

COMPUTER-ASSISTED INSTRUCTION: THE APPLICATION OF THEOREM-PROVING
TO ADAPTIVE RESPONSE ANALYSIS

by

Adele Goldberg

TECHNICAL REPORT NO. 203

May 25, 1973

PSYCHOLOGY AND EDUCATION SERIES

Reproduction in Whole or in Part Is Permitted for
Any Purpose of the United States Government

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

STANFORD UNIVERSITY

STANFORD, CALIFORNIA

ACKNOWLEDGMENTS

This dissertation is the result of several years full of help from many people. Dr. Patrick Suppes of Stanford University has been the central support in lending me both the resources of his research institute and his time for consultation and consolation.

I am indebted to Tobin Barrozo for his collaboration on my initial efforts with Chapters IV and VI, to Robert L. Smith for his patient reading of all the proofs, and to Maya Bar-Hillel, a trusted editor and friend. Especial thanks are due to David Levine, whose brain I often picked.

I appreciate the efforts extended for me by Dr. Roman Weil of the University of Chicago. Dr. Weil initially chaired my dissertation committee. The advice and encouragement of Dr. Victor Yngve will long be remembered. Thanks are also due to the other members of my dissertation committee: Robert L. Ashenurst, Joseph Goguen, and Bruce Kallick.

I am most grateful to Mrs. Dianne Kanerva for preparing this manuscript in its present form.

Support for this research was provided by National Science Foundation Grant NSFGJ-443 to Stanford University.

Faint, illegible text, possibly bleed-through from the reverse side of the page. The text is too light to transcribe accurately.

COMPUTER-ASSISTED INSTRUCTION: THE APPLICATION OF THEOREM-PROVING
TO ADAPTIVE RESPONSE ANALYSIS

Adele Goldberg

ABSTRACT

In a CAI system for teaching elementary mathematical logic, in order to provide tutorial advice similar to the suggestions a human tutor might give, a theorem-prover is employed as a proof-analyzer to generate appropriate dialogue with students who need help with a proof. It is shown that the heuristically programmed theorem-prover, by embodying techniques thought to be used by the students, constructs derivations for expressions in the elementary theory of Abelian groups within the constraints on rule usage imposed on these students. To demonstrate its capabilities, the theorem-prover was tested on a list of theorems and problems chosen arbitrarily from a standard curriculum in elementary algebra. The proof-analyzer mocks the adaptive behavior of a human tutor; it can determine relevant hints when a student requires help in completing a solution, and can encourage the students to discover diverse solution paths. The proof-analyzer is incorporated in a new instructional system for mathematical logic, an example of a "student-oriented" CAI system which emphasizes active participation of the students with the computer by providing methods by which a student can initially specify or extend the axiomatic theory he is studying, while still retaining the error analysis and teaching facilities of an interactive proof-checker.

1. The first part of the document discusses the importance of maintaining accurate records of all transactions.

2. It also emphasizes the need for regular audits to ensure the integrity of the financial data.

3. The document further outlines the procedures for handling discrepancies and resolving any issues that may arise.

4. Additionally, it provides guidelines for the proper use of funds and the allocation of resources.

5. The document also addresses the importance of transparency and accountability in all financial activities.

6. Furthermore, it discusses the role of the board of directors in overseeing the financial operations of the organization.

7. The document also highlights the need for ongoing communication and collaboration between all stakeholders.

8. Additionally, it provides information on the various financial reports that must be prepared and submitted.

9. The document also discusses the importance of staying up-to-date on changes in financial regulations and standards.

10. Finally, it concludes by reiterating the commitment to maintaining the highest standards of financial integrity.

11. The document is intended to serve as a guide for all employees and management in handling financial matters.

12. It is the responsibility of all individuals involved to ensure that these guidelines are followed at all times.

13. The document is subject to periodic review and updates as needed to reflect changes in the financial landscape.

14. For more information, please contact the Finance Department at [phone number] or [email address].

15. Thank you for your attention and cooperation in maintaining the financial health of our organization.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	1
ABSTRACT	ii
Chapter	
I. THE COMPUTER AS TUTOR	1
1. Humanizing the Mechanical Tutor	
2. A Tutor for Mathematical Logic	
3. Using a Theorem-Prover to Teach Theorem Proving	
4. A Generalized Instructional System for Elementary Mathematical Logic	
5. Research Design	
II. COMPUTER-ASSISTED INSTRUCTION	22
1. Background of CAI Research	
2. Mechanical Proof-Checkers	
2.1 Proof-Checkers with Immediate Feedback	
III. MECHANICAL THEOREM PROVING: A NEW APPLICATION	34
1. Background of Automatic Theorem Proving	
2. Existing Theorem-Proving Programs Are Not Appropriate as Proof-Analyzers	
3. Properties of an Appropriate Theorem-Prover	
IV. A THEOREM-PROVER FOR THE THEORY OF ABELIAN GROUPS	
1. The Language \mathcal{L}	
2. Some Preliminary Definitions	
3. The Rules of Inference	
4. Truth in \mathcal{L}	
5. Proof Methods	
6. Incompleteness and Decidability	
7. Some Heuristics Used by the Theorem-Prover	
V. THE THEOREM-PROVER AS PROOF-ANALYZER	122
1. Finding One or More Solutions	
1.1 The COLLECTOR	
1.2 An Example	
2. Completing Partial Solutions	
2.1 Information about $O+B=B$	
2.2 Comparison of the Theorem-Prover/Proof-Analyzer with the Stored-Hint Approach	
3. Lisa: A Case Study in Tutoring Elementary Algebra	
4. A Study of Experienced Human Tutors of Elementary Algebra	

Chapter

VI. A GENERALIZED INSTRUCTIONAL SYSTEM FOR TEACHING ELEMENTARY MATHEMATICAL LOGIC	200
1. Basic Features of the Instructional System	
2. Formal Description of the Instructional System	
2.1 The Vocabulary	
2.2 Proper Substitution: Instances of Axioms and Theorems	
2.3 The Command Language	
3. Error-Analysis for Commands--Syntax and Application	
4. Generating a Dialogue to Teach a Command	
VII. SUMMARY AND FINAL COMMENTS	253
REFERENCES	260
APPENDICES	
I. AXIOMS AND THEOREMS ON ADDITION	266
II. PROOFS FOR THEOREMS ON ADDITION	267
III. THE THEOREM-PROVING ALGORITHM	276
IV. MATERIAL FOR THE STUDY OF EXPERIENCED HUMAN TUTORS	282
V. SUMMARY OF THE COMMAND LANGUAGE PRESENTED IN CHAPTER VI	286

CHAPTER I

THE COMPUTER AS TUTOR

One of the main criticisms of computerized instruction alleges that it deprives the student of the dynamic and personalized interaction afforded him by an attentive human tutor. The intention of this dissertation is to answer this criticism by examining some of the features which distinguish the tutorial role and, in light of these features, studying the computer's ability to assume that role.

1. Humanizing the Mechanical Tutor

An attractive feature offered by a human tutor is his ability to provide a flexible environment in which the student can freely explore the material he is expected to understand. In such an environment the student can rehearse factual material, study relationships between theories, make up examples which clarify the ideas, and experiment with newly acquired skills. Through his guidance, the human tutor not only shows the student how to proceed, but is sensitive to his difficulties and misunderstandings as well. By analyzing the student's work, the tutor can tide him over rough spots, show him where he went wrong, and encourage him to explore promising paths.

An ideal tutor adjusts his teaching to the singular requirements of each learner. It is this notion of individualized attention, of shaping

instruction to the unique history of each individual, which is most frequently associated with the tutorial process. The tutor analyzes each student's work, and extracts information from each student's responses, especially when they are only partial or erroneous, so as to make decisions about how instruction should proceed. Thus the tutor provides an adaptive, or response-sensitive, teaching system.

If the computer is to simulate a human tutor, it must be such an adaptive system. The computer must make decisions relevant to the immediate needs of individual learners. Computer teaching systems which are nonadaptive, or response-insensitive, are static lesson drivers which present prestored questions, or analyze student responses only on the basis of comparisons with prestored answers. They do not consist of algorithms which enable an instructional system to develop dynamically; that is, they do not include routines which generate new problems, process unanticipated answers, analyze erroneous answers and find out what went wrong so as to comment on the error, and so on. The ability to program elaborate branching strategies has allowed utilization of some aspects of a student's performance in determining subsequent instruction. But the instruction is usually limited to a set of predetermined decision processes and is dependent on prestored remedial help sequences. Even when the designers of instructional programs manage to avoid passive multiple-choice or fill-in-the-blank response formats (developing, for example, flexible question-answering systems [Carbonell, 1970]), they tend to fall back on remedial branching schemes in order to provide help for the less skillful students.

Historically, computer-assisted (CAI) teaching systems, designed either for use in an actual classroom or for studying conceptual problems of an educational nature, have claimed the ability to adapt to individual needs. The appropriateness of these claims is dependent on the designer's redefinition of the word "individualization," a definition which is usually not as broad as the one used here. The next chapter traces the development of CAI systems, examining how, if at all, they include features associated with the tutorial process.

To the extent that CAI systems require constant and active participation from every student, and keep detailed statistics about each student's performance, the computerized teacher can carry out tasks which the human teacher of a class of forty to fifty students cannot. But to realize a computer system which tutors a particular subject, a system which is both dynamic and personalized, one must go beyond the issue of how to present the main body of material, or of how to obtain optimal teaching strategies. The question presently confronting CAI researchers asks how the computer can simulate a human tutor, and expects answers relevant to individual subject areas. For a solution, one must inquire into the behavior of a tutor for that subject area: what strategies does a tutor employ in encouraging the student to explore the material so as to understand the structure as well as the content of the subject; what rules, if any, does a tutor use to decide what to say to the student who requests help; more basically, where does the information for that decision come from; and, ultimately, can the process of tutoring, of providing especially attentive help, be understood sufficiently well so as to be embodied in a computer program.

Another significant feature of a human tutor is that he knows how to do the work the student is expected to do. The tutor can solve algebraic word problems or translate English sentences to German, solve integrals and analyze poetry. Much research has already been done in having computers perform tasks normally carried out by humans: tasks such as carrying on conversations in restricted subsets of English; problem-solving (with and without human interaction); retrieving answers to questions, or making inferences from large bases of information; playing games which are intellectually challenging, such as chess; and so on. Much of this work is applicable to having the computer act as a tutor, because, like its human counterpart, the computer-tutor should be able to solve the problems that students are expected to solve.¹ It is through this ability to carry out the student's tasks that the computer might individually examine each student's erroneous or incomplete work so as to generate a tutorial dialogue aimed at helping the student understand his error or continue his work, to give advice adapted to each student's individual need.

Implicit in the word "dialogue" is the idea that the computer can carry on a natural conversation with the student, much the way the human teacher does. This of course means that the program both understands and generates sentences in a natural language, a task which has yet to be fully accomplished (Simmons, 1970). However, the need to analyze student responses requires that the computer carry out at least a limited conversation. (Note that "responses" includes whatever vehicle the student is

¹"The first rule of teaching is to know what you are supposed to teach. The second rule of teaching is to know a little more than what you are supposed to teach" (G. Polya, 1957, p. 173).

permitted to use to demonstrate that he has carried out the task set for him.) Dialogues in restricted subsets of English have already been successfully implemented for situations in which the information needed to generate statements is already available, or is directly obtainable from immediate storage or sentences in anticipated formats (Minsky, 1968). The tutorial dialogue can, in general, be restricted to the vocabulary and grammar of the subject area being taught, avoiding the unsolved problem of conversing in the full natural language. Thus an investigation of the dialogue problem focuses greatest attention on the basic problem of obtaining the information from which the dialogue can be generated.

2. A Tutor for Mathematical Logic

Take, as an example, an interactive program for teaching elementary mathematical logic, in particular, for teaching the students skills in constructing mathematically valid proofs. Here the student's problem could be to construct a proof that an expression E , say in elementary algebra, is a consequent of a set of statements already designated as true. The student's response is a proof, that is, a sequence of steps or lines L_1, \dots, L_n , each justified in some valid way, such that L_n is E . The steps would either be names of logical rules, proof procedures, or instances of axioms or already established theorems; and the lines, the well-formed expressions inferred by each step. A proof is well defined in terms of either the sequence of steps or the actual lines. In an interactive system the student types the steps, using commands from a specially devised command language. The computer determines the validity of each

step, and, if the step is acceptable, generates the corresponding expression as a new line of the proof.

A minimal facility such a CAI system must provide is that of a proof-checker to verify that the student correctly demonstrates that the expression E is in fact derivable from a given set of premises. The proof-checker would check each step of the proof as the student makes it to determine if the step is valid. Here the student can be taught the applicability of inference rules and the proper use of axioms and proven theorems.

To clarify what is meant by "proof-checker," an example from the Stanford logic and algebra program is given (Suppes, 1971). The student works on the keyboard of a standard teletypewriter; he types commands which are requests to have the computer generate lines of a proof. Example commands the student can use are AA, CP, and WP. AA is the mnemonic for "affirm the antecedent" (a rule of inference with which a formula Q may be inferred from formulas P and $P \rightarrow Q$), WP signifies "working premise," and CP is the conditional proof procedure as shown below. The student's work might look like:¹

<u>WP</u>	(1)	<u>$P \rightarrow (P \rightarrow Q)$</u>
<u>WP</u>	(2)	<u>P</u>
<u>1.2AA</u>	(3)	$P \rightarrow Q$
<u>2.3AA</u>		
		LINE 2 MUST BE A CONDITIONAL.
<u>3.2AA</u>	(4)	Q

¹Henceforth, the convention will be to underline the information typed by the student. All other information (proof line numbers, proof lines, error messages) is computer output.

3.4AA

LINE 4 MUST BE THE ANTECEDENT OF LINE 3.

2.4CP

(5)

 $P \rightarrow Q$ 1.5CP

(6)

 $(P \rightarrow (P \rightarrow Q)) \rightarrow (P \rightarrow Q)$

The above is a proof of the expression shown in line 6. The "commands" are the sequences of numbers and mnemonics at the left margin of the proof. Line numbers referenced in the command must refer to a particular type of expression, such as a conditional statement, or to a line generated by a particular code, such as a working premise. Otherwise, an error message is printed. The commands have certain formats (e.g., AA requires two line references); the student receives an error message if the command syntax is incorrect or if the command cannot be applied to the lines referenced.

Examples of other proof-checkers are illustrated in the next chapter. The comments which follow are applicable to these other proof-checkers as well as the one described above.

In constructing a proof in an interactive system, the student may find that he has taken several steps but is unable to finish. The computer-teacher, in assuming a tutorial role, is expected to help the student by giving him advice on how to utilize his own partial proofs to arrive at complete ones. The tutor might also, after examining the student's partial or completed work, suggest alternative solution paths. That is, he could point out another strategy known to lead to a successful proof, and, in this manner, encourage the student to explore diverse solution paths for each problem.

The ability to decide what strategies to discuss, and when it is necessary to make further suggestions, depends on the tutor's knowing

what the student has already done and what other paths are available. The computer which purports to tutor a student constructing proofs should first find out what the student has done so far with the problem. Using this information, the computer can initiate a dialogue which will direct the student towards a successful solution of the problem. There are many different solutions which constitute derivations of a true expression in a theory; for, that true expression, there may not be a proof procedure which uses a natural¹ approach for setting up at least one possible derivation. The dialogue problem is compounded by the fact that the student may have attempted several different proofs in a given response, several or none of which can be completed successfully. In such a case, the question arises as to which solution path, if any, the tutor should discuss, and whether the student should be made aware of other valid approaches he may have started.

In the past, instructional programs such as the proof-checker presented above have virtually ignored the problem of figuring out what the student has already done. Instead, the programs offer hints which direct the student to finding a solution thought up by the course author at the time he wrote the curriculum. These hints are usually prestored with the problem on a peripheral device. They are rarely responsive to the student's needs, as no attempt is made to determine those needs. Comments produced from predetermined hints are often inappropriate or irrelevant: inappropriate in the sense that the student may have already carried out the suggestions, or irrelevant to the task of completing a valid solution already started by the student.

¹By "natural" is implied a conventional mathematical proof such as those found in Suppes and Hill, 1964.

Consider another technique which was suggested for helping the students using the Stanford program: store solutions for each problem, and then compare the student's partial proof, line by line, with the stored ones in search of a likely candidate for discussion. Since it would probably be impossible to store all possible solutions for each problem, the hints might still be irrelevant. Moreover, other than declaring the similarity between a stored proof and the student's partial one, this technique lacks the information needed to explain why a particular suggestion was made at all.

Several possible features of an interactive system for teaching proof construction exist which introduce further arguments against the stored-hint or stored-proof approach. The two suggested here, although not available in the Stanford program, are incorporated in a new instructional system to be presented later.

The first feature permits the curriculum writer to prohibit the use of certain commands in a problem. (This is useful in encouraging the student to try new derivations.) The set of stored proofs for that problem should exclude solutions using the disallowed commands. But, further, suppose the instructional system were designed so as to let the student freely explore the command language; the proof-checker does not monitor the student's work with the intention to direct the search for a solution. In such a system, the student could successfully complete a derivation using a prohibited command. The computer-tutor would let the student finish his solution, but might then comment on its incorrectness due to the special restriction, and request a different solution.

Suppose now that the student requests help, that his partial proof can be successfully completed, but that the student has used a prohibited

command. A tutor depending on either stored hints or stored proofs could not give advice on an acceptable solution while still commenting on the student's partial proof; this is because no information about that proof is available.

As a second feature, what if the instructional program were flexible enough to let the student make up his own derivation problems? The tutor could not have any anticipated hints or solutions to these problems, and so would not be able to help those students adventuresome enough to make up their own examples. These difficulties arise because the so-called proof-analyzer, the tutor, cannot find its own solutions to the derivation problems.

3. Using a Theorem-Prover to Teach Theorem Proving

A solution for the dialogue problem, for helping the student who requests aid in constructing a mathematically valid proof, is presented in this dissertation. The solution is to employ a mechanical theorem-prover, one which is capable of doing the problems the students must do, as a proof-analyzer. In its new role as a proof-analyzer, the theorem-prover examines the student's incomplete proof and extracts information which can then be used to give hints and advice on how the student might continue his work. The theorem-prover assumes that the proof-checker portion of the instructional system has verified the correctness of the student's partial proof. And so the theorem-prover can use the student's steps, much the way it uses a set of premises, as a guideline for discovering one or more valid derivations. (This analysis process is described

in detail in Chapter V.) The computer-tutor then uses these derivations as a base from which to generate a simple tutorial dialogue.

Because the theorem-prover may discover more than one valid derivation, it can provide information for discussing diverse solution paths. From the set of derivations determined by the theorem-prover, the computer-tutor can decide whether there are other solutions which the student should try; for example, solutions which require the student to use recently learned commands or recently proven theorems, or solutions which require significantly fewer steps than the student's first derivation. In encouraging the students to try diverse solution paths, the computer-tutor can attempt to keep the students from becoming "set" for a particular solution approach and, thus, unable to handle new problem situations.

The existence of an appropriate theorem-prover, one which can do the problems which the students are expected to do, does not necessarily imply that the theorem-prover can be made to interact with an instructional system such as the one illustrated earlier. This interaction is of the following nature. As part of a problem in the curriculum, the student is told what expression to prove, and within what constraints. He then tries to find a solution. If the student cannot find the solution, then the instructional system gives this same information to the theorem-prover. This information includes any work already done by the student. The theorem-prover attempts to find one or more proofs, especially ones using solution steps already thought up by the student. The prover tells the instructional system, the computer-tutor, about these solutions. Then the tutor, in turn, based on strategies expressed in Chapter V, determines what suggestions to make to the student.

Subsequent chapters will explain how two programs, a theorem-prover and an instructional system, operate and communicate with one another. However, one comment should be made here: that is, if the theorem-prover makes use of rules and procedures which the students use, the task of examining the steps already taken by the student, of using the student's ideas, is facilitated. This is one reason, of several to be cited, why a heuristically programmed theorem-prover was chosen as the proof-analyzer, rather than one with a more formal basis.

Several problems arise in using a theorem-prover as a mechanical device for extracting information from an incomplete proof. A central one is the question of justifying the choice of the type of theorem-prover itself. Not every theorem-prover will behave correctly in an interactive tutorial mode. A theorem-prover embodying heuristics thought to be used by the students was chosen. The inference rules used by the program were empirically determined by observing students doing proofs, from experience tutoring these students, and from investigations in using numerous versions of the program. The underlying idea behind the design of the program was to have the theorem-prover produce proof steps which one could expect a student to produce, not steps requiring converging inferences so large that only a computer could handle them.

The purpose in carrying out empirical studies, even those of an informal nature, was to provide information on how the theorem-prover might incorporate methods used by the students. The idea was less to imitate the good students than to have the theorem-prover use strategies which could be taught to the less skillful students. For example, the theorem-prover could use methods for reducing the problem to a simpler

one, for organizing the given information to determine which rules are applicable, and for manipulating the proof steps so as to reduce the difference between a given step and the desired expression. Heuristics could especially be developed to recognize similarities between axioms, or proven theorems, and the expression to be obtained, so that substitution sequences might be computed and subsidiary derivations established. If steps of a proof completed by the theorem-prover would correspond to particular strategies, then these strategies could be commented upon in the tutorial dialogue. The basis of the dialogue would be to teach the strategies, to point out those methods the theorem-prover found could be used to generate a solution. In the survey on mechanical theorem-provers given in Chapter III, the inappropriateness of some other types of theorem-proving programs is noted. Although the rules of inference used by the heuristic theorem-prover are generally characterized, and some results of a mathematical nature offered, subsequent discussions and examples are restricted to an Abelian group under addition. This is done for several reasons. One, several efforts have already been made by CAI researchers to develop systems for teaching elementary algebra. Their experience has provided some empirical data on the types of difficulties students have in constructing elementary proofs. Second, from a curriculum for elementary algebra developed at Stanford University, a set of theorems and derivation problems about addition were readily available.¹ The theorem-prover could thus be designed

¹Since July 1, 1970, the author has been responsible for the operation of the Stanford logic and algebra program.

to find proofs for the theorems, and then be tested on a fairly large selection of the derivation problems. Third, the Stanford program uses the stored hints approach for giving the students help. Difficulties arising from inappropriate hints were recorded and could be compared with the help sequences afforded by the theorem-prover/proof-analyzer. Human tutors, experienced in helping students of the Stanford program, were also available. A study could be carried out to compare the human tutor's work with that of the theorem-prover/proof-analyzer, and to further decide how well the analysis techniques work. Finally, these techniques can be examined with a view toward their application to other areas of instruction.

Subsequent chapters will demonstrate the ability of the theorem-prover to construct proofs of certain true statements of elementary algebra, and will include some mathematical results concerning the set of inference rules.

Then they will exhibit how this theorem-prover can

1. examine a partial proof produced by a student using an instructional system of the kind described earlier (that is, one utilizing the response format of the Stanford logic and algebra program);
2. determine one or more solution paths which will successfully continue that proof;
3. adapt to alternative solution paths taken by the student, even when those alternatives appear in a single incomplete proof; and
4. provide information from which a simple tutorial dialogue can be generated.

The tutorial dialogue employs the kind of questioning suggested by Polya (1957) for helping the student solve the immediate problem, and developing the student's ability so that he may solve future problems himself.

Consequently, in the initial dialogue shown in Chapter V, attempts were made to use general suggestions which do not give the answer away. It is unproductive to merely present a correct derivation to a student who cannot understand how it is possible to discover such an argument. Skinner, in outlining the resemblances between programmed instruction and tutoring, stated that a good tutor does not lecture; instead by his hints and questioning he helps the pupil to find and state answers for himself (Skinner, 1958). And so the hints try to show the students where to find solution ideas. Polya commented that the teacher should make suggestions which could have occurred to the student himself. By using procedures which mirror those of the student, the theorem-prover comes up with more natural and more instructive hints.

