (IN X IBM-7094) IN X COMPUTER)))
 4   (IN MAX IBM-7094)
 5   (FA (Y) (IF (IN Y GIRL). (IN Y PERSON)))
 6   (FA (X) (IF (IN X KEYPUNCH-OPERATOR) (IN X GIRL)))

FORGET IN 1
 DONE
FORGET IN 1
 DONE
LIST IN
 LISTING OF PREDICATE IN
 1   (FA (X) (IF (IN X IBM-7094) (IN X COMPUTER)))
 2   (IN MAX IBM-7094)
 3   (FA (Y) (IF (IN Y GIRL) (IN Y PERSON)))
 4   (FA (X) (IF (IN X KEYPUNCH-OPERATOR) (IN X GIRL)))

S    (FA (X Y) (EQV (IS X Y) (IS Y X)))
     OK

S    (FA (Y Z W) (IF (AND (IS Y Z) (IS Z W)) (IS Y W)))
     OK

S    (IN JOHN TEACHER)
     OK

S    (IS JOHN JACK)
     OK

Q    (IN JACK TEACHER)
     NO PROOF FOUND

CONTINUE
 NO PROOF FOUND

UNWIND
 (NO PROOF)
```

```
S    (FA (X Y Z) (IF (AND (IN X Y) (IS Z X) (IN Z Y)))
     OK

Q    (IN JACK TEACHER)
     YES
UNWIND

SUMMARY
1    IS (JOHN,JACK)                                        AXIOM
2    IN(JOHN,TEACHER)                                      AXIOM
3    -IN(JACK, TEACHER)                                    NEG OF THM
4    -IS(Z,X)    -IN(X,Y)
      IN(Z,Y)                                              AXIOM
5    -IN(X,TEACHER)   -IS(JACK,X)                          FROM 3,4
6    -IS(JACK, JOHN)                                       FROM 2,5
7    IS(X,Y)    -IS(Y,X)                                   AXIOM
8    -IS(JOHN,JACK)                                        FROM 6,7
(CONTRADICTION FROM CLAUSES 1 AND 8)

(12 CLAUSES GENERATED)

S    (FA (S) (IF (IN S FIREMEN) (OWNS S PAIR-OF-RED-SUSPENDERS)))
     OK

Q    (OWNS PAIR-OF-RED-SUSPENDERS PAIR-OF-RED-SUSPENDERS)
     NO PROOF FOUND

S    (FA (X) (IF (IN X FIRECHIEF) (IN X FIREMEN)))
     OK

Q    (FA (X) (IF (IN X FIRECHIEF)(OWNS X PAID-OF-RED-SUSPENDERS)))
     YES

Q    (EX (X) (IF (IN X FIRECHIEF)(OWNS X PAIR-OF-RED-SUSPENDERS)))
     YES   WHEN X = S

S    (OWNS ALFRED LOG-LOG-DECITRIG)
     OK

S    (IN LOG-LOG-DECITRIG SLIDE-RULE)
     OK

Q    (EX (X) (AND (IN X SLIDE-RULE) (OWNS ALFRED X)))
     YES   WHEN   X = LOG-LOG-DECITRIG

S    (IN VERNON TECH-MAN)
     OK

S    (FA (X) (IF (IN X TECH-MAN) (IN X ENGINEERING-SUTDENT)))
     OK

S    (FA (X) (IF (IN X ENGINEERING-STUDENT) (EX (Y) (AND (IN Y SLIDE RULE)
       OWNS X Y )))))
     OK

S    (FA (X) (IN (IN X TECH-MAN)(IN X ENGINEERING-STUDENT)))
     OK

Q    (EX (X)(AND (IN X SLIDE-RULE) (OWNS VERNON X)))
     YES   WHEN   X = SK7(VERNON)
```

C. A Simple Chemistry Example

This section presents the results of testing the question-answering program QA3 on the problem set used by W. S. Cooper.[39] The subject was simple chemistry. For his question-answering system Cooper used a restricted English language input. The statements and questions were translated by hand into first-order logic before being given to QA3. Coles' English-to-logic program sometimes translates these sentences into different but still logically equivalent logic statements.

QA3 was able to answer all 23 of the answerable questions. Cooper's program answered 19 of them, failing on Questions 19, 20, 22, and 23. Slagle's Deducom[31] was able to answer 7 of the answerable questions--namely, Questions 1, 2, 3, 5, 6, 11, and 23.

It took about two hours to translate all the facts and questions into logic. It took about two hours to type all statements and questions into the computer and receive answers.

There were 38 facts, translating into 38 clauses, the longest clause having 3 literals. There were 17 different constants, 16 different predicate letters, and no functions. There were 24 questions, the longest translating into 2 clauses. The longest clause in a question had 2 literals. The proofs were not difficult.

One detail should be mentioned. Cooper interprets the sentence "All P's are Q's" to mean

$$(\exists x)P(x) \wedge (\forall x)\ [P(y) \supset Q(y)]$$

to avoid the possibility that $(\exists x)P(x)$ is false. This explains the translations rendered for Questions 11 and 17.

The following abbreviations are used in the facts and questions:

Abbreviations of Chemical Names

MA	Magnesium
MAO	Magnesium Oxide
O	Oxygen
FES	Ferrous Sulfide
FE	Iron
S	Sulfur

Abbreviations of Chemical Names Cont'd

N	Nitrogen
H	Hydrogen
C	Carbon
CU	Copper
H2SO4	Sulfuric Acid
NACL	Sodium Chloride

1. Facts

The facts given QA3 are listed below. The first line of each fact is the English language representation. The second line, prefaced by "S," is the first-order logic translation. QA3 responds with OK if it accepts the statement. (All were accepted.)

```
1.  Magnesium is a metal.
    S (METAL MA)
    OK

2.  Magnesium burns rapidly.
    S  (BURNSRAPIDLY MA)
    OK

3.  Magnesium oxide is a white metallic oxide.
    S (AND(WHITE MAO)(METALLIC MAO)(OXIDE MAO)
    OK

4.  Oxygen is a nonmetal.
    S (NONMETAL O)
    OK

5.  Ferrous sulfide is a dark-gray compound that is brittle.
    S (AND(DARKGRAY FES)(COMPOUND FES)(BRITTLE FES))
    OK

6.  Iron is a metal.
    S (METAL FE)
    OK

7.  Sulfur is a nonmetal.
    S (NONMETAL S)
    OK

8.  Gasoline is a fuel.
    S (FUEL GASOLINE)
    OK
```

```
9.  Gasoline is combustible.
    S (COMBUSTIBLE GASOLINE)
    OK

10. Combustible things burn.
    S (FA(X)(IMP(COMBUSTIBLE X)(BURNS X)))
    OK

11. Fuels are combustible.
    S (FA(X)(IMP(FUEL X) (COMBUSTIBLE X)))
    OK

12. Ice is a solid.
    S (SOLID ICE)
    OK

13. Steam is a gas.
    S (GAS STEAM)
    OK

14. Magnesium is an element.
    S (ELEMENT MA)
    OK

15. Iron is an element.
    S (ELEMENT FE)
    OK

16. Sulfur is an element.
    S (ELEMENT S)
    OK

17. Oxygen is an element.
    S (ELEMENT O)
    OK

18. Nitrogen is an element.
    S (ELEMENT N)
    OK

19. Hydrogen is an element.
    S (ELEMENT H)
    OK

20. Carbon is an element.
    S (ELEMENT C)
    OK

21. Copper is an element.
    S (ELEMENT CU)
    OK
```

However, Statements 14 through 21 can be written as:

```
S (AND(ELEMENT MA)(ELEMENT FE)(ELEMENT S)(ELEMENT O)
      (ELEMENT N)(ELEMENT H)(ELEMENT C)(ELEMENT CU))
OK
```

22. Salt is a compound.
```
S (COMPOUND SALT)
OK
```

23. Sugar is a compound.
```
S (COMPOUND SUGAR)
OK
```

24. Water is a compound.
```
S (COMPOUND WATER)
OK
```

25. Sulfuric acid is a compound.
```
A (COMPOUND H2SO4)
OK
```

Similarly, Statements 21 through 25 can be written as:

```
S (AND(ELEMENT CU)(COMPOUND SALT)(COMPOUND SUGAR)
      (COMPOUND WATER)(COMPOUND H2SO4))
OK
```

26. Elements are not compounds.
```
S (FA(X)(IMP(ELEMENT X)(NOT(COMPOUND X))))
OK
```

27. Salt is sodium chloride.
```
S (IS SALT NACL)
OK
```

28. Sodium chloride is salt.
```
S (IS NACL SALT)
OK
```

29. Oxides are compounds.
```
S (FA(X)(IMP(OXIDE X)(COMPOUND X)))
OK
```

30. Metals are metallic.
```
S (FA(X)(IMP(METAL X)(METALLIC X)))
OK
```

31. No metal is a nonmetal.
```
    S (FA(X)(IMP(METAL X)(NOT(NONMETAL X))))
    OK
```

32. Dark-gray things are not white.
```
    S (FA(X)(IMP(DARKGRAY X)(NOT(WHITE X))))
    OK
```

33. A solid is not a gas.
```
    S (FA(X)(IMP(SOLID X)(NOT(GAS X))))
    OK
```

34. Any thing that burns rapidly burns.
```
    S (FA(X)(IMP(BURNSRAPIDLY X) (BURNS X)))
    OK
```

In addition to Cooper's axioms, QA3 required the following axioms:

35. Ferrous sulfide is a sulfide.
```
    S (SULFIDE FES)
    OK
```

The following three facts were stated directly in logic rather than in English.

36. Equality is reflexive. (The predicate "IS" is used for equality, following Cooper's phrasing.)
```
    S (FA(X)(IS X X))
    OK
```

37. Equality is symmetric.
```
    S (FA(X Y)(IMP(IS X Y)(IS Y X)))
    OK
```

38. Equals can be substituted for equals. (Only one instance of this axiom schema was needed.)
```
    S (FA(X Y)(IMP(AND(IS X Y)
      (COMPOUND X))(COMPOUND Y)))
    OK
```

2. Questions and Answers

The questions and answers, along with a few proofs are listed below. The first line of question is the English language question. The second line, beginning with "Q", is the first-order logic question actually typed into QA3. The answer is prefaced by an "A". Notice that Cooper's questions are statements requiring yes or no answers. QA3 sometimes gives additional information.

1. Magnesium is a metal?
   Q (METAL MA)
   A YES

2. Magnesium is not a metal?
   Q (NOT(METAL MA))
   A NO

3. Magnesium is a nonmetal?
   Q (NONMETAL MA)
   A NO

4. Magnesium is not a nonmetal?
   Q (NOT(NONMETAL MA))
   A YES

5. Magnesium is a metal that burns rapidly?
   Q (AND(METAL MA) (BURNSRAPIDLY MA))
   A YES

6. Magnesium is magnesium?
   Q (IS MA MA)
   A YES

7. Some oxides are white?
   Q (EX(X)(AND(WHITE X)(OXIDE X)))
   A YES   X = MAO

8. No oxide is white?
   Q (NOT(EX(X)(AND(OXIDE X)(WHITE X))))
   A NO

The proof is exhibited by typing the command UNWIND.

```
UNWIND

SUMMARY

        1   OXIDE(MAO)                          AXIOM
        2   -OXIDE(X)   -WHITE(X)               NEG OF THM
        3   -WHITE(MAO)                         FROM 1,2
        4   WHITE(MAO)                          AXIOM
        5   CONTRADICTION                       FROM 3,4

        3 CLAUSES LEFT
        2 CLAUSES GENERATED
        3 CLAUSES ENTERED
        2 RESOLUTIONS OUT OF 5 TRIES
        SUBSUMED Ø TIMES OUT OF 2 TRIES
        FACTORED Ø TIMES OUT OF 2 TRIES
```

9. Oxides are not white?
```
Q (FA(X)(IMP(OXIDE X)(NOT(WHITE X))))
A NO
```

10. Magnesium oxide is an oxide?
```
Q (OXIDE MAO)
A YES
```

11. Every oxide is an oxide?
```
Q (AND(EX(X)(OXIDE X))(FA(Y)(IMP(OXIDE Y)(OXIDE Y))))
A YES, X = MAO
```

12. Ferrous sulfide is dark gray?
```
Q (DARKGRAY FES)
A YES
```

13. Ferrous sulfide is a brittle compound?
```
Q (AND(COMPOUND FES)(BRITTLE FES))
A YES
```

14. Ferrous sulfide is not brittle?
```
Q (NOT(BRITTLE FES))
A NO
```

15. Some sulfides are brittle?
```
Q (EX(X)(AND(SULFIDE X)(BRITTLE X)))
A YES, X = FES
```

16. Ferrous sulfide is not a compound that is not dark gray?
```
Q (NOT(AND(COMPOUND FES)(NOT(DARKGRAY FES))))
A YES
```

17. Anything that is not a compound is not ferrous sulfide?
```
Q (AND(EX(X)(NOT(COMPOUND X)))(FA(Y)(IMP(NOT(COMPOUND Y))
      (NOT(IS Y FES)))))
A YES, X = MA
```

The proof is given below. Note that line 3, ~ COMPOUND(MA) is the resolvent of the two axioms (neither is in the set of support) in Line 1 and Line 2. This resolvent in Line 3 is then used twice, being resolved against Lines 4 and 7. This example illustrates how the extended set of support strategy in QA3 produces a useful lemma.

Also, the proof illustrates the use of equality axioms (Lines 9 and 11), rather than an automatic treatment of equality.

```
UNWIND

SUMMARY

      1   ELEMENT(MA)                         AXIOM
      2   -COMPOUND(X)   -ELEMENT(X)          AXIOM
      3   -COMPOUND(MA)                       FROM 1,2
      4   COMPOUND(X)   -COMPOUND(SK5)        NEG OF THM
      5   -COMPOUND(SK42)                     FROM 3,4
      6   COMPOUND(FES)                       AXIOM
      7   COMPOUND(X)   IS(SK42)FES)          NEG OF THM
      8   IS(SK42,FES)                        FROM 3,7
      9   IS(Y,X)   -IS(X,Y)                  AXIOM
     10   IS(FES,SK42)                        FROM 8,9
     11   COMPOUND(Y)   -IS(X,Y)
          -COMPOUND(X)                        AXIOM
     12   COMPOUND(SK42)   -COMPOUND(FES)     FROM 10,11
     13   COMPOUND(SK42)                      FROM 6,12
     14   CONTRADICTION                       FROM 5,13

     48 CLAUSES LEFT.
     40 CLAUSES GENERATED
     50 CLAUSES ENTERED
     40 RESOLUTIONS OUT OF 292 TRIES
     SUBSUMED 12 TIMES OUT OF 579 TRIES
     FACTORED 0 TIMES OUT OF 7 TRIES
```

18. No dark gray thing is a sulfide?

```
    Q (NOT(EX(X)(AND(DARKGRAY X)(SULFIDE X))))
    A NO, X = FES
```

19. Ferrous sulfide is white?

```
    Q (WHITE FES)
    A NO
```

20. Sodium chloride is a compound?

```
    Q (COMPOUND NACL)
    A YES
```

```
UNWIND

SUMMARY

1   IS(SALT,NACL)                                AXIOM
2   -COMPOUND(NACL)                              NEG OF THM
3   -COMPOUND(X)   -IS(X,Y)   COMPOUND(Y)        AXIOM
4   -IS(X,NACL)   -COMPOUND(X)                   FROM 2,3
5   -COMPOUND(SALT)                              FROM 1,4
6   COMPOUND(SALT)                               AXIOM
7   CONTRADICTION                                FROM 5,6
```

```
10 CLAUSES LEFT
4 CLAUSES GENERATED
10 CLAUSES ENTERED
4 RESOLUTIONS OUT OF 25 TRIES
SUBSUMED 0 TIMES OUT OF 35 TRIES
FACTORED 0 TIMES OUT OF 1 TRIES
```

```
21. Salt is an element?
    Q (ELEMENT SALT)
    A NO

22. Sodium chloride is an element?
    Q (ELEMENT NACL)
    A NO

23. Gasoline is a fuel that burns?
    Q (AND(FUEL GASOLINE)(BURNS GASOLINE))
    A YES
```

The following question is Cooper's example of an unanswerable question.