4. A Generalized Instructional System for Elementary Mathematical Logic

So far, the discussion on individualization in a CAI program has focused attention on the computer's ability to help a student who is encountering difficulty. Hypothetical features of a computerized system for teaching elementary mathematical logic have been introduced, specifically to highlight reasons for using a theorem-prover to analyze a student's partial or erroneous proofs. Several of these features reflect ideas known by educators to be effective teaching aids. In fact, the theorem-prover does interact with an instructional system embodying many of these ideas. This interactive CAI system serves as an environment in which research on developing and on teaching nonlogical axiomatic theories can be carried out.¹

¹A more detailed account appears in Goldberg (1971).

The system is a new approach to using a computer to teach mathematical logic. It was designed to increase the active participation of the students with the computer-teacher by providing methods by which a student can (a) experiment freely with his new skills in proof construction, (b) receive immediate feedback on errors, (c) ask about previously learned material, (d) make up his own examples, and (e) receive advice and comments on his work which take into account only the material he knows (i.e., that has been already taught or has developed himself). Such features are available because the instructional program is actually a powerful interpretive system in which an individual, be he student, teacher, or researcher, can develop and then study a nonlogical axiomatic system along whatever lines he himself specifies. The justification for such an elaborate facility rests on two conjectures. The first is that if the student can develop his own examples, then he might develop a better understanding of what an axiomatic theory is, i.e., where theorems and rules come from, and how dependency relationships between theorems affect the ability to find proofs (Suppes and Crothers, 1967). The second appears in Polya: "The mathematical experience of the student is incomplete if he never had an opportunity to solve a problem invented by himself" (Polya, 1957, p. 68).

The interpretive system is able to support formalizations of the full predicate calculus with identity. The user can specify a vocabulary and a set of axioms with corresponding names, prove and name theorems and lemmas, and derive new rules of inference. In so doing, he builds a command language for constructing proofs such as that suggested in Section 2.

The program "knows" only primitive rules of predicate calculus, but consists of routines which compute and learn new commands from the well-formed

formulas of the system. When the user attempts to construct a proof within the system he so specifies, he types commands from his command language. In order to interpret such commands, generalized processing routines check each command for correct syntax and usage, and compute appropriate error messages if the command is not a valid one. Moreover, the system is able to provide teaching sequences for these "learned" commands upon demand by the student. Thus the features for specifying axiomatic theories are built around a kind of interactive proof-checker which has broad tutoring facilities.

The instructional system presented here is representative of a mode of computer-based instruction which gives the student those computer facilities usually available only to a teacher or a curriculum-writer. This mode is contrasted with the usual CAI programs in which the teacher, or programmed algorithms, entirely determine the sequence of questions the students receive. It may be the case in the program for teaching logic, that the command language and a curriculum are constructed by a teacher, and then the students follow the curriculum. But the students always have the option to interrupt the linear sequence of problems and make up their own problems, or prove lemmas, or derive new rules of inference to help make the teacher's problems easier to solve. Here, the distinction between a teacher and a student is a narrow one, because the student himself can enlarge the command language or invent a "curriculum" for himself and his classmates.

A teacher-specified curriculum is not necessary for the operation of the instructional system. The student himself can specify the nonlogical axiomatic theory he is to study.¹ Such a CAI system is also a good

¹This might be a project for students in a Philosophy of Science course.

environment for trying out the kinds of logic tasks modeled after the R. L. Moore method of instruction. One such task is a "Finding-Axioms" exercise, in which the students are given a list of formulas and are asked to choose at most N as axioms. The rest are to be proven as theorems. Success in this exercise may rest on the skillful choice of the order of proving the theorems.

The Finding-Axioms exercise was used in an elementary course in logic at Stanford University, a computer-based course usually taught exclusively by the logic-algebra program. Several students, with no mathematics background, were experiencing difficulty in getting through the linearly organized curriculum when they were given the new exercises to try on the new instructional program. After working one or two Finding-Axioms exercises, one student expressed delight in finally "understanding what was going on." The opportunity to specify axiomatic systems, to make up lemmas, and organize the curriculum proved invaluable in this student's understanding of sentential logic and elementary algebra. More details on the Finding-Axioms exercises can be found in Goldberg and Suppes (1972).

Seventh graders, who also tried the exercises, had more difficulty with them than with the logic and algebra program. Their bottleneck was not in constructing the derivations, but in deciding which problem to work on next; in the context of the logic-algebra program, their only former encounter with proof construction, they were too used to being told exactly which expressions were axioms and which and in what order the rest were theorems to prove. Jerome Bruner discusses this problem in considering how to stimulate thought in the setting of a school:

By school age, children have come to expect quite arbitrary and, from their point of view, meaningless demands to be made upon them by adults--the result, most likely, of the fact that adults often fail to recognize the task of conversion necessary to make their questions have some intrinsic significance for the child. Children, of course, will try to solve problems if they recognize them as such. But they are not often either predisposed to or skillful in problem finding, in recognizing the hidden conjectural feature in tasks set them. But we know now that children in school can quite quickly be led to such problem finding by encouragement and instruction (Bruner, 1966, pp. 157-158; underlining mine).

The kind of CAI system offered here might provide such encouragement. Experience using the new instructional system for mathematical logic has been favorable as an experimental laboratory in which to explore new ideas about using theorem-provers for teaching about theorem proving, and as a better way to use the computer for instructional purposes.

5. Research Design

The outline of the research design presented in this section reflects the organization of subsequent chapters. It might also serve as a directive on how to read the dissertation.

The next chapter surveys the use of computers as teaching machines, and includes examples to illustrate proof-checker programs. More information is also given on the Stanford logic and algebra program. This leads to Chapter III, an expansion on the reasons why a heuristic theorem-prover was chosen as the method for extracting information from the student's work. Reasons are also given for choosing a heuristically based theorem-prover above other kinds of provers.

Chapter IV contains details about the theorem-prover itself: the characterization of the inference rules, which are shown to be complete for a subset of the elementary theory of Abelian groups; and a description of a heuristic that cuts down the amount of time spent in searching for a solution. If the tutor is to be used in actual classrooms, such heuristics will be needed.

The theorem-prover was written in LISP (McCarthy, 1962) for the Digital Equipment Corporation model PDP-10 computer at the Computer-Based Laboratory of the Institute for Mathematical Studies in the Social Sciences (IMSSS) at Stanford University. Its operation, and that of the instructional system with which it interacts, do not depend on any special resources of the Stanford laboratory; however, the time-sharing facilities certainly expedited the large-scale implementation required.

Solutions for theorems found in the IMSSS algebra curriculum are given in Appendix II, and described in Chapter IV to illustrate the style of proofs that are generated. Chapter V is devoted to demonstrating the adaptive capabilities of the theorem-prover to complete partial proofs. A case study was carried out in which a seventh-grade girl, Lisa, tried derivation problems. At the points in a proof at which she needed help in order to continue, she typed the command HELP. The theorem-prover then determined possible ways to successfully complete her work. Examples of a simple dialogue with Lisa, based on the information provided by the theorem-prover, are given. A second study was conducted, an investigation of the tutoring behavior of the human tutors who had been employed to help the students of the Stanford logic-algebra course. It provided further information about how well the proof analysis works.

Chapter VI summarizes the features of the framework in which the theorem-proving research was carried out, that is, a generalized instructional system for teaching elementary mathematical logic. This summary includes examples of the automatic tutoring facilities of the system, and the other ways in which mechanical theorem-proving procedures are incorporated in the teaching program.

CHAPTER II

COMPUTER-ASSISTED INSTRUCTION

1. Background of CAI Research

"Individualization of the learning process" is the dominant theme recurring through the now abundant articles, books, surveys, and progress reports on the subject of computer-assisted (-applications-to, -aided, -administered) instruction (which will henceforth all be implied by the acronym CAI). The stated goal of most authors is to provide improved learning conditions for school children: learning at a better, more conveniently administered, self-paced level. Then, they claim, the conventional teacher will not have to face the drudgery and repetition of presenting factual material, will be able to change frequently the content of presentations, and will have more time for classroom interactions. Also mentioned is the computer's ability to provide a controlled environment for doing research on teaching, since the teaching conditions can be maintained or manipulated easily, and to collect and analyze ample records of student performance.

The above paragraph, packed as it is with clichés of the rapidly expanding field, does summarize apparent intentions of CAI researchers. Although the above goal is not yet achieved, the last decade saw a surge of activity and experience in the use of computers in education. Aided by advances in computer technology and increased funding, many CAI

projects have been initiated. Some, such as that of the Stanford University group, have demonstrated clearly the feasibility of CAI. They have developed a computer network which reaches more than a thousand students a day to teach courses in elementary mathematics, logic, reading, and languages. Most of the work in the early and middle 1960's centered around problems of optimal teaching strategies, of increasing the complexity in the methods of evaluating and processing student responses, and of data-processing and managerial aids to the school systems. However, from a computer science point of view, programs in CAI have just recently begun to take advantage of the computational powers of the time-shared systems in which they operate. In the past, the instructional systems were input/output bound, with minimal use of computer processing for evaluative feedback and selection of presentation material. Programs are now being developed which have been directly influenced by results of research in artificial intelligence, e.g., on pattern recognition, CRT display processing, information retrieval for question-answering systems, symbolic manipulation and generation techniques.

Much of the significant work in CAI has been collected in a number of books (Atkinson and Wilson, 1969; Bushnell and Allen, 1967; Coulson, 1962). Substantial surveys can be found in Hickey (1968), and Zinn and McClintock (1970). It is impossible to discuss here all the relevant work that pertains to the present research. A review of artificial-intelligence projects on mechanical theorem-proving appears in Chapter III; some more recent CAI projects which are directly relevant are reviewed below.

Stolurow (1967) stated that a "model for adaptive, or personalized, instruction, specifies a set of response-dependent rules to be used by a teacher, or a teaching system, in making decisions about the nature of the

subsequent events to be used in teaching a student." CAI programs can be characterized by these rules or instructional strategies. The lowest level of CAI was the outgrowth of ideas of noncomputerized teaching machines put forth by Pressey (1926) and Skinner (1968). The basic idea is to present the text material in sequential steps, or frames, and to require mastery of each frame before progressing. The student responds by constructing an answer which is processed for an exact match. All the educational material is stored in bulk on peripheral devices. The computer is simulating an elaborate bookkeeper with a large file cabinet. Individualization is attained here only in the student's rate of completing the material.

Crowder (1962), in his "scrambled" textbooks, introduced the branching concept. The mechanical version, which is either called a "selective" computer teaching-machine or the tutorial mode of CAI, is characterized by three operations: storage, comparison, selection. The curriculum writer must determine all possible questions, statements, and hints he may want to present to the student. He must also decide on all alternative responses he suspects the students will make. The curriculum is stored on peripheral devices. The student is asked to answer a question by selecting from a multiple-choice list of anticipated correct and incorrect answers. The response and the list are compared, and the corresponding prestored information is selected as the next item to be displayed. Selection is based on the immediately preceding response. Individualization is exemplified in the selection of remedial sequences to assure understanding of the main trunk or source of items.

More advanced selective CAI programs are referred to as adaptive tutorial modes (although the use of the adjective "adaptive" advertises

more than the CAI systems actually provide). These programs base the selection of material on the history of a series of past responses. For example, the student's progress through lessons with problems of varying difficulty might depend on the student's total performance record on all or some of the previous lessons. Response processing has also been extended to partial answer analysis: checking for spelling mistakes, algebraic equivalents, partially correct translations in language drills, and so on. Despite the increased sensitivity to student performance, individualization here still seems to mean pacing the flow of material in order to get everyone to learn the same prescribed information.

In an effort to reduce the storage requirements and to increase the responsiveness of the teacher-student interaction, "generative" computer teaching machines are presently being programmed. Developed mostly for mathematics courses, they employ algorithms to generate problems and answers. Uttal (1969) has used generative techniques in a tutorial system for analytical geometry. The algorithm for generating problems mirrors that used for the diagnostics and remediation lessons. A help sequence teaches the student how to trace algebraic and arithmetic errors by sequentially decomposing an expression into subexpressions until the error is discovered. Uttal maintains that this algorithm simulates the human tutor's process of "going over the problem together."

These systems, mainly computer- (and, therefore, teacher-) controlled, utilize prescribed sequences of exact material. Even Wexler's (1970) more advanced system for generating questions and answers of a nonscientific nature still requires this anticipation of student needs. Some work has been done, notably with PIANIT at Systems Development Corporation, to let

the student ask for intermediate information or do some computations on the computer before responding. In all these situations, despite quite complex attempts to analyze performance and use the computer as a calculator or problem generator, the student is still under teacher control for what he can examine and therefore what he can learn. Moreover, the kind of help (tutoring) he can get is still very limited.

In simulation attempts the idea was to use the computer as a laboratory tool with which the student could test his intuition about possible results of changes in parameters or could, perhaps, make his own discoveries about the properties of a model. Another example is that of computer games involving more than one person. These games provide real-life experiences through models of an economic or political nature. Here the student is left on his own to explore the subject area. The more sophisticated students might benefit from this sort of self-discovery method, but what of the less skillful students who require more personalized help before they can learn how to test the material presented to them? Perhaps the games could serve as better, self-contained instructional devices if they co-existed with programs that can play them.

Three projects of interest attempt to provide less restrictive learning environments. The first, Uttal's work, has already been mentioned. The point to be made here concerns the coupling of generation techniques with adaptive response analysis. Help diagnostics are computed by using the same procedures used to generate problems. (Wexler, too, employs this coupling of generation and analysis procedures.) As will be seen in the work on theorem-proving, this coupling affords a significant vehicle for determining what the student was trying to do in order to respond

accordingly. It illustrates the idea that if the computer is to tutor a student, then the computer should be capable of doing what the student is asked to do in order to use that capability to "understand" what the student is doing. This need for knowledgeable computer-tutors was also emphasized by Siklóssy [1970].)

The second project is Carbonell's (1970) work on "information-structure-oriented" CAI versus the aforementioned "ad-hoc-frame-oriented" programs. His system is called SCHOLAR and is "capable of reviewing the knowledge of a student in a given context (e.g., geography) by maintaining a mixed-initiative dialogue with him in a rather comfortable subset of English." Basically a question-answering system, the student as well as the program can have the initiative to ask questions and be required to answer them. SCHOLAR can generate multiple-choice, true-false, yes-no, fill-in-the-blank, and what-why-where questions. By keeping track of the content of this dialogue, SCHOLAR can effect changes on the basis of relevancy and time considerations. By representing the "meaning" of words in the data-base in terms of a Quillian-like semantic network (Quillian, 1968), Carbonell has moved a large step away from keyword or template-matching response techniques. (Quillian [1969], too, has applied his work on natural-language processing to CAI research.) Carbonell does not, as he points out, consider the general error-analysis problem, although he does recognize errors in a student's response and pursues them in a limited manner. Compared with other CAI programs, Carbonell's program, by taking advantage of the work in natural-language processing, makes more use of the computer's computational powers to provide freer exploration by the student.

The theme of free exploration, of understanding through active participation, rather than just passively being taught, can be found in Papert's (1971) work, especially in the LOGO project. There the student is taught a simple programming language and is then allowed to initiate any task he wants. This project-oriented approach to teaching, where the student originates a problem and its solutions, learns to state them explicitly, and then proves the solution is correct, emphasizes learning to think. Papert's teaching methods, that is, having the students write and debug complete programs, emphasizes positive thinking about errors. The students learn to accept that wrong answers "are the interesting answers-- that they are pointers to the places where the student's work can be improved." This is obviously true for the tutor as well as for the student. To paraphrase Papert, the motivational aspect of this teaching approach is that the value of what the students do is proved by success in their goals rather than by the opinion or edict of a teacher. He works with a computer just "because it allows the structure, rewards, and pleasures of a complex project to take on a pure and clear shape, (thereby making) possible and, especially, (allowing) a new degree of articulate understanding of their essential unity."

2. Mechanical Proof-Checkers

Having reviewed the history of the instructional use of computers, this section turns to those CAI programs which specifically deal with teaching elementary mathematical logic. The intention here will be to clarify subsequent uses of the word "proof-checker." This is accomplished

by describing three kinds of computer-based programs which can be used to verify the correctness of proofs.

The first kind of proof-checker analyzes a complete proof, determining the validity of each line and filling in missing steps that would make the proof a rigorous one. "Rigorous," as used here, implies that no step in the deduction, no assumption or mediating inference, is omitted. Abraham's (1963) computer program to verify (but not invent) mathematical proofs exemplifies this kind of proof-checker. The idea came from suggestions in McCarthy's (1959) Advice Taker article, in particular, the notion of writing programs which could make deductive inference using some built-in "common sense." One part of Abraham's program translated a proof from a textbook or journal article into a canonical form. The other part applied ten inference rules to the transformed proof in order to assure that each line was a valid inference. The verification scheme could find errors in nonrigorous proofs by attempting to expand the arguments rigorously. Although it could have been used in checking students' proofs as well, the program was not actually employed in an educational setting.

The second kind of proof-checker qualifies as a homework grader. The first alternative to such a grading program is to use the computer as a batch-processing system, "marking" the students' solutions as a human teacher might. Clearly, the use of this sort of program benefits the teacher only (unless comparisons are to be made to a less conscientious teacher and any help serves the students' interests). In an interactive mode, the student constructs a rigorous proof, decides that it is complete, and then receives feedback from the computer as to the correctness of the solution. The student is free to make errors in the use of inference rules. But one error may be propagated so that the solution

is invalid. Since a good tutor does not permit false answers to remain uncorrected, the student would be asked to redo the problem. In fact, the student may have to redo the problem several times; and, most likely, it will take him longer to find a valid solution than if each error were immediately corrected.

The third kind of proof-checker examines each step of the proof as it is being constructed, rather than waiting until the student presents an entire solution. If the student attempts to use a rule of inference incorrectly, he is immediately informed of the error; he can then try the rule again, correcting his mistake. Objections to using immediate feedback methods rest on the goals of the curriculum. If, ultimately, the student will be expected to construct complete proofs without the aid of intermediary error analysis, then the opportunity to do so during the course might be necessary. Substantiating evidence will be needed before anything definite can be said. It may be the case that both interactive methods are useful; that checking each step as it is stated is preferable when the student is just learning the rules, and that later in the course the student could have the option to ask for comments after he decides he has completed a solution. In any case, the more difficult aspect of theorem proving is discovering goal solution strategies; the main concern of the computer-tutor is to help a student learn how to discover solution ideas. As similar tutorial requirements exist in either interactive situation, the method discussed in ensuing chapters will be that of immediate feedback; and the solution to the dialogue problem to be offered will be illustrated in terms of the response format used in the Stanford logic-algebra program.

2.1. Proof-Checkers with Immediate Feedback

Two attempts, other than the one described in Chapter VI, have been made to use this last type of proof-checker for teaching skills in proof construction. They are PROOF (Easley, Gelder, and Golden, 1964) at the University of Illinois, and the Stanford logic and algebra program (Suppes and Binford, 1965; Suppes and Ihrke, 1970; Suppes, 1971). Both instructional programs employ the kind of proof-checker which checks each step of the proof as it is being constructed. They both use a keyboard as the input device and both provide immediate feedback on errors made by the student. Each program is described below.

PROOF at the University of Illinois, Coordinated Science Laboratory.

The first report of PROOF described a program which was extremely limited by a clumsy keyset and the inability to reference more than one line (the last) of the proof. The student had to start with an instantiation of the problem to be proved, and, by algebraic manipulations on successive lines, using commands rigidly defined on the small keyset, reduce the expression to the form $X=X$. Thus, the proofs were given by a reduction to a tautology. The authors of PROOF proposed ways to give the student more control, to change the program so that they were no longer restricted to transformations along a linear chain, and to add new rules to the system. Although the new system was to be programmed for the PIATO III teaching logic, no second report seems available.

The Logic and Algebra Program at Stanford University, IMSSS.

The logic and algebra program at Stanford University is a natural deduction treatment for deriving expressions in an ordered field. The main thrust of the program

is to let the students construct derivations for expressions in sentential logic and elementary algebra. The students are taught a simple coded command language in which the codes represent the names of axioms, theorems, rules of inference, and some proof procedures. These codes, and how they are used, were illustrated in Chapter I. The student can request any valid command regardless of its relevance to finding the solution to the problem. He is provided a hint facility; hints are specified on the disk along with the problems. A timer is associated with each hint to determine the wait between the student's responses and offering (not giving) the hint. The student must explicitly request the hint before it is given. Because the hints are prestored, they may not always be relevant to assisting the student's search for a solution.

The logic and algebra program provides a self-contained two-year course directed at bright fourth-, fifth-, and sixth-graders. The student must solve a prescribed sequence of problems. However, he is free to explore the command language, to familiarize himself with the result of any command request. If he forgets how to use a command (the existence and exact order of line references and names, or what a command does), only syntactic error messages or human proctors can help him out.

The freedom given the student means that the program is not aware of what the student is doing. Aside from prestored hints, the program cannot offer any help in completing a derivation. Perhaps from the point of view of programming design, the logic and algebra program knows too much--is too "teacher oriented." Each rule is specifically programmed and each routine depends on particular knowledge of character meanings. The program handles only binary infix operators or one-place predicates. With much effort, it was reprogrammed to handle operators and rules of boolean algebra.

The instructional system described in Chapter VI was developed as a natural successor to the logic-algebra program, and many of its features result from experience obtained through operating in the schools with that program. The new system, which operates separately from the logic-algebra program, borrowed the student's response format, as shown in the example in Chapter I, from the Stanford program. However, the solution for the dialogue problem is especially important to the success of both instructional systems.

CHAPTER III

MECHANICAL THEOREM PROVING: A NEW APPLICATION

Criticisms that computer-based teaching systems lack the attentive and personalized interaction afforded the student by a human tutor are best answered in terms of a concrete example to the contrary. The example offered here addresses the need for individuality in teaching formal theorem proving. Individualization takes the form of helping students who make errors while working on a problem, and of helping students if they request hints on how to complete their partial solutions. It is realized further by determining methods for having the computer-teacher enrich the student's understanding of a problem, even if it has been successfully completed.

Methods the computer-teacher has for helping a student while he is working on a problem are given within the special framework of the instructional system for elementary mathematical logic to be outlined in Chapter VI. The question answered here is how a computer might give appropriate suggestions to the student on how to continue his work--appropriate in the sense that the suggestions are based on information determined from actual analysis of the proof steps already generated by the student.

Two methods for helping a student continue his work were offered earlier: storing hints with each problem, and storing complete proofs. These methods were rejected on several grounds, mainly because neither

was capable of taking the student's work fully into account when deciding what to say to him. Providing stored hints completely ignores the problem of examining the student's work. Comparing the student's work to stored answers in search of a likely candidate for discussion will probably utilize some of the student's proof steps, but will be insensitive to the more uncommon proofs a student may try. In both cases, the only apparent reason for suggesting a next step to take is that "it has been done that way before."

A method which is sensitive to the student's needs must be able to examine the student's partial proof to obtain information from which relevant and helpful suggestions can be made. The part of the instructional system which "conducts" the tutorial dialogue requires a base of information about the student's work: what steps he has already done, how these steps might enter into complete proofs, which of the several solution strategies seems to follow the student's ideas, what sorts of procedures entered into the discovery of that solution, and what other proof strategies will lead to successful solutions.