```
24. Some oxides are not white?
    Q(EX(X)(AND(OXIDE X)(NOT(WHITE X))))
    A NO PROOF FOUND
```

VI PROBLEM SOLVING

This section shows how our extended proof procedure can solve problems involving state transformations. We explore in particular the question of alternative predicate calculus representations for state-transformation problems. The "Monkey and Bananas" puzzle and the "Tower of Hanoi" puzzle are presented along with their solutions obtained by QA3.

Exactly how one can use logic and theorem proving for problem solving requires careful thought on the part of the user. Judging from my experience, and that of others using QA2 and QA3, one of the first difficulties encountered is the representation of problems, especially state-transformation problems, by statements in formal logic. Interest has been shown in seeing several detailed examples that illustrate alternate methods of axiomatizing such problems--i.e., techniques for "programming" in first-order logic. This section provides detailed examples of various methods of representation. After presenting methods in Secs. A and B, we provide a solution to the classic "Monkey and Bananas" problem in Sec. C. Next, Sec. D considers the "Tower of Hanoi" puzzle. Two related applications, robot problem solving and automatic programming, are discussed later in Sec. VII.

A. An Introduction to State-Transformation Methods

The concepts of states and state transformations have of course been in existence for a long time, and the usefulness of these concepts for problem solving is well known. The purpose of this section is not to discuss states and state transformations as such, but instead to show how these concepts can be *used* by an automatic resolution theorem prover. In practice, the employment of these methods has greatly extended the problem-solving capacity of QA2 and QA3. McCarthy and Hayes[40] present a relevant discussion of philosophical problems involved in attempting such formalizations.

First we will present a simple example. We begin by considering how a particular universe of discourse might be described in logic.

Facts describing the universe of discourse are expressed in the form of statements of mathematical logic. Questions or problems are stated as conjectures to be proved. If a theorem is proved, then the nature of our extended theorem prover is such that the proof is "constructive"--i.e., if the theorem asserts the existence of an object then the proof finds or constructs such an object.

At any given moment the universe under consideration may be said to be in a given state.

We will represent a particular state by a subscripted s--e.g., s_{17}. The letter s, with no subscript, will be a variable, ranging over states. A state is described by means of predicates. For example, if the predicate $AT(object_1,b,s_1)$ is true, then in state s_1 the object $object_1$ is at position b. Let this predicate be Axiom A1:

A1. $$AT(object_1,b,s_1) \quad .$$

The question "Where is $object_1$ in $state_1$?" can be expressed in logic as the theorem $(\exists x)AT(object_1,x,s_1)$. The answer found by using system QA3 to prove this theorem is "yes, x = b."

Changes in states are brought about by performing actions and sequences of actions. An action can be represented by an action function that maps states into new states (achieved by executing the action). An axiom describing the effect of an action is typically of the form

$$(\forall s)[P(s) \supset Q(f(s))]$$

where

s is a state variable

P is a predicate describing a state

f is an action function (corresponding to some action) that maps a state into a new state (achieved by executing the action)

Q is a predicate describing the new state.

(Entities such as P and f are termed "situational fluents" by McCarthy.[40])

As an example, consider an axiom describing the fact that object_1 can be pushed from point b to point c. The axiom is

A2. $(\forall s)[AT(\text{object}_1,b,s) \supset AT(\text{object}_1,c,push(\text{object}_1,b,c,s))]$.

The function $push(\text{object}_1,b,c,s)$ corresponds to the action of pushing object_1 from b to c. (Assume, for example, that a robot is the executor of these actions.)

Now consider the question, "Does there exist a sequence of actions such that object_1 is at point c?" Equivalently, one may ask, "Does there exist a state, possibly resulting from applying action functions to an initial state s_1, such that object_1 is at point c?" This question, in logic, is $(\exists s)AT(\text{object}_1,c,s)$, and the answer, provided by the theorem-proving program applied to Axioms A1 and A2, is "yes, $s = push(\text{object}_1, b,c,s_1)$."

Suppose a third axiom indicates that object_1 can be pushed from c to d:

A3. $(\forall s)[AT(\text{object}_1,c,s) \supset AT(\text{object}_1,d,push(\text{object}_1,c,d,s))]$.

Together, these three axioms imply that starting in state s_1, object_1 can be pushed from b to c, and then from c to d. This sequence of actions (a program for our robot) can be expressed by the composition of the two push functions, $push(\text{object}_1,c,d,push(\text{object}_1,b,c,s_1))$. The normal order of function evaluation, from the innermost function to the outermost, gives the correct sequence in which to perform the actions.

To find this solution to the problem of getting object_1 to position d, the following conjecture is posed to the theorem prover: "Does there exist a state such that object_1 is at position d?" or, stated in logic, $(\exists s)AT(\text{object}_1,d,s)$. The answer returned is "yes, $s = push(\text{object}_1,c,d, push(\text{object}_1,b,c,s_1))$."

The proof by resolution, given below, demonstrates how the desired answer is formed as a composition of action functions, thus describing a sequence of necessary actions. The mechanism for finding this answer is a special literal, the <u>answer literal</u>. This method of finding an answer is explained in detail in Sec. III. For our purposes here, we will just show how it works by example. At each step in the proof the answer literal will contain the current value of the object being constructed by the theorem prover. In this example the object being constructed is the sequence of actions s. So initially the answer literal <u>ANSWER</u>(s) is added to the clause representing the negation of the question. (One can interpret this clause, Clause 1, as "either object$_1$ is not at d in state s, or s is an answer.") The state variable s, inside the answer literal, is the "place holder" where the solution sequence is constructed. The construction process in this proof consists of successive instantiations of s. An instantiation of s can occur whenever a literal containing s is instantiated in the creation of a resolvent. Each instantiation of s fills in a new action or an argument of an action function. In general, a particular inference step in the proof (either by factoring or resolving) need not necessarily further instantiate s. For example, the step might be an inference that verifies that some particular property holds for the current answer at that step in the proof. The final step in the proof yields Clause 7, "an answer is push(object$_1$,c,d,push(object$_1$,b,c,s$_1$))," which terminates the proof.

<u>Proof</u>

1.	$\sim$AT(object$_1$,d,s) $\vee$ <u>ANSWER</u>(s)	Negation of theorem
2.	$\sim$AT(object$_1$,c,s) $\vee$ AT(object$_1$,d,push(object$_1$,c,d,s))	Axiom A3
3.	$\sim$AT(object$_1$,c,s) $\vee$ <u>ANSWER</u>(push(object$_1$,c,d,s))	Resolve 1,2
4.	$\sim$AT(object$_1$,b,s) $\vee$ AT(object$_1$,c,push(object$_1$,b,c,s))	Axiom A2
5.	$\sim$AT(object$_1$,b,s) $\vee$ <u>ANSWER</u>(push(object$_1$,c,d, push(object$_1$,b,c,s)))	Resolve 3,4
6.	AT(object$_1$,b,s$_1$)	Axiom A1

7. Contradiction Resolve 5,6

$\underline{\text{ANSWER}}(\text{push}(\text{object}_1,c,d,\text{push}(\text{object}_1,b,c,s_1)))$.

For the particular proof exhibited here, the order of generating the solution sequence during the search for the proof happens to be the same order in which the printout of the proof indicates that s is instantiated. This order consists of working backward from the goal by filling in the last action, then the next-to-last action, etc. In general, the order in which the solution sequence is generated depends upon the proof strategy, since the proof strategy determines the order in which clauses are resolved or factored. The proof that this method always produces correct answers, given in Sec. III-D, shows that the answers are correct regardless of the proof strategy used.

B. Refinements of the Method

The purpose of this section is to discuss variations of the formulation presented in the previous section and to show how other considerations such as time and conditional operations can be brought into the formalism.

1. An Alternative Formulation

The first subject we shall discuss is an alternative to the previously given formulation. We shall refer to the original, presented in Sec. VI-A, as Formulation I, and this alternative as Formulation II. Formulation II corresponds to a system-theoretic notion of state transformations. The state transformation function for a system gives the mapping of an action and a state into a new state. Let f represent the state transformation function, whose arguments are an action and a state and whose value is the new state obtained by applying the action to the state. Let $\{a_i\}$ be the actions, and nil be the null action. Let g be a function that maps two actions into a single composite action whose effect is the same as that of the argument actions applied sequentially. For example, axioms of the following form would partially define the state transformation function f:

B1. $(\forall s)[P(s) \supset Q(f(a_1,s))]$

B2. $(\forall s)[f(nil),s) = s]$

B3. $(\forall s,a_i,a_j)[f(a_j,f(a_i,s)) = f(g(a_i,a_j),s)]$.

The predicates P and Q represent descriptors of states. Axiom B1 describes the result of an action a_1 applied to the class of states that are equivalent in that they all have the property P(s). The resulting states are thus equivalent in that they have property Q(s). Axiom B2 indicates that the null action has no effect. The equation in B3 says that the effect of the composite action sequence $g(a_i,a_j)$ is the same as that of actions a_i and a_j applied sequentially. The question posed in this formulation can include an initial state--e.g., a question might be $(\exists x)Q(f(x,s_0))$, meaning "Does there exist a sequence of actions x that maps state s_0 into a state satisfying the predicate Q?" Observe that we are not insisting on finding a particular sequence of actions, but any sequence that leads us to a satisfactory state within the target class of states.

This representation is more complex, but has the advantage over the previous representation that both the starting state of a transformation and the sequence of actions are explicitly given as the arguments of the state-transformation function. Thus, one can quantify over, or specify in particular, either the starting state or the sequence, or both.

Next we shall show how other considerations can be brought into a state-transformation formalism. Both the original formulation (I) and the alternate (II) will be used as needed.

2. <u>No Change of State</u>

This kind of statement represents an implication that holds for a fixed state. An axiom typical of this class might describe the relationship between movable objects--e.g., if x is to the left of y and y is to the left of z, then x is to the left of z:

$$(\forall x,y,z,s)[\mathrm{LEFT}(x,y,s) \wedge \mathrm{LEFT}(y,z,s) \supset \mathrm{LEFT}(x,z,s)] \quad .$$

3. <u>Time</u>

Time can be a function of a state, to express the timing of actions and states. For example, if the function time(s) gives the time of an instantaneous state, in the axiom

$$(\forall s)[P(s) \supset [Q(f(s)) \wedge \mathrm{EQUAL}(\mathrm{difference}(\mathrm{time}(f(s)),\mathrm{time}(s)),\tau)]] \quad ,$$

where P(s) describes the initial state and Q(s) describes the final state, the state transformation takes τ seconds to complete.

4. <u>State-Independent Truths</u>

The following is an example of an axiom having state-independent functions and predicates:

$$(\forall x,y,z)[\mathrm{EQUAL}(\mathrm{plus}(x,17),z) \supset \mathrm{EQUAL}(\mathrm{difference}(z,x),17)] \quad ,$$

illustrating how functions and predicates are implicitly made state-independent by not taking states as arguments.

5. <u>Descriptors of Transformations</u>

A descriptor or modifier of an action may be added in the form of a predicate that takes as an argument the state transformation that is to be described. For example (in Formulation II),

WISHED-FOR(f(action,state),person)

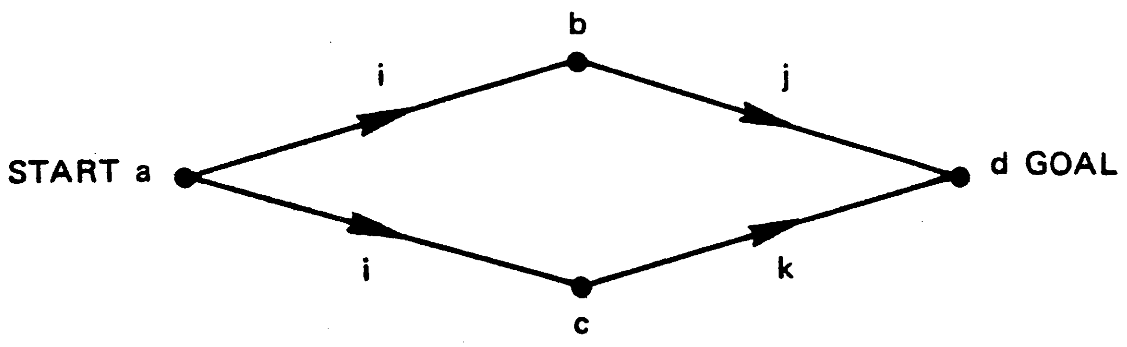
might indicate a wished-for occurence of an action;

LOCATION(f(action,state),place)

indicates that an action occurred at a certain place.

6. Disjunctive Answers

Consider a case in which an action results in one of two possibilities. As an example, consider an automaton that is to move from



a to d. The above figure shows that action i leads to either b or c from a. The function f is singlevalued but we don't know its value. The goal d can be reached from b by action j, or from c by action k. In the formalization given below it is possible to prove that the goal is reachable although a correct sequence of actions necessary to reach the goal is not generated. Instead the answer produced is a disjunction of two sequences--$j(i(s_0))$ or $k(i(s_0))$.

We use Formulation I. Axiom M1 specifies the starting state s_0 and starting position a. Axioms M2, M3, and M4 specify positions resulting from the allowed moves.

M1. $AT(a,s_0)$

M2. $(\forall s)[AT(a,s) \supset AT(b,i(s)) \vee AT(c,i(s))]$

M3. $(\forall s)[AT(b,s) \supset AT(d,j(s))]$

M4. $(\forall s)[AT(c,s) \supset AT(d,k(s))]$.

To find if the goal d is reachable, we ask the following question:

Question: $(\exists s)AT(d,s)$

to which an answer is:

Answer: Yes, $s = j(i(s_0))$ or $s = k(i(s_0))$.

The proof is:

Proof

1.	$\sim AT(d,s) \vee \underline{ANSWER}(s)$	Negation of theorem
2.	$\sim AT(b,s) \vee AT(d,j(s))$	Axiom M3
3.	$\sim AT(b,s) \vee \underline{ANSWER}(j(s))$	From 1,2
4.	$\sim AT(c,s) \vee AT(d,k(s))$	Axiom M4
5.	$\sim AT(c,s) \vee \underline{ANSWER}(k(s))$	From 1,4
6.	$\sim AT(a,s) \vee AT(b,i(s)) \vee AT(c,i(s))$	Axiom M2
7.	$\sim AT(a,s) \vee AT(b,i(s)) \vee \underline{ANSWER}(k(i(s)))$	From 5,6
8.	$\sim AT(a,s) \vee \underline{ANSWER}(j(i(s))) \vee \underline{ANSWER}(k(i(s)))$	From 3,7
9.	$AT(a,s_0)$	Axiom M1
10.	Contradiction	From 8,9

$$\underline{ANSWER}(j(i(s_0))) \vee \underline{ANSWER}(k(i(s_0))) \quad .$$

Observe that Clause 8 has two answers, one coming from Clause 3 corresponding to the action k and one from Clause 7 corresponding to the action j. This shows how an "or" answer can arise.

7. Answers with Conditionals

A conditional operation such as "*if* p *then* q *else* r" allows a program to branch to either operation q or operation r, depending on the outcome of the test condition p. By allowing a conditional operation, a better solution to the above problem is made possible--namely, "beginning in state s_0 take action i; if at b take action j, otherwise take action k."

Consider the problem above that yields disjunctive answers. The information in the above problem formulation, Axioms M1 through M4, plus additional information, allows the creation of a program with a conditional and a test operation. The following additional information is needed, which we shall furnish in the form of axioms.

The first addition needed is a conditional operation, along with a description of what the operation does. Since our programs are in the form of functions, a conditional <u>function</u> is needed. One such possible function is the LISP conditional function "cond" which will be discussed in Sec. VII-B. However, another function, a simple "select" function, is slightly easier to describe and will be used here. The function select(x,y,z,w) is defined to have the value z if x equals y and w otherwise.

M5. $(\forall x,y,z,w)[x = y \supset \text{select}(x,y,z,w) = z]$

M6. $(\forall x,y,z,w)[x \neq y \supset \text{select}(x,y,z,w) = w]$.

The second addition needed is a test operation, along with a description of what it does. Since our programs are in the form of functions, a test <u>function</u> is needed. We shall use "atf," meaning "at-function." The function "atf" applied to a state yields the location in that state--e.g., $\text{atf}(s_0) = a$. The atf function is described by

M7. $(\forall x,s)[AT(x,s) \equiv (\text{atf}(s) = x)]$.

These axioms lead to the solution

$$s = \text{select}(\text{atf}(i(s_0)),b,j(i(s_0)),k(i(s_0))) ,$$

meaning "if at b after applying i to s_0, take action j, otherwise action k."

Although the new axioms allow the conditional solution, just the addition of these axioms does not guarantee that disjunctive answers will not occur. To prevent the possibility of disjunctive answers, we simply tell the theorem prover not to accept any clauses having two answers that don't unify. This method will disallow all "constructive" proofs that yield more than one answer literal.

What may be a preferable problem formulation and solution can result from the use of the alternative state formulation (II) exemplified in Axioms B1, B2, and B3 above. Recall that f(i,s) is the state transformation function that maps action i and state s into a new state; the function g(i,j) maps the action i and the action j into the sequence of the two actions--i then j. The interrelation of f and g is described by

B3. $$(\forall i,j,s)[f(j,f(i,s)) = f(g(i,j),s)] \quad .$$

Axioms M1 through M4 remain the same but Axioms M5, M6, and M7 are replaced. The new select function is described by the two axioms:

M5′. $(\forall i,j,s,p,b)[\text{test}(p,s) = b \supset f(\text{select}(p,b,i,j),s) = f(i,s)]$

M6′. $(\forall i,j,s,p,b)[\text{test}(p,s) \neq b \supset f(\text{select}(p,b,i,j),s) = f(j,s)]$

where the function <u>test</u> applies the test condition p (which will correspond to atf for this problem) to state s. The test condition atf is defined by

M7′. $(\forall x,s)[AT(x,s) \equiv (\text{test}(atf,s) = x)]$.

The new solution is

$$s = f(g(i,\text{select}(atf,b,j,k)),s_0) \quad .$$

Further discussion of program writing, including recursion, is given in Sec. VII-B.

Another method of forming conditional answers is possible. This involves inspecting an existence proof such as the one given in Sec. VI-B-6, above. First, such a proof is generated in which clauses having multiple answers are allowed. The conditional operation is constructed by observing the two literals which are resolved upon to generate the two-answer clause. For example, in the above proof Clauses 3 and 7 resolve to yield 8. This step is repeated below, using the variable s′ in 3 to emphasize that s′ is different from s in 7.

Clause 3. $\sim AT(b,s') \vee \underline{ANSWER}(j(s'))$

Clause 7. $\sim AT(a,s) \vee AT(b,(i(s))) \vee \underline{ANSWER}(k(i(s)))$

Clause 8. $\sim AT(a,s) \vee \underline{ANSWER}(j(s)) \vee \underline{ANSWER}(k(i(s)))$.

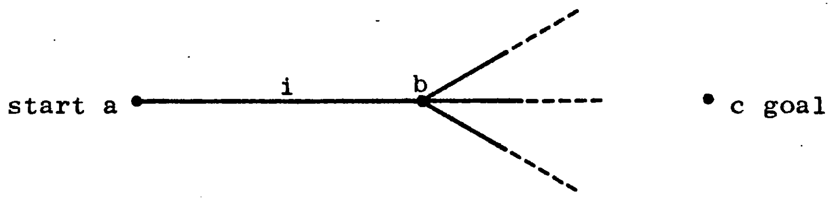
Clause 3 may be read as "if at $\underline{b}$ in state s′, the answer is to take action j when in state s′." Clause 7 may be read as "if not at $\underline{b}$ in state i(s) and if at $\underline{a}$ in state s, the answer is to take action k when in state i(s)." Observing that the resolution binds s′ to i(s) in Clause 8, one knows from Clauses 3 and 7 the test condition by which one decides which answer to choose in Clause 8: "if at $\underline{a}$ in state s the answer depends on i(s); if at $\underline{b}$ in i(s) take action j; otherwise take action k."

This discussion illustrates that the creation of a clause with two answer literals indicates that a conditional operation is needed to create a single conditional answer. This information provides a useful heuristic for the program-writing applications of QA3: When a clause having two answer literals is about to be generated, let the proof strategy call for the axioms that describe the conditional operation (such as M5 and M6). These axioms are then applied to create a single conditional answer.

Waldinger and Lee[22] have implemented a program-writing program PROW that also uses a resolution theorem prover to create constructive proofs, but by a different method than that of QA3. (The second method for creating conditionals by combining two answers is closely related to a technique used in PROW.) Information about the following is embedded in the PROW program: (1) the target program operations, (2) the general relationship of the problem statement and axioms to the allowed target program operations including the test conditions, and (3) the syntax of the target language. In QA3 this information is usually in the axioms--such as Axioms M5, M6, and M7. (The distinction is not entirely clearcut; for example, PROW could use axioms such as M5 and M6, and QA3 uses some knowledge of the target language to simplify the answers produced.)

8. Acquisition of Information

Another situation that arises in problem solving is one in which at the time the problem is stated and a solution is to be produced, there is insufficient information to completely specify a solution. More precisely, the solution cannot name every action and test condition in advance. As an example, consider a robot that is to move from a to c. The action i leads from a to b but no path to c is known, as illustrated below.



However, once point b is reached, more information can be acquired--for example, a guide to the area lives at b and will provide a path to point c if asked. Or perhaps once point b is reached, the robot might use its sensors to observe or discover paths to c.

To formalize this, assume that the action ask-path(b,c) will result in a proper path to c, when taken at b. For simplicity, assume that the name of the path is equal to the state resulting from asking the question. Using Formulation II, one suitable set of axioms is:

N1. $AT(a,s_0) \wedge PATH(a,b,i)$

N2. $(\forall s,x,y,j)[AT(x,s) \wedge PATH(x,y,j) \supset AT(y,f(j,s))]$

N3. $(\forall s)[AT(b,s) \supset PATH(b,c,f(\text{ask-path}(b,c),s)) \wedge$
$AT(b,f(\text{ask-path}(b,c),s))]$

where PATH(a,b,i) means that i is a path from a to b. The question $(\exists s)AT(c,s)$ results in the solution,

$$\text{"yes, } s = f(f(\text{ask-path}(b,c),f(i,s_0)),f(i,s_0))\text{"}.$$

Axiom N3 illustrates an important aspect of this formalism for problem solving: If a condition (such as the robot's) is made state-dependent, then we must specify how this condition changes when the state is changed. Thus, in Axiom N3 we must indicate that the robot's location is not changed by asking for a path. In a pure theorem-proving formalism, this means that if we want to know any condition in a given state, we must prove what that condition is. If a large number of state-dependent conditions need to be known at each state in a solution, then the theorem prover must prove what each condition is at each state in a conjectured solution. In such a case the theorem prover will take a long time to find the solution. McCarthy[40] refers to this problem as the <u>frame problem</u>, where the word "frame" refers to the frame of reference or the set of relevant conditions. Discussion of a method for easing this problem is presented in Sec. VII-A.

9. <u>Assignment Operations</u>

An assignment operation is one that assigns a value to a variable. An example of an assignment is the statement $a \leftarrow h(a)$, meaning that the value of a is to be changed to the value of the function h(a). In our representation we shall use an assignment function--i.e., assign(a,h(a)). Using Formulation II this function is described by the axiom

$$(\forall a,a_0,s)[\mathrm{VALUE}(a,a_0,s) \supset \mathrm{VALUE}(a,h(a_0),f(\mathrm{assign}(a,h(a)),s))]$$

where the predicate $\mathrm{VALUE}(a,a_0,s)$ means that variable a has value a_0 in state s.

C. <u>An Example: The Monkey and the Bananas</u>

To illustrate the methods described earlier, we present an axiomatization of McCarthy's[13] "Monkey and Bananas" problem.

The monkey is faced with the problem of getting a bunch of bananas hanging from the ceiling just beyond his reach. To solve the problem, the monkey must push a box to an empty place under the bananas, climb on top of the box, and then reach the bananas.

The constants are monkey, box, bananas, and under-bananas. The functions are reach, climb, and move, meaning the following:

reach(m,z,s)	The state resulting from the action of m reaching z, starting from state s
climb(m,b,s)	The state resulting from the action of m climbing b, starting from state s
move(m,b,u,s)	The state resulting from the action of m moving b to place u, starting from state s.

The predicates are:

MOVABLE(b)	b is movable
AT(m,u,s)	m is at place u in state s
ON(m,b,s)	m is on b in state s
HAS(m,z,s)	m has z in state s
CLIMBABLE(m,b,s)	m can climb b in state s
REACHABLE(m,b,s)	m can reach b in state s.

The axioms* are:

MB1. MOVABLE(box)

MB2. AT(box,place_b,s_0)

MB3. $(\forall x) \sim$AT(x,under-bananas,s_0)

*The astute reader will notice that the axioms leave much to be desired. In keeping with the "toy problem" tradition we present an unrealistic axiomatization of this unrealistic problem. The problem's value lies in the fact that it is a reasonably interesting problem that may be familiar to the reader.

MB4. $(\forall b,p_1,p_2,s)[[AT(b,p_1,s) \wedge MOVABLE(b) \wedge (\forall x) \sim AT(x,p_2,s)] \supset$
$[AT(b,p_2,move(monkey,b,p_2,s)) \wedge$
$AT(monkey,p_2,move(monkey,b,p_2,s))]]$

MB5. $(\forall s)CLIMBABLE(monkey,box,s)$

MB6. $(\forall m,p,b,s)[[AT(b,p,s) \wedge CLIMBABLE(m,b,s)] \supset$
$[AT(b,p,climb(m,b,s)) \wedge ON(m,b,climb(m,b,s))]]$

MB7. $(\forall s)[[AT(box,under\text{-}bananas,s) \wedge ON(monkey,box,s)] \supset$
$REACHABLE(monkey,bananas,s)]$

MB8. $(\forall m,z,s)[REACHABLE(m,z,s) \supset HAS(m,z,reach(m,z,s))]$

The question is "Does there exist a state s (sequence of actions) in which the monkey has the bananas?"

$$\text{QUESTION:} \quad (\exists s)HAS(monkey,bananas,s) \quad .$$

The answer is yes,

$$s = reach(monkey,bananas,climb(monkey,box,$$
$$move(monkey,box,under\text{-}bananas,s_0))) \quad .$$

By executing this function, the monkey gets the bananas. The monkey must, of course, execute the functions in the usual order, starting with the innermost and working outward. Thus he first moves the box under the bananas, then climbs on the box, and then reaches the bananas.

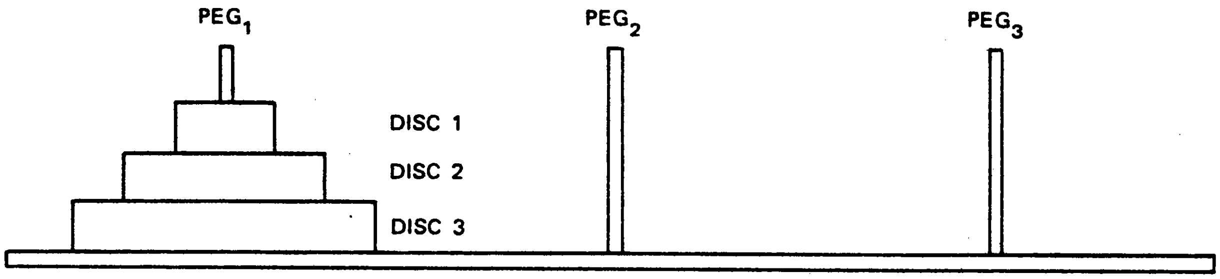
The printout of the proof is given in Appendix B.

D. Formalizations for the Tower of Hanoi Puzzle

The first applications of our QA2 and QA3 programs were to "question-answering" examples. Commonly used question-answering examples have short proofs, and usually there are a few obvious formulations for a given subject area. (The major difficulty in question-answering problems usually is searching a large data base, rather than finding a long and difficult proof.) Typically, any reasonable formulation works well. As one goes on to problems like the Tower of Hanoi puzzle, more effort is required to find a representation that is suitable for efficient problem solving.

This puzzle has proved to be an interesting study of representation. Several people using QA3 have set up axiom systems for the puzzle. Apparently, a "good" axiomatization--one leading to quick solutions--is not entirely obvious, since many axiomatizations did not result in solutions. In this section we will present and compare several alternative representations, including ones that lead to a solution.

There are three pegs--peg_1, peg_2, and peg_3. There are a number of discs each of whose diameter is different from that of all the other discs. Initially all discs are stacked on peg_1, in order of descending size. The three-disc version is illustrated below. The object of the



puzzle is to find a sequence of moves that will transfer all the discs from peg_1 to peg_3. The allowed moves consist of taking the top disc from any peg and placing it on another peg, but a disc can never be placed on top of a smaller disc.

In order to correctly specify the problem, any formalization must: (1) specify the positions of the discs for each state, (2) specify how actions change the position of the discs, and (3) specify the rules of the game--i.e., what is legal.

Let the predicate ON specify disc positions. In the simplest representation the predicate ON specifies the position of one disc--e.g., ON($disc_1$,peg_1,s) says that in state s $disc_1$ is on peg_1. This representation requires one predicate to specify the position of each disc. The relative position of each disc either must be specified by another statement, or else if two discs are on the same peg it must be implicitly understood that they are in the proper order. Perhaps the simplest extension is to allow the predicate another argument that specifies the position of the position of the disc--i.e., $ON(disc_1, peg_1, position_2, s)$. Again, this requires many statements to specify a complete configuration.

Since a move can be construed as constructing a stack of discs, and since a stack can be represented as a list, consider, as an alternative representation, a <u>list</u> as a representation of a stack. Let the function $\ell(x,y)$ represent the list that has x as its first element (representing the top disc in the stack) and y as the rest of the list (representing the rest of the discs in the stack). This function ℓ corresponds to the "cons" function in LISP. Let nil be the empty list. The statement $ON(\ell(disc_1, \ell(disc_2, nil)), peg_1, s)$ asserts that the stack having top disc, $disc_1$, and second disc, $disc_2$, is on peg_1. This representation illustrates a useful technique in logic--namely, the use of functions as the construction (and selection) operators. This notion is consistent with the use of action functions as constructors of sequences.

Next, consider how to express possible changes in states. Perhaps the simplest idea is to say that a given state implies that certain moves are legal. One must then have other statements indicating the result of each move. This method is a bit lengthy. It is easier to express in one statement the fact that given some state, a new state is the result of a move. Thus one such move to a new state is described by $(\forall s)[ON(\ell(disc_1, nil), peg_1, s) \wedge ON(nil, peg_2, s) \wedge ON(\ell(disc_2, \ell(disc_3, nil)), peg_3, s) \supset ON(nil, peg_1, move(disc_1, peg_1, peg_2, s)) \wedge ON(\ell(disc_1, nil), peg_2, move(disc_1, peg_1, peg_2, s)) \wedge ON(\ell(disc_2, \ell(disc_3, nil)), peg_3, move(disc_1, peg_1, peg_2, s))]$.

With this method it is possible to enumerate all possible moves and configuration combinations. However, it is still easier to use variables to represent whole classes of states and moves. Thus, $(\forall s, x, y, z, p_i, p_j, p_k, d)[ON(\ell(d,x), p_i, s) \wedge ON(y, p_j, s) \wedge ON(z, p_k, s) \supset ON(x, p_i, move(d, p_i, p_j, s)) \wedge ON(\ell(d,y), p_j, move(d, p_i, p_j, s)) \wedge ON(z, p_k, move(d, p_i, p_j, s))]$ specifies a whole class of moves. The problem here is that additional restrictions must be added so that illegal states cannot be part of a solution. In the previous formalism, one could let the axioms enumerate just the legal moves and states, and thus prevent incorrect solutions.

The first method for adding restrictions is to have a predicate that restricts moves to just the legitimate states. Since the starting state is legal, one might think that only legal states can be reached. However, the resolution process (set-of-support strategy) typically works backward from the goal state toward states that can reach the goal state--such states are sometimes called "forcing states." Thus, illegal but forcing states can be reached by working backward from the goal state. This does not allow for incorrect solutions, since the only forcing states that can appear in the solution must be those reached from the starting state (which is a legal state). The restriction of moving only <u>to</u> new states thus prevents an error. But the search is unnecessarily large, since the theorem prover is considering illegal states that cannot lead to a solution. So a better solution is to eliminate these illegal forcing states by allowing moves only <u>from</u> the legal states <u>to</u> legal states. This is perhaps the best specification, in a sense. Such an axiom is $(\forall s,x,y,z,p_i,p_j,p_k,d)[\mathrm{ON}(\ell(d,x),p_i,s) \wedge \mathrm{ON}(y,p_j,s) \wedge \mathrm{ON}(z,p_k,s) \wedge \mathrm{LEGAL}(\ell(d,x)) \wedge \mathrm{LEGAL}(\ell(d,y)) \wedge \mathrm{DISTINCT}(p_i,p_j,p_k) \supset \mathrm{ON}(x,p_i,\mathrm{move}(d,p_i,p_j,s)) \wedge \mathrm{ON}(\ell(d,y),p_j,\mathrm{move}(d,p_i,p_j,s)) \wedge \mathrm{ON}(z,p_k,\mathrm{move}(d,p_i,p_j,s))]$. The predicate LEGAL(x) is true if and only if the discs are listed in order of increasing size. (One can "cheat" and have a simpler axiom by omitting the predicate that requires that the state resulting from a move have a legal stack of discs. Since the set-of-support strategy forces the theorem prover to work backward starting from a legal final state, it will only consider legal states. However, one is then using an axiomatization that, by itself, is incorrect.) The additional LEGAL predicate is a typical example of how additional information in the axioms results in a quicker solution. The predicate $\mathrm{DISTINCT}(p_i,p_j,p_k)$ means no two pegs are equal.