In order to provide this information, a general and knowledgeable proof-analyzer is needed--a proof-analyzer which can come up with its own answers to problems the students must solve, but which can also adjust its solution search to solve the problem with respect to work already done by the student. A computer program which can fulfill this need for a knowledgeable proof-analyzer is a theorem-prover, one which is capable of finding the sorts of proofs the students are expected to discover, and one which can communicate its results to the instructional system in which the students study. The intention in using the theorem-prover is (a) to

assure that the computer-tutor comes up with correct responses which help the student complete his work, and (b) to teach the students good solution methods which the tutor knows will be useful in obtaining a valid proof to the given derivation problem. Note that if there is some correspondence between the solution methods employed by the theorem-prover and the methods which the tutor teaches the students, then the tutor can discuss the theorem-prover's actual solution search. So it would be insufficient for the theorem-prover merely to accomplish mechanically the task of discovering solutions without attempting to simulate the processes human problem-solvers supposedly use. To confirm their usefulness in teaching unskilled students, the computer-tutor's theorem-prover might employ techniques determined from methods thought to be used by good students; as such, the program would attempt to imitate these good students.

The computer-tutor to be offered in succeeding chapters, with a theorem-prover as proof-analyzer, is not necessarily offered as a theory of human problem-solving, although the similarities between this tutor's theorem-prover and the programs of Newell and Simon (1963a and 1963b), for which psychological studies have already been conducted, indicate that such an offering would not be an unreasonable one. Newell and Simon claim that their detailed protocols (verbatim recording of all that the subject and the experimenter say during an experiment) support the contention that what is going on inside the human system is quite akin to the symbol manipulations going on inside their General Problem-Solver (GPS) program. These same symbol manipulations form the basis of the tutor's theorem-prover. The investigation of the theorem-prover's techniques, or heuristics, was benefited by the work of Newell and Simon on GPS, as well as

by personal experience in tutoring students learning to construct algebraic proofs.

Having decided that a mechanical theorem-prover might serve as a proof-analyzer, and having given some hint as to the type of theorem-prover actually constructed for the task, the next questions to answer are: what are the properties of the appropriate theorem-prover; how can a theorem-prover learn about the student's work and communicate its results to an instructional system; is a theorem-prover already available which can be adapted to operate within an interactive educational system; and does the use of a theorem-prover as proof-analyzer approximate the behavior of human tutors? First, what kind of theorem-provers already exist?

1. Background of Automatic Theorem Proving

The mechanical proving of mathematical theorems has occupied the investigations of artificial-intelligence researchers since the late 1950's. Mathematicians and psychologists have both participated in constructing computer programs for generating mathematical proofs or for checking proofs for correctness (until recently, mostly in the field of first-order logic). These programs are valuable in advancing the understanding of the human problem-solving process as well as in obtaining new formal results.¹ They may also yet prove valuable in teaching the notion of mathematical proof.

¹Note, however, that new and interesting results found by mechanical theorem-provers (which have been published) have not been obtained by a mechanical theorem-prover acting independently, but only in a highly interactive mode with the human user.

Mechanical generation of mathematical proofs, as opposed to proof-checking (Abrahams, 1963; Mauer, 1966), has typically relied on two approaches. The more powerful in terms of the size of the class of problems it can handle is exemplified by resolution-based theorem-provers, that is, programs based on the single inference rule called the Resolution Principle (Robinson, 1965). The resolution principle is applicable to all deduction problems that can be reformulated as statements in the first-order predicate calculus. (See Green [1969] for an example of resolution as applied to a question-answering system.) Proofs using this method are sometimes quite difficult to follow; the number of statements generated usually expands rapidly. Therefore, in order to write a program based on the resolution principle, a number of search strategies had to be devised which could limit the amount of computing and memory space required (Luckham, 1970).

The second approach, rather than using a few strategies to refine a formal algorithm, is based mainly on heuristics. Heuristic programming is, by now, a familiar concept in computer science. One of the earliest programs based on heuristic devices discovers proofs to theorems in the sentential calculus: the Logic Theorist (LT) program (Newell, Shaw, and Simon, 1963). The program operates by attempting to eliminate differences between a "given" expression and one which is to be proved. Elimination is carried out by replacing substructures of the given expression to transform it into the expression to be proved; the replacements are applications of substitution or rewrite rules.

A heuristic is a strategy to solve a given problem, whose use may incur a significant risk. If it is applicable, it may greatly reduce

the effort expended in finding a solution. However, there is no guarantee that it is, indeed, applicable to a given problem. The performance of LIT depends on a heuristic, a similarity test, to decide which subproblems to try to prove so that a difference might be removed.

Constructing a proof heuristically is generally viewed as a problem in symbolic manipulation; thus LIT emphasizes finding syntactic similarities between the structures of logical formulas. LIT introduced the "means-end" analysis technique (or problem reduction)¹ in which the program begins with the statement to be proven. It then attempts to reduce the statement to a known statement, or one which can be easily established by substitution rules or standard transformations. This is also sometimes referred to as "working backward." In this reduction process, subproblems are set up which must be solved by techniques used to solve the original problem. So the search procedure is recursive.

A second example of a heuristic theorem-prover is the Fortran Deductive System (FDS) (Quinlan and Hunt, 1968a and 1968b). Like LIT, the FDS is a problem-solver which reduces differences between an initial state (or structure) and a goal state, using rewrite rules. The rewrite rules are axioms of the deduction system, and the goals, the theorems to be established. If the goal must be of the form $\eta = \delta$, then the initial state becomes η , and the goal is δ . The resulting solutions are "line-by-line" proofs; each rewrite rule can be applied to the results of the previous rewrite only. The authors of FDS call these solutions "difference-discoverable." A look-ahead process is used to choose subproblems which

¹Nilsson (1971) describes other problem-solving methods, called state space transformations and formal logic (e.g., the resolution principle); they are used in mechanical theorem-proving programs.

are less difficult than the original problem, but which help solve that problem. This process orders the attempts to apply rewrite rules.

Heuristic programs, such as LT and FDS, are usually syntactic. The program's ability to solve problems depends on the language in which the problems are stated and the particular structure of the statements. This approach is useful, and, in many cases, sufficient. But another search technique, called the search for representation, employs a semantic model. Often the inclusion of some model of the intended interpretation of the problem domain (to act as a filter or guide for using the more syntactic procedures) can improve the algorithms by eliminating unlikely solution paths. Few programs have successfully included the use of a semantic model. A good example of one which does use a semantic model is Gelernter's (1963) geometry theorem-proving program. This program considers as candidates for subproblems only those statements that are true about a diagram-like representation of the problem.

2. Existing Theorem-Proving Programs Are Not Appropriate as Proof-Analyzers

As stated in the introduction to this chapter, the computer-tutor can obtain information for a dialogue from a knowledgeable proof-analyzer, one which can (a) generate solutions like those produced by students learning how to construct proofs, and (b) be adapted to the instructional system in which the students study. Can the sorts of theorem-provers described in Section 1 serve this purpose?

The resolution principle is a very powerful theorem-prover. One of the problems with this approach is that it tends to take very large proof

steps, proofs which are hard or impossible for a person to comprehend. Since the object here is not to prove assertions, but to explain to students the reasoning which leads to them, resolution-based programs are not suitable. Moreover, although the proofs might be translated to the student's response format, the tutor could not give an adequate explanation of why a next step should be taken. The reason a particular proof step was taken was due to resolution, not to rules the students are customarily taught.¹ If the tutor gives rules without reasons, the unmotivated rules may not be understood.

A suggestive trait of some heuristic programs is that they perform in a manner like that used by humans, and so are likely to produce results similar to those achieved by the human problem-solver. A reasonable sort of heuristic cannot aim at unfailing rules, but it may endeavor to study procedures typically useful in solving problems, to attempt to imitate the behavior a human mathematician displays in searching for and constructing a proof (and thus the procedures which might be useful to teach to mathematics students). The basic goal-subgoal search techniques of LT and FDS do seem to produce proofs like that illustrated in Chapter I and Appendix II, albeit in different formats. But the FDS algorithm is effective only if the theorem has a proof embodying the "line-by-line" property. FDS must start with the left side of an identity and reduce it to the right. It cannot find solutions of a "natural" sort, in which the proof combines two or more chains of inference. The algorithm used by FDS might benefit students using the University of Illinois PROOF program.

¹This is not to say that the resolution principle should not be taught to students of mathematical logic!

LT is better suited to the proof-analysis problem. Some modifications are, of course, needed so the program may govern a new domain. Perhaps the search procedure could be improved by expanding the LT similarity test. LT's choice routine is purely syntactical: it compares the goal with the list of axioms (rewrite rules) until it finds one syntactically identical with the goal. A new test could look further to examine semantic properties of the axioms. The purpose of this expansion would be to try to consider the mathematical content of the axioms and theorems, not just their surface structure.

3. Properties of an Appropriate Theorem-Prover

Hence, a different theorem-prover was required to act as proof-analyzer, one utilizing heuristic techniques of previous programs, but one especially adapted to the rules and procedures available to the students using the instructional system introduced in Chapter I. This new theorem-prover has the following properties.

1. The structure of the goal (theorem) and axioms are examined to symbolically manipulate one of the axioms (the initial statement) until the goal is obtained.
2. Differences between axioms or theorems and goals are detected.
3. Subproblems are set up to be proved, mirroring the use of subsidiary derivations, which attempt to reduce the differences.
4. The size of the steps of a proof are reasonable so that the students can definitely produce these steps.
5. The heuristics employed to determine the proof steps correspond to reasons which may occur to the students for taking

such steps, reasons of pattern recognition and comparison; thus the theorem-prover takes steps for reasons which the students may be taught to use.

The theorem-prover formulated resembles the LP and GPS programs, which are basically ways of achieving a goal by setting up subgoals (sub-problems) whose attainment leads to the attainment of the initial goal (problem). It does not, however, benefit from GPS-like tables which tell the program which rules reduce each difference, and which differences are more difficult to resolve. The theorem-prover was designed to prove theorems of an Abelian group under addition, theorems used in the Stanford logic-algebra curriculum (see Appendix I). Proofs for these theorems are given in Appendix II as examples of the style of proofs produced by the prover, a style which reflects that used by the students of the Stanford program in that the only rules used are those available to the students.

One aspect of this style is that the students are not given the statement to be transformed into the desired expression, i.e., the initial statement. Rather, they must figure out for themselves some instance of an axiom or an established theorem which can be so manipulated. Moreover, necessary transformations may require establishment of an expression which in turn requires choosing an initial statement, and so the recursive nature of the deduction proceeds. Thus, the theorem-prover uses a simple syntactic algorithm, modified by a semantic filter, or FEATURE TEST, for choosing these initial statements.

Another aspect of the proof style is that of working forward. By "working forward" is meant that the algorithm chooses an initial statement and tries to transform it into the statement to be proven. If the initial

statement is a true one and each transformation preserves truth, then the sequence of transformations constitutes a proof. However, if a sound choice method for picking the initial statement is not known, working forward can be more arbitrary than other heuristics.

The instructional system requires the students to present a proof resembling a working-forward approach to solving the problem. This is not to say that the students actually solve the problem in this manner, but only that the end product looks as though they did. The distinction is significant in the case where the student must do all his work on the computer. If he works backward, as does the LIT program, then by the time he begins typing, he should have a clear idea of how to complete the proof. Frequent examples of proofs where the student began the derivation with statements which will actually lead to a solution, but the student "got stuck" and had to request help, suggests that working backward is not always a sufficient search technique.

Norton's experience in designing a program to prove some theorems of group theory gives evidence that a procedure which only reduces desired conclusions to establish statements is not adequate (Norton, 1966). He had to do a certain amount of working forward to counter problems in determining and evaluating variable substitutions. (Determining variable substitutions is one of the hardest parts of discovering solutions.) And so the theorem-prover combines techniques of working backward with those of working forward.

The theorem-prover's system of inference rules, the FEATURE TEST, and other heuristic techniques actually employed, are presented in full in the next chapter.

CHAPTER IV

A THEOREM-PROVER FOR ABELIAN GROUPS

The purpose of this chapter is to give a formalization of a theorem-prover of the sort projected in the preceding chapters, and to discuss its adequacies for the proof-analysis task. In particular, the theorem-prover to be described is an idealization of part of the actual computer program that was written in order to test empirically the use of a theorem-prover as a proof-analyzer. It is an algorithm for attempting to find proofs of assertions in the theory of Abelian groups by using heuristics for pattern recognition and manipulation.

We are providing a model of the essence of the theorem-prover in order to analyze what this prover can and cannot do in its proposed role of providing tutorial information. Such analyses (even of a simplified version of the system) are necessary, before the empirical tests, so that the inherent capabilities and handicaps of the basic design of the tutorial technique can be understood. Such mathematical modeling of a teaching system, although not widely employed by CAI researchers, should be an essential part of the methodology of CAI program development.

In subsequent sections, we specify the formal system. This consists of (a) a language understood by the theorem-prover and denoted by \mathcal{L} ; and (b) rules of inference used by the prover for manipulating expressions in \mathcal{L} . A special heuristic, the FEATURE TEST, is also given. This heuristic is useful in guiding the prover's search for a proof. The presentation of

the (idealized) theorem-prover is completed by specifying an order in which the inference rules can be applied to formulas in \mathcal{L} . The theorem-prover, then, is an algorithm which tries to discover a sequence of inference rules which, when applied to formulas in \mathcal{L} , produces a proof for some designated well-formed formula.

There are basic mathematical questions to be answered about the theorem-prover, questions having to do with the prover's ability to serve as a proof-analyzer. For one, are the prover's results valid proofs, or will the instructional system be given faulty information to pass on to the students? Can a class of expressions be formulated for which the theorem-prover is guaranteed to find a proof? This is of both practical and theoretical interest. From a practical point of view, knowing what the theorem-prover can do can help in determining problems to give the students, with some assurance that the program can be used to tutor a student working on those problems. There are methods, of course, for circumventing situations in which the theorem-prover cannot find a solution; these are presented in the next chapter. But knowing the form of provable formulas permits use of the theorem-prover on problems suggested by the students themselves.

The language and the inference rules used by the theorem-prover are presented in the first sections of this chapter. Definitions for provability and truth in the system are followed by a demonstration that if a formula is provable by the theorem-prover, then it is indeed true. This requires proof that each rule only permits an inference from true statements to a true statement. It is then shown that the set of rules and proof methods is incomplete with respect to all theorems of the theory

of Abelian groups, and that there exists a subset of true expressions of that theory for which proofs can always be constructed by the theorem-prover.

The seventh and final section of this chapter is a more informal presentation of heuristics included in the actual computer program. The desire to use heuristics as a means of decreasing the tutor's response time and as a way of producing proofs that capture some "teachable tricks" was discussed in an earlier chapter. We note here that even the idealized theorem-prover includes such a heuristic, in the choice of initial patterns, which limits the number of axioms considered in order to limit the amount of time spent searching for a solution. Quick response time, so that the computer can retain the student's attention, is, of course, an essential ingredient of a CAI system.

This chapter, as well as Appendix II, contains examples that illustrate proofs found by the theorem-prover; the examples in Appendix II are printed in a format resembling that used by students of the Stanford logic-algebra program.

1. The Language \mathcal{L}

We define the language \mathcal{L} used by the theorem-prover to be the set of all expressions generated by the following grammar W . W is a non-ambiguous context-free grammar with the set of terminal symbols

$\mathcal{A} = \{A, B, C, D, =, \Rightarrow, +, -, ', \neg, O, (,)\}$; the nonterminal set $\{F, V, T\}$; and the following production rules:

W1. $V \rightarrow A$	W8. $T \rightarrow (T + T)$
W2. $V \rightarrow B$	W9. $T \rightarrow (T - T)$
W3. $V \rightarrow C$	W10. $T \rightarrow (-T)$
W4. $V \rightarrow D$	W11. $F \rightarrow (T = T)$
W5. $V \rightarrow V'$	W12. $F \rightarrow (F \Rightarrow F)$
W6. $T \rightarrow 0$	W13. $F \rightarrow (\neg F)$
W7. $T \rightarrow V$	

Note that F , V , and T are read as formula, variable, and term, respectively. We introduce the following notation:

\mathcal{S}^+ is the set of all nonempty strings of terminal symbols of finite length;

$\mathcal{L}_V = \{\alpha \in \mathcal{S}^+ \mid V \xrightarrow{W} \alpha\}$, the set of all variables generated by W if V is the starting symbol;

$\mathcal{L}_T = \{\alpha \in \mathcal{S}^+ \mid T \xrightarrow{W} \alpha\}$, the set of all terms generated by W if T is the starting symbol;

$\mathcal{L}_F = \{\alpha \in \mathcal{S}^+ \mid F \xrightarrow{W} \alpha\}$, the set of all formulas generated by W if F is the starting symbol; and

$$\mathcal{L} = \mathcal{L}_V \cup \mathcal{L}_T \cup \mathcal{L}_F .$$

Note that we will use the words "formula" and "well-formed formula" synonymously in referring to elements of \mathcal{L}_F . The expressions listed below are examples of well-formed formulas. They are used in subsequent sections and referred to by the given name. They are, in fact, the axioms for Abelian group theory.

$$\text{CA } ((A + B) = (B + A))$$

$$\text{AS } (((A + B) + C) = (A + (B + C)))$$

$$AI \quad ((A + (-A)) = 0)$$

$$Z \quad ((A + 0) = A)$$

$$N \quad ((A + (-B)) = (A - B))$$

2. Some Preliminary Definitions

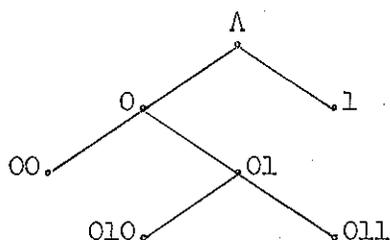
In this section we introduce some notation and definitions in order to specify the seven inference rules. As a parsing algorithm is not an interesting part of the theorem-prover, we can simply assume that expressions can be rewritten, by the program, into a derivation tree for that expression.

A binary tree is a subset $T \subseteq \{0,1\}^*$ such that (a) $w1 \in T \Rightarrow w0 \in T$, and (b) $wn \in T, n \in \{0,1\}^* \Rightarrow w \in T$. We let $\Lambda \in T$ denote the empty word, and $\#S$ denote the cardinality of any set S . Intuitively, the elements of T are the nodes of the tree, and $\#T$ is the number of nodes in T . The following vocabulary will be used for trees.

1. Λ , the empty word, is the root of the tree;
2. If $w \in T$ and $wn \in T$ for all $n \in \{0,1\}^*$, then wn is a descendent of w (note that, under this definition, w is a descendent of w);
3. If $w, w0 \in T$, then $w0$ is the left direct descendent of w ;
4. If $w, w1 \in T$, then $w1$ is the right direct descendent of w ;
5. If $w \in T$ but $w0, w1 \notin T$, then w is a leaf of T ; and
6. The subtree whose root is $w \in T$, denoted T/w , is the tree $\{n \mid n \in \{0,1\}^* \text{ and } wn \in T\}$.

An example of a binary tree is $\{\Lambda, 0, 1, 00, 01, 010, 011\}$, illustrated

as:

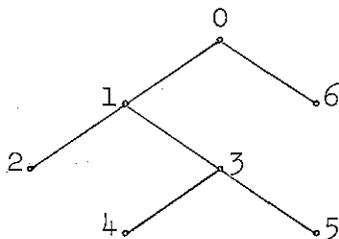


A labeled binary tree $\langle T, \ell \rangle$ is a tree T with a labeling function $\ell: T \rightarrow A$, where A is a set of symbols.

To specify the inference rules as binary tree manipulations, we give a labeling of nodes which is essentially a lexicographic ordering with $\Lambda < 0 < 1$, $L: T \rightarrow [N]$, where $\#T = N$ and $[N] = \{0, 1, 2, \dots, N-1\}$, defined by

- (i) $L(\Lambda) = 0$;
- (ii) If $w, w0 \in T$, then $L(w0) = 1 + L(w)$;
- (iii) If $w, w1 \in T$, then $L(w1) = L(w0) + \#(T/w0)$.

We call $w \in T$, where $L(w) = \#T$, the rightmost leaf of the tree. For the tree illustrated above, this labeling is:

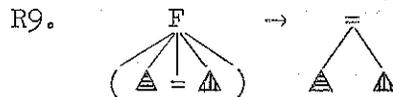
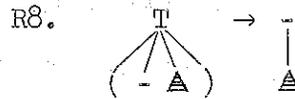
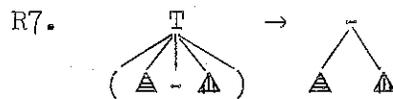
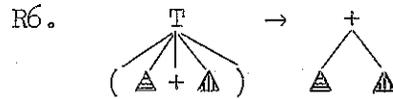
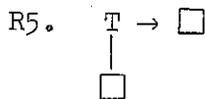
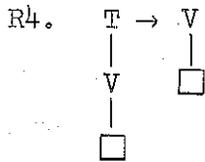
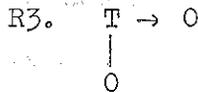
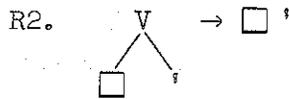
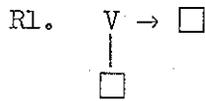


Henceforth, we feel free to refer to $w \in T$ by its label $L(w)$.

The theorem-prover manipulates "patterns" of expressions in \mathcal{L} . We will define a pattern of $E \in \mathcal{L}$ to be a simplification of the derivation

tree of E to an "operator-operand" structure. The pattern $P(E)$ will, of course, be a labeled binary tree $\langle T, l \rangle$ with $l: T \rightarrow \mathcal{L}$.

The subtree manipulation or replacement system used to obtain the pattern for E consists of an infinite set of subtree replacement rules generated from the finite specifications presented below as rule-schemata R1-R11. Note that we present the rules as diagrams since this is somewhat clearer and more intuitive than a formal algorithm for pruning and replacing nodes of the tree. The diagrams use the notation introduced by Rosen (1971) of hatched triangles and squares for "arbitrary trees"; we are to substitute any tree structure for the hatched triangles and any variable for the squares.¹ The set of all instances of a schema obtainable in this way is the set of rewrite rules generated by the schema.



¹By saying "any" we (unnecessarily) allow rules that should never be applicable to a derivation tree of $E \in \mathcal{L}$.



We "apply" rewrite rules to a tree T by first matching the left half of a rule against a subtree of T . Second, we replace that subtree by the right half of the rule so that T becomes a new tree, T' . We then try to rewrite T' , etc. Since our rules are schemata, "matching" means we see whether a subtree is an instance¹ of the left half of a rule and replace it with the corresponding instance of the right half. We continue until no more rules can be applied to the resulting tree. Rosen calls such a tree the "normal form."

Definition 1. The pattern for $E \in \mathcal{L}$ is the normal form of the derivation tree for E using subtree replacement rules R1-R11.

Since W is context-free and unambiguous, each $E \in \mathcal{L}$ has a unique derivation tree. The proof of the uniqueness of the pattern for E then follows from results on subtree replacement systems reported in Rosen. According to Rosen, the normal form is unique if the rule-schemata meet two conditions: unequivocal (set of rules generated by the rule-schemata is a partial function) and closed. The closure property says that if a rule $\phi \rightarrow \psi$ is applicable to the left half of another rule $\phi_0 \rightarrow \psi_0$, the result being ϕ_1 , then we can apply $\phi \rightarrow \psi$ to ψ_0 at appropriate places (defined by a "residue map" of subtrees of ϕ_0 to subtrees of ψ_0) to obtain ψ_1 such that the result, $\phi_1 \rightarrow \psi_1$, is also a rule of the system. (That is, every instance of every schema is a rule of the system.)