The clauses generated during the search that are concerned with illegal states are subsumed by ~LEGAL predicates such as $(\forall s) \sim\mathrm{LEGAL}(\ell(\mathrm{disc}_2,(\mathrm{disc}_1,x)))$. The stacks are formed by placing one new disc on top of a legal stack. If the new top disc is smaller than the old top disc then it is of course smaller than all the others on the stack. Thus the legal stack axioms need only to specify that the top disc is smaller than the second disc for a stack to be legal. This blocks the construction of incorrect stacks.

One complete axiomatization is as follows:

AX1. $(\forall x,y,z,m,n,p_i,p_j,p_k)[ON(\ell(d(m),x),p_i,s) \wedge ON(y,p_j,s) \wedge ON(z,p_k,s) \wedge DISTINCT(p_i,p_j,p_k) \wedge LEGAL(\ell(d(m),x)) \wedge LEGAL(\ell(d(n),y)) \supset ON(x,p_i,move(d(m),p_i,p_j,s)) \wedge ON(\ell(d(m),y),p_j,move(d(m),p_i,p_j,s)) \wedge ON(z,p_k,move(d(m),p_i,p_j,s))]$

AX2. $(\forall m,n,x)[LEGAL(\ell(d(m),\ell(d(n),x))) \equiv LESS(m,n)] \wedge (\forall n)LEGAL(\ell(d(n),nil)) \wedge LEGAL(nil)$.

Instead of naming each disc, the disc number n is an argument of the function d(n) that represents the nth disc. This representation illustrates how the proof procedure can be shortened by solving frequent decidable subproblems with special available tools--namely, the LISP programming language. The theorem prover uses LISP (the "lessp" function) to evaluate the LESS(n,m) predicate--a very quick step. This predicate evaluation mechanism has the effect of generating, wherever needed, such axioms as ~LESS(3,2) or LESS(2,3) to resolve against or subsume literals in generated clauses. Similarly, LISP evaluates the DISTINCT predicate.

Note that the move axiom, AX1, breaks up into three clauses, each clause specifying the change in the stack for one particular peg. The process of making one move requires nine binary resolutions, and two binary factorings of clauses.

Still other solutions are possible by using special term-matching capabilities in QA3 that extend the unification and subsumption algorithms to include list terms, set terms, and certain types of symmetries.

In another axiomatization, the complete configuration of the puzzle in a given state is specified by the predicate ON. ON(x,y,z,s) means that in state s, stack x is on peg_1, stack y is on peg_2, and stack z is on peg_3. Thus, if the predicate $\text{ON}(\ell(d_1,\ell(d_2,\text{nil}))),\text{nil},\ell(d_3,\text{nil}),s_k)$ holds, the stack $d_1 - d_2$ is on peg_1 and d_3 is on peg_3. The predicate LEGAL again indicates that a given stack of discs is allowed.

Two kinds of axioms are required--move axioms and legal stack axioms. One legal stack axiom is $\text{LEGAL}(\ell(d_1,\ell(d_2,\text{nil})))$. One move axiom is $(\forall d,x,y,z,s)[\text{ON}(\ell(d,x),y,z,s) \wedge \text{LEGAL}(\ell(d,x)) \wedge \text{LEGAL}(\ell(d,y)) \supset \text{ON}(x,$ $\ell(d,y),z,\text{move}(d,p_1,p_2,s))]$. This axiom states that disc d can be moved from peg_1 to peg_2 if the initial stack on peg_1 is legal and the resultant stack on peg_2 is legal.

In this last-mentioned formalization, using 13 axioms to specify the problem, QA3 easily solved this problem for the three-disc puzzle. During the search for a proof, 98 clauses were generated but only 25 of the clauses were accepted. Of the 25, 12 were not in the proof. The solution entails seven moves, thus passing through eight states (counting the initial and final states). The 12 clauses not in the proof correspond to searching through 5 states that are not used in the solution. Thus the solution is found rather easily. Of course, if a sufficiently poor axiomatization is chosen--one requiring an enumeration of enough correct and incorrect disc positions--the system becomes saturated and fails to obtain a solution within time and space constraints. An important factor in the proof search is the elimination of extra clauses corresponding to alternate paths that reach a given state. In the above problem it happens that the subsumption heuristic eliminates 73 of these redundant clauses. However, this particular use of subsumption is problem-dependent, thus one must examine any given problem formulation to determine whether or not subsumption will eliminate alternative paths to equivalent states.

The four-disc version of the puzzle can be much more difficult than the three-disc puzzle in terms of search. At about this level of difficulty one must be somewhat more careful to obtain a low-cost solution.

Ernst[41] formalizes the notion of "difference" used by GPS and shows what properties these differences must possess for GPS to succeed on a problem. He then presents a "good" set of differences for the Tower of Hanoi problem. Utilizing this information, GPS solves the problem for four discs, considering no incorrect states in its search. Thus Ernst has chosen a set of differences that guide GPS directly to the solution.

Another method of solution is possible. First, solve the three-disc puzzle (using the answer statement). Then ask for a solution to the four-disc puzzle. The solution then is: Move the top three discs from peg_1 to peg_2; move $disc_4$ from peg_1 to peg_3; move the three discs on peg_2 to peg_3. This method produces a much easier solution. But this can be considered as cheating, since the machine is "guided" to a solution by being told which subproblem to first solve and store away. The use of the differences by GPS similarly lets the problem solver be "guided" toward a solution.

There is another possibly more desirable solution. The four-disc puzzle can be posed as the problem, with no three-disc solution. If the solution of the three-disc puzzle occurs during the search for a solution to the four-disc puzzle, and if it is automatically recognized and saved as a lemma, then the four-disc solution should follow easily.

Finally, if an induction axiom is provided, the axioms imply a solution in the form of a recursive program that solves the puzzle for an arbitrary number of discs. Aiko Hormann[42] discusses the related solutions of the four-disc problem by the program GAKU (not an automatic theorem-proving program). The solutions by lemma finding, induction, and search guided by differences have not been run on QA3.

VII SAMPLE PROBLEM-SOLVING APPLICATIONS

This section presents four sample problem-solving applications: robot problem solving, automatic program writing, self-description, and scene description.

A. Applications to the Robot Project

1. Introduction to Robot Problem Solving

In this section we discuss how theorem-proving methods are being tested for several applications in the Stanford Research Institute Artificial Intelligence Group's automaton (robot). We emphasize that this section describes work that is now in progress, rather than work that is completed. These methods represent explorations in problem solving, rather than final decisions about how the robot is to do problem solving. An overview of the current status of the entire robot project is provided by Nilsson.[30] Coles[8] has developed an English-to-logic translator that is part of the robot.

We use theorem-proving methods for three purposes, the simplest being the use of QA3 as a central information storage and retrieval system that is accessible to various parts of the system as well as the human users. The data base of QA3 is thus one of the robot's models of its world, including itself.

A second use is as an experimental tool to test out a particular problem formulation. When a suitable formulation is found, it may then be desirable to write a faster or more efficicient specific program that implements this formulation, perhaps involving little or no search. If the special program is not as general as the axiom system is, so that the special program fails in certain cases, the axioms can be retained to be used in the troublesome cases. Both solutions can be made available by storing, as the first axiom to be tried, a special axiom that describes the special solution. The predicate-evaluation mechanism can then call LISP to run the special solution. If it fails, the other axioms will then be used.

The third use is as a real-time problem solver. In the implementation we are now using, statements of logic--clauses--are the basic units of information. Statements are derived from several sources: teletype entries, axioms stored in memory, clauses or statements generated by the theorem prover, and statements evaluated by programs--subroutines in LISP, FORTRAN, or machine language. These programs can use robot sensors and sensory data to verify, disprove, or generate statements of logic.

The SRI robot is a cart on wheels, having a TV camera and a range-finder mounted on the cart. There are bumpers on the cart, but no arms or grasping agents, so the only way the robot can manipulate its environment is by simple pushing actions. Given this rather severe restriction of no grasping, the robot must be clever to effectively solve problems involving modifying its world. We present below some axioms for robot problem solving.

The first axiom describes the move routines of the robot:

R1. $$(\forall s,p_1,p_2,path_{12})[AT(robot,p_1,s) \wedge PATH(p_1,p_2,path_{12},s) \supset AT(robot,p_2,move(robot,path_{12},s))] \ .$$

This action says that if the robot is at p_1 and there is a path to p_2, the robot will be at p_2 after moving along the path. The predicate PATH indicates there exists a robot-path, $path_{12}$, from place p_1 to place p_2. A robot-path is a path adequate for the robot's movement. The terms p_1 and p_2 describe the position of the robot.

In general, it may be very inefficient to use the theorem prover to find the $path_{12}$ such that $PATH(p_1,p_2,path_{12})$ is true. Several existing FORTRAN subroutines, having sophisticated problem-solving capabilities of their own, may be used to determine a good path through obstacles on level ground. We will show later a case where the theorem prover may be used to find a more obscure kind of path. For the less obscure paths Axiom R1 is merely a description of the semantics of these FORTRAN programs, so that new and meaningful programs can be generated by QA3 by using the efficient path-generating programs as subprograms.

The "predicate-evaluation" mechanism is used to call the FORTRAN path-finding routines. The effect of this evaluation mechanism is the same as if the family of axioms of the form PATH(p_1,p_2,$path_{12}$) for all p_1 and p_2 such that $path_{12}$ exists, were all stored in memory and available to the theorem prover.

The second axiom is a push axiom that describes the effect of pushing an object. The robot has no arm or graspers, just a bumper. Its world consists of large objects such as boxes, wedges, cubes, etc. These objects are roughly the same size as the robot itself.

The basic predicate that specifies the position of an object is ATO, meaning at-object. The predicate

$$\text{ATO}(\text{object}_1,\text{description}_1,\text{position}_1,s_1)$$

indicates that object_1, having structural description "description_1", is in position "position_1", in state "s_1". At the time of this writing, a particular set of "standard" structure descriptions has not yet been selected. So far several have been used. The simplest description is a point whose position is at the estimated center of gravity of the object. This description is used for the FORTRAN "push in a straight line" routine. Since all the objects in the robot's world are polyhedrons, reasonably simple complete structural descriptions are possible. For example, one structural description consists of the set of polygons that form the surface of the polyhedron. In turn, the structure of the polygons is given by the set of vertices in its boundary. Connectivity of structures can be stated explicitly or else implied by common boundaries. The position of an object is given by a mapping of the topologically described structure into the robot's coordinate system. Such structural descriptions may be given as axioms or supplied by the scene-analysis programs used by the robot.

A basic axiom describing the robot's manipulation of an object is:

R2. $(\forall s, obj_1, desc_1, pos_1, pos_2)[ATO(obj_1, desc_1, pos_1, s) \wedge$
$MOVABLE(obj_1) \wedge ROTATE\text{-}TRANSLATE\text{-}ABLE(desc_1, pos_1, pos_2) \wedge$
$OBJECT\text{-}PATH(desc_1, pos_1, pos_2, path_{12}, s) \supset$
$ATO(obj_1, desc_1, pos_2, push(obj_1, path_{12}, s))]$.

This axiom says that if object 1, described by description 1, is at position 1, and object 1 is movable, and object 1 can be theoretically rotated and translated to the new position 2, and there is an object-path from 1 to 2, then object 1 will be at position 2 as a result of pushing it along the path. The predicate ROTATE-TRANSLATE-ABLE($desc_1$,pos_1,pos_2) checks the necessary condition that the object can be theoretically rotated and translated into the new position. The predicate OBJECT-PATH($desc_1$,pos_1,pos_2,$path_{12}$) means that pos_2 is the estimated new position resulting from pushing along push-path, $path_{12}$.

Let us now return to the frame problem. More specifically, in a state resulting from pushing an object, how can we indicate the location of objects that were not pushed? One such axiom is:

R3. $(\forall obj_1, obj_2, desc_1, pos_1, path_{12}, s)[ATO(obj_1, desc_1, pos_1, s) \wedge$
$\sim SAME(obj_1, obj_2) \supset ATO(obj_1, desc_1, pos_1, push(obj_2, path_{12}, s))]$.

This axiom says that all objects that are not the same as the pushed object are unmoved. The predicate evaluation mechanism is used to evaluate SAME and speed up the proof. One can use this predicate evaluation mechanism, and perhaps other fast methods for handling classes of deductions (such as special representations of state-dependent information and special programs for updating this information--which is done in the robot), but another problem remains. Observe that Axiom R3 assumes that only the objects directly pushed by the robot move. This is not always the case, since an object being pushed might accidentally strike another object and move it. This leads to the question of dealing with the real world and using axioms to approximate the real world.

2. Real-World Problem Solving: Feedback

Our descriptions of the real world, axiomatic or otherwise, are at best only approximations. For example, the new position of an object moved by the robot will not necessarily be accurately predicted, even if one goes to great extremes to calculate a predicted new position. The robot does not have a grasp on the object, so that some slippage may occur. The floor surface is not uniform and smooth. The weight distribution of objects is not known. There is only rudimentary kinesthetic sensing feedback--namely, whether or not the bumper is still in contact with the object. Thus it appears that a large feedback loop iterating toward a solution is necessary: Form a plan for pushing the object (possibly using the push axiom), push according to the plan, back up, take a look, see where the object is, compare the position to the desired position, start over again. The new position (to some level of accuracy) is provided by the sensors of the robot. This new position is compared to the position predicted by the axiom. If the move is not successful, the predicate (provided by sensors in the new state) that reasonably accurately gives the object's position in the new state must be used as the description of the initial state for the next attempt.

This feedback method can be extended to sequences of actions. Consider the problem: Find s_f such that $P_3(s_f)$ is true. Suppose the starting state is s_0, with property $P_0(s_0)$. Suppose the axioms are as follows:

$$P_0(s_0)$$

$$(\forall s)[P_0(s) \supset P_1(f_1(s))]$$

$$(\forall s)[P_1(s) \supset P_2(f_2(s))]$$

$$(\forall s)[P_2(s) \supset P_3(f_3(s))] \quad .$$

The sequence of actions $f_3(f_2(f_1(s_0)))$ transforms state s_0 with property $P_0(s_0)$ into state s_f having property $P_3(s_f)$.

The solution is thus $s_f = f_3(f_2(f_1(s_0)))$.

Corresponding to each "theoretical" predicate $P_i(s)$ is a "real-world" predicate $P_i'(s)$. The truth value of $P_i'(s)$ is determined by sensors and the robot's internal model of the world. It has built-in bounds on how close its measurements must be to the correct values in order to assert that it is true.* The proof implies the following description of the result after each step of execution of $f_3(f_2(f_1(s_0)))$:

Actions and Successive States	Predicted Theoretical Results	Predicted Real-World Results
s_0	$P_0(s_0)$	$P_0'(s_0)$
$s_1 = f_1(s_0)$	$P_1(s_1)$	$P_1'(s_1)$
$s_2 = f_2(s_1)$	$P_2(s_2)$	$P_2'(s_2)$
$s_f = f_3(s_2)$	$P_3(s_3)$	$P_3'(s_f)$

To measure progress after, say, the ith step, one checks that $P_i'(s_i)$ is true. If not, then some other condition $P_i''(s_i)$ holds and a new problem is generated, given $P_i''(s_i)$ as the starting point. If new information is present, such as is the case when the robot hits an obstacle that is not in its model, the model is updated before a new solution is attempted. The position of this new object of course invalidates the previous plan--i.e., had the new object's position been known, the previous plan would not have been generated.

The new solution may still be able to use that part of the old solution that is not invalidated by any new information. For example, if $P_i''(s_i)$ holds, it may still be possible to reach the jth intermediate state and then continue the planned sequence of actions from the jth state. However, the object-pushing axiom is an example of an axiom that probably will incorrectly predict results and yet no further information, except for the new position, will be available. For this case, the best

*At this time, a many-valued logic having degrees of truth is not used, although this is an interesting possibility.

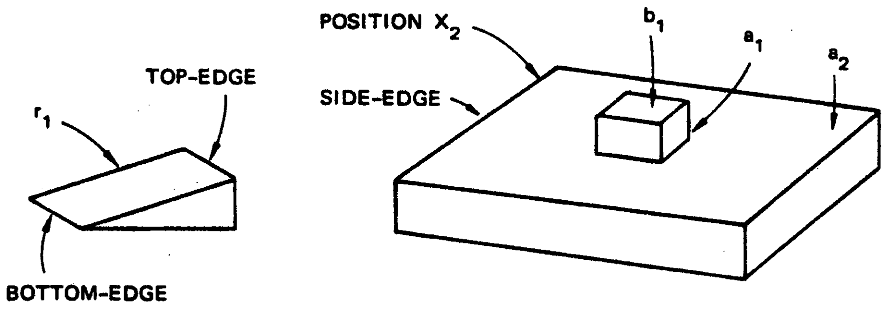
approach is probably to iterate toward the target state by repeated use of the push axiom to generate a new plan. Hopefully, the process converges.

For a given axiomatization, feedback does not necessarily make it any easier to find a proof. However, knowing that the system uses feedback allows us to choose a simpler and less accurate axiom system. Simple axiom systems can then lead to shorter proofs.

One can envision formalizing this entire problem-solving process, including the notion of feedback, verifying whether or not a given condition is met, updating the model, recursively calling the theorem prover, etc. The author has not attempted such a formalization, although he has written a first-order formalization of the theorem prover's own problem-solving strategy. This raises the very interesting possibility of self-modification of strategy; however, in practice such problems lie well beyond the current theorem-proving capacity of the program.

3. A Simple Robot Problem

Now let us consider a problem requiring the use of a ramp to roll onto a platform, as illustrated below.



TA-7494-5

The goal is to push the box b_1 from position a_1 to a_2. To get onto the platform, the robot must push the ramp r_1 to the platform, and then roll up the ramp onto the platform.

A simple problem formulation can use a special ramp-using axiom such as:

R4. $(\forall x_1, x_2, s, \text{top-edge}, \text{bottom-edge}, \text{ramp}_1)[\text{AT-RAMP}(\text{ramp}_1, \text{top-edge}, x_2, \text{bottom-edge}, x_1, s) \land \text{AT-PLATFORM}(\text{side-edge}, x_2, s) \supset \text{AT}(\text{robot}, x_2, \text{climb}(\text{ramp}_1, x_1, x_2, s))]$.

with the obvious meaning. Such a solution is quick but leaves much to be desired in terms of generality.

A more general problem statement is one in which the robot has a description of its own capabilities, and a translation of this statement of its abilities into the basic terms that describe its sensory and human-given model of the world. It then learns from a fundamental level to deal with the world. Such a knowledge does not make for the quickest solution to a frequently encountered problem, but certainly does lend itself to learning, greater degrees of problem solving, and self-reliance in a new problem situation.

Closer to this extreme of greatest generality is the following axiomatization:

R5. $(\forall x_1, x_2, r)[\text{RECTANGLE}(r, x_1, x_2) \land \text{LESSP}(\text{maxslope}(r), k_0) \land \text{LESSP}(r_0, \text{width}(r)) \land \text{CLEAR}(\text{space}(r, h_0), s) \land \text{SOLID}(r) \supset \text{PATH}(x_1, x_2, r)]$.

This axiom says that r describes a rectangle having ends x_1 and x_2. The maximum slope is less than a constant k_0, the width of r is greater than the robot's width w_0, the space above r to the robot's height h_0 is clear, and the rectangle r has a solid surface.

Two paths can be joined as follows:

R6. $(\forall x_1, x_2, x_3, r_1, r_2)[\text{PATH}(x_1, x_2, r_1) \land \text{PATH}(x_2, x_3, r_2) \supset \text{PATH}(x_1, x_3, \text{join}(r_1, r_2))]$.

From these two axioms (R5 and R6), the push axiom (R2), and a recognition of a solid object that can be used as a ramp, a solution can be obtained in terms of <u>climb</u>, <u>push</u>, <u>join</u>, etc. This more general method will probably be more useful if the robot will be required to construct a ramp, or recognize and push over a potential ramp that is standing on its wide end.

The danger in trying the more general methods is that one may be asking the theorem prover to rederive some significant portion of math or physics, in order to solve some simple problem.

B. Automatic Programming

1. Introduction

The automatic writing, checking, and debugging of computer programs are problems of great interest both for their independent importance and as useful tools for intelligent machines. This section shows how a theorem prover can be used to solve certain automatic programming problems. The formalization given here will be used to precisely state and solve the problem of automatic generation of programs, including recursive programs, along with concurrent generation of proofs of the correctness of these programs. Thus any programs automatically written by this method have no errors.

We shall take LISP[43,44] as our example of a programming language. In the LISP language, a function is described by two entities: (1) its value, and (2) its side effect. Side effects can be described in terms of their effect upon the state of the program. Methods for describing state-transformation operations, as well as methods for the automatic writing of programs in a state-transformation language, were presented in Secs. VI-A and B. For simplicity, in this section we shall discuss "pure" LISP, in which a LISP function corresponds to the standard notion of a function--i.e., it has a value but no side effect.

Thus we shall use pure LISP 1.5 without the program feature, which is essentially the lambda calculus. In this restricted system, a LISP program is merely a function. For example, the LISP function car applied to a list returns the first element of the list. Thus if the variable x has as value the list (a b c), then car(x) = a. The LISP function cdr yields the remainder of the list; thus cdr(x) = (b c), and car(cdr(x)) = b. There are several approaches one may take in formalizing LISP; the one given here is a simple mapping from LISP's lambda calculus to the predicate calculus. LISP programs are represented by functions. The syntax of pure LISP 1.5 is normal function composition, and the corresponding syntax for the formalization is also function composition. LISP "predicates" are represented in LISP--and in this formalization--as functions having either the value nil (false) or else a value not equal to nil (true). The semantics are given by axioms relating LISP functions to list structures--e.g., $(\forall x,y)$car(cons(x,y)) = x, where cons(x,y) is the list whose first element is x and whose remainder is y.

In our formulation of programming problems, we emphasize the distinction between the program (represented as a function in LISP) that solves a problem, and a test for the validity of a solution to a problem (represented as a predicate in logic). It is often much easier to construct the predicate than it is to construct the function. Indeed, one may say that a problem is not well defined until an effective test for its solution is provided.

For example, suppose we wish to write a program that sorts a list. This problem is not fully specified until the meaning of "sort" is explained; and the method of explanation we choose is to provide a predicate R(x,y) that is true if list y is a sorted version of list x and false otherwise. (The precise method of defining this relation R will be given later.)

In general, our approach to using a theorem prover to solve programming problems in LISP requires that we give the theorem prover two sets of initial axioms:

(1) Axioms defining the functions and constructs of the subset of LISP to be used

(2) Axioms defining an input-output relation such as the relation R(x,y), which is to be true if and only if x is any input of the appropriate form for some LISP program and y is the corresponding output to be produced by such a program.

Given this relation R, and the LISP axioms, by having the theorem prover prove (or disprove) the appropriate question we can formulate the following four kinds of programming problems: checking, simulation, verifying (debugging), and program writing. These problems may be explained using the sort program as an example, as follows:

(1) <u>Checking</u>: The form of the question is R(a,b) where a and b are two given lists. By proving R(a,b) true or false, <u>b</u> is checked to be either a sorted version of <u>a</u> or not. The desired answer is accordingly either yes or no.

(2) <u>Simulation</u>: The form of the question is $(\exists x)R(a,x)$, where <u>a</u> is a given input list. If the question $(\exists x)R(a,x)$ is answered yes, then a sorted version of x exists and a sorted version is constructed by the theorem prover. Thus the theorem prover acts as a sort program. If the answer is no, then it has proved that a sorted version of x does not exist (an impossible answer if <u>a</u> is a proper list).

(3) <u>Verifying</u>: The form of the question is $(\forall x)R(x,g(x))$, where g(x) is a program written by the user. This mode is known as verifying, debugging, proving a program correct, or proving a program incorrect. If the answer to $(\forall x)R(x,g(x))$ is yes, then g(x) sorts every proper input list and the program is correct. If the answer is no, a counterexample list c, which the program will not sort, must be constructed by the theorem prover. This mode requires induction axioms to prove that looping or recursive programs converge.

(4) <u>Program Writing</u>: The form of the question is $(\forall x)(\exists y)R(x,y)$. In this synthesis mode the program is to be constructed or else proved impossible to construct. If the answer is yes, then a program--say, f(x)--must be constructed that will sort all proper input lists. If the answer is no, an unsortable list (impossible, in this case) must be produced. This mode also requires induction axioms. The form of the

problem statement shown here is oversimplified for the sake of clarity. The exact form will be shown later.

In addition to the possibility of the "yes" answer and the "no" answer, there is always the possibility of a "no proof found" answer if the search is halted by some time or space bound. The elimination of disjunctive answers, which is assumed in this section, was explained in Sec. VI-B.

These methods are summarized in the following table. The reader may view R(x,y) as representing some general desired input-output relationship.

Programming Problem	Form of Question	Desired Answer
Checking	$R(a,b)$	yes or no
Simulation	$(\exists x)R(a,x)$	yes, $x = b$ or no
Verifying	$(\forall x)R(x,g(x))$	yes or no, $x = c$
Program Writing	$(\forall x)(\exists y)R(x,y)$	yes, $y = f(x)$ or no, $x = c$

We now present an axiomatization of LISP followed by two axiomatizations of the sort relation R (one for a special case and one more general).

2. Axiomatization of a Subset of LISP

All LISP functions and predicates will be written in small letters. The functions "equal(x,y)", "atom(x)", and "null(x)" evaluate to "nil" if false and something not equal to "nil"--say "T"--if true. The predicates of first-order logic that are used to describe LISP are written in capital letters. These, of course, have truth values.

The version of LISP described here does not distinguish between an S-expression and a copy of that S-expression. There is some redundancy in the following formulation, in that certain functions and predicates could have been defined in terms of others; however, the redundancy allows us to state the problem more concisely. Also, some axioms could have been eliminated since they are derivable from others, but are included for clarity. The variables x, y, and z are bound by universal quantifiers, but the quantifiers are omitted for the sake of readability wherever possible. The formulation is given below:

<u>Predicates</u>	<u>Meaning</u>
NULL(x)	x = nil
LIST(x)	x is a list
ATOM(x)	x is an atom
x = y	x is equal to y

<u>Functions</u>	<u>Meaning</u>
car(x)	The first element of the list x.
cdr(x)	The rest of the list x.
cons(x,y)	If y is a list then the value of cons(x,y) is a new list that has x as its first element and y as the rest of the list--e.g., cons(1,(2 3)) = (1 2 3). If y is an atom instead of a list, cons(x,y) has as value a "dotted pair"--e.g., cons(1,2) = (1.2).
cond(x,y,z)	The conditional statement, <u>if</u> x = nil <u>then</u> y <u>else</u> z. Note that the syntax of this function is slightly different than the usual LISP syntax.
nil	The null (empty) list containing no elements.
equal(x,y)	Equality test, whose value is "nil" if x does not equal y.

atom(x)	Atom test, whose value is "nil" if x is not an atom.
null(x)	Null test, whose value is "nil" if x is not equal to nil.

Axioms

L1.	$x = car(cons(x,y))$
L2.	$y = cdr(cons(x,y))$
L3.	$\sim ATOM(x) \supset x = cons(car(x),cdr(x))$
L4.	$\sim ATOM(cons(x,y))$
L5.	$ATOM(nil)$
L6.	$x = nil \supset cond(x,y,z) = z$
L7.	$x \neq nil \supset cond(x,y,z) = y$
L8.	$x = y \equiv equal(x,y) \neq nil$
L9.	$ATOM(x) \equiv atom(x) \neq nil$
L10.	$NULL(x) \equiv null(x) \neq nil$.

3. A Simplified Sort Problem

Before examining a more general sort problem, consider the following very simple special case. Instead of a list-sorting program, consider a program that "sorts" a dotted pair of two distinct numbers--i.e., given an input pair the program returns as an output pair the same two numbers, but the first number of the output pair must be smaller than the second. To specify such a program, we must define the simple version of R, $R_0(x,y)$. Let us say that a dotted pair of numbers is "sorted" if the first number is less than the second. Thus, $R_0(x,y)$ is true if and only if y equals x when x is sorted and y is the reverse of x when x is not sorted. Stated more precisely, we have:

P1. $$(\forall x,y)\{R_0(x,y) \equiv [[car(x) < cdr(x) \supset y = x] \wedge [car(x) \nless cdr(x) \supset car(y) = cdr(x) \wedge cdr(y) = car(x)]]\} \ .$$

The correspondence of the LISP "lessp" function to the "less-than" relation is provided in the following axiom:

P2. $(\forall x,y)[\text{lessp}(x,y) \neq \text{nil} \equiv x < y]$.

Using the predicate R_0 we will give examples of four programming problems and their solutions:

(1) Checking:

Q: $R_0(\text{cons}(2,1),\text{cons}(1,2))$

A: yes

(2) Simulation:

Q: $(\exists x)R_0(\text{cons}(2,1),x)$

A: yes, $x = \text{cons}(1,2)$

(3) Verifying:

Q: $(\forall x)R_0(x,\text{cond}(\text{lessp}(\text{car}(x),\text{cdr}(x)),x,$
$\text{cons}(\text{cdr}(x),\text{car}(x)))$

A: yes

Thus the program supplied by the user is correct.

(4) Program writing:

Q: $(\forall x)(\exists x)R_0(x,y)$

A: yes, $y = \text{cond}(\text{lessp}(\text{car}(x),\text{cdr}(x)),x,$
$\text{cons}(\text{cdr}(x),\text{car}(x)))$

Translated into a more readable form, the program is:

<u>if</u> car(x) < cdr(x) <u>then</u> x <u>else</u> cons(cdr(x),car(x)) .

Given only the necessary axioms--L1, L2, L6, L7, P1, and P2--QA3 found a proof that constructed the sort program shown above. A limited form of the paramodulation[45,46] rule of inference was used to handle equality.

We now turn to a more difficult problem.

4. The Sort Axioms

The definition of the predicate R is in terms of the predicates ON and SD. The meaning of these predicates is given below:

R(x,y)	A predicate stating that if x is a list of numbers with no number occurring more than once in the list, then y is a list containing the same elements as x, and y is sorted--i.e., the numbers are arranged in order of increasing size.
ON(x,y)	A predicate stating that x is an element on the list y.
SD(x)	A predicate stating that the list x is sorted.

First we define R(x,y), that y is a sorted version of x, as follows:

S1. $(\forall x,y)\{R(x,y) \equiv [(\forall z)[ON(z,x) \equiv ON(z,y)] \wedge SD(y)]\}$.

Thus, a sorted version y of list x contains the same elements as x and is sorted.

Next we define, recursively, the predicate ON(x,y):

S2. $(\forall x,y)\{ON(x,y) \equiv [\sim ATOM(y) \wedge [x = car(y) \vee ON(x,cdr(y))]]\}$.

This axiom states that x is on y if and only if x is the first element of y or if x is on the rest of y.

Next we define the meaning of a sorted list:

S3. $(\forall x)\{SD(x) \equiv [NULL(x) \vee [\sim ATOM(x) \wedge NULL(cdr(x))] \vee [\sim ATOM(x) \wedge \sim NULL(cdr(x)) \wedge car(x) \leq car(cdr(x)) \wedge SD(cdr(x))]]\}$.

This axiom states that x is sorted if and only if x is empty, or x contains only one element, or the first element of x is less than the second element and the rest of x is sorted.

To simplify the problem statement we assume that the arguments of the predicates and functions range only over the proper type of objects--i.e., either numbers or lists. In effect, we are assuming that the input list will indeed be a properly formed list of numbers. (The problem statement could be modified to specify correct types by using predicates such as NUMBERP(x)--true only if x is, say, a real number.)

The problem is made simpler by using a "merge" function. This function and a predicate P describing the merge function are named and described as follows:

sort(x)	A LISP sort function (to be constructed) giving as its value a sorted version of x.
merge(x,u)	A LISP merge function merging x into the sorted list u, such that the list returned contains the elements of u, and also contains x, and this list is sorted.
P(x,u,y)	A predicate stating that y is the result of merging x into the sorted list u.

We define P(x,u,y), that y is u with x merged into it:

S4. $$(\forall x,u,y)\{P(x,u,y) \equiv [SD(u) \supset [SD(y) \wedge (\forall z)(ON(z,y) \equiv (ON(z,u) \vee z = x))]]\} \;.$$

Thus P(x,u,y) holds if and only if the fact that u is sorted implies that y contains x in addition to the elements of u, and y is sorted. One such merge function is merge(x,u) = cond(null(u),cons(x,u),cond(lessp(x,car(u)), cons(x,u),cons(car(u),merge(x,cdr(u))))).

The axiom required to describe the merge function is:

S5. $$(\forall x,u)P(x,u,merge(x,u)) \;.$$

This completes a description of the predicates ON, SD, R, and P. Together, these specify the input-output relation for a sort function and a merge function. Before posing the problems to the theorem prover, we need to introduce axioms that describe the convergence of recursive functions.

5. Induction Axioms

In order to prove that a recursive function converges to the proper value, the theorem prover requires an induction axiom. An example of an induction principle is that if one keeps taking "cdr" of a finite list, one will reach the end of the list in a finite number of steps. This is analogous to an induction principle for the non-negative integers--i.e., let "P" be a predicate, and "h" a function. Then, for finite lists,

$$[P(h(nil)) \wedge (\forall x)[\sim ATOM(x) \wedge P(h(cdr(x))) \supset P(h(x))]] \supset (\forall z)P(h(z))$$

is analogous to

$$[P(h(0)) \wedge (\forall n)[n \neq 0 \wedge P(h(n-1)) \supset P(h(n))]] \supset (\forall m)P(h(m))$$

for non-negative integers.

There are other kinds of induction criteria besides the one given above. Unfortunately, for each recursive function that is to be shown to converge, the appropriate induction axiom must be carefully formulated by the user. The induction axiom also serves the purpose of introducing the <u>name</u> of the function to be written. We will now give the problem statement for the sort program, introducing appropriate induction information where necessary.

6. The Sort Problem

The following examples illustrate the four kinds of problems:

(1) Checking:

Q: R(cons(2,cons(1,nil)),cons(1,cons(2,nil)))

A: yes

(2) Simulation:

Q: $(\exists x)R(\mathrm{cons}(2,\mathrm{cons}(1,\mathrm{nil})),x)$

A: yes, $x = \mathrm{cons}(1,\mathrm{cons}(2,\mathrm{nil}))$

(3) Verifying: Now consider the verifying or debugging problem. Suppose we are given a proposed definition of a sort function and we want to know if it is correct. Suppose the proposed definition is

S6. $(\forall x)[\mathrm{sort}(x) \equiv \mathrm{cond}(\mathrm{null}(x),\mathrm{nil},\mathrm{merge}(\mathrm{car}(x),$
$\mathrm{sort}(\mathrm{cdr}(x))))]$.

Thus sort is defined in terms of car, cdr, cond, null, merge, and sort. Each of these functions except sort is already described by previously given axioms. We also need the appropriate induction axiom in terms of sort. Of course, the particular induction axiom needed depends on the definition of the particular sort function given. For this sort function the particular induction axiom needed is

S7. $[R(\mathrm{nil},\mathrm{sort}(\mathrm{nil})) \wedge (\forall x)[\sim ATOM(x) \wedge R(\mathrm{cdr}(x),$
$\mathrm{sort}(\mathrm{cdr}(x))) \supset R(x,\mathrm{sort}(x))]] \supset (\forall y)R(y,\mathrm{sort}(y))$.

The following conjecture can then be posed to the theorem prover:

Q: $(\forall x)R(x,\mathrm{sort}(x))$

A: yes

(4) Program writing: The next problem is that of synthesizing or writing a sort function. We assume, of course, that no definition such as S6 is provided. Certain information needed for this particular problem might be considered to be a part of this particular problem statement rather than

a part of the data base. We shall phrase the question so that in addition to its primary purpose of asking for a solution, the question provides three more pieces of information: (1) The question assigns a name to the function that is to be constructed. A recursive function is defined in terms of itself, so to construct this definition the name of the function must be known (or else created internally). (2) The question specifies the number of arguments of the function that is to be considered. (3) The question (rather than an induction axiom) gives the particular inductive hypothesis to be used in constructing the function.

In this form, the question and answer are

$$\text{Q:} \quad (\forall x)(\exists y)\{R(nil,y) \wedge [[\sim ATOM(x) \wedge R(cdr(x), sort(cdr(x)))] \supset R(x,y)]\}$$

$$\text{A:} \quad \text{yes, } y = cond(equal(x,nil),nil,merge(car(x), sort(cdr(x))))\ .$$

Thus the question names the function to be "sort" and specifies that it is a function of one argument. The question gives the inductive hypothesis--that the function sorts cdr(x)--and then asks for a function that sorts x. When the answer y is found, y is labeled to be the function sort(x).

Using this formulation, QA3 was unable to write the sort program in a reasonable amount of time, although the author did find a correct proof within the resolution formalism.* The creation of the merge function can also be posed to the theorem prover by the same methods.

*In Appendix C the problem is reformulated using a different set of axioms. In the new formulation QA3 created the sort program "sort(x) = cond(x,merge(car(x),sort(cdr(x))),nil)."

7. Discussion of Automatic Programming Problems

The axioms and conjectures given here illustrate the fundamental ideas of automatic programming. However, this work as well as earlier work by Simon,[47] Slagle,[31] Floyd,[48] Manna,[49] and others provides merely a small part of what needs to be done. Below we present discussion of issues that might profit from further investigation.

Loops. One obvious extension of this method is to create programs that have loops rather than recursion. A simple technique exists for carrying out this operation. First, one writes just recursive functions. Many recursive functions can then be converted into iteration--i.e., faster-running loops that do not use a stack. McCarthy[50] gives criteria that determine how to convert recursion to iteration. An algorithm for determining cases in which recursion can be converted to iteration, and then performing the conversion process, is embedded in modern LISP compilers. This algorithm could be applied to recursive functions written by the theorem-proving program.

Separation of Aspects of Problem Solving. Let us divide information into three types:

(1) Information concerning the problem description and semantics. An example of such information is given in the axiom $AT(a,s_0)$, or Axiom S1 which defines a sorted list.

(2) Information concerning the target programming language, such as the axiom $[x = nil \supset cond(x,y,z) = z]$.

(3) Information concerning the interrelation of the problem and the target language, such as $[LESS(x,y) \equiv lessp(x,y) \neq nil]$.

These kinds of information are not, of course, mutually exclusive.

In the axiom systems presented, no distinction is made between such classes of information. Consequently, during the search for a proof the theorem prover might attempt to use axioms of type 1 for purposes where it needs information of type 2. Such attempts lead nowhere and generate useless clauses. However, as discussed in Sec. VI-B-6, we can place in the proof strategy our knowledge of when such information is to be used, thus leading to more efficient proofs. One such method--calling for the conditional axioms at the right time, as discussed in Sec. VI-B-6--has been implemented in QA3.

The PROW program of Waldinger and Lee[22] provides a very promising method of separating the problem of proof construction from the problem of program construction. In their system, the only axioms used are those that describe the subject--i.e., state the problem. Their proof that a solution exists does not directly construct the program. Instead, information about the target-programming language, as well as information about the relationship of the target-programming language to the problem-statement language, is in another part of the PROW program--the "post-processor." The post-processor then uses this information to convert the completed proof into a program. The post-processor also converts recursion into loops and allows several target programming languages.*

If our goal is to do automatic programming involving complex programs, we will probably wish to do some optimization or problem solving on the target language itself. For this reason we might want to have axioms that can give the semantics of the target language, and also allow the intercommunication of information in the problem-statement language with information in the target language. Two possible ways to do this effectively suggest themselves:

(1) Use the methods presented here, in which all information is in first-order logic. To gain efficiency, use special problem-solving strategies that minimize unnecessary interaction.

(2) Use a higher-order logic system, in which the program construction is separated from the proof construction, possibly by being at another level. The program construction process might then be described in terms of the first-order existence proof.

*It would be possible to use the "PROW techniques" in QA3 and vice-versa.

Problem Formulation. The axiomatization given here has considerable room for improvement: Missing portions of LISP include the program feature and the use of lambda to bind variables. The functions to be written must be named by the user, and the number of arguments must also be specified by the user.

Heuristics for Program-Writing Problems. Two heuristics have been considered so far. The first consists of examining the program as it is constructed (by looking inside the answer literal). Even though the syntax is guaranteed correct, the answer literal may contain various nonsense or undefined constructions [such as car(nil)]. Any clause containing such constructed answers should be eliminated. Another heuristic is to actually run the partial program by a pseudo-LISP interpreter on a sample problem. The theorem prover knows the correct performance on these sample problems because they have either been solutions or else counterexamples to program-simulation questions that were stored in memory, or else they have been provided by the user. If the pseudo-LISP interpreter can produce a partial output that is incorrect, the partial program can be eliminated. If done properly, such a method might be valuable, but in our limited experience its usefulness is not yet clear.

Higher-Level Programming Concepts. A necessary requirement for practical program writing is the development of higher-level concepts (such as the LISP "map" function) that describe the use of frequently employed constructs (functions) or partial constructs.

Induction. The various methods of proof by induction should be studied further and related to the kinds of problems in which they are useful. The automatic selection or generation of appropriate induction axioms would be most helpful.

Program Segmentation. Another interesting problem is that of automatically generating the specifications for the subfunctions to be called before writing these functions. For example, in our system the sort problem was divided into two problems: First, specify and create a merge function; next, specify a sort function and then construct this function in terms of the merge function. The segmentation into two problems and the specification of each problem was provided by the user.

C. Self-Description

One of the goals for future problem solving by a theorem-proving-based system is the ability to deal with self-descriptions. In this section a portion of the operation of QA3 itself is axiomatized. The resolution theorem-proving strategy--namely, the unit-preference strategy with the set-of-support strategy--is formalized.

It is intended that this strategy axiomatization be a step toward self-usable self-descriptions of programs. The uses fall into two categories:

(1) Given a description of a strategy, the program will be able to carry out the strategy.

(2) Given a description of a strategy, the program will be able to reason about, make inferences about, or modify, a strategy.

There are at least three methods (and combinations thereof) by which a program could carry out an axiomatically-described strategy. The first is to use a theorem prover to prove (by its own strategy) that there exists a proof (by the described strategy) of a given theorem. The theorem prover operates according to its own strategy--say, Strategy I. The axiomatically-described strategy is, say, Strategy II. The object constructed by Strategy I will be a complete proof search according to Strategy II. Such a technique would be very slow. A second and faster method to carry out an axiomatically-described strategy would be to build an interpreter of strategy axioms, using extensions of techniques such as predicate evaluation. A third method is to have the theorem prover prove the existence of, and hence write, a special program (algorithm) that will carry out the proof search according to the described strategy.

The other use of strategy description is reasoning about strategies. Once we have the description in the language of logic, the theorem prover can make inferences about the strategy. One can imagine a proof that a strategy is complete, a proof that one strategy dominates another under certain conditions, or a proof that proves the existence of a better strategy and creates it. A logical strategy description could also provide a dialogue language in which a user can discuss strategy with the program, discussing, for example, the feasibility of a proposed strategy change.

In practice, these uses of a strategy description are beyond the capabilities of the program. The methods given here do establish one possible approach, but I believe that a practical system would require a very careful and uniform problem formulation, better than the one given here, possibly in a suitable, hierarchically-organized higher-order logic.

The axioms are first-order, in that variables are allowed only at the level of terms. However, the terms are allowed to range over wffs of first-order logic. The wffs are treated as symbol string terms.

First, a very simple axiomatization of theorem proving is presented to illustrate the basic ideas. Then, the formalization is modified to show how more information about proof strategies can be introduced.

1. Rules of Inference

This first set of axioms illustrates how clauses can be inferred by resolution. Upper-case variables will be used to represent types of variables. Subscripts indicate specific individuals of each type. The following are the variable types, relations, and axioms that will be used.

Variable Types	Representation
Literal	L_i
Clause	C_i
Set of clauses	B_i
Well-formed formula of first-order logic	S_i

Relations	Meaning
RESOLVE(C_1,C_2,C_3)	C_3 is a resolvent of C_1 and C_2.
INFER(B_i,C_i)	C_i is inferred from B_i by successive resolution or factorings.
MEMBER(C_i,B_i)	C_i is a member of the set of clauses B_i.

Axioms

T1. $(\forall C_i, B_i)[\mathrm{MEMBER}(C_i, B_i) \supset \mathrm{INFER}(B_i, C_i)]$

T2. $(\forall B_i, C_i, C_j, C_k)[\mathrm{INFER}(B_i, C_i) \wedge \mathrm{INFER}(B_i, C_j) \wedge \mathrm{RESOLVE}(C_i, C_j, C_k) \supset \mathrm{INFER}(B_i, C_k)]$

Observe that $\mathrm{RESOLVE}(C_i, C_j, C_k)$ is decidable. That is, if we are given any C_i and C_j, then there is a program that could determine if there is a resolvent C_k and produce it if it exists. In a proof we might use the predicate evaluation mechanism to produce such a C_k.

2. Proof by Refutation

To describe proof by refutation we need the following additional functions:

Functions	Meaning
$\mathrm{not}(S_i)$	The negation of a statement S_i
$\mathrm{union}(B_i, B_j)$	The union of sets B_i and B_j
$\mathrm{clauses}(S_i)$	The set of clauses representing statement S_i
null	The null clause.

Now let the theorem to be proved be denoted by S_f. Assume the theorem is to follow from a set of axioms. This set of axioms will be represented by the conjunction of the axioms S_0, forming a single statement.

We say that the predicate $\mathrm{PROVES}(S_0, S_f)$ is true if and only if there exists a refutation proof of S_f from S_0. This fact is described by the axiom

T3. $(\forall S_0, S_f)[\mathrm{INFER}(\mathrm{union}(\mathrm{clauses}(S_0), \mathrm{clauses}(\mathrm{not}(S_f))), \mathrm{null}) \supset \mathrm{PROVES}(S_0, S_f)]$

where infer is defined by T1 and T2. Thus, to show that S_f follows from S_0, we show $\mathrm{PROVES}(S_0, S_f)$ to be valid.

3. Set-of-Support Strategy

To express the set-of-support strategy, T2 must be modified. Let SUPPORT(C_i) be a predicate indicating that clause C_i has support from the theorem. The negation of the theorem S_f will be the initial set of support. The modified version of T2 is:

T2′. $$\{(\forall C_i)[\text{MEMBER}(C_i,\text{clauses}(\text{not}(S_f))) \supset \text{SUPPORT}(C_i)]\} \wedge \{(\forall B_i,C_i,C_j,C_k)\{[\text{INFER}(B_i,C_i) \wedge \text{INFER}(B_i,C_j) \wedge \text{RESOLVE}(C_i,C_j,C_k) \wedge [\text{SUPPORT}(C_i) \vee \text{SUPPORT}(C_j)]] \supset \text{INFER}(B_i,C_k) \wedge \text{SUPPORT}(C_k)\}\} \quad .$$

4. Unit-Preference Strategy

The above formulations of provability are nonsequential, in the sense that the order of creation of clauses by resolution is not specified. We show below how the unit-preference strategy, which is sequential, can be described. Several new concepts are needed:

Expression	Meaning
length(C_i)	A function whose value is the length of the clause C_i (number of literals).
ORDER(C_i,j)	A predicate meaning that the clause C_i is the jth clause created.
RESOLVE(C_a,C_b,L)	This RESOLVE predicate is different than the previous RESOLVE predicate. RESOLVE(C_a,C_b,L) is true if and only if the third argument L is the list of all possible resolvents created by resolving C_a against C_b on all possible combinations of literals. If there are no resolvents, L is the empty list.
TRY(B_i,n,C_i,C_j)	A predicate meaning that on the nth step of the proof search from clauses B_i, the resolution of C_i against C_j is attempted. Thus, n counts the attempted resolutions.

The unit-preference strategy does not completely specify the order in which clauses are resolved--e.g., it does not specify which two unit clauses should be resolved first. The axioms below give an ordering down to the clause level, but not at the literal level. The axiom presumes that all possible resolvents of two clauses are created in one step. Axioms T4 and T5 specify the next pair of candidates for resolution.

$$\begin{aligned}
\text{T4.}\quad & (\forall n,a,b,x,y,C_a,C_b,C_x,C_y,B_i)\{[[\sim((C_a = C_x \wedge C_b = C_y) \vee (C_a = C_y \wedge C_b = C_x))] \wedge \\
& \quad \mathrm{INFER}(B_i,C_a) \wedge \mathrm{INFER}(B_i,C_b) \wedge \mathrm{INFER}(B_i,C_x) \wedge \mathrm{INFER}(B_i,C_y) \wedge \\
& \quad \mathrm{ORDER}(C_a,a) \wedge \mathrm{ORDER}(C_b,b) \wedge \mathrm{ORDER}(C_x,x) \wedge \mathrm{ORDER}(C_y,y) \wedge \\
& \quad (\forall m)[m < n \supset [\sim\mathrm{TRY}(B_i,m,C_a,C_b) \wedge \sim\mathrm{TRY}(B_i,m,C_x,C_y) \wedge \\
& \qquad \sim\mathrm{TRY}(B_i,m,C_b,C_a) \wedge \sim\mathrm{TRY}(B_i,m,C_y,C_x)]] \wedge \\
& \quad [\mathrm{length}(C_a) = 1 \vee \mathrm{length}(C_b) = 1 \vee (\forall C_u,C_v)[(\mathrm{length}(C_u) = 1 \vee \\
& \qquad \mathrm{length}(C_v) = 1) \supset (\exists \ell)(\ell < n \wedge \mathrm{TRY}(B_i,\ell,C_u,C_v))]] \wedge a < b \wedge \\
& \quad [\mathrm{length}(C_a) + \mathrm{length}(C_b) \leq \mathrm{length}(C_x) + \mathrm{length}(C_y)] \wedge \\
& \quad [[\mathrm{length}(C_a) + \mathrm{length}(C_b) = \mathrm{length}(C_x) + \mathrm{length}(C_y)] \supset \\
& \qquad [[\min(\mathrm{length}(C_a),\mathrm{length}(C_b)) \leq \min(\mathrm{length}(C_x),\mathrm{length}(C_y))] \wedge \\
& \qquad\quad [a = x \supset b < y]]] \wedge \\
& \quad [\mathrm{SUPPORT}(C_a) \vee \mathrm{SUPPORT}(C_b)] \wedge n \geq 0] \supset \mathrm{TRY}(B_i,n,C_a,C_b)\}
\end{aligned}$$

$$\begin{aligned}
\text{T5.}\quad & (\forall C_a,C_b,C_i,C_j,m,B_i)\{[\mathrm{TRY}(B_i,m,C_i,C_j) \wedge (C_a \neq C_i \vee C_b \neq C_j)] \supset \\
& \quad \sim\mathrm{TRY}(B_i,m,C_a,C_b)\} \quad .
\end{aligned}$$

Two more axioms, T6 and T7, are required to specify the assignment of an order to each clause. The predicate "NEXTORDER(n,m)" states that the resolvents created on the nth try are to be ordered sequentially starting with m. "ASSIGNORDER(L,q,k)" implies that the clauses on the list L are assigned orders q+1 through q+k. First(L) and rest(L) are functions referring respectively to the first element of the list L and the rest of the list L (the empty list has zero length).

$$\begin{aligned}
\text{T6.}\quad & (\forall n,q,k,L,C_a,C_b,B_i)\{[\mathrm{NEXTORDER}(n,q) \wedge \mathrm{TRY}(B_i,n,C_a,C_b) \wedge \\
& \quad \mathrm{RESOLVE}(C_a,C_b,L) \wedge \mathrm{length}(L) = k] \supset \\
& \quad [\mathrm{NEXTORDER}(n+1,q+k) \wedge \mathrm{ASSIGNORDER}(L,q,k)]\}
\end{aligned}$$

T7. $(\forall q,k,L)\{[\mathrm{ASSIGNORDER}(L,q,k) \wedge k \neq 0] \supset$
$[\mathrm{ORDER}(\mathrm{first}(L),q+1) \wedge \mathrm{ASSIGNORDER}(\mathrm{rest}(L),q+1,k-1)]\}$.

The clauses in the initial set, $B_0 = \mathrm{UNION}(\mathrm{clauses}(S_0),\mathrm{clauses}(\mathrm{not}(S_f)))$, are assumed to be assigned orders in such a way that longer clauses have higher orders. The initial condition for assigning orders to generated clauses is given by

T8. NEXTORDER(0,b)

where b is the number of clauses in B_0. To complete the formalization, T2$'$ must be modified to use TRY:

T2$''$. $\{(\forall C_i)[\mathrm{MEMBER}(C_i,\mathrm{clauses}(\mathrm{not}(S_f))) \supset \mathrm{SUPPORT}(C_i)]\} \wedge$
$\{(\forall C_i,C_j,S_j,n,L,B_i)\{[\mathrm{TRY}(B_i,n,C_i,C_j) \wedge \mathrm{RESOLVE}(C_i,C_j,L) \wedge \mathrm{ON}(C_k,L)] \supset$
$[\mathrm{INFER}(S_j,C_k) \wedge \mathrm{SUPPORT}(C_k)]\}\}$.

The predicate $\mathrm{ON}(C_k,L)$ means that the clause C_k is on the list L.

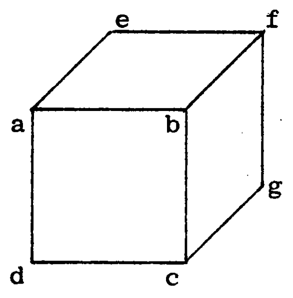
The complete set of axioms describing a unit-preference, set-of-support proof is T1, T2$''$, T3, T4, T5, T6, T7, and T8. To see if S_0 follows from S_f by this strategy, the statement $\mathrm{PROVES}(S_0,S_f)$ must be shown to follow from T1, T2$''$, T3, ..., T8. The course of the proof must necessarily imply a sequence of true statements of the form $\mathrm{TRY}(B_i,0,C_i,C_j),\mathrm{TRY}(B_i,1,C_k,C_\ell),\ldots,\mathrm{TRY}(B_i,n,C_p,C_q)$, where C_p and C_q resolve to yield the empty clause.

D. Pattern Recognition--Scene Description

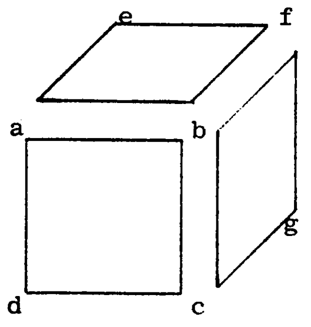
This section presents a pattern-recognition problem consisting of finding, in a line drawing of a scene, a two-dimensional projection of a cube. The problem is a scene-description or scene-analysis task: Given a set of line segments the problem is to find a cube and describe it in terms of its vertices.

This problem is an interesting exercise in developing heuristics for using the theorem prover. This problem is a study in the *specialization* of the theorem prover to a particular well-defined problem, rather than an effort toward *generality*. The initial problem formulation and proof strategy selected resulted in an extremely inefficient search. An improvement in efficiency resulted from several changes: a better representation, an extension of the unification algorithm to automatically handle certain equalities, and the use of a measure of progress so that a hill-climbing search method could be used. Using these heuristics the theorem prover is made to perform very well on this scene-description problem. However, because the heuristics are aimed at this particular problem rather than a more general problem, we have not established that theorem-proving methods would be applicable to more difficult scene-description problems.

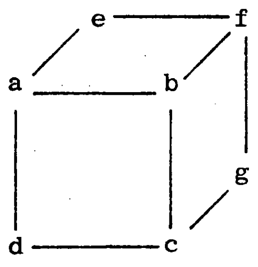
The scene consists of nine line segments connected together to form a two-dimensional projection of a cube, shown below. The cube can be



The Cube



Its Three Faces



Its Nine Edges

decomposed into its three faces and further decomposed into its nine edges or line segments. The input data for the problem consist of the nine line segments, where each line segment is named by its two end points. The line drawing is assumed perfect (a mechanism for postulating the existence of missing lines is discussed later).

The axiomatization discusses three kinds of objects--lines, quadrilaterals, and cubes--which are defined as follows:*

*It might be better to characterize the problem by some other means, such as regions, connections, and cubes.

(1) A line is the basic element.

(2) A quadrilateral is a set of four suitably connected lines.

(3) A cube is a set of three suitably connected quadrilaterals.

Now consider two alternative axiomatizations within this framework. The first formulation uses the predicate LINE(a,b) to represent the fact that in the scene there is a line segment $\underline{ab}$ between points a and b. As the second alternative, the line segment from a to b can be represented by a single term p(a,b), where p(a,b) denotes the line joining together the arguments a and b. In this alternative formulation, which we shall use, the predicate LINE(p(a,b)) means that in the scene there is a line segment $\underline{ab}$.

Now assume that we are given as an axiom this fact that the line segment $\underline{ab}$ is in the scene, and we wish to deduce the fact that line segment $\underline{ba}$ is in the scene. This deduction is correct, since a line segment is not considered to be directional. For alternative 1, the axiom

$$(\forall x,y)[\mathrm{LINE}(x,y) \supset \mathrm{LINE}(y,x)]$$

is needed. For alternative 2, the equality axiom

$$(\forall x,y)[p(x,y) = p(y,x)]$$

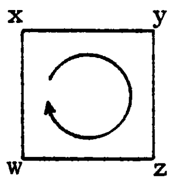
along with an equality substitution axiom or mechanism is needed. The addition of either one of these two clauses to the data base results in the deduction of many extra clauses during the search for the cube.

Using the second alternative, the search can be narrowed by extending the unification algorithm to allow the two terms p(a,b) and p(b,a) to unify. Thus, as an example, the deduction of LINE(p(a,b)) from LINE(p(b,a)) follows in one resolution step. This is a way of automatically treating a particular kind of equality.

Consider the definition of a quadrilateral,[*] [†]

C1: $(\forall x,y,z,w)\{[\text{LINE}(p(x,y)) \wedge \text{LINE}(p(y,z)) \wedge \text{LINE}(p(z,w)) \wedge \text{LINE}(p(w,x))] \supset \text{QUADRILATERAL}(p(x,y,z,w))]\}$,

which can be illustrated as follows:



This axiom states that four suitably connected line segments form a quadrilateral, p(x,y,z,w). The term p(x,y,z,w) is thus allowed a variable number of arguments and denotes the line segments <u>xy</u>, <u>yz</u>, <u>zw</u>, and <u>wx</u>. The equality mechanism for the single line segment allows the line segments to be named arbitrarily, from either end. It allows the quadrilateral to be named starting from any vertex, and tracing around the vertices in a particular direction--say, clockwise. Thus the two quadrilaterals <u>xyzw</u> and <u>yzwx</u> are considered equal. The same extension of the unification algorithm is used to handle this case. The precise statement of this treatment of equality is as follows: the function letter p is allowed an indefinite number of arguments; any two terms $p(a_1,a_2,\ldots,a_m)$ and $p(b_1,b_2,\ldots,b_n)$ unify if and only if $m = n$ and there exists a <u>cyclic permutation</u> of $b_1,b_2,\ldots,b_n$ that unifies with $a_1,a_2,\ldots,a_m$.[§]

---

[*] Instead of a quadrilateral one might assume that some geometry information is available, so that one might look for, say, a perspective transformation of a rectangular parallelipiped.

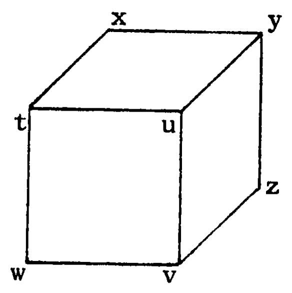
[†] The reader may have observed that Axiom C_1 admits as quadrilaterals a wide class of line graphs that might be subgraphs of a given graph. For example, it admits non-convex quadrilaterals as well as convex quadrilaterals crossed by diagonal line segments. A more restrictive definition might be advantageous if there were many lines in the scene.

[§] The subsumption algorithm is also made compatible with this special equality.

Now consider the definition of a two-dimensional projection of a cube:[*]

C2. $(\forall t,u,v,w,x,y,z)\{[\mathrm{QUADRILATERAL}(p(t,u,v,w)) \wedge \mathrm{QUADRILATERAL}(p(y,u,t,x)) \wedge \mathrm{QUADRILATERAL}(p(v,u,y,z))] \supset \mathrm{CUBE}(p(p(t,u,v,w),p(y,u,t,x),p(v,u,y,z)))\}$,

which can be illustrated as follows:

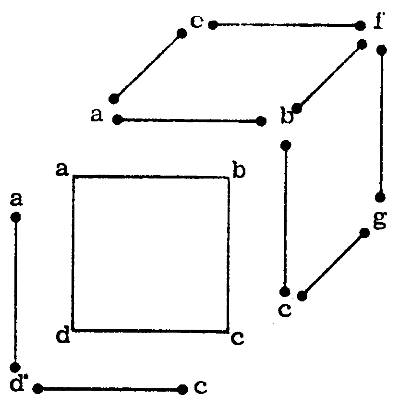


Again the function p is used, with p(a,b,c) representing the cube whose quadrilaterals (clockwise) are a, b, and c.

The problem is posed to the theorem prover by giving as axioms C1, C2, and the line segments. A line segment is presented as an axiom such as LINE(p(a,b)). It is presumed that these lines are supplied by a line-finding scene-analysis program. The quadrilaterals can be given as data instead of the lines, or else any suitable combination of quadrilaterals and lines can be provided as data.

In a test run, one quadrilateral and nine lines were given as the input data, as shown below.

[*]In dealing with complicated line graph structures, one might want a more restricted definition of a cube.



This scene is described by the following axioms:

QUADRILATERAL(p(a,b,c,d))

LINE(p(a,b))

LINE(p(b,c))

LINE(p(c,d))

LINE(p(d,a))

LINE(p(a,e))

LINE(p(f,g))

LINE(p(f,b))

LINE(p(g,c))

LINE(p(e,f)) .

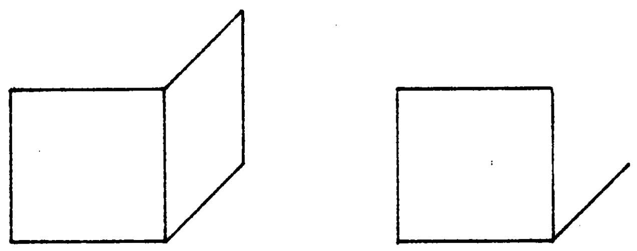
The problem was then posed as the question $(\exists x)$CUBE(x). Even with the addition of the special treatment of equality, the search does not proceed as desired. One form of undesired intermediate clause made two of the vertices of a quadrilateral or two vertices of the cube be the same vertex. Such a deduction can lead to a proof only in the case where the cube is seen from the edge in such a manner that two vertices coincide. Suppose, for the sake of exploring another search-narrowing trick, we will admit only cubes whose two-dimensional projections merge no vertices. Then we can use the new predicate DISTINCT($\ell(x_1,x_2,...,x_n)$) that is true if and only if the points $x_1, x_2, \ldots, x_n$ are distinct. The term function letter ℓ stands here for a list of indefinite length. Axioms C_1 and C_2 are revised, yielding

C1′. $(\forall x,y,z,w)\{[\mathrm{LINE}(p(x,y)) \wedge \mathrm{LINE}(p(y,z)) \wedge \mathrm{LINE}(p(z,w)) \wedge \mathrm{LINE}(p(w,x)) \wedge \mathrm{DISTINCT}(\ell(x,y,z,w))] \supset \mathrm{QUADRILATERAL}(p(x,y,z,w))\}$

C2′. $(\forall t,u,v,w,x,y,z)\{[\mathrm{QUADRILATERAL}(p(t,u,v,w)) \wedge \mathrm{QUADRILATERAL}(p(y,u,t,x)) \wedge \mathrm{QUADRILATERAL}(p(v,u,y,z)) \wedge \mathrm{DISTINCT}(\ell(t,u,v,w,x,y,z))] \supset \mathrm{CUBE}(p(p(t,u,v,w),p(y,u,t,x),P(v,u,y,z)))\}$.

The predicate evaluation mechanism is then used to evaluate the predicate DISTINCT. When a clause is generated in which two arguments (ground terms or not) of DISTINCT are the same, the literal is effectively subsumed and the clause is deleted.

One further improvement was made by using a hill-climbing proof strategy instead of the unit-preference proof strategy. For this particular problem a good measure of progress is available--namely, how much of the cube is constructed. For example, a partial solution consisting of two completed quadrilaterals is further along than a solution consisting of one completed quadrilateral plus one more edge, as shown below:



To measure progress a value is computed for each clause generated in the proof search. The user creates an evaluation function that assigns this value to a clause. The value of a potential resolvent is predicted before the resolvent is actually generated. The next clause generated at any step is the clause that is predicted to have the highest value. The value of an initial clause is zero; one point is added for each line added to the cube being constructed, and thus four points are added when a quadrilateral is completed.

Together, all these methods finally result in a search that finds a proof and generates no incorrect nodes at all. The proof is shown in Appendix D. This completes an illustration of how one can "tune" the theorem prover to work well on a particular problem.