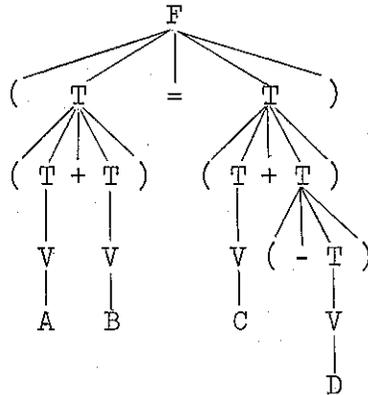
¹"Instance" is defined formally by Rosen (p. 50) and later by us in this section.

That our rule set is closed is easy to check. The resulting rule $\phi_1 \rightarrow \psi_1$ is necessarily of the same form as $\phi_0 \rightarrow \psi_0$. Since every rule of the form $\phi_0 \rightarrow \psi_0$ is in the rule set, $\phi_1 \rightarrow \psi_1$ is in it also. Furthermore, each rule is an instance of a unique rule-schema because there are no two rule-schemata such that both the left halves and the right halves are trees with identical labels at the roots.

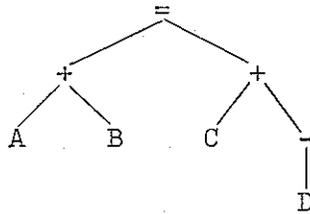
We need to show that the required residue map exists. Our replacement system fulfills the three properties set down by Rosen: (1) parameters (hatched triangles and squares) occur only at leaves of each part of a rule-schema; (2) no parameter occurs twice in the left half; and (3) every parameter that occurs in the right half also occurs in the left half. So the residue map is the natural one given by Rosen (his Lemma 1) of mapping each parameter on the left half to the identical parameters occurring in the right half. The replacement system is closed. Closure means that the chance that applying one rule might interfere with the ability to apply another is a harmless property.

It is left to show that no two rules have the same left half. But this follows from the fact that each rule is an instance of a unique rule-schema and no rule-schemata have identical left halves. So the subtree replacement system is unequivocal closed; by Rosen's Main Theorem, the pattern is unique.

As an example of a pattern, take $E = ((A + B) = (C + (-D)))$. The derivation tree in W is:



and the pattern is the tree $\{A, 0, 1, 00, 01, 10, 11, 110\}$ with labeling l :



Examples of two other expressions and their corresponding patterns are shown in Figure 1.

We will use the following definition of length to carry out proofs in the next sections.

Definition 2.

- (1) The length of the pattern for a variable in \mathcal{L}_V is:

$$\text{length}(v) = 0 \quad v \text{ is } A, B, C, \text{ or } D$$

$$\text{length}(v') = 1 + \text{length}(v) \quad v \in \mathcal{L}_V.$$

- (2) The length of the pattern for a term in \mathcal{L}_T is identical to the

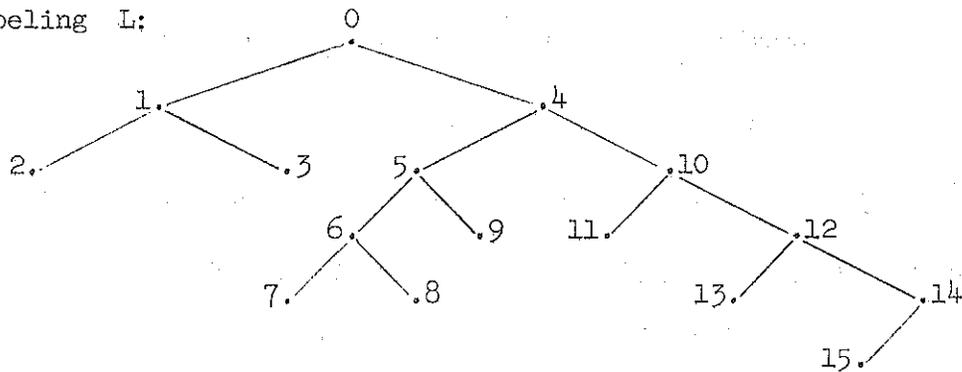
length of the term, where we define the length of a term to be:

$$\text{length}(t) = 0 \quad t \in \mathcal{L}_T$$

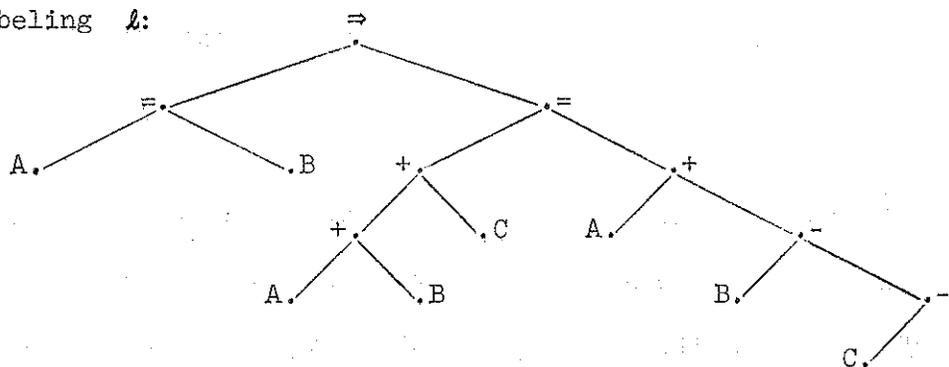
$$\text{length}(0) = 0$$

$$E_1 = ((A = B) \Rightarrow (((A + B) + C) = (A + (B - (-C)))))$$

P_1 with labeling L :

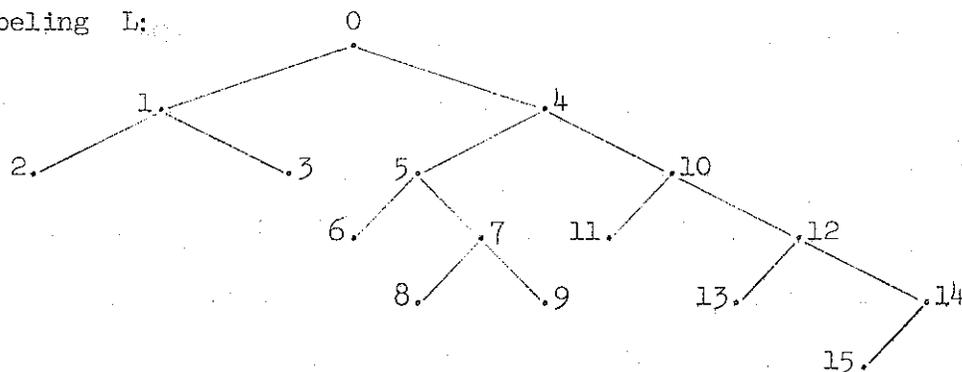


P_1 with labeling \mathcal{L} :



$$E_2 = ((A = B) \Rightarrow ((A + (B + C)) = (A + (B - (-C)))))$$

P_2 with labeling L :



P_2 with labeling \mathcal{L} :

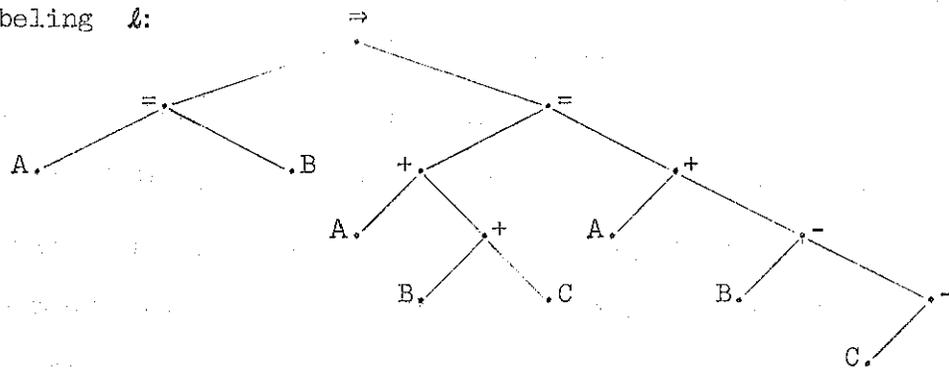


Fig. 1.--Patterns of expressions E_1 and E_2 .

$$\text{length}(t_1 + t_2) = 1 + \text{length}(t_1) + \text{length}(t_2) \quad t_1, t_2 \in \mathcal{L}_T$$

$$\text{length}(t_1 - t_2) = 1 + \text{length}(t_1) + \text{length}(t_2) \quad t_1, t_2 \in \mathcal{L}_T$$

$$\text{length}(-t) = 1 + \text{length}(t) \quad t \in \mathcal{L}_T$$

(3) The length of the pattern for a formula in \mathcal{L}_F is identical to the length of the formula, where we define the length of a formula to be:

$$\text{length}(t_1 = t_2) = 0 \quad t_1, t_2 \in \mathcal{L}_T$$

$$\text{length}(f_1 \Rightarrow f_2) = 1 + \text{length}(f_1) + \text{length}(f_2) \quad f_1, f_2 \in \mathcal{L}_F$$

$$\text{length}(\neg f) = 1 + \text{length}(f) \quad f \in \mathcal{L}_F.$$

In manipulating the patterns, the prover might replace part of a pattern or subpattern by another pattern according to some rules. Let $P = \langle T, \ell \rangle$. Then the subpattern $S(P, w)$ of P is the labeled subtree $\langle T/w, \ell' \rangle$ where $w \in T$ and ℓ' is the restriction of ℓ to T/w . Obviously, by virtue of the definition of pattern, $S(P_1, w_1) = S(P_2, w_2)$ if and only if $T_1 = T_2$ (set equality) and $\ell_1(w) = \ell_2(w)$ for all $w \in T_1 = T_2$, where P_1 is $\langle T_1, \ell_1 \rangle$ and P_2 is $\langle T_2, \ell_2 \rangle$.

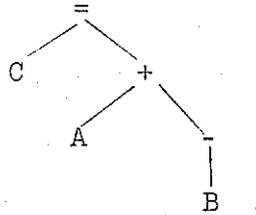
For a tree T and $w \in T$, let $w \setminus T = T - \{wn \mid n \in T/w\}$ (set subtraction). That is, $w \setminus T$ is the operation of removing subtree T/w from T . Let $T[w, T'] = w \setminus T \cup \{wn \mid n \in T'\}$, the result of removing subtree T/w from T and replacing it with T' .

As an example, let $T = \{\Lambda, 0, 1, 10, 11, 110\}$, and $T' = \{\Lambda, 0, 1\}$. Then $T/1 = \{\Lambda, 0, 1, 10\}$; $1 \setminus T = T - \{1n \mid n \in T/1\} = T - \{1, 10, 11, 110\} = \{\Lambda, 0\}$. $T[1, T'] = \{\Lambda, 0\} \cup \{1n \mid n \in T'\} = \{\Lambda, 0\} \cup \{1, 10, 11\} = \{\Lambda, 0, 1, 10, 11\}$.

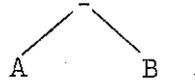
We can easily extend this notation to define what we mean by the substitution of a pattern for a subpattern, $P[w, P']$. Let $P = \langle T, \ell \rangle$,

and $P' = \langle T', \ell' \rangle$. Then $P[w, P'] = \langle T'', \ell'' \rangle$, where $w \in T$, $T'' = T[w, T']$, $\ell''(w_i) = \ell(w_i)$ for all $w_i \in w \setminus T''$, and $\ell''(w_i) = \ell'(w_i)$ for all $w_i \in \{wn \mid n \in T'\} = T''/w$.

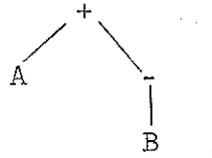
Continuing the above example, let E be $(C = (A + (-B)))$. Then P is



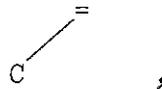
Let P' be



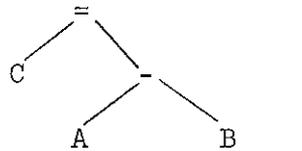
Clearly, $P = \langle T, \ell \rangle$ and $P' = \langle T', \ell' \rangle$ for T and T' shown in the above example. Then, for $l \in T$, P/l is



$l \setminus P$ is



and $P[l, P']$ is



To define what we mean by a substitution instance of P , we must introduce some more notation. We call an ordered sequence $\Sigma = \langle w_1, T_1 \rangle \langle w_2, T_2 \rangle \dots \langle w_m, T_m \rangle$ a substitution sequence if the w_i 's are all distinct words (nodes) and the T_i 's are trees. Then

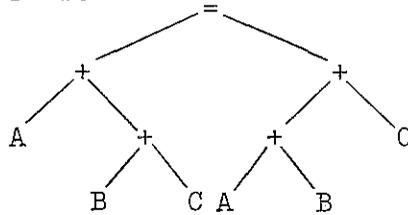
$T[\Sigma] = T[w_1, T_1][w_2, T_2] \dots [w_m, T_m]$, where $w_{i+1} \in T[w_1, T_1] \dots [w_i, T_i]$, $1 \leq i < m$. If, for some i , $w_{i+1} \notin T[w_1, T_1] \dots [w_i, T_i]$, then $T[\Sigma]$ is not defined.

The extension of the notation to patterns is obvious. We let $\Sigma = \langle w_1, P_1 \rangle \langle w_2, P_2 \rangle \dots \langle w_m, P_m \rangle$, where P_i is the pattern for some expression in \mathcal{L} . Then $P[\Sigma] = P[w_1, P_1][w_2, P_2] \dots [w_m, P_m]$, where $w_{i+1} \in P[w_1, P_1] \dots [w_i, P_i]$, $1 \leq i < m$. Again, if, for some i , $w_{i+1} \notin P[w_1, P_1] \dots [w_i, P_i]$, then $P[\Sigma]$ is not defined.

Suppose $P = \langle T, \ell \rangle$ and consider a sequence $\sigma = \langle v_1, P_1 \rangle \langle v_2, P_2 \rangle \dots \langle v_k, P_k \rangle$ such that $v_i \in \mathcal{L}_v$ and P_i is the pattern of some term in \mathcal{L}_T . We want to specify a substitution sequence Σ_i which will effect the substitution of each occurrence of v_i in P by P_i .

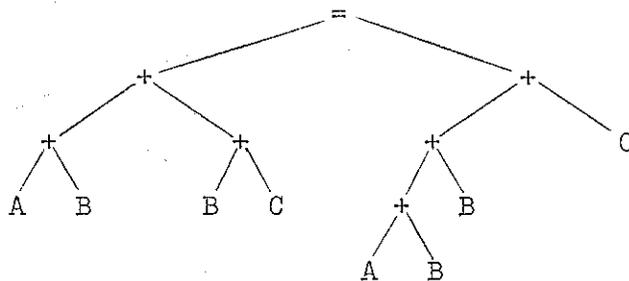
(*) For $\ell^{-1}(v_i) = \{w_1, \dots, w_m\}$, with the w_i 's distinct and lexicographically ordered, let Σ_i be the sequence $\langle w_1, P_i \rangle \dots \langle w_m, P_i \rangle$.

As an example, let P be



(So $T = \{A, 0, 1, 00, 01, 010, 011, 10, 11, 100, 101\}$). Let $\sigma = \langle A, (A + B) \rangle \langle B, (C + D) \rangle \langle C, A \rangle$. Then, $v_1 = A$, $\ell^{-1}(v_1) = \{00, 100\}$, and $\Sigma_1 = \langle 00, (A + B) \rangle \langle 100, (A + B) \rangle$. The substitution $P[\Sigma_1]$ results in the replacement of all occurrences of A in P by $(A + B)$.

That is, $P[\Sigma_1]$ is



If $\sigma = \langle v_1, P_1 \rangle \dots \langle v_k, P_k \rangle$ meets certain restrictions, we can concatenate all sequences Σ_i , $1 \leq i \leq k$, in order to obtain a substitution instance of P .

Definition 3. Suppose $P = \langle T, \ell \rangle$ and $\sigma = \langle v_1, P_1 \rangle \langle v_2, P_2 \rangle \dots \langle v_k, P_k \rangle$ such that, for $1 \leq i \leq k$,

- (i) for each i , $v_i \in \mathcal{L}_V$, P_i is the pattern for some term in \mathcal{L}_T ;
 - (ii) for each leaf w of T , there is an i such that $\langle \ell(w), P_i \rangle$ occurs in σ ; and
 - (iii) if $v_i = v_j$ for $\langle v_i, P_i \rangle, \langle v_j, P_j \rangle$ occurring in σ , then $P_i = P_j$.
- Let Σ_i , for $1 \leq i \leq k$, be defined as above (*). Then, for $\Sigma = \Sigma_1 \Sigma_2 \dots \Sigma_k$, the concatenation of all Σ_i , $P[\sigma] = P[\Sigma]$ is a substitution instance of P .

Continuing the above example, we have

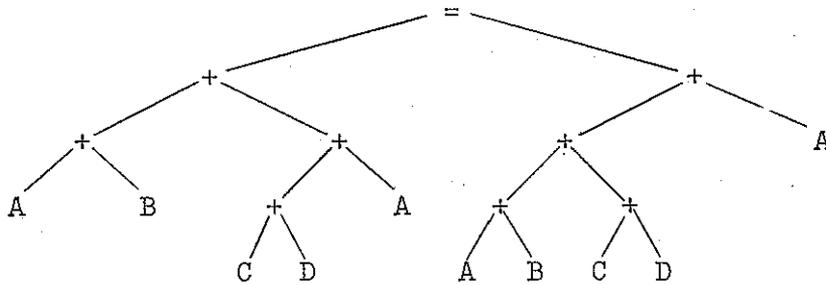
$$1. v_2 = B, \Sigma_2 = \langle 010, (C + D) \rangle \langle 101, (C + D) \rangle;$$

$$2. v_3 = C, \Sigma_3 = \langle 011, A \rangle \langle 11, A \rangle;$$

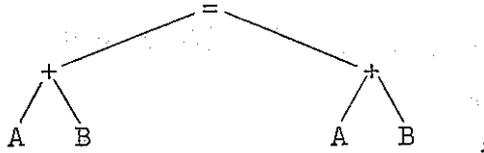
$$3. \Sigma = \Sigma_1 \Sigma_2 \Sigma_3; \text{ and}$$

$$4. P[\sigma] = P[\Sigma], \text{ a substitution instance of the associativity axiom}$$

for addition, is



General substitution is affected by the order in which $\langle w_i, P_i \rangle$ are specified. (A quick check of $P[\Sigma]$ where P is



$\Sigma = \langle 01, (A + B) \rangle \langle 0, E \rangle$ and then $\Sigma = \langle 0, E \rangle \langle 01, (A + B) \rangle$ convinces one of this point. The first substitution is defined, the second is not.) However, the construction of Σ in Definition 3 depends on the fact that the order in which pairs $\langle w_i, P_i \rangle$ appear in Σ does not change the result.

Theorem 1. The order in which pairs $\langle w_i, P_i \rangle$ appear in the substitution sequence $\Sigma = \langle w_1, P_1 \rangle \dots \langle w_m, P_m \rangle$ does not change the substitution result.

Proof. We show that, for any adjacent pairs in Σ , $\langle w_i, P_i \rangle$ and $\langle w_j, P_j \rangle$, $P[w_i, P_i][w_j, P_j] = P[w_j, P_j][w_i, P_i]$.
 Since w_i is a leaf of T , $T/w_i = \{w_j\}$; $T[w_i, T_i] = w_i \setminus T \cup \{w_i^n \mid n \in T_i\}$
 $= (T - \{w_i\}) \cup \{w_i^n \mid n \in T_i\}$. The corresponding identities hold for w_j .
 The sets $\{w_i^n \mid n \in T_i\}$ and $\{w_j^n \mid n \in T_j\}$ are disjoint because (a) $w_i \neq w_j$ (Definition 3), and (b) for all $n \in T_j$, w_j^n cannot begin with a sequence

equal to w_i (nor w_i^n with a w_j) since such an equality would imply that w_i (w_j) is not a leaf.

We will use the following result in the proof:

If T, A, B are sets, $A \subseteq B$, then $(T - A) \cup B = T \cup B$.

$$\begin{aligned}
 (T - A) \cup B &= \{x \mid (x \in T \text{ and } x \notin A) \text{ or } x \in B\} \\
 &= \{x \mid (x \notin B \text{ and } x \in T \text{ and } x \notin A) \text{ or } x \in B\} \\
 &= \{x \mid (x \notin B \text{ and } x \in T) \text{ or } x \in B\} \text{ since } x \notin B \Rightarrow x \notin A \\
 &= \{x \mid x \in T \text{ or } x \in B\} \\
 &= T \cup B .
 \end{aligned}$$

We can now show that any transposition of adjacent pairs in Σ yields a substitution sequence Σ' such that $P[\Sigma] = P[\Sigma']$. Since such transpositions generate the group of all permutations, we can then conclude that any permutation Σ' of Σ yields the same substitution result.

Part 1.

$$\begin{aligned}
 T[w_i, T_i] &= (T - \{w_i\}) \cup \{w_i^n \mid n \in T_i\} = T \cup \{w_i^n \mid n \in T_i\} \\
 T[w_i, T_i][w_j, T_j] &= ((T \cup \{w_i^n \mid n \in T_i\}) - \{w_j\}) \cup \{w_j^n \mid n \in T_j\} \\
 &= (T \cup \{w_i^n \mid n \in T_i\}) \cup \{w_j^n \mid n \in T_j\} \\
 &= (T \cup \{w_j^n \mid n \in T_j\}) \cup \{w_i^n \mid n \in T_i\} \\
 &= [(T - \{w_j\}) \cup \{w_j^n \mid n \in T_j\}] \cup \{w_i^n \mid n \in T_i\} \\
 &= T[w_j, T_j] \cup \{w_i^n \mid n \in T_i\} \\
 &= (T[w_j, T_j] - \{w_i\}) \cup \{w_i^n \mid n \in T_i\} \\
 &= T[w_j, T_j][w_i, T_i] .
 \end{aligned}$$

Part 2.

Let $P^1 = P[w_i, P_i] = \langle T^1, l^1 \rangle$.

Then $l^1(w) = l(w)$ for all $w \in T - \{w_i\}$,

$l^1(w) = l_i(w)$ for all $w \in \{w_i n \mid n \in T_i\}$.

Let $P^2 = P^1[w_j, P_j] = \langle T^2, l^2 \rangle$.

Then $l^2(w) = l^1(w)$ for all $w \in T^1 - \{w_j\}$,

$l^2(w) = l_j(w)$ for all $w \in \{w_j n \mid n \in T_j\}$.

That is,

$l^2(w) = l(w)$ for all $w \in T - \{w_i\} - \{w_j\}$,

$l^2(w) = l_i(w)$ for all $w \in \{w_i n \mid n \in T_i\}$,

$l^2(w) = l_j(w)$ for all $w \in \{w_j n \mid n \in T_j\}$,

because the sets $T - \{w_i\} - \{w_j\}$, $\{w_i n \mid n \in T_i\}$, $\{w_j n \mid n \in T_j\}$ are disjoint. We get the same result when we carry out the above analysis

for $P[w_j, P_j][w_i, P_i]$. ▣

3. The Rules of Inference

The theorem-prover has both a language \mathcal{L} and a set of rules for obtaining patterns of some elements of \mathcal{L} from other patterns. The rules used by the (idealized) theorem-prover are presented in this section. Each rule operates on patterns in order to produce a new pattern. In the following definitions of the rules, let Φ be a (possibly empty) sequence of rules, ψ a sequence of patterns. We will use $\Phi_{[n]}$ to denote a sequence of n rules; $\Phi_{[0]}$ is the empty sequence (with similar meanings for $\psi_{[n]}$ and $\psi_{[0]}$). The i th rule in the sequence is Φ_i ; the pattern obtained by application of rule Φ_i (with respect to $\Phi_{[i-1]}$) is denoted by

$\psi_i = \langle \Gamma_i, \mathcal{L}_i \rangle$. The asterisk, $*$, denotes the operation of concatenating an element onto a sequence of n elements.

Let AX be a set of patterns for elements of \mathcal{L}_F . Elements of AX are called axioms. Each of the seven inference rules has four parts: the first part adds the rule to a given sequence of rules Φ ; the second defines the pattern obtained by application of that rule; the third specifies the set A , elements of which refer to patterns commonly called assumptions or premises; and the fourth defines the set DA , elements of which refer to "discharged" patterns. An element of DA either refers to (a) a premise which has been used as the antecedent in forming a material conditional, or (b) a rule whose justification for being in the sequence of rules depends on a premise such as that described in (a). Presence in the set DA indicates that a rule can no longer be referenced by subsequent applications of the inference rules. Each node w is referred to below by its label $L(w)$; $A_0 = DA_0 = \emptyset$.

IR1. SUBSTITUTION $P[\sigma]$

If $P[\sigma]$ is defined for some substitution sequence σ and $P \in AX$, then

$$\Phi_{[n+1]} = \Phi_{[n]} * \text{SUB}(\sigma, P),$$

$$\psi_{[n+1]} = \psi_{[n]} * P[\sigma],$$

$$A_{n+1} = A_n, \text{ and}$$

$$DA_{n+1} = DA_n.$$

The second inference rule, the replacement rule, allows us to obtain a new pattern from patterns ψ_i and ψ_j by replacing the k th subpattern in ψ_i by the right-hand side of the identity ψ_j . Such a replacement

is only permitted if the left-hand side of ψ_j is identical to the subpattern.

IR2. REPLACEMENT RE(i,j,k)

If $i, j \leq n$, $i, j \notin DA_n$, $k \in T_i$, $\ell_j(0)$ is the identity symbol $=$, and $S(\psi_i, k) = S(\psi_j, l)$, then

$$\Phi_{[n+1]} = \Phi_{[n]} * \text{RE}(i, j, k),$$

$$\Psi_{[n+1]} = \Psi_{[n]} * \psi_i[k, S(\psi_j, m)], \quad \text{for } m \text{ the right direct descendent of the root of } T_j,$$

$$A_{n+1} = A_n, \quad \text{and}$$

$$DA_{n+1} = DA_n.$$

Observe that we cannot add a new rule to the sequence if the inference depends on a rule referenced in the set DA .

IR3. COMMUTE EQUALS CE(i)

If $i \leq n$, $\ell_i(0)$ is $=$, and $i \notin DA_n$, then

$$\Phi_{[n+1]} = \Phi_{[n]} * \text{CE}(i),$$

$$\Psi_{[n+1]} = \Psi_{[n]} * \psi_i[\Sigma],$$

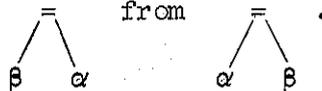
where, for $\alpha = S(\psi_i, l)$ and

$$\beta = S(\psi_i, l + \#\alpha), \quad \Sigma = \langle l, \beta \rangle \langle l + \#\beta, \alpha \rangle,$$

$$A_{n+1} = A_n, \quad \text{and}$$

$$DA_{n+1} = DA_n.$$

In pictures, IR3 forms the pattern



IR4. DETACHMENT DE(i,j)

If $i, j \leq n$, $i, j \notin DA_n$, $\mathcal{L}_i(0)$ is \Rightarrow , and $\psi_j = S(\psi_i, l)$, then

$$\Phi_{[n+1]} = \Phi_{[n]} * DE(i, j),$$

$$\Psi_{[n+1]} = \Psi_{[n]} * S(\psi_i, k),$$

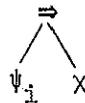
for k the right direct descendent
of the root of T_i ,

$$A_{n+1} = A_n, \text{ and}$$

$$DA_{n+1} = DA_n.$$

Inference Rule 4 is just the classical law of detachment which says that

from patterns ψ_j in the form \Rightarrow and ψ_i we can infer χ .

IR5. REPETITION REP(i)

If $i \leq n$ and $i \notin DA_n$, then

$$\Phi_{[n+1]} = \Phi_{[n]} * REP(i),$$

$$\Psi_{[n+1]} = \Psi_{[n]} * \psi_i,$$

$$A_{n+1} = A_n, \text{ and}$$

$$DA_{n+1} = DA_n.$$

None of the above rules changes the set of premises, A , nor the set of "discharged" premises, DA . The next rule allows us to introduce a new pattern as an assumption. This premise can be discharged by forming a conditional statement from it and the last pattern in the sequence (IR7).

IR6. ASSUMPTION AR(P)

If P is a pattern for some formula in \mathcal{L}_P , then

$$\Phi_{[n+1]} = \Phi_{[n]} * AR(P),$$

$$\psi_{[n+1]} = \psi_{[n]} * P,$$

$$A_{n+1} = A_n \cup \{n+1\}, \text{ and}$$

$$DA_{n+1} = DA_n.$$

IR7. CONDITIONAL PROOF CP(i,n)

If $i \leq n$, such that $i \in A_n$ (i.e., ψ_i is a premise), then

$$\phi_{[n+1]} = \phi_{[n]} * CP(i,n),$$

$$\psi_{[n+1]} = \psi_{[n]} * P,$$

where $S(P,1) = \psi_i$, $\ell_{n+1}(0)$ is \Rightarrow ,

and $S(P,1 + \#\psi_i) = \psi_n$,

$$A_{n+1} = A_n - \{i\}, \text{ and}$$

$$DA_{n+1} = DA_n \cup \{j \mid i \in DEP(j)\},$$

where we define $DEP(j)$, the set of all $i \in A_n$, such that ϕ_j derives (indirectly or directly) from i ,

as follows:

If ϕ_j is IRL, $DEP(j) = \phi$;

If ϕ_j is CE(k) or REP(k),

$$DEP(j) = DEP(k);$$

If ϕ_j is RE(k,m,n) or DE(k,m),

$$DEP(j) = DEP(k) \cup DEP(m);$$

If ϕ_j is AR(P), $DEP(j) = \{j\}$; and

If ϕ_j is CP(k,m), $DEP(j) = DEP(m)$

$$\cup DEP(k).$$

IR7, the deduction rule, states that we can take an assumption made (a premise referenced in the set A_n) and form a conditional statement from it and the pattern specified by the last rule in the sequence. We must

then discharge the assumption from this active set and add the reference to it, as well as to all the rules depending on it, to an inactive set, DA.

Given the seven inference rules generating the pattern sequence ψ , we can now define what we mean by a proof of a formula $E \in \mathcal{L}$.

Definition 4. $\phi_{[n]}$ is a derivation of ψ_n from $\{\psi_i \mid i \in A_n\}$ if and only if, for each i , $1 \leq i \leq n$, ϕ_i is one of the inference rules IR1-IR7.

Definition 5. $\phi_{[n]}$ is a proof of ψ_n if and only if $\phi_{[n]}$ is a derivation of ψ_n from ϕ (i.e., there are no active assumptions in $\phi_{[n]}$).

Definition 6. $E \in \mathcal{L}$ is provable if and only if there exists a $\phi_{[n]}$ such that $\phi_{[n]}$ is a proof of $P(E)$.

4. Truth in \mathcal{L}

A structure M for the language \mathcal{L} is an ordered 2-tuple $\langle D, R \rangle$. D , the domain of discourse, is a nonempty set; R is a function which assigns to each n -place predicate and n -place operation symbol in \mathcal{L} an n -place relation and n -place function defined on D , respectively. The truth of a well-formed formula in \mathcal{L} in M is defined through the notion of satisfaction in M , which is a relation between assignments and formulas. In this section, we will define what is meant by truth for a pattern and prove that, if $E \in \mathcal{L}$ is provable, then E is true in M . The completeness theorem for logic then tells us that the theorem-prover will not construct a proof for an untrue statement.

Definition 7. Let $M = \langle D, R \rangle$ be a structure for \mathcal{L} . Then an assignment on M is a function $g: \mathcal{L}_V \rightarrow D$.

Definition 8. Let $M = \langle D, R \rangle$ be a structure for \mathcal{L} , and g be an assignment on M . Then the extension of g to terms, $g_T: \mathcal{L}_T \rightarrow D$, is defined by:

$$\begin{aligned} g_T(t) &= g(t) && \text{if } t \in \mathcal{L}_V, \\ &= R(t) && \text{if } t \text{ is } 0, \\ &= R(f)(g_T(t_1) \dots g_T(t_n)) && \text{if } t \text{ is } (ft_1 \dots t_n). \end{aligned}$$

In the examples we will use in this chapter, f is one of $+$, $-$, or unary $-$, and $R(f)$ is just the usual arithmetical functions.

Definition 9. Let $M = \langle D, R \rangle$ be a structure for \mathcal{L} , and g be an assignment on M . Then the extension of g to formulas (satisfaction of a formula by g), $g_F: \mathcal{L}_F \rightarrow \{0, 1\}$, is defined by:

$$\begin{aligned} \text{(i)} \quad g_F(t_1 = t_2) &= 1 \text{ if and only if } g_T(t_1) = g_T(t_2), \quad t_1, t_2 \in \mathcal{L}_T; \\ &= 0 \text{ otherwise.} \\ \text{(ii)} \quad g_F(f_1 \Rightarrow f_2) &= 1 \text{ if and only if } g_F(f_1) = 0 \text{ or } g_F(f_2) = 1, \\ & \quad f_1, f_2 \in \mathcal{L}_F; \\ &= 0 \text{ otherwise.} \\ \text{(iii)} \quad g_F(\neg f_1) &= 1 \text{ if and only if } g_F(f_1) = 0, \quad f_1 \in \mathcal{L}_F; \\ &= 0 \text{ otherwise.} \end{aligned}$$

If $g_F(\theta) = 1$, we say that θ is satisfiable in M , and that g satisfies θ . In the next definition, we extend Definitions 7, 8, and 9 to patterns.

Definition 10. Suppose $E \in \mathcal{L}$, and $M = \langle D, R \rangle$ is a structure for \mathcal{L} . Then

$$g(P(E)) = g(E) \quad \text{for } E \in \mathcal{L}_V,$$

$$g_T(P(E)) = g_T(E) \quad \text{for } E \in \mathcal{L}_T,$$

$$g_F(P(E)) = g_F(E) \quad \text{for } E \in \mathcal{L}_F.$$

That is, a pattern $P(E)$, $E \in \mathcal{L}_F$, is satisfiable in a structure $M = \langle D, R \rangle$ for \mathcal{L} if and only if E is satisfiable in M .

Definition 11. A formula E is true in a structure $M = \langle D, R \rangle$ for \mathcal{L} if every assignment on M satisfies E in M . A pattern $P(E)$ is true in a structure $M = \langle D, R \rangle$ for \mathcal{L} if and only if E is true in M .

Definition 12. Let A be a set of formulas in \mathcal{L}_F , $M = \langle D, R \rangle$ be a structure for \mathcal{L} . Then M is a model for A if and only if every element of A is true in M . Similarly, if AX is a set of patterns for formulas in \mathcal{L}_F , then M is a model for AX if and only if every element of AX is true in M .

Our stated intention in this section is to show that if a formula in \mathcal{L} is provable from AX (the axioms), then it is true in every model for AX . The arguments, which are based on satisfaction of patterns, follow easily if we prove several lemmas. In the following, let $M = \langle D, R \rangle$ be a model for AX , and g be an assignment on M .

Lemma 1. If $A \in \mathcal{L}_F$ is true in M , $\sigma = \langle v_1, P_1 \rangle \langle v_2, P_2 \rangle \dots \langle v_n, P_n \rangle$ such that $A[\sigma]$ is a substitution instance of A , then $A[\sigma]$ is true in M .
[IR1, SUBSTITUTION RULE]

Proof. The proof is by induction on the length of A (Definition 2).

Atomic case ($\text{length}(A) = 0$). A is in the form $(\alpha = \beta)$, and α, β are patterns for terms in \mathcal{L}_T ; $A[\sigma]$ is in the form $(\alpha^* = \beta^*)$, for $\alpha^* = \alpha[\sigma]$ and $\beta^* = \beta[\sigma]$. By Definition 9, $g_F(A) = 1$ only if

$g_{\mathbb{T}}(\alpha) = g_{\mathbb{T}}(\beta)$. To show $A[\sigma]$ is true, we prove any assignment on M satisfies $A[\sigma]$. Pick any assignment h on M and define the assignment g on M , g satisfies A , by:

- (1) $g_{\mathbb{T}}(v) = h_{\mathbb{T}}(P)$ if $\langle v, P \rangle$ is an item in σ , and
- (2) $g_{\mathbb{T}}(v) = h_{\mathbb{T}}(v)$ for all other v the pattern for a variable

in \mathcal{L}_V .

We have to show that $g_{\mathbb{T}}(\alpha) = g_{\mathbb{T}}(\beta)$ implies $h_{\mathbb{T}}(\alpha^*) = h_{\mathbb{T}}(\beta^*)$. To do so, we first show that $g_{\mathbb{T}}(t) = h_{\mathbb{T}}(t^*)$ for any t the pattern of a term in $\mathcal{L}_{\mathbb{T}}$ and $t^* = t[\sigma]$.

Subproof. This part of the proof is by induction on the length of a pattern for a term.

Atomic case ($\text{length}(t) = 0$). Either t is the pattern for a variable in \mathcal{L}_V or t is 0 . In the first case, if the pair $\langle t, P \rangle$ is an item in σ , t^* is P . By (1) above, $g_{\mathbb{T}}(t) = h_{\mathbb{T}}(P) = h_{\mathbb{T}}(t^*)$. Otherwise, t^* is t and, by (2), $g_{\mathbb{T}}(t) = h_{\mathbb{T}}(t^*)$. If t is 0 , then t^* is also 0 since no pair $\langle 0, P \rangle$ is an item in σ . By (2), $g_{\mathbb{T}}(t) = h_{\mathbb{T}}(t^*)$.

Inductive hypothesis. Assume $g_{\mathbb{T}}(s) = h_{\mathbb{T}}(s^*)$ for all patterns of terms in $\mathcal{L}_{\mathbb{T}}$, $s^* = s[\sigma]$ a substitution instance of s , and $\text{length}(s) < \text{length}(t)$.

For $1 \leq i \leq n$, t is in the form $(fs_1 \dots s_n)$, where $\text{length}(s_i) < \text{length}(t)$; t^* is in the form $(fs_1^* \dots s_n^*)$, where $s_i^* = s_i[\sigma]$. By Definition 8, $g_{\mathbb{T}}(t) = R(f)(g_{\mathbb{T}}(s_1) \dots g_{\mathbb{T}}(s_n))$ and $h_{\mathbb{T}}(t^*) = R(f)(h_{\mathbb{T}}(s_1[\sigma]) \dots h_{\mathbb{T}}(s_n[\sigma]))$. By the inductive hypothesis, $g_{\mathbb{T}}(s_i) = h_{\mathbb{T}}(s_i[\sigma])$, for $1 \leq i \leq n$. Therefore, substituting $h_{\mathbb{T}}(s_i[\sigma])$ for $g_{\mathbb{T}}(s_i)$, $g_{\mathbb{T}}(t) = h_{\mathbb{T}}(t^*)$. By the induction, $g_{\mathbb{T}}(t) = h_{\mathbb{T}}(t^*)$ for

all t such that t is the pattern of a term in \mathcal{L}_T and t^* is a substitution instance of t .

Returning to the main proof, we now have that $g_T(\alpha) = h_T(\alpha^*)$ and $g_T(\beta) = h_T(\beta^*)$. By the transitivity property of identity, $h_T(\alpha^*) = h_T(\beta^*)$. Therefore, h satisfies $A[\sigma]$. Since we carried out this proof for any assignment h on M , $A[\sigma]$ is satisfied by any assignment on M and so is true in M .

Inductive hypothesis. Assume that, if $B \in \mathcal{L}_F$ is true in M , $\text{length}(B) < \text{length}(A)$, $B[\sigma']$ is a substitution instance of B for some $\sigma' = \langle v_1, P_1 \rangle \langle v_2, P_2 \rangle \dots \langle v_n, P_n \rangle$, then $B[\sigma']$ is true in M .

A is either of the form $(f_1 \Rightarrow f_2)$ or $(\neg f_1)$, where f_1, f_2 are patterns for some formulas in \mathcal{L}_F , and $\text{length}(f_1)$ and $\text{length}(f_2) < \text{length}(A)$. Suppose A is in the form $(f_1 \Rightarrow f_2)$. Then $A[\sigma]$ is in the form $(f_1[\sigma] \Rightarrow f_2[\sigma])$. If A is true, then for all assignments g on M , either $g_F(f_1) = 0$ or $g_F(f_2) = 1$. If $g_F(f_2) = 1$, then $g_F(f_2[\sigma]) = 1$ by the inductive hypothesis, and so $A[\sigma]$ is true in M . Else, $g_F(f_1) = 0$. To say $A[\sigma]$ is true in M , we have to show that $g_F(f_1[\sigma]) = 0$. Assume $g_F(f_1[\sigma]) = 1$. Clearly we can construct a σ' such that $f_1[\sigma][\sigma'] = f_1$. Since $\text{length}(f_1[\sigma]) < \text{length}(A)$, we know by the inductive hypothesis that $f_1[\sigma][\sigma']$ is true. That is, f_1 is true in M , contradicting the assumption that $g_F(f_1) = 0$. Hence, $g_F(f_1[\sigma])$ must be 0.

Suppose A is in the form $(\neg f_1)$. A is true only if $g_F(f_1) = 0$. By the same argument used above, we can show that $g_F(f_1[\sigma]) = 1$ implies the contradictory result $g_F(f_1) = 1$. Therefore, $g_F(f_1[\sigma])$ must be 0, and $A[\sigma] = (\neg f_1[\sigma])$ is true in M .

By the induction, for all $A \in \mathcal{L}_F$, A true in M , $A[\sigma]$ a substitution instance of A is true in M . ▣

Lemma 2. If g satisfies the identity P_1 , then, for $\alpha = S(P_1, 1)$ and $\beta = S(P_1, 1 + \# \alpha)$, g satisfies $P_2 = P_1[1, \# \beta][1 + \# \beta, \alpha]$. [IR3, COMMUTE EQUALS RULE]

Proof. Clearly, P_1 is in the form $(\alpha = \beta)$. By the substitution, we have that $S(P_2, 1) = \beta = S(P_1, 1 + \# \alpha)$, $S(P_2, 1 + \beta) = \alpha = S(P_1, 1)$. That is, P_2 is in the form $(\beta = \alpha)$. Since g satisfies P_1 , $g_T(\alpha) = g_T(\beta)$. By the commutability property of identity, this is identical to saying that $g_T(\beta) = g_T(\alpha)$ (i.e., g assigns the same value to both sides of the identity P_2). By Definitions 9 and 10, g satisfies P_2 . ▣

Lemma 3. Let $\eta[\Sigma]$ be defined for η the pattern for some term in \mathcal{L}_T , $\Sigma = \langle w, \beta \rangle$, $S(\eta, w) = \alpha$, such that g satisfies $(\alpha = \beta)$. Then $g_T(\eta) = g_T(\eta[\Sigma])$.

Proof. By induction on the length of η (Definition 2).

Atomic case ($\text{length}(\eta) = 0$). Either $\eta = P(t)$ for $t \in \mathcal{L}_V$ or t is 0. In both cases, because we assume that α is a subpattern of η , $\eta = \alpha = \beta$. So, $g_T(\eta) = g_T(\alpha) = g_T(\beta)$. $\eta[\Sigma] = \beta$ and, so, $g_T(\eta[\Sigma]) = g_T(\beta) = g_T(\eta)$.

Inductive hypothesis. Assume $g_T(\delta) = g_T(\delta[\Sigma])$ for $\delta[\Sigma]$ defined, $\Sigma = \langle w, \beta \rangle$, $S(\delta, w) = \alpha$, and g satisfies $(\alpha = \beta)$, such that $\text{length}(\delta) < \text{length}(\eta)$.

η is in the form $(ft_1 \dots t_n)$. By Definitions 8 and 10, $g_T(\eta) = R(f)(g_T(t_1) \dots g_T(t_n))$, where $\text{length}(t_i) < \text{length}(\eta)$, $1 \leq i \leq n$.

$\eta[\Sigma]$ is in the form $(ft_1^* \dots t_n^*)$, where $t_i^* = t_i[\Sigma]$ if α is in t_i , $t_i^* = t_i$ otherwise. $g_{\mathbb{T}}(\eta[\Sigma]) = R(f)(g_{\mathbb{T}}(t_1^*) \dots g_{\mathbb{T}}(t_n^*))$. By the inductive hypothesis, $g_{\mathbb{T}}(t_i^*) = g_{\mathbb{T}}(t_i)$. Replacing each $g_{\mathbb{T}}(t_i^*)$ in $g_{\mathbb{T}}(\eta[\Sigma])$ by $g_{\mathbb{T}}(t_i)$, we have $g_{\mathbb{T}}(\eta[\Sigma]) = g_{\mathbb{T}}(\eta)$.

By the induction, therefore, $g_{\mathbb{T}}(\eta) = g_{\mathbb{T}}(\eta[\Sigma])$ for any η the pattern of a term in $\mathcal{L}_{\mathbb{T}}$, $\Sigma = \langle w, \beta \rangle$, $S(\eta, w) = \alpha$, g satisfies $(\alpha = \beta)$. ▀

Lemma 4. Given patterns P_1 and P_2 , suppose P_2 is an identity and, for some w in P_1 , $S(P_1, w) = S(P_2, l)$. If g satisfies P_1 and P_2 , then g satisfies $P_3 = P_1[w, S(P_2, m)]$, where $m = l + \#S(P_2, l)$.

[IR2, REPLACEMENT RULE]

Proof. The identity P_2 is in the form $(\alpha = \beta)$, $\beta = S(P_2, m)$. g satisfies P_2 , so $g_{\mathbb{T}}(\alpha) = g_{\mathbb{T}}(\beta)$ by Definitions 9 and 10. The proof is by induction on the length of P_1 .

Atomic case ($\text{length}(P_1) = 0$). P_1 is in the form $(\rho = \delta)$, for ρ, δ patterns of terms in $\mathcal{L}_{\mathbb{T}}$, and $S(P_1, k) = \delta$. $P_3 = P_1[\Sigma]$, $\Sigma = \langle w, \beta \rangle$, and $S(P_1, w) = \alpha$. Since g satisfies P_1 , $g_{\mathbb{T}}(\rho) = g_{\mathbb{T}}(\delta)$. g satisfies P_3 only if either

- (1) $g_{\mathbb{T}}(\rho[\Sigma]) = g_{\mathbb{T}}(\delta)$ for $L(w) < k$, or
- (2) $g_{\mathbb{T}}(\rho) = g_{\mathbb{T}}(\delta[\Sigma])$ for $L(w) \geq k$.

By Lemma 3, $g_{\mathbb{T}}(\rho) = g_{\mathbb{T}}(\rho[\Sigma])$, case (1), or $g_{\mathbb{T}}(\delta) = g_{\mathbb{T}}(\delta[\Sigma])$, case (2). In either case, g satisfies P_3 .

Inductive hypothesis. Assume that, if g satisfies θ , $\Sigma = \langle w, \beta \rangle$, $S(\theta, w) = \alpha$, such that w in θ , $\text{length}(\theta) < \text{length}(P_1)$, then g satisfies $P_3 = \theta[\Sigma]$.

P_1 is either in the form $(f_1 \Rightarrow f_2)$ or $(\neg f_1)$, f_1, f_2 patterns for formulas in \mathcal{L}_F , and $\text{length}(f_1)$ and $\text{length}(f_2) < \text{length}(P_1)$.