This cube-recognition problem leads to a method by which missing lines can be postulated, thus generating requests for the line finder to look again in a particular place. Recall that as the proof progresses, the theorem prover requests additional data, in the form of clauses, from memory. Suppose that in a search for a cube all lines but one are filled in so that the measure of progress is very high. Since the lines that would eventually connect to the missing line are filled in, the end points of the missing line are known. Because of the high progress measure, when the theorem prover requests this missing line from memory the request could be channeled to the line finder, asking the line finder to look harder in that place.

VIII DISCUSSION AND CONCLUSIONS

A. Adequacy of Theorem Proving for Question Answering

The method of theorem proving by resolution has been demonstrated to be an adequate deduction technique for many question-answering tasks. The answer construction mechanism greatly extends the question-answering power of the theorem-proving method. The simple measure of relevance used for selecting clauses from the data base--whether or not they resolve with the best candidate clause in the active clause set--is adequate for easy problems but needs improvement. The simple memory organization--indexing of clauses by predicate letters and length, with the clauses sharing as much common substructure as possible--is adequate for the question-answering tasks considered so far and is not a limiting factor in the system's performance. For the subjects treated it has been possible to adequately express the semantics in the language of first-order logic.

B. Theorem Proving and Problem Solving

The first applications of QA2 and QA3 were to "question answering." Typical question-answering applications are usually easy for a resolution-type theorem prover. Examples of such easy problem sets given QA3 include the questions done by Raphael's SIR,[4] Slagle's DEDUCOM,[3] and Cooper's chemistry question-answering program.[39] Usually there are a few obvious formulations for some subject area, and any reasonable formulation works well. As one goes to harder problems like the Tower of Hanoi puzzle, and program-writing problems, good and reasonably well-thought-out representations are necessary for efficient problem solving.

As problems become more difficult, not only are representations more critical, but the proper selection of strategies becomes increasingly important. The theorem prover may be considered an "interpreter" for a high-level assertational or declarative language--logic. As is the case with most high-level programming languages the user may be somewhat distant from the efficiency of "logic" programs unless he knows something about the strategies of the system.

Some representations are better than others only because of the particular strategy used to search for a proof. It would be desirable if the theorem prover could adopt the best strategy for a given problem and representation, or even change the representation. I don't believe these goals are impossible, but at present they have not been reached. However, a library of strategy programs and a strategy language is slowly evolving in QA3. To change strategies in the present version the user must know about set-of-support and other program parameters such as level bound and term-depth bound. To radically change the strategy, the user presently has to know the LISP language and must be able to modify certain strategy sections of the program. In practice, several individuals who have used the system have modified the search strategies to suit their needs. To add and debug a new heuristic or to modify a search strategy where reprogramming is required seems to take from a few minutes to several days. Ultimately it is intended that the system will be able to write simple strategy programs itself, and "understand" the semantics of its strategies.

C. An Experimental Tool

The program QA3 as well as its predecessor QA2 has served as a usable experimental tool for several researchers. The computer program is reasonably clean and well-documented (as experimental programs go). It is provided with many user-oriented features such as editing facilities for the data base, extensive on-line tracing of proof searches, controls on the search process, and statistics on each search (cf. Sec. IV).

One experimental use of the theorem-proving program is to test problem formulations. In exploring difficult problems it can be useful to write a computer program to test a problem formulation and solution technique. The machine tends to sharpen one's understanding of the problem. I believe that in some problem-solving applications the "high-level language" of logic along with a theorem-proving program can be a quick programming method for testing ideas. One reason is that a representation in the form of an axiom system can correspond quite closely to one's conceptualization of a problem. Another reason is that it is sometimes easier to reformulate an axiom system than to rewrite a problem-solving program, and this ease of reformulation facilitates exploration. As mentioned earlier, part or all of the problem formulation (and possibly some solutions) can be saved as axioms and used as part of the final problem-solving mechanism if desired.

Raphael, Coles, and others[28] have begun to study some medical question-answering applications in a project supported by the National Library of Medicine. One experiment successfully utilized a data base of 300 clauses to suggest suitable drugs for particular cases. This experiment used the predicate evaluation mechanism in a special treatment of the exhaustive enumeration of finite sets. This particular project has emphasized the need to develop special fast search and retrieval methods for "easy" questions in a large data base.

Kling[29] has used and modified QA3 in a research project concerning the use of analogy to discover difficult mathematical proofs in geometry and algebra. He uses a previously solved problem and its resolution proof as a model for a newly posed allegedly analogous one. Both problems (theorems) are posed on a common data base, and the analogy is used to provide relevance-criteria for deciding which subset of the data base should be used for solving the new problem. In addition, various "cues" such as interesting (analogous) lemmas are extracted from the model proof, proved in the analog case and added to CLAUSELIST. The analogy system (ZORBA) uses QA3 in the last step of a process that began with the analogy generation and cue extraction.

D. A Brief Comparison to Other Systems

The program has been tested on several question sets used by earlier question-answering programs. The subjects for the first question set, reported by Green and Raphael,[1] consisted of some set membership, set inclusion, part-whole relationship, and similar problems.

Raphael's SIR[4,5] gave a similar but larger problem set also having the interesting feature of requiring facts or axioms from several subjects to interact in answering a question. SIR used a different subroutine to answer each type of question, and when a new relation was added to the system, not only was a new subroutine required to deal with that relation but also changes throughout the system were usually necessary to handle the interaction of the new relation with the previous relations. This programming difficulty was the basic obstacle in enlarging SIR. Raphael proposed a "formalized question answerer" as the solution. QA3 was tested on the SIR problem set with the following results: All the facts programmed into or told to SIR were entered into the QA3 memory as axioms of first-order logic, and QA3 answered essentially all the questions answered by SIR. The questions missed used the special SIR heuristic, the "exception principle." It was possible to hand-translate, as they were read, questions and facts stated in SIR's restricted English, into first-order logic.

Slagle, in his paper on Deducom,[31] a question-answering system, presented a broader, though less interactive, problem set consisting of gathered questions either answered by programs of, or else proposed by, Raphael,[4,5] Black,[51] Safier,[52] McCarthy,[13] Cooper,[39] and Simon.[53] Deducom was considered one of the best question-answering systems using non-English inputs. Included in this set were several examples of sequential processes, including one of McCarthy's End Game Questions,[13] Safier's Mikado Question,[52] McCarthy's Monkey-and-Bananas Question,[13] and one of Simon's State Description Compiler Questions.[53] Using the technique discussed in Sec. VI to describe processes, it was possible to axiomatize for QA3 all the facts and to answer all the questions printed in Slagle's paper. Furthermore, QA3 overcame some of the defects of deducom: QA3 could answer all answerable questions, the order of presenting the axioms did not affect its ability to answer questions, and no redundant facts were required. QA3 was then tested on the entire set of 23 questions presented by Cooper.[39] QA3 correctly answered all the questions, including four not answered by Cooper's program and sixteen not answered by Deducom.

In addition to these common question-answering problems, QA3 also solved the Wolf, Goat, and Cabbage puzzle in which a farmer must transport the wolf, goat, and cabbage across the river in a boat that can hold only himself and one other. The wolf cannot be left alone with the goat and the goat cannot be left alone with the cabbage. QA3 has also solved the Tower of Hanoi puzzle (see Sec. VI-D) and some simple analogy puzzles.

In all of the problems mentioned above, QA3 was given the facts and questions in first-order logic, whereas Raphael's program and Cooper's program used a restricted English input. However, in a test run Coles' program translated Cooper's questions from English into logic, and QA3 was able to answer all the questions.

The General Problem Solver (GPS) of Newell, Shaw, and Simon, discussed at length in Newell and Ernst,[54] has solved many problems, some rather difficult. QA3 can do the easier GPS problems, but it does not perform as well on some of the most difficult. The difference is that GPS is designed so that if the user supplies "differences" that specify which subproblem to attempt next, the search procedure effectively uses this information to narrow its search. Such search guidance is not built into QA3. It would be of interest to introduce the GPS search strategy or a similar search strategy into a resolution program such as QA3. An advantage of QA3 is that the language of mathematical logic is more elegant and often easier to use, in my own opinion, than the transformation language of GPS. QA3 is also more of a true question-answering system than GPS, having storage and retrieval capabilities and a larger interactive data base (rather than necessarily being tuned like GPS for one problem at a time).

E. Alternate Approaches

A detailed comparison of all the known possible alternate approaches to question answering and problem solving would be very valuable, but unfortunately no one has yet undertaken this task. In this section I will mention a few of the more obvious approaches and provide references. Simmons[14,15] provides a description of some methods that have already been implemented for use in question-answering systems.

One large class of candidates for the basis of a question-answering system consists of the various classical kinds of logic. These include propositional logic, first-order logic, higher-order logic, and modal logic. (Many working question-answering programs use some comparable systems of logic but defy such simple categorizations.) The higher-order logics and modal logics can be more powerful than first-order in their ability to express concepts, and first-order is in turn more powerful than propositional. If this is so, why use anything except a higher-order logic in a question-answering system? The answer lies in the present state of knowledge about methods of using each system. For propositional logic there exist fast, tested decision procedures. First-order logic is not decidable; however, there exist slower but reasonable machine-implementable proof procedures. In general not as much is known about how one can implement a practical higher-order system. One method for using logic that may be feasible in certain cases is to state a problem in, say, modal logic and then translate it into first-order logic so that a first-order proof procedure may be used. McCarthy and Hayes[40] present a relevant philosophical discussion of logics. Hewitt[55] presents a programming system intended for the implementation of a higher-order-logic theorem prover. Robinson[56] presents a higher-order logic system.

In addition to the more nearly classical logical approaches to constructing a problem solver, several problem-solving systems utilizing other approaches have been proposed and several have been implemented. Because all such systems (as well as QA3) are relatively new, and because the systems use quite different mechanisms (at least on the surface), a detailed comparison to resolution theorem-proving methods is difficult, and remains an open question.

A subject method closely related to logic is set theory. Set-theoretic methods can be imbedded in logic (and vice versa). But sometimes one would rather speak explicitly in terms of predicates, and sometimes one would rather speak explicitly in terms of sets, especially in problems involving the enumeration of finite sets. As far as I know, the present state of knowledge about what question-answering and problem-solving procedures could be used effectively within a set-theoretic framework is not as advanced as knowledge of first-order-logic proof procedures. Suppes[57] discusses "Set Theoretical Structures in Science," and Sandwall[58] discusses a promising machine-implementable set-based question-answering system.

Burstall developed "A Combinatory Approach to Relational Question Answering and Syntax Analysis."[59] His system is based upon combinators, which are functions having functions as arguments and functions as values.

In another, quite different approach, Fikes[60] discusses a problem-solving system in which problems are stated as ALGOL-like procedures and then a problem-solving program finds the correct values of variables left constrained but unspecified in the problem statement.

One feature of many of these methods that has struck me is that there is an underlying similarity in the development of each of the diverse approaches to the development of resolution theorem proving. Each approach seems to first enter a phase in which it is discovered that the approach is "incomplete" in some practical sense. Typically there is a quick and effective strategy for easy problems--corresponding to a depth-first unit-preference strategy. Later comes a difficult transition to case analyses, and breadth-first search--corresponding to the non-unit strategies. Initial strategies tend to resemble the set-of-support strategy. Matching procedures are at first often not as general as possible, so that each problem-solving step unnecessarily binds variables to incorrect values. Later one sees the need for sophisticated and versatile subject-dependent strategies, and better problem representations. More elaborate matching procedures are desired, such as those described in Sec. VII-D. Larger steps of deduction are desired--corresponding, say, to maximal clashes.[19] One might conjecture that a researcher developing a new approach to question answering would do well to borrow from the store of resolution and other well-developed methods such as GPS and translate these methods into his approach.

F. Limitations and Improvements

In this section I shall discuss two limitations on the performance of QA3 and what can be done to improve performance. The first limitation is that the system is slow. The second is that it cannot solve difficult or highly specialized problems: it cannot do real game-playing (checkers, chess, etc.) requiring a great deal of analysis and special data structures; it cannot write long or complex programs; and it becomes inundated if supplied with too many possible relevant facts about its problem areas.

1. Speed

What do we mean by saying the system is too slow? We mean that on some problems the time required to answer the question is large, even though the proof strategy is well suited to the problem, the representation is the desired representation, and a theorem-prover seems to be a suitable problem-solving mechanism. On such problems, an examination of the program's internal operations indicates that the number and type of LISP operations being done on a typical problem is quite reasonable. The easiest questions, such as the chemistry questions, take several seconds. The particular Monkey and Bananas problem formulation given in Sec. VI-C requires one minute and fourteen seconds for a proof. These times are all console (real) time, not CPU time, since QA3 is running under a time-sharing system--the SDS 940.

The major cause of this slowness is the computer system in which QA3 is programmed. The program is written in a version of LISP implemented by Bolt, Beranek, and Newman for the SDS 940. The 940 has only 16K 24-bit words for the user, but LISP uses a paging system and drum to extend the effective memory size to 125K or more. The price one pays is that this version of LISP is very slow--e.g., a function call or a "cons" takes about 1.5 milliseconds. Since the 940 word is only 24 bits, there is only one LISP cell per computer word. The QA3 program occupies about 25K words. On large problems, the QA3 program, the LISP system, and free storage have required about 100K and more of storage. A detailed analysis of where time was going revealed that time was fairly evenly distributed among the many subprograms of QA3. The key algorithms such as subsumption and unification were programmed about as well as possible, using the known tricks within this version of LISP. Two possibilities for increasing speed are: (1) convert to machine language (or FORTRAN, etc.), and (2) switch to a new machine. Because one of the goals of this system has been to maintain flexibility, it would probably be a mistake to recode the program in machine language or FORTRAN. The flexibility of the LISP language has been very valuable for writing, debugging, modifying, and experimenting with the program. Fortunately a faster machine is available; the system is being transferred to a PDP-10, a computer with a larger word--36 bits--so that there are two LISP cells per computer word. The PDP-10 has a large core memory (up to 256K) and a fast LISP system. In summary, one severe limitation is the system in which QA3 is programmed, and the limitation can be overcome by a larger, faster system.

The LISP language has been adequate, but the proposed LISP-2 language, if it existed, would seem to be an excellent language in which to implement a new question-answering system.

2. <u>Difficult Questions</u>

Another kind of limitation is the inability of QA3 to handle a difficult question. In a typical case, a user will try a set of axioms and find that the search for a solution takes too long. By observing the search process, the user feels that the search is quite unreasonable for the problem. It may be the case that the program is not well suited to the problem (such as difficult game-playing). On the other hand, it may be the case that the program's performance can be improved. By observing how and why the search process is poor, the user often sees how simple changes will lead to the desired results. We list four such changes that are possible:

(1) <u>Representation Changes</u>. The Tower of Hanoi example illustrated how successively better representations led to easier solutions.

(2) <u>Strategy Changes</u>. The cube-finding problem illustrated how a measure of progress allowed a very efficient hill-climbing proof strategy.

(3) <u>Predicate Evaluation</u>. The predicate evaluation mechanism discussed in the cube-finding problem and in the Tower of Hanoi problem used special LISP programs to quickly trim poor nodes from the search tree and add fast computational ability to the theorem prover. The LISP program can of course use special data structures in its computations.

(4) Special Term-Matching During Unification. The special equality mechanism discussed in the cube-finding problem also decreased the number of clauses produced during the search.

A still better improvement is theoretically possible, though not yet practical: The user could ask QA3 tor write its own special program to solve the problem at hand. The LISP sort program problem illustrated how the theorem prover has the potential to go from a simulation mode to a program-writing mode in which the theorem prover can write a fast program that quickly solves the particular problem.

On the problems studies so far, the user tends to see good ideas for improvements faster than he is able to implement them. To help alleviate such a condition the present version of QA3 is gradually being modified to make each of the above methods easier to use.

3. A Framework for a General Machine Intelligence

One of the unstated but implicit goals of this research has been the development of a framework and a system in which to embed the many aspects of intelligence that will ultimately be necessary for a true machine intelligence. This goal has not of course been reached, but some light has been shed, and some directions for the future are clearer now.

Many possibly important aspects of machine intelligence have been discussed in detail herein. One such ability is program writing in the system's own language. Automatic program writing will facilitate effective self-modification and will allow automatic specialization. By specialization I mean the ability to automatically improve performance on a particular task by creating better and better programs for such tasks. The key to this ability is the capacity for describing and "understanding" the semantics of the programs. The rest of this task is to develop good methods, constructs, systems, etc. for efficient automatic programming. As an example of such a process, initially the machine will have a set of rules that describe a process, such as the rules for describing a cube. In a slow "interpretive" mode the machine can deal with these rules to recognize and describe a cube. When such a process is deemed sufficiently important, the machine will create a special program for recognizing a cube. If no further modification of this program is necessary, then the "semantics" of the internal operation of this special program will not be saved (perhaps analogous to an "unconscious" stored subroutine). If the program is to be modified or if subprograms are to be used later in other operations, the semantics of its internal operation will be saved to enable such a process.

Another important ability is the communication of information among problem-solving subsystems. Specific problem-solving subroutines cannot operate effectively by themselves, especially in changing environments and changing requirements. For example, to reach a given goal the machine may need to first recognize an object. The recognition of the object requires moving the machine to another position. The recognition process might integrate visual information such as texture, outline, and color with temporal information, (It's afternoon), contextual information (such as "I know there is an x somewhere in this room and it's not anywhere else, so this may well be an x").

Such integration of types of information requires a versatile and clean interface for the many subprograms. Each subroutine must be able to request additional information from any other subsystem. Likewise any subsystem must be able to send information such as answers to requests or other useful but unrequested data to other subsystems. Such an attempt was made in QA2 and QA3 in that in various applications the "theorem proven" could request and accept needed information from LISP, MEMORY, sensors, teletype, FORTRAN, etc.

Although QA2 and QA3 possessed rudimentary abilities of the kind described, they were not really adequate. The system organization was not sufficiently clean and versatile to allow a multitude of intercommunications and diverse problem solvers to effectively cooperate in achieving their goals on difficult problems.

The next system being designed will hopefully come closer to this ideal and also overcome the limitations mentioned in Sec. VIII-F-2, above.

4. Next System

In addition to the modifications being made to QA3, Robert A. Yates and I are designing an entirely new system, called QA4. The specification of the new system is not yet complete, so that we cannot yet say exactly what it will consist of, but several features now being developed will probably be part of QA4: The new system will use a higher-order-logic language; it will include a strategy language for describing storage and retrieval operations, proof-finding strategies, and general problem-solving strategies; and it will include special primitive set operations and special internal representations for finite sets.

The design goals of the system include greater flexibility than QA3, more usable self-descriptive capabilities, more usable automatic program-writing capabilities, ease of memory reorganization, ease of changing strategies, ease of changing representation, ease of changing inference mechanisms, and greater ability to specialize the system for hard-problem domains. The system will be more semantically oriented and less syntactically oriented than QA3. The system is intended to approach more closely the goal of the advice taker--i.e., it will be able to take more advice about its performance but will require less knowledge on the part of the user about its internal operations and representations. Such a system is of course difficult to design, but preliminary results are promising.

G. Problems for Research

We summarize here several broad, important research problems worthy of further work. Good solutions to these problems would contribute to the field of artificial intelligence.

1. Automatic Representation Changes

An important problem is that of creating a system that can automatically find substantial improvements in its representations of information. In a paper illustrating the importance of representation changes, Amarel[61] discusses seven successively better representations for the missionary and cannibals puzzle. With each improved representation, the problem becomes easier. Amarel also indicates the factors that make each change possible. Can such a process be automated? Can a theorem-proving system be made to examine its axioms and revise them to yield better but logically equivalent axioms?

2. Automatic Strategy Changes

An important problem is devising a system that can automatically find substantial improvements in its problem-solving strategies. Can a theorem prover be made to observe its axioms and performance and then find differences,[41] metrics, indicators of relevance, or other means of successfully guiding search and selecting strategies? Can a theorem prover be made to construct new strategies and/or prove new strategies to be better under particular conditions?

3. Automatic Programming

Automatic program writing seems to be a field of great importance in itself and especially for artificial intelligence. Much research today requires constant reprogramming. The self-modifying machines of the future might well use automatic program-writing facilities. The work on automatic program writing reported here and elsewhere is just a small beginning.

4. Answer Construction

The concept of answer construction is worthy of further study. Under what conditions and how can one find a "better" answer? What is a best answer? In a constructive proof, the answer clause contains a partially-constructed answer. Can this answer be used to guide the proof search or provide a "meaning" for a step in the proof?

5. Better Automatic Theorem Provers

Better automatic theorem provers lead to better question-answering systems and better problem-solving systems. In addition to the need for better theorem-proving formalisms there is much room for improvement within the resolution formalism.

One of the most important requirements for improving theorem provers is that of finding better proof strategies. It seems likely that no one fixed strategy will be best for all problems, so the key is finding flexible and suitable strategies. An important consideration in developing a new strategy is that the strategy should avoid redundancy. This can be done by two means: (1) avoid the creation of new but unnecessary inferences, and (2) create new inferences but eliminate unnecessary ones. It seems especially difficult to change proof strategies and still avoid the creation of unnecessary inferences. It is easier to change proof strategies and eliminate unnecessarily created inferences, although such a system will usually be less efficient. The subsumption algorithm provides a quite general means of eliminating unnecessarily created clauses. Improvements of the subsumption algorithm would be quite worthwhile.

Also important to resolution theorem proving is the development of efficient techniques for treating the equality relation, techniques for treating finite sets, and techniques for enumeration and testing of elements of sets. An important part of each of those problems is that of providing good strategies that tell us when to employ these techniques.

6. Undertaking More Realistic Applications

Increases in the level of realism and difficulty of an application of a question-answering system can lead to new problems and force new solutions. One important question-answering application would be the use of a very large, interactive data base where difficult questions are asked. Another important and difficult application is one where the system must interact with the real world through sensors and effectors, such as the SRI robot project.

7. Comparison of Methods

An important and difficult problem is that of comparing and evaluating the known approaches to question answering and problem solving. What are the domains of applicability of each of these techniques? Do any of the known systems provide the right framework in which to embed a general intelligence?

Appendix A

DESCRIPTION OF RESEARCH PROJECT

The plan for this research has been to design, implement, experiment with, and evaluate an evolving series of question-answering systems (in the form of computer programs). The research has been carried out under the supervision of Dr. Bertram Raphael.

The first step taken was to choose some simple subject areas. These first subjects included part-whole relationships, set memberships, set inclusions, spatial relations, family relations, and other relatively simple-to-formalize subjects.

QA1 was the first system implemented. This system was described in detail, first in Ref. 62, and later in a revised and published version of the same paper.[1] It was largely an attempt to improve on the SIR system of Raphael. The major advantage of QA1 over SIR lies in the ability of QA1 to hold in its list-structured memory logical statements about how various kinds of facts might interact. Thus QA1 does not require as many separate _ad hoc_ question-answering routines as did SIR. The data representation and memory organization of QA1 were adequate but the deduction techniques required improvement, so the control language and logical deduction programs of QA1 were left in rather rough form. To progress further, a decision was reached to start anew, and to base the new work upon relevant research in the field of automatic theorem proving.

The next version, QA2 (also described in Refs. 62 and 1), thus used first-order logic and an automatic theorem prover applying J. A. Robinson's resolution techniques. First, it was necessary to devise ways in which a pure theorem prover could be extended to a question-answering system. QA2 was then implemented; it was successful on all the simple problems that had been selected. The next step was to formalize more difficult subject areas that are basically processes involving changes of state, including writing computer programs in LISP, describing the actions of a robot, and theorem proving itself. These harder problems also served as goals for the next question-answering system, QA3. QA3 is conceptually similar to QA2, using first-order logic and theorem proving by resolution, but QA3 is more sophisticated, has more frills, and is a much more efficient program. The goal of the project was not to study and improve theorem provers as such, but to a certain extent this has been necessary in order to use the latest theorem-proving techniques to solve hard problems.

QA1 was programmed in LISP on the Q32 computer of Systems Development Corporation in Santa Monica. QA2 was also written in LISP on the Q32 computer, and was then transferred to the SDS 940 of the Artificial Intelligence Laboratory of Stanford Research Institute. The slowness of QA2 on the 940 helped provide impetus for seeking efficiency in a new system. Thus QA3 was programmed on the SDS 940. The author programmed QA1. For the programming effort on QA2 and QA3, Bob Yates joined the author, providing a considerable contribution.

Appendix B

THE MONKEY AND BANANAS PROOF

The axioms for the Monkey and Bananas problem are listed below, followed by the proof. The term SK24(S,P2,P1,B) that first appears in clause 16 of the proof is a Skolem function generated by the elimination of (∀x) in the conversion of axiom MB4 to quantifier-free clause form. (One may think of it as the object that is not at place P2 in state S.)