Suppose P_1 is in the form $(f_1 \Rightarrow f_2)$, $S(P_1, k) = f_2$. g satisfies P_1 , so either $g_F(f_1) = 0$ or $g_F(f_2) = 1$. There are two cases to consider. Either $w \geq k$, or $w < k$.

Case 1. $w \geq k$. Then $P_3 = P_1[\Sigma]$ is in the form $(f_1 \Rightarrow f_2[\Sigma])$. If $g_F(f_1) = 0$, then g satisfies P_3 . If $g_F(f_2) = 1$, then, by the inductive hypothesis, $g_F(f_2[\Sigma]) = 1$, and g satisfies P_3 .

Case 2. $w < k$. $P_3 = P_1[\Sigma]$ is in the form $(f_1[\Sigma] \Rightarrow f_2)$. If $g_F(f_2) = 1$, then g satisfies P_3 . If $g_F(f_1) = 0$, we must show that $g_F(f_1[\Sigma]) = 0$. Assume that $g_F(f_1[\Sigma]) = 1$. Clearly, we can construct a Σ' such that $f_1[\Sigma][\Sigma'] = f_1$. By the inductive hypothesis, if $g_F(f_1[\Sigma]) = 1$, then $g_F(f_1[\Sigma][\Sigma']) = 1 = g_F(f_1)$. This contradicts the assumption that $g_F(f_1) = 0$. Therefore, $g_F(f_1[\Sigma])$ must be 0 and g satisfies P_3 .

Suppose P_1 is in the form $(\neg f_1)$. g satisfies P_1 only if $g_F(f_1) = 0$. To prove that g satisfies $P_3 = (\neg f_1[\Sigma])$, we show that the assumption that $g_F(f_1[\Sigma]) = 1$ leads to a contradiction. This is done in the same manner as above. Therefore, $g_F(f_1[\Sigma]) = 0$ and g satisfies P_3 .

By the induction, for any P_1 the pattern for a formula in \mathcal{L}_F , if $S(P_1, w) = \alpha$, $\Sigma = \langle w, \beta \rangle$, g satisfies $(\alpha = \beta)$, then g satisfies $P_3 = P_1[\Sigma]$. ▀

Lemma 5. Given patterns P_1 and P_2 , suppose P_1 is in the form of a material conditional and $P_2 = S(P_1, 1)$. If g satisfies P_1 and P_2 , then g satisfies $S(P_1, k)$, where $k = 1 + \#S(P_1, 1)$. [IR4, DETACHMENT RULE]

Proof. P_1 is in the form $(f_1 \Rightarrow f_2)$ where f_1 is $S(P_1, 1)$ and f_2 is $S(P_1, k)$. By Definitions 9 and 10, g satisfies P_1 if and only if $g_F(S(P_1, 1)) = 0$ or $g_F(S(P_1, k)) = 1$. g satisfies P_2 , that is, $g_F(P_2) = 1$. Since $P_2 = S(P_1, 1)$, we have that $g_F(S(P_1, 1)) = 1$. Therefore, $g_F(S(P_1, k))$ must be 1. ▣

Lemma 6. If $\phi_{[i]}$ is a derivation of Ψ_i from $AS = \{\Psi_i \mid i \in A_n\}$, and g satisfies each $\Psi_i \in AS$, then g satisfies Ψ_n .

Proof. The proof is by induction on the length of the pattern sequence $\Psi_{[n]}$ for each $\Psi_1 \in AS$.

Atomic case ($n = 1$). If AS is the empty set, $\phi_{[1]}$ must come from SUBSTITUTION (IR1). By Lemma 1, g satisfies $\Psi_1 = A[\sigma]$, $A \in AX$, σ a substitution sequence. If AS is not empty, then Ψ_1 might belong to AS if $\phi_{[1]}$ is obtained by IR6. Then g satisfies Ψ_1 by assumption.

Inductive hypothesis. Assume, for all $j \leq k$, if $\phi_{[j]}$ is a derivation of Ψ_j from $AS = \{\Psi_i \mid i \in A_j\}$, and if g satisfies each $\Psi_i \in AS$, then g satisfies Ψ_j .

Now suppose $\phi_{[k+1]}$ is a derivation of ϕ_{k+1} from $AS = \{\Psi_i \mid i \in A_{k+1}\}$, and g satisfies each $\Psi_i \in AS$. We want to show g satisfies Ψ_{k+1} . That g satisfies Ψ_{k+1} if $\phi_{[k+1]}$ is obtained from inference rules 1-4 follows from Lemmas 1, 4, 2 and 5, respectively. If $\phi_{[k+1]}$ is obtained from IR5 (REPETITION), then Ψ_{k+1} is the same as some Ψ_i in $\Psi_{[k]}$ and g satisfies Ψ_i by the inductive hypothesis. If $\phi_{[k+1]}$ is obtained from IR6 (ASSUMPTION), then $k+1 \in A_{k+1}$, and $\Psi_{k+1} \in AS$. So, g satisfies Ψ_{k+1} by assumption. If $\phi_{[k+1]}$ follows from IR7 (CONDITIONAL PROOF), then Ψ_{k+1} is in the form $(\Psi_1 \Rightarrow \Psi_k)$ for $1 \leq i \leq k$. By Definitions 9

and 10, g satisfies Ψ_{k+1} only if either $g_F(\Psi_k) = 1$ or $g_F(\Psi_1) = 0$.
 But $g_F(\Psi_k) = 1$ by the inductive hypothesis.

By the induction, the lemma holds for all $k \geq 1$. ▣

Theorem 2. Let $M = \langle D, R \rangle$ be a model for AX . If $E \in \mathcal{L}$ is
provable, then E is true in M .

Proof. If E is provable, then there exists a rule sequence $\Phi_{[n]}$ such that $\Phi_{[n]}$ is a derivation of $P(E)$ from the empty set. Every assignment g on M satisfies the empty set. By Lemma 6, every assignment g on M must satisfy $\Psi_n = P(E)$. By Definition 10, $P(E)$ is satisfiable in M if and only if E is satisfiable in M . Therefore, every assignment on M satisfies E . By Definition 11, E is true in M . ▣

5. Proof Methods

So far, we have defined what we mean by a proof of a formula $E \in \mathcal{L}$, but have not indicated what proof procedures the theorem-prover uses in order to discover that proof. There are four procedures specified in this section. Each will be defined as a function that constructs a sequence of m rules from a sequence of n rules, $0 \leq n \leq m$. The theorem-proving algorithm, then, specifies an order in which these procedures are attempted.

Given the algorithm, we will be able to define a class of true statements of an Abelian group for which the theorem-prover can always find a proof. This proof of the adequacy of the algorithm for a class of formulas is offered in the next section.

Suppose the theorem-prover must find a proof, or a derivation, for some formula $E \in \mathcal{L}$. We call $P(E)$ the goal. According to the first method we will present, the prover chooses a pattern which can be transformed into the goal by sequential applications of the inference rules. If $\phi_{[n]}$ is a proof or derivation for the goal, then ϕ_1 is the initial pattern chosen according to the next definition. Note that it is obvious from the specifications of the inference rules that the initial pattern must either be an assumption or a substitution instance of an axiom. The particular substitution is chosen in such a way as to pick an initial pattern that closely resembles the goal. So, for example, we specify that the main connectives of ψ_1 and the goal, G , must be identical. The definition clearly omits, in many cases, instances of axioms that might be useful in solving the goal. This limited definition was chosen, however, to cut down the breadth of the theorem-prover's search in order to save time.

Definition 13. $I = \langle T_1, l_1 \rangle$ is an initial pattern for $G = \langle T_2, l_2 \rangle$ if and only if

$$(i) \quad l_1(0) = l_2(0),$$

$$(ii) \quad \text{either (a) } I = P \text{ (for } P \in \mathcal{L} \text{) or (b) } I = P[\sigma] \text{ (for } P[\sigma] \text{ defined), and}$$

(iii) one of the following is true:

$$(a) \quad I = G,$$

$$(b) \quad S(I, l) = S(G, l), \text{ or}$$

$$(c) \quad S(I, l + \#S(I, l)) = S(G, l + \#S(G, l)).$$

For now we assume that any pattern in AX which fulfills one of (i), (ii) or (iii) above can be taken as an initial pattern. We emphasize this point because later we will discuss a FEATURE TEST heuristic. The purpose of this heuristic is to alter the choice of an initial pattern.

To rewrite one pattern so that it is identical to another pattern, the theorem-prover must find out how the two patterns differ and then reduce that difference. It compares the patterns to determine if they are identical, and, if not, forms their difference set.

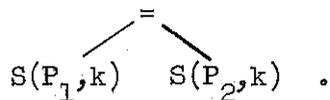
Definition 14. The difference set of P_1 and P_2 , $D(P_1, P_2)$, is $\{k \mid 1 \leq k \leq \min(\#P_1, \#P_2), \text{ and } S(P_1, k) \neq S(P_2, k)\}$.

Suppose, in searching for a proof of G , the theorem-prover begins with an initial pattern I for G and then determines the difference set of G and I . If $D(I, G)$ is not empty, the prover tries to use the rules of inference to construct a proof of $S(G, i) = S(I, i)$, for some $i \in D(I, G)$. The intent is to show that I can be rewritten by replacing $S(I, i)$ by the equivalent $S(G, i)$. To accomplish this task, the theorem-prover sets up a new goal which we call the subgoal.

Definition 15. Given patterns P_1, P_2 , and $k \in D(P_1, P_2)$, the k th subgoal, $SG(P_1, P_2, k)$ is the pattern $\langle T, \ell \rangle$ such that

- (i) $\ell(0)$ is the identity symbol $=$,
- (ii) $S(SG(P_1, P_2, k), 1) = S(P_1, k)$, and
- (iii) $S(SG(P_1, P_2, k), 1 + \#S(P_1, k)) = S(P_2, k)$.

A simple diagram of the subgoal $SG(P_1, P_2, k)$ is:



As an example of a subgoal, we refer to P_1 and P_2 in Figure 1. The difference set is $D(P_1, P_2) = \{4, 5, 6, 7\}$. For $4 \in D(P_1, P_2)$,

$S(P_1, 4) = ((A + B) + C)$ and $S(P_2, 4) = (A + (B + C))$. The subgoal $SG(P_1, P_2, 4)$ is $((A + B) + C) = (A + (B + C))$.

The proof for the subgoal $SG(I, G, k)$ is a subsequence of the proof for G . If we have $\Phi_{[m]}$ and assume that $\Psi_n = I$ and $\Psi_m = SG(I, G, k)$, $m \geq n$, then the theorem-prover can form a new pattern by application of the REPLACEMENT rule, $RE(n, m, k)$.

The theorem-prover's basic proof method is to transform one pattern into another by reducing the difference between the two patterns in the manner outlined above. This method consists mainly of two functions, one iterative, the other recursive. We first examine the recursive part of the method. Suppose we compute the difference set of two patterns, I and G . Using a function we call GOAL (Definition 17), we search for some $k \in D(I, G)$ such that a proof for $SG(I, G, k)$ takes the form of sequential applications of the REPLACEMENT rule. We explicitly define this proof form as $SUBGOAL^n(\Phi_{[k]}, G)$.

In this definition, the recursion terminates (at level $n = 0$) either (a) when the prover finds an initial pattern identical to G ; or (b) when $\Psi_{[k]}$ contains a P such that G is identical to P , or G is identical to the application of the COMMUTE EQUALS rule to P . Otherwise (level $n \neq 0$), the prover picks an initial pattern for G , or picks some pattern already in the pattern sequence, $\Psi_{[k]}$. Call this pattern P' . The prover tries to show that the difference between P' and the goal can be reduced. Here, the difference yields a subgoal $SG(P', G, m)$. The proof of the subgoal, Φ' , is appended to $\Phi_{[k]} * \theta$ (where θ is IR1, IR3, IR5 or IR6) to form the sequence $\Phi_{[j]}$. A new rule sequence is obtained by the REPLACEMENT rule: $\Phi_{[j+1]} = \Phi_{[j]} * RE(k + 1, j, m)$. The new

pattern, Ψ_{j+1} , should be useful in reducing the difference between G and the initial pattern chosen at level $n - 1$.

Definition 16.

$\text{SUBGOAL}^0(\Phi_{[k]}, G) = \Phi_{[k+1]}$, if and only if $G = \Psi_{k+1}$ and $\Phi_{[k+1]}$ is obtained from either IR1, IR3, IR5, or IR6 [subgoal is provable in one step].

$\text{SUBGOAL}^n(\Phi_{[k]}, G) = \Phi_{[j]} * \text{RE}(k+1, j, m)$ if and only if

(i) $\Phi_{[k+1]}$ is obtained from IR1, IR3, IR5, or IR6, where Ψ_{k+1} must be an initial pattern for G [select some initial pattern I];

(ii) $m \in D(\Psi_{k+1}, G)$ [determine difference between subgoal and I];

(iii) $\Phi_{[j]} = \text{SUBGOAL}^{n-1}(\Phi_{[k+1]}, \text{SG}(\Psi_{k+1}, G, m))$ [difference can be shown equivalent]; and

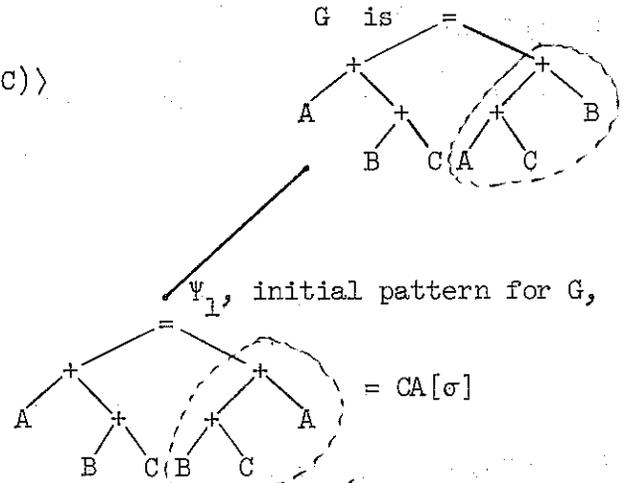
(iv) Suppose, for each $\text{SUBGOAL}^i(\Phi_{[k]}, G)$, $i < n$, we denote G by G^i . Then $\text{SG}(\Psi_{k+1}, G, m) \neq G^i$, for all $0 \leq i < n$ [no attempt to prove same expression twice].

Restriction (iv) prevents the prover from trying to find a proof for a subgoal which is like a previously established, but not yet resolved, subgoal.

Take, as an example of a pattern whose proof takes the form of $\text{SUBGOAL}^n(\Phi_{[k]}, G)$, $G = ((A + (B + C)) = ((A + C) + B))$. The example is restricted to axioms of an associative system without the CE rule (IR3). Thus, $\text{AX} = \{\text{CA}, \text{AS}\}$. An initial pattern for G is $\text{CA}[\sigma] = ((A + (B + C)) = ((B + C) + A))$. We will follow the construction of the proof with diagrams of trees whose nodes are labeled by patterns, the initial patterns and the subgoals. The leaves of the tree are patterns that are immediately solvable by one of the inference rules. The subpatterns marked by dotted circles make up the next subgoal. Proof of $G = \text{SUBGOAL}^n(A, G)$, $n \geq 0$, is the following steps (follow the diagrams by the arrows pointing to the sequence).

1. $SUBGOAL^n(\Lambda, G)$

$$\Phi_1 = SUB(\sigma, CA), \quad \sigma = \langle A, A \rangle \langle B, (B + C) \rangle$$



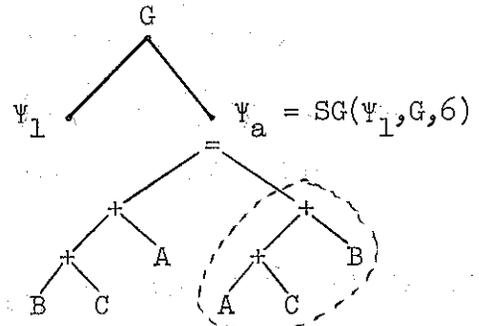
2. $D(\Psi_1, G) = \{6, 7, 8, 10\}$

$$\text{Let } \Psi_a = SG(\Psi_1, G, 6)$$

$$= ((B + C) + A) = ((A + C) + B))$$

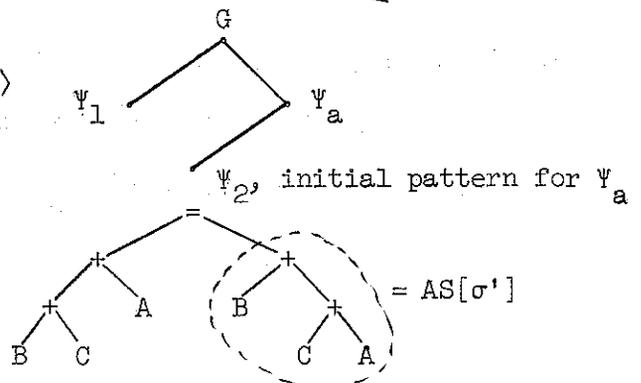
Then, $SUBGOAL^n(\Lambda, G)$

$$= SUBGOAL^{n-1}(\Phi_{[1]}, \Psi_a) * RE(1, a, 6)$$



3. $SUBGOAL^{n-1}(\Phi_{[1]}, \Psi_a)$

$$\Phi_2 = SUB(\sigma', AS), \quad \sigma' = \langle A, B \rangle \langle B, C \rangle \langle C, A \rangle$$



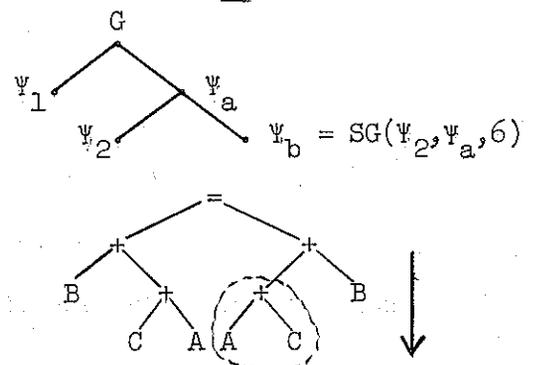
4. $D(\Psi_2, \Psi_a) = \{6, 7, 8, 10\}$

$$\text{Let } \Psi_b = SG(\Psi_2, \Psi_a, 6)$$

$$= ((B + (C + A)) = ((A + C) + B))$$

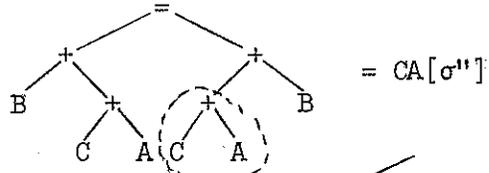
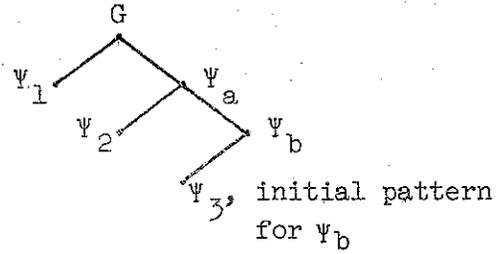
Then, $SUBGOAL^{n-1}(\Phi_{[1]}, \Psi_a)$

$$= SUBGOAL^{n-2}(\Phi_{[2]}, \Psi_b) * RE(2, b, 6)$$



5. $\text{SUBGOAL}^{n-2}(\Phi_{[2]}, \Psi_b)$

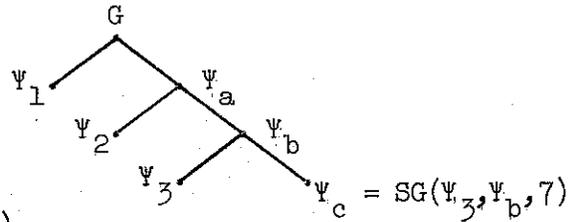
$\Phi_3 = \text{SUB}(\sigma''', CA), \sigma''' = \langle A, B \rangle \langle B, (C + A) \rangle$



6. $D(\Psi_3, \Psi_b) = \{6, 7, 8, 9\}$

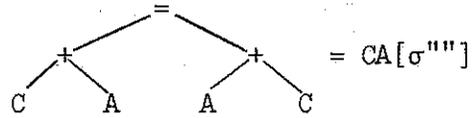
Let $\Psi_c = \text{SG}(\Psi_3, \Psi_b, 7)$
 $= ((C + A) = (A + C))$

Then, $\text{SUBGOAL}^{n-2}(\Phi_{[2]}, \Psi_b)$
 $= \text{SUBGOAL}^{n-3}(\Phi_{[3]}, \Psi_c) * \text{RE}(3, c, 7)$



7. $\text{SUBGOAL}^{n-3}(\Phi_{[3]}, \Psi_c) = \Phi_{[3]} * \text{SUB}(\sigma''', CA)$

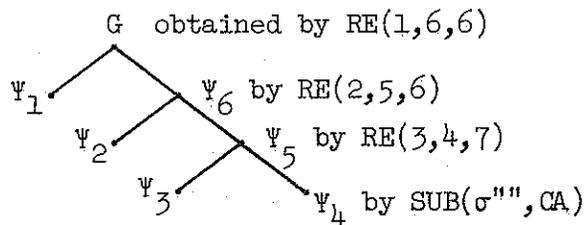
$\Phi_4 = \text{SUB}(\sigma''', CA), \sigma''' = \langle A, C \rangle \langle B, A \rangle$



8. Therefore, $n = 3$, and a proof for G is $\text{SUBGOAL}^3(\Lambda, G)$.

Assigning 4 to c , 5 to b , 6 to a , we have

$\text{SUBGOAL}^3(\Lambda, G) = \Phi_{[7]} = ((\Phi_{[4]} * \text{RE}(3, 4, 7)) * \text{RE}(2, 5, 6)) * \text{RE}(1, 6, 6)$



(Notice in the final diagram that the subgoal at each node is obtained by $\text{RE}(i, j, k)$, where i and j are the node's direct descendants.)

The function GOAL searches the difference set $D(I, G)$ for an integer j such that a derivation of $\text{SG}(I, G, j)$ is $\text{SUBGOAL}^n(\Phi, \text{SG}(I, G, j))$, $n \geq 0$.

Definition 17. $GOAL(I,G,j)(\Phi) = \theta$ if and only if θ is

- (i) ϕ if $j \leq 0$;
- (ii) j if $j \in D(I,G)$ and there is an $n \geq 0$, $k > 0$, such that
 $\Phi_{[k]} = SUBGOAL^n(\Phi, SG(I,G,j))$;
- (iii) $GOAL(I,G,j-1)(\Phi)$ otherwise.

Now suppose we want to find a proof for G and we can find a $k \in D(I,G)$ such that $k \neq \phi$, $k = GOAL(I,G,\#G)$. Then we can, by IR2, replace $S(I,k)$ by $S(G,k)$ and form a new pattern, I' . We continue the search for a proof of G by computing $D(I',G)$ and trying to further reduce any differences between the last pattern formed and G . This procedure continues until either (a) $D(I',G) = \phi$, that is, the proof is complete; or (b) $GOAL(I',G,\#G) = \phi$, in which case we say that the difference between I and G cannot be further reduced by this method. This iterative part of the proof procedure is defined below as $REPLACE^n(\Phi,G)$.

Definition 18.

$REPLACE^0(\Phi_{[i-1]},G) = \Phi_{[j]} * RE(i,j,k)$ where

- (i) $\Phi_{[i]}$ is obtained from either IR1, IR3, IR5, or IR6, where Ψ_i must be an initial pattern for G [there is a difference between G and the initial pattern I];
- (ii) $k = GOAL(\Psi_i,G,\#G)(\Phi_{[i]})$, $k \neq \phi$ [that difference is derivable by definition of SUBGOAL]; and
- (iii) $\Phi_{[j]} = SUBGOAL^m(\Phi_{[i]}, SG(\Psi_i,G,k))$, $m \geq 0$ [the replacement does not, however, generate G].

$REPLACE^n(\Phi, G) = \Phi_{[j]} * RE(i, j, k)$ where

(i) $\Phi_{[i]} = REPLACE^{n-1}(\Phi, G)$ [differences have already been reduced;

this is a further attempt];

(ii) $k = GOAL(\Psi_i, G, \#G)(\Phi_{[i]})$, $k \neq \phi$ [the kth subgoal is provable]; and

(iii) $\Phi_{[j]} = SUBGOAL^m(\Phi_{[i]}, SG(\Psi_i, G, k))$, $m \geq 0$ [$\Phi_{[j]}$ is the proof].

The proof procedure we call REDUCE is now easily stated.

Method 1. REDUCE(Φ)

G is obtainable by REDUCE(Φ) if and only if there is an $n \geq 0$, $k > 0$, such that $\Phi_{[k]} = REPLACE^n(\Phi, G)$, and $G = \Psi_k$.

Several examples of the REDUCE method follow.

1. A simple example is $G = ((C + D) = (C + D))$ and $CA \in AX$. Then

$G = \Psi_3$, where $\Phi_{[3]} = REPLACE^0(\Lambda, G)$ in the following way.

$REPLACE^0(\Lambda, G) = \Phi_{[2]} * RE(1, 2, 4)$ where

(i) $\Phi_{[1]} = SUB(\sigma, CA)$, $\sigma = \langle A, C \rangle \langle B, D \rangle$

$\Psi_1 = ((C + D) = (D + C))$

(ii) $GOAL(\Psi_1, G, 6)(\Phi_{[1]}) = 4$

because $D(\Psi_1, G) = \{4\}$ and

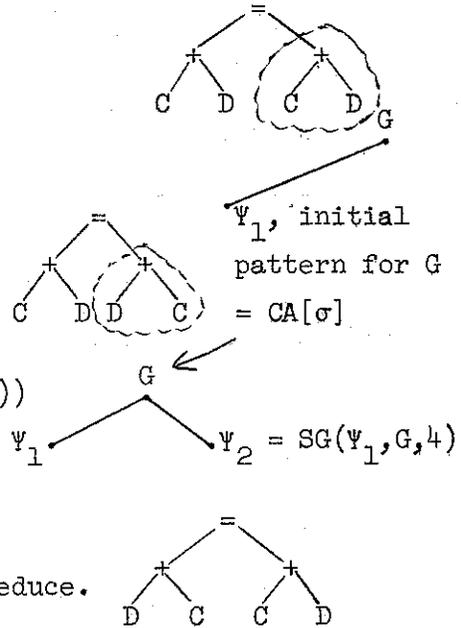
$\Phi_{[2]} = SUBGOAL^0(\Phi_{[1]}, SG(\Psi_1, G, 4))$

$= SUBGOAL^0(\Phi_{[1]}, ((D + C) = (C + D)))$

$= \Phi_{[1]} * SUB(\sigma', CA)$,

$\sigma' = \langle A, D \rangle \langle B, C \rangle$

(iii) There was only a single difference to reduce.

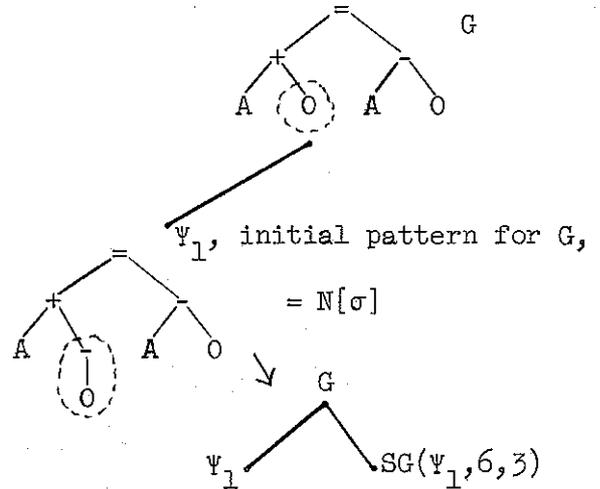


2. Let G be $((A + 0) = (A - 0))$, $AX = \{N, AI, AS, CA, Z\}$.

Then $G = \Psi_7$, $\Phi_{[7]} = \text{REPLACE}^0(\Lambda, G)$ in the following way.

$$\text{REPLACE}^0(\Lambda, G) = \Phi_{[j]} * \text{RE}(1, j, 3)$$

(i) $\Phi_1 = \text{SUB}(\sigma, N)$, $\sigma = \langle A, A \rangle \langle B, 0 \rangle$



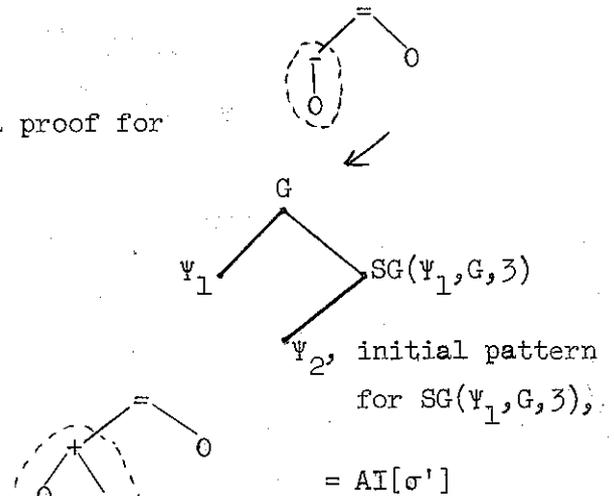
(ii) $D(\Psi_1, G) = \{1, 3\}$,

$\text{GOAL}(\Psi_1, G, 6)(\Phi_{[1]}) = 3$, because a proof for

$\text{SG}(\Psi_1, G, 3)$ is constructed by:

(1) $\text{SUBGOAL}^n(\Phi_{[1]}, \text{SG}(\Psi_1, G, 3))$

$\Phi_2 = \text{SUB}(\sigma', AI)$, $\sigma' = \langle A, 0 \rangle$



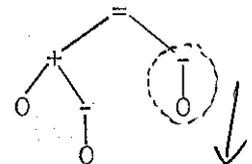
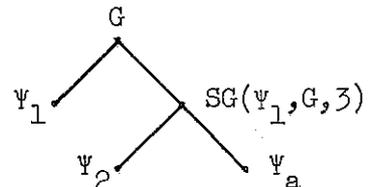
(2) $D(\Psi_2, \text{SG}(\Psi_1, G, 3)) = \{1, 2, 3\}$

Let $\Psi_a = \text{SG}(\Psi_2, \text{SG}(\Psi_1, G, 3), 1)$

$= ((0 + (-0)) = (-0))$

Then $\text{SUBGOAL}^n(\Phi_{[1]}, \text{SG}(\Psi_1, G, 3))$

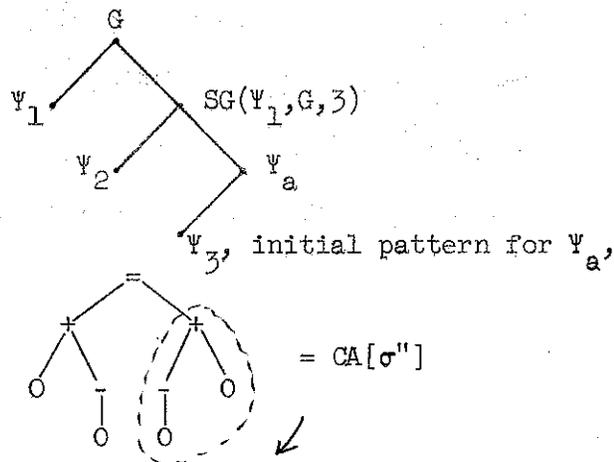
$= \text{SUBGOAL}^{n-1}(\Phi_{[2]}, \Psi_a) * \text{RE}(2, a, 1)$



(3) $\text{SUBGOAL}^{n-1}(\Phi_{[2]}, \Psi_a)$

$\Phi_3 = \text{SUB}(\sigma'', \text{CA}),$

$\sigma'' = \langle A, 0 \rangle \langle B, (-0) \rangle$



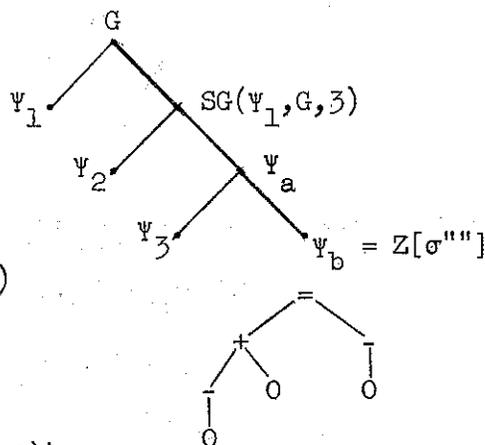
(4) $D(\Psi_3, \Psi_a) = \{5, 6\}$

Let $\Psi_b = \text{SG}(\Psi_3, \Psi_a, 5)$

$= (((-0) + 0) = (-0))$

Then $\text{SUBGOAL}^{n-1}(\Phi_{[2]}, \Psi_a)$

$= \text{SUBGOAL}^{n-2}(\Phi_{[3]}, \Psi_b) * \text{RE}(3, b, 5)$



(5) $\text{SUBGOAL}^{n-2}(\Phi_{[3]}, \Psi_b)$

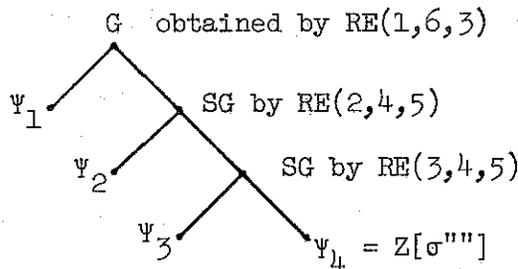
$= \Phi_{[3]} * \text{SUB}(\sigma''', Z), \quad \sigma''' = \langle A, (-0) \rangle$

Therefore, $n = 2$; a proof for the subgoal is $\text{SUBGOAL}^2(\Phi_{[1]}, \text{SG}(\Psi_1, G, 3))$.

Assigning 4 to b, 5 to a, we have that

$\text{SUBGOAL}^2(\Phi_{[1]}, \text{SG}(\Psi_1, G, 3)) = (\Phi_{[4]} * \text{RE}(3, 4, 5)) * \text{RE}(2, 4, 5)$.

Therefore, $j = 6$.



(iii) Then

$\text{REDUCE}^0(\Lambda, G) = \Phi_{[6]} * \text{RE}(1, 6, 3)$.

3. Let G be $((A + (-0)) + (C + 0)) = ((0 + C) + (A + 0))$. Then $G = \Psi_9$

where $\Phi_{[9]} = \text{REPLACE}^1[\Lambda, a)$ in the following way:

(i) $\text{REPLACE}^0(\Lambda, G) = \Phi_{[3]}$ where

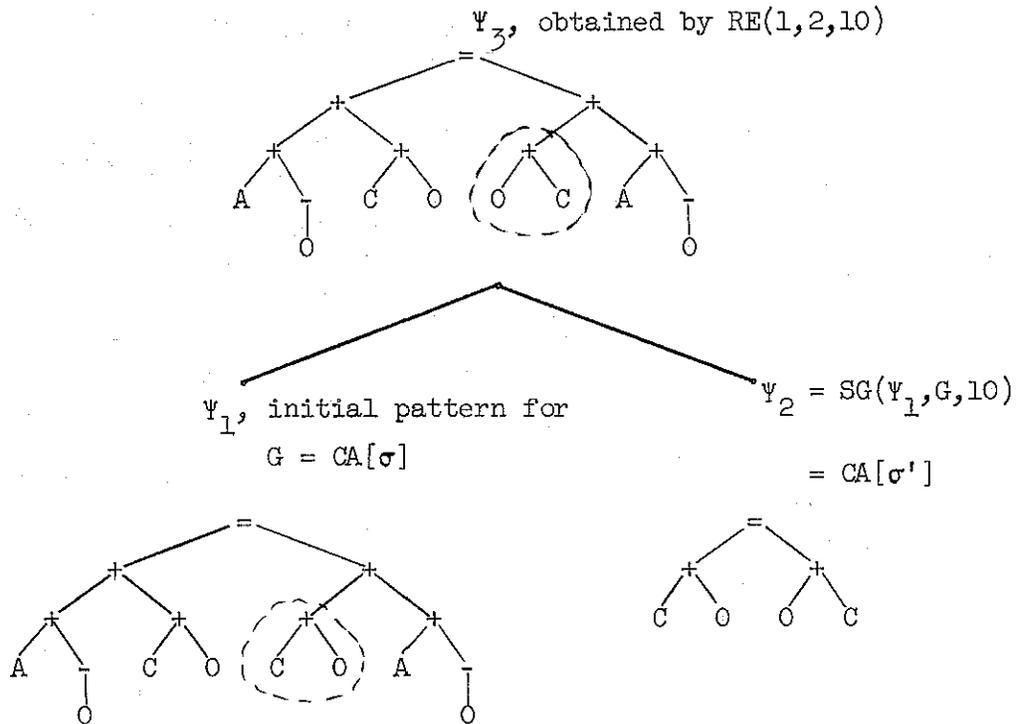
$$\Phi_{[1]} = \text{SUB}(\sigma, CA),$$

$$\sigma = \langle A, (A + (-0)) \rangle \langle B, (C + 0) \rangle$$

$$\Phi_{[2]} = \text{SUBGOAL}^0(\Phi_{[1]}, \text{SG}(\Psi_1, G, 10))$$

$$= \text{SUBGOAL}^0(\Phi_{[1]}, ((C + 0) = (0 + C)))$$

$$= \Phi_{[1]} * \text{SUB}(\sigma', CA), \sigma' = \langle A, C \rangle \langle B, 0 \rangle$$



$$\Phi_{[3]} = \Phi_{[2]} * \text{RE}(1,2,10)$$

$$\Psi_3 = (((A + (-0)) + (C + 0)) = ((0 + C) + (A + (-0))))$$

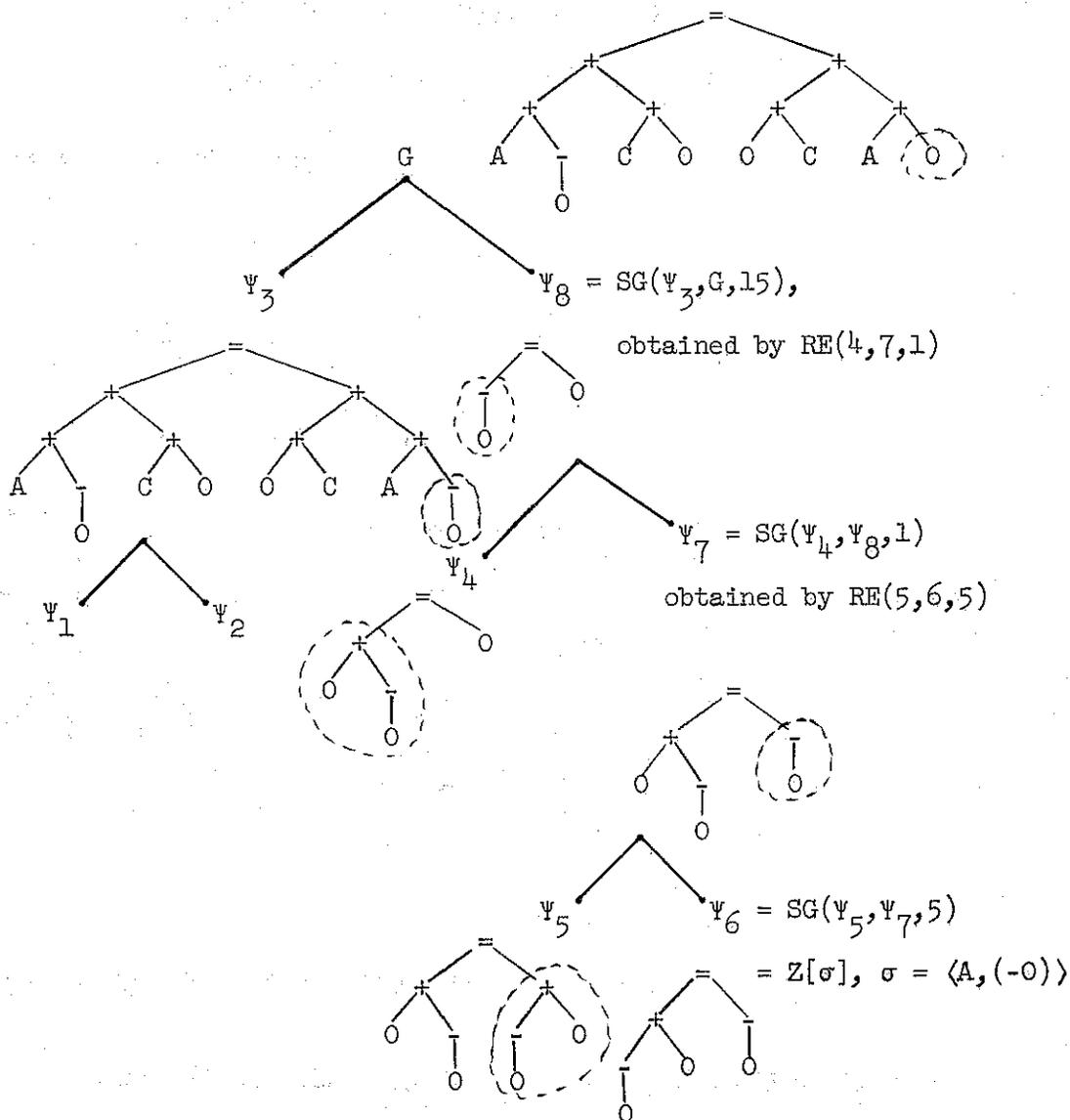
(ii) $\text{REPLACE}^1(\Lambda, G) = \Phi_{[9]} = \Phi_{[8]} * \text{RE}(3,8,15)$ where

$$\Phi_{[3]} = \text{REPLACE}^0(\Lambda, G)$$

$$\begin{aligned} \Phi[8] &= \text{SUBGOAL}^2(\Phi[3], \text{SG}(\Psi_3, G, 15)) \\ &= \text{SUBGOAL}^2(\Phi[3], ((-0) = 0)) \end{aligned}$$

$$\Psi_8 = ((-0) = 0)$$

The five-rule proof of $((-0) = 0)$ was constructed in Example 2 and is only repeated here in the diagram.



The second proof method is useful in finding a proof for G only if G is in the form of a material conditional. The antecedent of G is immediately included in the pattern sequence by IR6, $\Phi_{[1]} = AR(S(G,1))$. The prover then attempts to find a derivation for the consequent of G , $S(G,1 + \#S(G,1))$, given $\Phi_{[1]}$.

In searching for a derivation of the consequent of G , the theorem-prover can try to use any of the four proof methods, including Method 2. It might choose Ψ_1 as the initial pattern for G and construct the derivation using Method 1, REDUCE($\Phi_{[1]}$); it may prove a subgoal using IR5, $\Phi_{[k]} * REP(1)$, $k \geq 1$; or it may not reference Φ_1 in a rule until the last step. If $\Phi_{[j]}$ is the derivation of the consequent of G , where $\Psi_1 = S(G,1)$, then the last rule of the proof or derivation for G is IR7, $\Phi_{[j+1]} = \Phi_{[j]} * CP(1,j)$.

Method 2. CONDITIONAL-PROOF(Φ)

G is obtainable by CONDITIONAL-PROOF(Φ) if and only if

- (i) $G = \langle T, \mathcal{L} \rangle$ and $\mathcal{L}(0)$ is \Rightarrow [G is a material conditional];
- (ii) $\Phi_{[j]} = \Phi * AR(S(G,1))$, IR5 [we assume the antecedent of G]; and
- (iii) there exists a $k \geq 0$ such that $\Phi_{[j+k]} = \Phi_{[j]} * \Phi_{[k]}$ is a derivation of $S(G,1 + \#S(G,1))$ [there is a proof for the consequent].

Then $\Phi_{[j+k]} * CP(j, j+k)$ is a derivation of G .

Some examples of Method 2 follow.

1. Theorem 7 of Appendix II.

2. Let G be $((B = A) \Rightarrow (A = B))$. Then $\Phi_1 = AR((B = A))$, $\Psi_1 = (B = A)$.

The derivation of $S(G,4) = (A = B)$ is:

$$\Phi_2 = \Phi_1 * CE(1)$$

The proof of G is

$$\Phi_3 = \Phi_2 * CP(1,2)$$

$$\Psi_3 = ((B = A) \Rightarrow (A = B)).$$

3. Let G be $((A = B) \Rightarrow ((A + B) = (A + A)))$. Then $\Phi_1 = AR((A = B))$,

$\Psi_1 = (A = B)$. Let $G' = S(G,1 + \#S(G,1)) = ((A + B) = (A + A))$, the

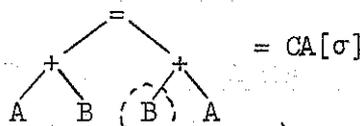
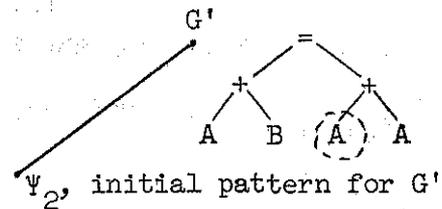
consequent of G .

(i) For $k = 3$, G' is obtained by $REDUCE(\Phi_1)$ in the following way.

$$REPLACE^O(\Phi_1, G') = \Phi_4 \text{ where}$$

$$\Phi_2 = \Phi_1 * SUB(\sigma, CA),$$

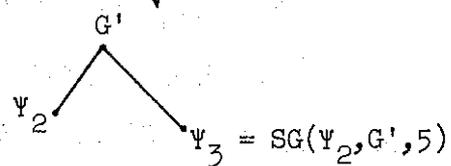
$$\sigma = \langle A,A \rangle \langle B,B \rangle$$



$$\Phi_3 = SUBGOAL^O(\Phi_2, SG(\Psi_2, G', 5))$$

$$= SUBGOAL^O(\Phi_1, (B = A))$$

$$= \Phi_2 * CE(1)$$



$$\Phi_4 = \Phi_3 * RE(2,3,5)$$

(ii) The proof for G is $\Phi_5 = \Phi_4 * CP(1,4)$.

The third method, which uses the DETACHMENT rule (IR4), attempts to find an initial pattern I such that (a) I is in the form of a conditional, (b) the goal is obtainable from the consequent of I by applications of the inference rules, and (c) a proof can be constructed for the antecedent of I . Since the consequent of I is to be transformed into the goal, I is chosen in such a way as to have the consequent of I be similar to G , i.e., fulfill the requirements listed in the definition of an initial pattern for G .

Method 3. DETACHMENT($\Phi_{[n]}$)

G is obtainable by DETACHMENT($\Phi_{[n]}$) if and only if there is a pattern $I = \langle T, \ell \rangle$ such that

(i) $\ell(0)$ is \Rightarrow [I is a conditional];

(ii) $I' = S(I, \ell + \#S(I, \ell))$ fulfills the three conditions of an initial pattern for G [I' is the consequent of I];

(iii) there is a $j \geq 0$ such that $\Phi_{[k]} = \Phi_{[n]} * \Phi_{[j]}$ is a derivation or proof of $S(I, \ell)$ [antecedent of I is provable];

(iv) $\Phi_{[k+1]}$ is obtainable by IR1, IR5, or IR6, such that $I = \psi_{k+1}$ [add I to the pattern sequence]; and

(v) $(\Phi_{[k+1]} * DE(k+1, k)) * \Phi_{[m]}$ is a derivation of G [G provable from consequent of I].