```
LIST MONKEY
MB1  (MOVABLE BOX)
MB2  (FA(X)(NOT(AT X UNDER-BANANAS SØ)))
MB3  (AT BOX PLACEB SØ)
MB4  (FA(B P1 P2 S)(IF(AND(AT B P1 S)(MOVABLE B)(FA(X)(NOT(AT X P2 S))))(AND
     (AT MONKEY P2(MOVE MONKEY B P2 S))(AT B P2(MOVE MONKEY B P2 S)))))
MB5  (FA(S)(CLIMBABLE MONKEY BOX S))
MB6  (FA(M P B S)(IF(AND(AT B P S)(CLIMBABLE M B S))(AND(AT B P(CLIMB
     M B S))(ON M B(CLIMB M B S)))))
MB7  (FA(S)(IF(AND(AT BOX UNDER-BANANAS S)(ON MONKEY BOX S))(REACHABLE
     MONKEY BANANAS S)))
MB8  (FA(M B S)(IF(REACHABLE M B S)(HAS M B(REACH M B S))))
DONE

Q    (EX(S)(HAS MONKEY BANANAS S))
A    YES, S = REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,MOVE(MONKEY,BOX,
     UNDER-BANANAS,SØ)))

PROOF
1    -AT(X,UNDER-BANANAS,SØ)                                AXIOM
2    AT(BOX,PLACEB,SØ)                                      AXIOM
3    CLIMBABLE(MONKEY,BOX,S)                                AXIOM
4    -HAS(MONKEY,BANANAS,S)                                 NEG OF THM
          ANSWER(S)
5    HAS(M,B,REACH(M,B,S))  -REACHABLE(M,B,S)               AXIOM
6    -REACHABLE(MONKEY,BANANAS,S)                           FROM 4,5
          ANSWER(REACH(MONKEY,BANANAS,S))
```

```
7     REACHABLE(MONKEY,BANANAS,S)  -AT(BOX,UNDER-BANANAS,S)     AXIOM
      -ON(MONKEY,BOX,S)

8     -AT(BOX,UNDER-BANANAS,S)  -ON(MONKEY,BOX,S)               FROM 6,7

           ANSWER(REACH(MONKEY,BANANAS,S))

9     ON(M,B,CLIMB(M,B,S))  -AT(B,P,S)  -CLIMBABLE(M,B,S)       AXIOM

10    -AT(BOX,UNDER-BANANAS,CLIMB(MONKEY,BOX,S))                FROM 8,9
      -AT(BOX,P,S)  -CLIMBABLE(MONKEY,BOX,S)

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))

11    -AT(BOX,UNDER-BANANAS,CLIMB(MONKEY,BOX,S))                FROM 3,1Ø
      -AT(BOX,P,S)

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))

12    AT(B,P,CLIMB(M,B,S))  -AT(B,P,S)  -CLIMBABLE(M,B,S)       AXIOM

13    -AT(BOX,XX1,S)  -AT(BOX,UNDER-BANANAS,S)                  FROM 11,12
      -CLIMBABLE(MONKEY,BOX,S)

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))

14    -AT(BOX,XX1,S)  -AT(BOX,UNDER-BANANAS,S)                  FROM 3,13

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))

15    -AT(BOX,UNDER-BANANAS,X)                                  FACTOR 14

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,S)))

16    AT(B,P2,MOVE(MONKEY,B,P2,S))  -MOVABLE(B)  -AT(B,P1,S)    AXIOM
      AT(SK24(S,P2,P1,B),P2,S)

17    -MOVABLE(BOX)  -AT(BOX,P1,S)  AT(SK24(S,UNDER-BANANAS,    FROM 15,16
      P1,BOX),UNDER-BANANAS,S)

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,
           MOVE(MONKEY,BOX,UNDER-BANANAS,S))))

18    -MOVABLE(BOX)  AT(SK24(SØ,UNDER-BANANAS,PLACEB,BOX),      FROM 2,17
      UNDER-BANANAS,SØ)

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,
           MOVE(MONKEY,BOX,UNDER-BANANAS,SØ))))

19    -MOVABLE(BOX)                                             FROM 1,18

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,
           MOVE(MONKEY,BOX,UNDER-BANANAS,SØ))))

20    MOVABLE(BOX)                                              AXIOM

21    CONTRADICTION                                             FROM 19,2Ø

           ANSWER(REACH(MONKEY,BANANAS,CLIMB(MONKEY,BOX,
           MOVE(MONKEY,BOX,UNDER-BANANAS,SØ))))

11    CLAUSES LEFT
28    CLAUSES GENERATED
22    CLAUSES ENTERED
27    RESOLUTIONS OUT OF 91 TRIES
SUBSUMED 23 TIMES OUT OF 179 TRIES
FACTORED 1 TIMES OUT OF 25 TRIES
```

Appendix C

THE SORT PROOF

The following axiomatization[*] is shorter than that given in Sec. VII-B. This axiomatization results in a proof that creates a sort program.

These axioms use the SAME predicate instead of the ON predicate. SAME(x,y) holds if and only if the lists x and y contain the same elements (not necessarily in the same order). SAME can be defined in terms of ON as:

U0. $(\forall x,y)[\mathrm{SAME}(x,y) \equiv (\forall z)[\mathrm{ON}(z,x) \equiv \mathrm{ON}(z,y)]]$,

although this definition is not needed for the sort proof. The only information needed about SAME is the definition of R in terms of SAME. Similarly, we do not need the definition of SD, just the definition of R in terms of SD and the description of merge in terms of SD. First, the predicate R is defined in terms of SAME and SD:

U1. $(\forall x,y)[R(x,y) \equiv [\mathrm{SAME}(x,y) \wedge \mathrm{SD}(y)]]$.

Next, the merge function is described by Axioms U2 and U3:

U2. $(\forall x,y)[\mathrm{SD}(y) \supset \mathrm{SD}(\mathrm{merge}(x,y))]$.

Axiom 2 states that if the input list to merge is sorted, then the output is sorted.

U3. $(\forall u,x,y)[[\mathrm{SD}(y) \wedge \mathrm{SAME}(x,y)] \supset \mathrm{SAME}(\mathrm{cons}(u,x),\mathrm{merge}(u,y))]$.

Axiom U3 may be thought of as follows: Let y be the sorted input list to merge(u,y). The new element to be added is u. The set of elements in the list cons(u,x) is just the elements of x plus the element u. If y and x have the same elements, then the lists merge(u,y) and cons(u,x) have the same elements.

---

[*]The axiomatization is based on a suggestion by R. Yates.

We also state the terminating condition on R, namely that the sorted version of the empty list is the empty list itself,

U4. $(\forall x)[x = \text{nil} \supset R(x,\text{nil})]$.

(Axiom U4 could be derived from U0 and the definition of R given in Sec. VII-B, but we shall take it as an axiom to simplify the proof.) One of the fundamental LISP axioms (comparable to L3 in Sec. VII-B) will be used:

U5. $(\forall x)[x \neq \text{nil} \supset x = \text{cons}(\text{car}(x),\text{cdr}(x))]$.

Since we are assuming a domain of lists for U5, x is either the empty list "nil" or else a non-empty list. In case it is a non-empty list, we say that x is equal to cons(car(x),cdr(x)).

We will use an equality axiom to specify the substitutivity property of the equality relation. The particular one needed is

U6. $(\forall x,y,z)[[x = y \wedge \text{SAME}(y,z)] \supset \text{SAME}(x,z)]$,

which allows us to substitute equal terms for equal terms in the first argument of the SAME predicate. No other equality axioms are used.

The machine form of the axioms and the proof is given below. A discussion of the rather complicated proof follows the listing of the proof.

The only axioms used in the proof are listed below in the QA3 input form:

```
U1.   (FA(X Y)(IFF(R X Y)(AND(SAME X Y)(SD Y))))
U2.   (FA(X Y)(IF(SD Y)(SD(MERGE X Y))))
U3.   (FA(X Y U)(IF(AND(SD Y)(SAME X Y))(SAME(CONS U X)(MERGE U Y))))
U4.   (FA(X)(IF(EQUAL X NIL)(R X NIL)))
U5.   (FA(X)(IF(NOT(EQUAL X NIL))(EQUAL X(CONS(CAR X)(CDR X)))))
U6.   (FA(X Y Z)(IF(AND(EQUAL X Y)(SAME Y Z))(SAME X Z)))   .
```

The question, answer, and proof are:

```
Q     (FA(X)(EX(Y)(AND(IF(EQUAL X NIL)(R X Y))(IF(AND(NOT(EQUAL X NIL))
         (R(CDR X)(SORT(CDR X))))(R X Y))))

A     YES, Y = COND(X,MERGE(CAR(X),SORT(CDR(X))),NIL)

UNWIND

SUMMARY

 1    -R(SK62,Y)                                                 NEG OF THM
           ANSWER(Y)
 2    R(X,NIL)  -EQUAL(X,NIL)                                    AXIOM
 3    -EQUAL(SK62,NIL)                                           FROM 1,2
           ANSWER(NIL)
 4    EQUAL(X,CONS(CAR(X),CDR(X)))  EQUAL(X,NIL)                 AXIOM
 5    SD(MERGE(X,Y))  -SD(Y)                                     AXIOM
 6    R(X,Y)  -SAME(X,Y)  -SD(Y)                                 AXIOM
 7    -SAME(SK62,Y)  -SD(Y)                                      FROM 1,6
           ANSWER(Y)
 8    -SD(Y)  -SAME(SK62,MERGE(X,Y))                             FROM 5,7
           ANSWER(MERGE(X,Y))
 9    SD(Y)  -R(X,Y)                                             AXIOM
10    -SAME(SK62,MERGE(X,Y))  -R(XX16,Y)                         FROM 8,9
           ANSWER(MERGE(X,Y))
11    EQUAL(SK62,NIL)  R(CDR(SK62),SORT(CDR(SK62)))              NEG OF THM
           ANSWER(XX1)
12    -SAME(SK62,MERGE(X,SORT(CDR(SK62))))  EQUAL(SK62,NIL)      FROM 10,11
           ANSWER(MERGE(X,SORT(CDR(SK62))))
13    -SAME(SK62,MERGE(X,SORT(CDR(SK62))))                       FROM 3,12
           ANSWER(COND(SK62,MERGE(X,SORT(CDR(SK62))),NIL))
14    SAME(X,Z)  -EQUAL(X,Y)  -SAME(Y,Z)                         AXIOM
15    -EQUAL(SK62,Y)  -SAME(Y,MERGE(X,SORT(CDR(SK62))))          FROM 13,14
           ANSWER(COND(SK62,MERGE(X,SORT(CDR(SK62))),NIL))
16    EQUAL(SK62,NIL)  -SAME(CONS(CAR(SK62),CDR(SK62)),
                           MERGE(XX117,SORT(CDR(SK62))))         FROM 4,15
           ANSWER(COND(SK62,MERGE(XX117,SORT(CDR(SK62))),NIL))
```

```
17   -SAME(CONS(CAR(SK62),CDR(SK62)),MERGE(XX117,
        SORT(CDR(SK62))))                                   FROM 3,16

          ANSWER(COND(SK62,MERGE(XX117,SORT(CDR(SK62))),NIL))

18   R(CDR(SK62),SORT(CDR(SK62)))                           FROM 3,11

          ANSWER(COND(SK62,XX1,NIL))

19   SD(SORT(CDR(SK62)))                                    FROM 18,9

          ANSWER(COND(SK62,XX1,NIL))

20   SAME(CONS(U,X),MERGE(U,Y))  -SD(Y)  -SAME(X,Y)         AXIOM

21   SAME(CONS(U,X),MERGE(U,SORT(CDR(SK62))))
     -SAME(X,SORT(CDR(SK62)))                               FROM 19,20

          ANSWER(COND(SK62,XX1,NIL))

22   -SAME(CDR(SK62),SORT(CDR(SK62)))                       FROM 17,21

          ANSWER(COND(SK62,MERGE(CAR(SK62),SORT(CDR(SK62))),NIL))

23   SAME(X,Y)  -R(X,Y)                                     AXIOM

24   -R(CDR(SK62),SORT(CDR(SK62)))                          FROM 22,23

          ANSWER(COND(SK62,MERGE(CAR(SK62),SORT(CDR(SK62))),NIL))

25   EQUAL(SK62,NIL)                                        FROM 24,11

          ANSWER(COND(SK62,MERGE(CAR(SK62),SORT(CDR(SK62))),NIL))

26   CONTRADICTION                                          FROM 3,25

          ANSWER(COND(SK62,MERGE(CAR(SK62),SORT(CDR(SK62))),NIL))

115 CLAUSES LEFT

286 CLAUSES GENERATED

115 CLAUSES ENTERED

552 RESOLUTIONS OUT OF 2403 TRIES

SUBSUMED 220 TIMES OUT OF 19059 TRIES

FACTORED 170 TIMES OUT OF 393 TRIES
```

The strategy is somewhat "tuned" for this problem (and hopefully for other programming problems). A preference is given to clauses whose answers do not contain many nested occurrences of any one function. Clauses having the answer "nil" are not preferred. The preferences are handled by increasing the level of nonpreferred clauses beyond their normal level.

Often answer simplification is possible. For example, the function cond(x,cond(x,y,w),z) is equivalent to the shorter function cond(x,y,z). QA3 can automatically make this simplification (as shown in Clauses 17 and 26).

As discussed in Sec. VII-B, the "cond" axioms (L6 and L7) are not used explicitly. Instead, a special mechanism simulates the use of these axioms. To see this, observe that the answer in Clause 13 is a conditional answer constructed from the two answers in Clauses 3 and 12. However, this operation is equivalent to using the cond axioms. To see this, we give below a simple resolution derivation showing how two clauses having two different answers can be combined by a standard resolution proof.

Suppose the two clauses are:

A1. $a = \text{nil} \vee P \vee$ ANSWER(b)

A2. $a \neq \text{nil} \vee Q \vee$ ANSWER(c)

where P and Q represent arbitrary, possibly empty, disjunctions of literals. Note that A1 has the answer b and A2 has the answer c. A1 and A2 may be considered analogous, respectively, to Clauses 12 and 3 in the above proof, where "a" corresponds to SK62, "b" corresponds to merge(x,sort(cdr(x))), and "c" corresponds to nil. We will now derive A13 by a conventional, unabbreviated resolution proof. Clause A13 will be seen to have a single conditional answer, cond(a,b,c), and is analogous to Clause 13 of the above proof.

The clauses describing the conditional operation are:

A3. $x = \text{nil} \vee \text{cond}(x,y,z) = y$

A4. $x \neq \text{nil} \vee \text{cond}(x,y,z) = z$.

An axiom describing the substitutivity of equality in the ANSWER predicate is as follows:

A5. $y \neq x \vee$
$\sim$ANSWER(x) $\vee$ ANSWER(y) .

The answers from A1 and A2 can be combined by the following sequence of resolutions:

A6.	$y \neq b \vee a = \text{nil} \vee P \vee$ ANSWER(y)	From A1,A5
A7.	$y \neq c \vee a \neq \text{nil} \vee Q \vee$ ANSWER(y)	From A2,A5
A8.	$a = \text{nil} \vee x = \text{nil} \vee P \vee$ ANSWER(cond(x,b,z))	From A3,A6
A9.	$a \neq \text{nil} \vee x \neq \text{nil} \vee Q \vee$ ANSWER(cond(x,y,c))	From A4,A7
A10.	$a = \text{nil} \vee P \vee$ ANSWER(cond(a,b,z))	Factor A8
A11.	$a \neq \text{nil} \vee Q \vee$ ANSWER(cond(a,y,c))	Factor A9
A12.	$P \vee Q \vee$ ANSWER(cond(a,b,z)) $\vee$ ANSWER(cond(a,y,c))	From A10,A11
A13.	$P \vee Q \vee$ ANSWER(cond(a,b,c))	Factor A12

The special mechanism for combining answers speeds up this process and shortens the proof.

Appendix D

THE CUBE PROOF

The following axioms represent picture data:

```
(QUADRILATERAL (P A B C D))
(LINE (P A B))
(LINE (P B C))
(LINE (P C D))
(LINE (P D A))
(LINE (P A E))
(LINE (P F G))
(LINE (P F B))
(LINE (P G C))
(LINE (P E F))
```

The following two axioms define a quadrilateral and a cube, respectively.

```
(FA (X Y Z W) (IF (AND (LINE (P X Y)) (LINE (P Y Z)) (LINE (P Z W))
(LINE (P W X)) (DISTINCT (L X Y Z W))) (QUADRILATERAL (P X Y Z W))))

(FA (X1 X2 X3 X4 X5 X6 X7) (IF (AND (QUADRILATERAL (P X1 X2 X3 X4))
(QUADRILATERAL (P X6 X2 X1 X5)) (QUADRILATERAL (P X3 X2 X6 X7)) (DISTINCT
(L X1 X2 X3 X4 X5 X6 X7))) (CUBE (P (P X1 X2 X3 X4) (P X6 X2 X1 X5)
(P X3 X2 X6 X7)))))
```

The question, answer, and proof are:

```
Q (EX(X)(CUBE X))

    YES, X = P(P(F,B,A,E),P(C,B,F,G),P(A,B,C,D))

UNWIND

SUMMARY  01/07/69 1545:07

1     LINE(P(F,G))                                           AXIOM
2     LINE(P(F,B))                                           AXIOM
3     LINE(P(B,C))                                           AXIOM
4     LINE(P(E,F))                                           AXIOM
5     LINE(P(A,E))                                           AXIOM
6     LINE(P(A,B))                                           AXIOM
7     QUADRILATERAL(P(A,B,C,D))                              AXIOM
```

```
8     -CUBE(X)                                                NEG OF THM
          ANSWER(X)

9     CUBE(P(P(X1,X2,X3,X4),P(X6,X2,X1,X5),P(X3,X2,X6,X7)))   AXIOM
      -QUADRILATERAL(P(X3,X2,X6,X7))
      -QUADRILATERAL(P(X1,X2,X3,X4))
      -QUADRILATERAL(P(X6,X2,X1,X5))
      -DISTINCT(L(X1,X2,X3,X4,X5,X6,X7))

10    -QUADRILATERAL(P(X3,X2,X6,X7))                          FROM 8,9
      -QUADRILATERAL(P(X1,X2,X3,X4))
      -QUADRILATERAL(P(X6,X2,X1,X5))
      -DISTINCT(L(X1,X2,X3,X4,X5,X6,X7))

          ANSWER(P(P(X1,X2,X3,X4),P(X6,X2,X1,X5),P(X3,X2,X6,X7)))

11    -QUADRILATERAL(P(X1,B,A,X4))                            FROM 7,10
      -QUADRILATERAL(P(C,B,X1,X5))
      -DISTINCT(L(X1,B,A,X4,X5,C,D))

          ANSWER(P(P(X1,B,A,X4),P(C,B,X1,X5),P(A,B,C,D)))

12    QUADRILATERAL(P(X,Y,Z,W))  -LINE(P(W,X))                AXIOM
      -LINE(P(Y,Z))  -LINE(P(X,Y))
      -LINE(P(Z,W))  -DISTINCT(L(X,Y,Z,W))

13    -QUADRILATERAL(P(C,B,X,X5))                             FROM 11,12
      -DISTINCT(L(X,B,A,W,X5,C,D))
      -LINE(P(W,X))   -LINE(P(B,A))
      -LINE(P(X,B))   -LINE(P(A,W))
      -DISTINCT(L(X,B,A,W))

          ANSWER(P(P(X,B,A,W),P(C,B,X,X5),P(A,B,C,D)))

14    -QUADRILATERAL(P(C,B,X,X5))                             FROM 6,13
      -LINE(P(W,X))  -LINE(P(X,B))
      -LINE(P(A,W))  -DISTINCT(L(X,B,A,W))

          ANSWER(P(P(X,B,A,W),P(C,B,X,X5),P(A,B,C,D)))

15    -QUADRILATERAL(P(C,B,X,X5))                             FROM 5,14
      -DISTINCT(L(X,B,A,E,X5,C,D))
      -LINE(P(E,X))  -LINE(P(X,B))
      -DISTINCT(L(X,B,A,E))

          ANSWER(P(P(X,B,A,E),P(C,B,X,X5),P(A,B,C,D)))

16    -QUADRILATERAL(P(C,B,F,X5))                             FROM 2,15
      -DISTINCT(L(F,B,A,E,X5,C,D))
      -LINE(P(E,F))

          ANSWER(P(P(F,B,A,E),P(C,B,F,X5),P(A,B,C,D)))

17    -QUADRILATERAL(P(C,B,F,X5))                             FROM 4,16
      -DISTINCT(L(F,B,A,E,X5,C,D))

          ANSWER(P(P(F,B,A,E),P(C,B,F,X5),P(A,B,C,D)))
```

```
18    -DISTINCT(L(F,B,A,E,W,C,D))                             FROM 17,12
      -LINE(P(W,C))  -LINE(P(B,F))
      -LINE(P(C,B))  -LINE(P(F,W))
      -DISTINCT(L(C,B,F,W))

          ANSWER(P(P(F,B,A,E),P(C,B,F,W),P(A,B,C,D)))

19    -DISTINCT(L(F,B,A,E,W,C,D))                             FROM 3,18
      -LINE(P(W,C))  -LINE(P(B,F))
      -LINE(P(F,W))  -DISTINCT(L(C,B,F,W))

          ANSWER(P(P(F,B,A,E),P(C,B,F,W),P(A,B,C,D)))

20    -DISTINCT(L(F,B,A,E,W,C,D))                             FROM 2,19
      -LINE(P(W,C))  -LINE(P(F,W))
      -DISTINCT(L(C,B,F,W))

          ANSWER(P(P(F,B,A,E),P(C,B,F,W),P(A,B,C,D)))

21    -LINE(P(G,C))                                           FROM 1,2Ø

          ANSWER(P(P(F,B,A,E),P(C,B,F,G),P(A,B,C,D)))

22    LINE(P(G,C))                                            AXIOM

23    CONTRADICTION                                           FROM 21,22

          ANSWER(P(P(F,B,A,E),P(C,B,F,G),P(A,B,C,D)))

18    CLAUSES LEFT

23    CLAUSES GENERATED

18    CLAUSES ENTERED

23    RESOLUTIONS OUT OF 83 TRIES

      SUBSUMED 7 TIMES OUT OF 252 TRIES

      FACTORED Ø TIMES OUT OF Ø TRIES
```

REFERENCES

1. C. Green and B. Raphael, "The Use of Theorem-Proving Techniques in Question-Answering Systems," Proc. 23rd Nat. Conf. ACM (Thompson Book Company, Washington, D.C., 1968).

2. C. Green, "Theorem Proving by Resolution as a Basis for Question-Answering Systems," Machine Intelligence 4, D. Michie and B. Meltzer, Eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).

3. C. Green, "Application of Theorem Proving to Problem Solving," Proc. International Joint Conference on Artificial Intelligence, D. E. Walker and L. M. Norton, Eds., Washington, D.C., 7-9 May 1969 (to be published).

4. B. Raphael, "A Computer Program Which 'Understands'," Proc. FJCC pp. 577-589 (1964).

5. B. Raphael, "SIR, A Computer Program for Semantic Information Retrieval," in Semantic Information Processing, M. Minsky, Ed. (MIT Press, Cambridge, Massachusetts, and London, England, 1968).

6. C. H. Kellogg, "A Natural Language Compiler for On-Line Data Management," AFIPS Conference Proceedings, Vol. 33, pp. 473-493 (Thompson Book Co., Washington, D.C., 1968).

7. L. S. Coles, "An On-Line Question-Answering System with Natural Language and Pictorial Input," Proc. Nat. Conf. ACM (1968).

8. L. S. Coles, "Talking with a Robot in English," Proc. International Joint Conference on Artificial Intelligence, D. E. Walker and L. M. Norton, Eds., Washington, D.C., 7-9 May 1969 (to be published).

9. B. F. Green, Jr., A. K. Wolf, C. Chomsky, and K. Laughery, "BASEBALL: An Automatic Question Answerer," Computers and Thought, E. A. Feigenbaum and J. Feldman, Eds. (McGraw-Hill Book Company, Inc., 1963).

10. R. K. Lindsay, "Inferential Memory as the Basis of Machines Which Understand Natural Language," Computers and Thought, E. A. Feigenbaum and J. Feldman, Eds. (McGraw-Hill Book Company, Inc., 1963).

11. K. M. Colby and D. C. Smith, "Dialogues Between Humans and an Artificial Belief System," Proc. International Joint Conference on Artificial Intelligence, D. E. Walker and L. M. Norton, Eds., Washington, D.C., 7-9 May 1969 (to be published).

12. J. McCarthy, "Programs with Common Sense," Proc. Symposium on Mechanization of Thought Processes (Her Majesty's Stationery Office, London, England, 1959).

13. J. McCarthy, "Situations, Actions, and Causal Laws," Memo No. 2, Stanford Artificial Intelligence Project, Stanford University, Stanford, California (July 1963).

14. R. F. Simmons, "Answering English Questions by Computer: A Survey," COMM. ACM, Vol. 8, No. 1 (January 1965).

15. R. F. Simmons, "Natural Language Question Answering Systems: 1969," TNN-87, University of Texas Computation Center, Austin (January 1969).

16. W. A. Wood, "Semantics for a Question-Answering System," Ph.D. Thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts (August 1967). Also Report NSF-19, Harvard Computation Laboratory.

17. D. G. Bobrow, J. B. Fraser, M. R. Quillian, "Automated Language Processing," Annual Review of Information Science and Technology, (Interscience New York, 1967, Vol. 2).

18. J. A. Robinson, "The Present State of Mechanical Theorem Proving," to appear in Proceedings the Fourth Systems Symposium, Cleveland, Ohio, November 19-20, 1968.

19. J. A. Robinson, "A Review of Automatic Theorem-Proving," Proc. Symp. Appl. Math., Vol. 19, Amer. Math. Soc., Providence, R.I. (1967).

20. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," J. ACM, Vol. 12, No. 1, pp. 23-41 (January 1965).

21. J. A. Robinson, "The Generalized Resolution Principle," Machine Intelligence 4, D. Michie and B. Meltzer, Eds., (Edinburgh University Press, Edinburgh, Scotland, (1968).

22. R. J. Waldinger and R. C. T. Lee, "PROW: A Step Toward Automatic Program Writing," Proceedings of the International Joint Conference on Artificial Intelligence, D. E. Walker and L. M. Norton, Eds., May 7-9, 1969, Washington, D.C. (to be published).

23. G. Sussman, Project MAC, MIT, Cambridge, Mass. (private communication).

24. R. Burstall, University of Edinburgh, Edinburgh, Scotland (private communication.)

25. J. L. Darlington, "Machine Methods for Proving Logical Arguments Expressed in English," Mech. Trans., Vol. 8, pp. 41-67 (June-October 1965).

26. J. L. Darlington, "Theorem Proving and Information Retrieval," Machine Intelligence 4, D. Michie and B. Meltzer, Eds., (Edinburgh University Press, Edinburgh, Scotland, 1969).

27. J. L. Darlington, "Theorem Provers as Question Answerers," Proceedings of the International Joint Conference on Artificial Intelligence, D. E. Walker and L. M. Norton, Eds., May 7-9, 1969, Washington, D.C. (to be published).

28. J. H. Chadwick, L. S. Coles, O. W. Whitby, B. Raphael, and J. H. Jones, "Medical Applications of Remote Electronic Browsing," Final Report to EDUCOM, University of Pittsburgh, (1969).

29. R. E. Kling, "Reasoning by Analogy with Application to Resolution Logic," to appear in Proceedings of the International Conference on Cybernetics, (1969).

30. N. J. Nilsson, "A Mobile Automaton: An Application of Artificial Intelligence Techniques," Proceedings of the International Joint Conference on Artificial Intelligence, D. E. Walker and L. M. Norton, Eds., May 7-9, 1969, Washington, D.C. (to be published).

31. J. R. Slagle, "Experiments with a Deductive, Question-Answering Program," Comm. ACM, Vol. 8, pp. 792-798 (December 1965).

32. D. C. Cooper, "Theorem Proving in Computers," Advances in Programming and Non-Numerical Computation, L. Fox, Ed., (Pergamon Press, 1966).

33. L. Wos, G. A. Robinson, and D. F. Carson, "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving," J. ACM, Vol. 12, No. 4, pp 536-541 (October 1965).

34. L. Wos, D. Carson, and G. Robinson, "The Unit Preference Strategy in Theorem Proving," Proc. AFIPS 1964 FJCC, Vol. 26, Pt. II, pp. 615-621 (Spartan Books, 1964).

35. L. Wos, G. A. Robinson, D. F. Carson, and L. Shalla, "The Concept of Demodulation in Theorem Proving," J. ACM, Vol. 14, No. 4, pp. 698-709 (October 1967).

36. J. R. Guard, F. C. Oglesby, J. H. Bennett, and L. G. Settle, "Semi-Automated Mathematics," J. ACM, Vol. 16, No. 1, pp. 49-62 (January 1969).

37. J. A. Robinson, "Heuristic and Complete Processes in the Mechanization of Theorem Proving," Systems and Computer Science, J. F. Hart and S. Takasu, Eds., pp. 116-124, (University of Toronto Press, 1967).

38. M. Davis, "Eliminating the Irrelevant from Mechanical Proofs," Proc. 15th Symp. in Appl. Math., Amer. Math. Soc., Providence, R.I., pp. 15-30 (1963).

39. W. S. Cooper, "Fact Retrieval and Deductive Question Answering Information Retrieval Systems," J. ACM, Vol. 11, pp. 117-137 (April 1964).

40. J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," Machine Intelligence 4, D. Michie and B. Meltzer, Eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).

41. G. Ernst, "Sufficient Conditions for the Success of GPS," Report No. SRC-68-17, Systems Research Center, Case Western Reserve University, Cleveland, Ohio (July 1968).

42. A. Hormann, "How a Computer System Can Learn," IEEE Spectrum (July 1964).

43. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, LISP 1.5 Programmer's Manual (The MIT Press, Cambridge, Massachusetts, 1962).

44. C. Weissman, LISP 1.5 Primer (Dickenson Publishing Company, Inc., Belmont, California, 1967).

45. L. Wos and G. Robinson, "Paramodulation and Set of Support," IRIA Symposium on Automatic Demonstration at Versailles, France, December 16-21, (proceedings to be published).

46. G. Robinson and L. Wos, "Paramodulation and Theorem-Proving in First-Order Theories with Equality," Machine Intelligence 4, B. Meltzer and D. Michie, Eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).

47. H. Simon, "Experiments with a Heuristic Compiler," J. ACM, Vol. 10, pp. 493-506 (October 1963).

48. R. W. Floyd, "The Verifying Compiler," Computer Science Research Review, Carnegie Mellon University (December 1967).

49. Z. Manna, "The Correctness of Programs," J. Computer and Systems Sciences, Vol. 3 (1969).

50. J. McCarthy, "Towards a Mathematical Science of Computation," Proceedings ICIP (North Holland Publishing Company, Amsterdam, 1962).

51. F. Black, A Deductive Question-Answering System, Harvard University Ph.D. Thesis (1964).

52. F. Safier, "The Mikado as an Advice Taker Problem," Memo, Stanford Artificial Intelligence Project, Stanford University (July 1963).

53. H. Simon, "Experiments with a Heuristic Compiler," J. ACM, Vol. 10, pp. 493-506 (October 1963).