As an example of Method 3, let G be $((-(-A)) = A)$, and let $AI, ST \in AX$ where ST is $((((A + B) = 0) \Rightarrow (A = (-B))))$. Also, let $I = ST[\sigma] = (((A + (-A)) = 0) \Rightarrow (A = (-(-A))))$, $\sigma = \langle A, A \rangle \langle B, (-A) \rangle$. The antecedent of I is $((A + (-A)) = 0)$ and the consequent of I is $(A = (-(-A)))$.

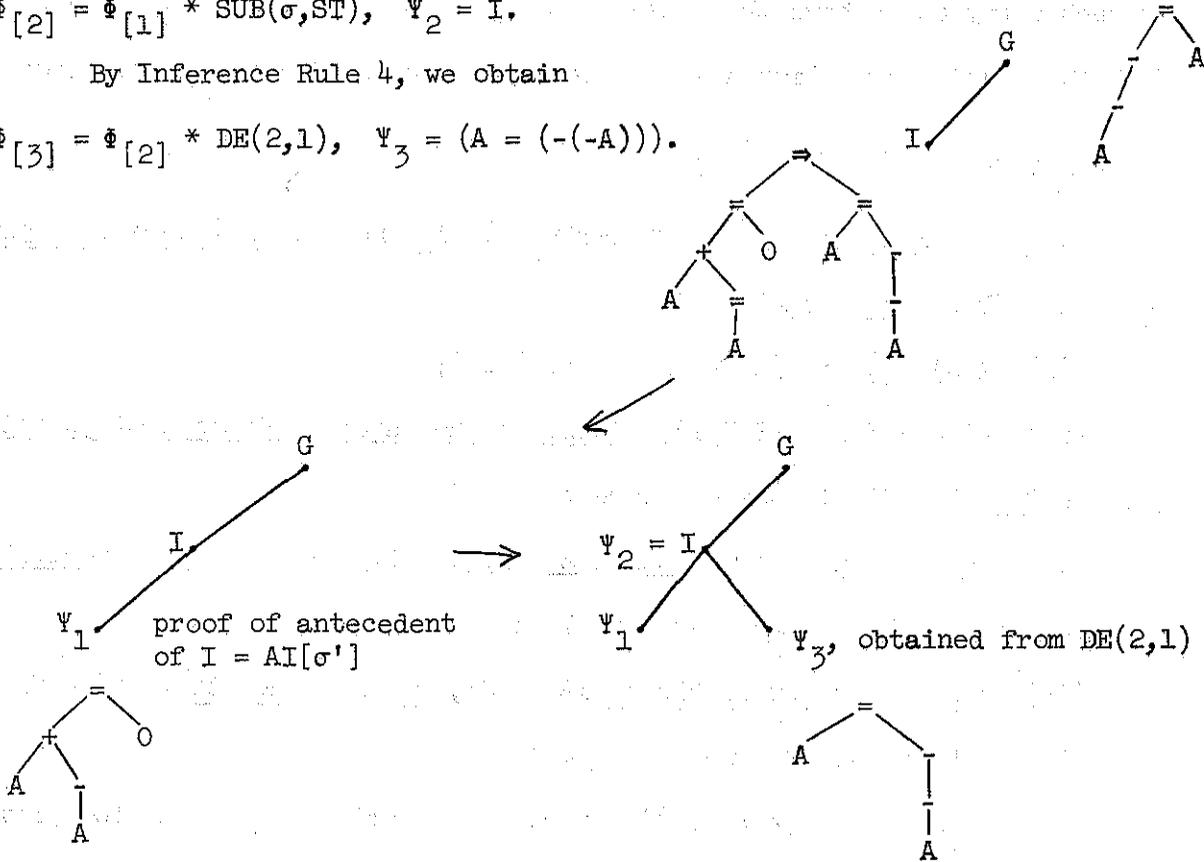
We must first show that the antecedent of I is provable. In this case, the proof is by substitution, $\Phi_{[1]} = \text{SUB}(\sigma', AI)$, $\sigma' = \langle A, A \rangle$, and $\Psi_1 = ((A + (-A)) = 0)$. From the definition of Method 3, we note that $n = 0$ and $j = k = 1$.

Now, using $I = \text{ST}[\sigma]$ given above, we add I to the pattern sequence:

$\Phi_{[2]} = \Phi_{[1]} * \text{SUB}(\sigma, \text{ST}), \Psi_2 = I.$

By Inference Rule 4, we obtain:

$\Phi_{[3]} = \Phi_{[2]} * \text{DE}(2,1), \Psi_3 = (A = (-(-A))).$



We can now try to prove G given $\Phi_{[3]}$. Thus, Ψ_3 can be an initial pattern. G is equal to CE(3); so the proof is $\Phi_{[4]} = \Phi_{[3]} * \text{CE}(3)$. Note that G is listed as Theorem 13, and ST as Theorem 10 in Appendix I.

The fourth method attempts to find a proof for a pattern G by first trying to simplify G. This simplification takes the form of replacing subpatterns of G by other equivalent (but not identical) subpatterns of G. For example, a proof for G, where G is $((A + B) + (- (B + A))) = 0$

is easily found if $S(G,2) = (A + B)$ is replaced by $S(G,6) = (B + A)$. A proof for the pattern formed from $S(G,2)$ and $S(G,6)$ is $\text{SUB}(\sigma, \text{CA})$, $\sigma = \langle A, A \rangle \langle B, B \rangle$. After showing that $((A + B) = (B + A))$, we can simplify G by replacing $S(G,2)$ by $(B + A)$. The new pattern, $G' = (((B + A) + (-(B + A))) = 0)$, is a substitution instance of the AI axiom. So the proof of G takes the form of $\phi_{[1]} = \text{SUB}(\sigma, \text{AI})$, $\sigma = \langle A, (B + A) \rangle$, plus the replacement of $S(P_1, 2) = (B + A)$ by $(A + B)$, the reverse replacement carried out in the simplification process.

In the following specification of Method 4, we indicate an exhaustive search of G for equivalent subpatterns. The next two definitions are the required search algorithms. $\text{FINDA}(P_1, P_2, k)$ searches the pattern P_2 for a subpattern equivalent, but not identical, to P_1 . $\text{FINDB}(P_1, P_2, k)$ increments k looking for a subpattern, $S(P_1, k)$, which FINDA finds is equivalent to a subpattern of P_2 .

In the following, let $P_1 = \langle T_1, l_1 \rangle$ and $P_2 = \langle T_2, l_2 \rangle$.

Definition 19. $\text{FINDA}(P_1, P_2, k) = \theta$ if and only if θ is

- (i) ϕ if $k \notin T_2$;
- (ii) k if P_1 is not syntactically identical to $S(P_2, k)$ and, for P_3 equal to $\begin{array}{c} \diagup \\ P_1 \\ \diagdown \end{array} = \begin{array}{c} \diagup \\ S(P_2, k) \\ \diagdown \end{array}$, there is an $n \geq 0$, $j > 0$, such that

$\phi_{[j]} = \text{SUBGOAL}^n(\Lambda, P_3)$ is a derivation of P_3 ;

- (iii) $\text{FINDA}(P_1, P_2, k + 1)$ otherwise.

Definition 20. $\text{FINDB}(P_1, P_2, k) = \theta$ if and only if θ is

- (i) ϕ if $k \notin T_1$;
- (ii) k if $\text{FINDA}(S(P_1, k), P_2, 1) \neq \phi$;
- (iii) $\text{FINDB}(P_1, P_2, k + 1)$ otherwise.

The fourth method, which we call the SUBSET-TEST, uses FINDA and FINDB in order to modify the pattern for which the theorem-prover is trying to find a proof. The recursive definition of $\text{SUBSET}^n(\Phi, G)$ specifies the simplification process. When $\text{SUBSET}^n(\Phi, G) = \text{SUBSET}^{n-1}(\Phi, G)$, for some $n > 0$, we say that no further simplification replacements can be made. If $\text{SUBSET}^n(\Phi, G)$, the simplified goal, is different from the original goal, then the prover tries to find a derivation for it. The proof of the original goal is then the proof of $\text{SUBSET}^n(\Phi, G)$ followed by the sequence of replacements which "reverse" the simplification replacements. That is, if α were replaced by β , then the reverse action is to replace the appropriate β by α .

Definition 21.

$$\text{SUBSET}^0(\Phi, G) = G.$$

$$\text{SUBSET}^n(\Phi, G) = \theta \text{ if and only if } \theta \text{ is}$$

- (i) $G'[j, S(G', k)]$ where $G' = \text{SUBSET}^{n-1}(\Phi, G)$, $j = \text{FINDB}(G', G', 1)$,
 $k = \text{FINDA}(S(G', j), G', 1)$, $j, k \neq \Phi$;
- (ii) $\text{SUBSET}^{n-1}(\Phi, G)$ otherwise.

Clearly, by "simplifying," we mean to decrease the number of distinct subpatterns in the goal G . The fourth proof method, then, follows.

Method 4. SUBSET-TEST(Φ)

G is obtainable by SUBSET-TEST(Φ) if and only if there exists an $n > 0$, $j > 0$, such that $\text{SUBSET}^n(\Phi, G) = \text{SUBSET}^{n-1}(\Phi, G)$, $\text{SUBSET}^n(\Phi, G) \neq G$, and $\Phi[j]$ is a derivation of $\text{SUBSET}^n(\Phi, G)$.

Another example may clarify how the actual derivation for G may be obtained from a proof of $\text{SUBSET}^n(\Phi, G)$. Let G be $((A + B) + (C + D)) = ((D + C) + (B + A))$. Then

$$\text{SUBSET}^0(\Lambda, G) = G;$$

$$\text{SUBSET}^1(\Lambda, G) = G'[2, S(G', 12)] \text{ where } G' \text{ is } \text{SUBSET}^0(\Lambda, G),$$

$$= G'[2, (B + A)] \text{ since } \text{FINDB}(G', G', 1) = 2 \text{ because}$$

$$\text{FINDA}(S(G', 2), G', 1) = 12 \text{ and the proof of}$$

$$P_3 = ((A + B) = (B + A)) \text{ is } \text{SUBGOAL}^0(\Lambda, P_3) = \text{SUB}(\sigma, CA),$$

$$\sigma = \langle A, A \rangle \langle B, B \rangle,$$

$$= (((B + A) + (C + D)) = ((D + C) + (B + A))) \text{ [this is the first modified goal];}$$

$$\text{SUBSET}^2(\Lambda, G) = G''[5, S(G'', 9)] \text{ where } G'' \text{ is } \text{SUBSET}^1(\Lambda, G),$$

$$= G''[5, (D + C)] \text{ since } \text{FINDB}(G'', G'', 1) = 5 \text{ because}$$

$$\text{FINDA}(S(G'', 5), G'', 1) = 9 \text{ and the proof of}$$

$$P_3 = ((C + D) = (D + C)) \text{ is } \text{SUBGOAL}^0(\Lambda, P_3) = \text{SUB}(\sigma, CA),$$

$$\sigma = \langle A, C \rangle \langle B, D \rangle,$$

$$= (((B + A) + (D + C)) = ((D + C) + (B + A))) \text{ [this is the second modified goal];}$$

$$\text{SUBSET}^3(\Lambda, G) = \text{SUBSET}^2(\Lambda, G).$$

$$\text{SUBSET}^3(\Lambda, G) \text{ is obtainable by IRL, } \text{SUB}(\sigma', CA), \sigma' = \langle A, (B + A) \rangle \langle B, (D + C) \rangle.$$

Then the proof of G is $\Phi[5]$ where

$$\Phi[1] = \text{SUB}(\sigma, CA), \sigma' \text{ given above,}$$

$$\Psi_1 = (((B + A) + (D + C)) = ((D + C) + (B + A))),$$

$$\Phi[2] = \Phi[1] * \text{SUB}(\sigma, CA), \sigma = \langle A, B \rangle \langle B, A \rangle, \text{ the reverse of the first simplification,}$$

$$\Psi_2 = ((B + A) = (A + B)),$$

$$\Phi[3] = \Phi[2] * RE(1,2,2),$$

$$\Psi_3 = (((A + B) + (D + C)) = ((D + C) + (B + A))),$$

$$\Phi[4] = \Phi[3] * SUB(\sigma, CA), \quad \sigma = \langle A, D \rangle \langle B, C \rangle, \quad \text{the reverse of the second simplification,}$$

$$\Psi_4 = ((D + C) = ((C + D))),$$

$$\Phi[5] = \Phi[4] * RE(3,4,5).$$

The complete theorem-proving algorithm is simply: Given an initial pattern, a rule sequence and a set of allowable premises, G is provable if one of the following steps succeeds.

Step 1. Find a pattern in the pattern sequence that is identical to G (proof by IR5).

Step 2. Determine if G is a substitution instance of an axiom (proof by IRL).

Step 3. Find a premise that is identical to G (proof by IR6).

Step 4. Try Methods 1, 2, 3 and 4, in that order.

Step 5. If G is an identity of the form $(\alpha = \beta)$, try Steps 1-4 for the commuted identity $(\beta = \alpha)$ (the proof is then the proof of $(\beta = \alpha) = \Phi[j]$ concatenated with $CE(j)$.)

This algorithm is presented in Appendix III in a language resembling that of ALGOL. Examples given in this section can be used to check the algorithm.

6. Incompleteness and Decidability

It would be natural at this point to show that if $G \in \mathcal{L}$ is true, then G is provable. That the theory of Abelian groups is decidable was

proven by Wanda Szmielow (1950). But the system of rules specified in Sections 1 through 5, although intended to examine all expressions of an Abelian group, is not complete for that theory; there exists at least one true formula in \mathcal{L} which is not provable. An argument is given at the end of this section to show that one such formula does exist. The inability to know whether an expression can be proved by the theorem-prover introduces some problems in using the system within the framework of a tutorial program. These problems are dealt with in the next chapter. However, a decidable class of expressions in \mathcal{L} does exist. The proof that a derivation can always be found for expressions within this class is given next.

To summarize the results of this section, we are going to prove theorems and lemmas that show that the theorem-prover can find a proof for any identity pattern P of a formula in \mathcal{L} in which we can construct the partial mapping $f' : S(P,1) \rightarrow S(P,1 + \#S(P,1))$ by

1. excluding the identity element from the domain of f' ;
2. mapping any leaf in the left half of P that is not the descendent of a node labeled by a minus sign to an identical leaf in the right half of P ;
3. for any subpattern in the left half that is of the form $(-\alpha)$, mapping the subpattern to a subpattern in the right half that is of the form $(-\beta)$ only if we can construct $f' : \alpha \rightarrow \beta$.

We will make this mapping more explicit in the lemmas and theorems that follow. We note here that such a mapping between sides of an identity is easy to compute. We therefore are able to know immediately whether the theorem-prover is guaranteed to find a proof for the identity.

If the expression is in the form of a conditional and the rightmost consequent is an identity in which we can construct the above mapping,

then it is also easy to show that the theorem-prover can find a proof for the expression. This will be the final extension to the class of decidable formulas presented in this section.

The first lemma, Lemma 7, gives a necessary and sufficient condition for the truth of formulas of a commutative semigroup in which we have no identity elements and shows the form of all true formulas in this system.

Let \mathcal{A}_1 be a commutative semigroup with the additional assumption:

$(\forall x)(\forall y)(x + y \neq x)$. The intended interpretation is addition on the positive integers. The purpose of excluding special identities is so that we know the addition of any term will form a new term with a different value assignment. At no time do other properties of addition enter into the proof of the following lemmas. Lemma 7, in effect, proves that a necessary condition for the truth of a well-formed formula in \mathcal{A}_1 is that both sides of the identity have the same number of occurrences of the same individual variables.

Lemma 7. Let G be a well-formed formula in \mathcal{A}_1 and let w be the right direct descendent of node 0 of $P(G)$. Then G is true if and only if there exists an isomorphism between the leaves of $S(P,1)$ and $S(P,w)$.

Proof. Let m,n be leaves of $S(P,1)$; r,s be leaves of $S(P,w)$. We define the one-one,onto mapping f : leaves of $S(P,1) \rightarrow$ leaves of $S(P,w)$ by $f(S(P,m)) = S(P,r)$ if and only if $l(m) = l(r)$. We observe that all well-formed formulas of \mathcal{A}_1 are of the form $(\alpha = \beta)$, where α is β , or α distinct from β (α, β are well-formed terms). The terms are variables or are built from terms and the binary operator (we will

use $1 + 1$ although the operation could be something other than addition).

We want to show that in each case the expression is true if and only if the isomorphism f exists.

Case I. If f exists, then P is true (and therefore G is true).

a. α is a variable, $\mathcal{L}(m) = \alpha$, m is a leaf. By f , β must also be a variable, $\mathcal{L}(r) = \beta$, r a leaf, and $f(\alpha) = \beta$. Therefore, $\mathcal{L}(m)$ is $\mathcal{L}(r)$ and P is true by truth definition of identity.

b. α is in the form $(\gamma_1 + \gamma_2)$. For each individual leaf A_i of α there is an individual leaf B_i of β . Let $M = \langle D, R \rangle$ be a model for \mathcal{L}_1 , g be a value assignment on D . If $A_1, \dots, A_j \in \text{domain of } f$, then $g'(A_1) + \dots + g'(A_j) \in D$, $g'(A_i)$ is well-defined for $i = 1, \dots, j$. Suppose P is false. Then, where $B_1, \dots, B_j \in \text{range of } f$, we have $g'(B_1) + \dots + g'(B_j)$ distinct from $g'(A_1) + \dots + g'(A_j)$. By the results on pages 17 and 18 of van der Waerden's Modern Algebra (van der Waerden, 1947; his proofs are stated for groups but use only the properties of a commutative semigroup), a formula in \mathcal{L}_1 can be rearranged into a canonical form in which the variables are well ordered with assumed left association of the binary operator. That means that each A_i has a corresponding B_i in the canonical form which, by f , is the same variable. Also, each B_i has a corresponding A_i that is the same variable. So the distinct value assignment is impossible.

Case II. If P is true, then the isomorphism f exists.

a. P is in the form $(\alpha = \beta)$; α is a variable; f does not exist; β is either a variable or in the form $(\eta_1 + \eta_2)$. If β is a variable and α is not, then we have a one-element domain which gives

us that element as an identity, although we assumed an identity does not exist in \mathcal{A}_1 . So β must be α and $f(\alpha) = \beta$. If β is $(\eta_1 + \eta_2)$, then β is, in canonical form, $((B_1 + B_2) + \dots) + B_n$. If the B_i are all α , then α is an identity element. Since P is true for all value assignments, pick one in which B_n is α . Then, by commutivity, β is of the form $B_n + ((B_1 + B_2) + \dots)$. But the summation denotes an element in the domain which is then an identity element. So β cannot be in the form $(\eta_1 + \eta_2)$.

b. α is in the form $(\gamma_1 + \gamma_2)$. β is either a variable or of the form $(\eta_1 + \eta_2)$. If β is a variable, then we can pick an interpretation in which β denotes the same element as γ_1 and therefore γ_2 is an identity element. If α and β do not have the same number of occurrences of variables, we can put α and β in canonical form and group β (we assume β has the greater number of occurrences) such that β is in the form $(\eta_1 + \eta_2)$. η_1 is a term having the same number of occurrences of variables as α . Since P is true, $M = \langle D, R \rangle$ is a model for \mathcal{A}_1 , $g'(\alpha) = g'(\beta)$ for all value assignments g on D . Pick a g such that η_1 and α denote the same element of D . Then η_2 denotes an element in D that is an identity. Therefore, β must have the same number of occurrences of variables as α . Must β have the same number of occurrences of the same variables (in which case, f clearly exists in P)? Suppose α and β are in a canonical form so that P is $(\gamma_1 + \gamma_2) = (\eta_1 + \eta_2)$ and γ_1 has the same number of occurrences of the same variables as η_1 . That is, we have $P : (\gamma_1 + \gamma_2) = (\eta_1 + \eta_2)$. If γ_2 and η_2 are empty, we can construct f . Otherwise, γ_2 and η_2 have the same number of variables, each variable of γ_2 distinct from

those in η_1 . So we can select a value assignment g on D such that, for at least one A_i in γ_2 , B_i in η_2 , $g'(A_i) \neq g'(B_i)$. Hence, we can pick g such that $g'(\gamma_2) \neq g'(\eta_2)$. But then we can also pick g so that $g'(\gamma_1) + g'(\gamma_2) \neq g'(\gamma_1) + g'(\eta_2)$, contradicting the assumption that P is true. Such an assignment of values is impossible if all the variables in α and β are the same. ■

To show that all formulas true in \mathcal{L}_1 are provable, we will show that the theorem-prover can perform all the necessary permutations of terms and reassociations of the parentheses. In such a system, in which we only consider the CA and AS axioms (given in Section 1), formulas have only one of the forms shown in Lemma 8.

Definition 22. Let G be a formula in \mathcal{L}_1 where $P(G)$ has one of the following forms:

- (i) $(\alpha = \alpha)$, $\alpha \in \mathcal{L}_T$;
- (ii) $((\alpha + \beta) = (\beta + \alpha))$, $\alpha, \beta \in \mathcal{L}_T$; or
- (iii) $((\alpha + \beta) + \gamma) = (\alpha + (\beta + \gamma))$, $\alpha, \beta, \gamma \in \mathcal{L}_T$.

Then $PR(P(G))$, a permutation and/or regrouping of $P(G)$, is the smallest set of all P_n such that there exists a sequence $\langle P_n P_{n-1} \dots P_0 \rangle$ such that

- (i) $P_0 = P(G)$ and $\Psi_0 = P(G)$;
- (ii) $P_i = RE(i-1, j, k)$, where $\Psi_{i-1} = P_{i-1}$, $\Psi_i = CA[\sigma]$ or $AS[\sigma]$, for some substitution sequence σ , $k \in \Psi_{i-1}$, $S(\Psi_j, l) = S(\Psi_{i-1}, k)$.

We now show that the set of all true well-formed formulas of \mathcal{L}_1 consists of all patterns P' such that $P' \in PR(P)$, P is a substitution instance of CA or AS, or P is in the form $(\alpha = \alpha)$, $\alpha \in \mathcal{L}_T$.

Lemma 8. A pattern P of a formula in \mathcal{L}_1 is true if and only if P has one of the forms:

(i) $P \in \text{PR}((\alpha = \alpha)), \alpha \in \mathcal{L}_T;$

(ii) $P \in \text{PR}(((\alpha + \beta) = (\beta + \alpha))), \alpha, \beta \in \mathcal{L}_T;$ or

(iii) $P \in \text{PR}((((\alpha + \beta) + \gamma) = (\alpha + (\beta + \gamma))))), \alpha, \beta, \gamma \in \mathcal{L}_T.$

Proof. That $P_0 \in \text{PR}((\alpha = \alpha))$ is true in any intended interpretation of the system is clear. $P_0 \in \text{PR}(\text{CA}[\sigma])$ or $P_0 \in \text{PR}(\text{AS}[\sigma])$, for some substitution sequence σ , is true by Lemma 1. Assuming P_{n-1} is true for P_{n-1} in the form of (i), (ii) or (iii), we must show that P_n in the form of (i), (ii) or (iii) is true. But this follows from Lemmas 1 and 4.

Suppose we have a formula in \mathcal{L}_1 whose pattern P is true. By Lemma 7, we know there exists an isomorphism between the leaves of $S(P, l)$ and the leaves of $S(P, w)$, where w is the right direct descendent of the root of P . The proof of Lemma 7 gives us an effective algorithm for re-ordering $S(P, l)$ so that $S(P, l)$ is syntactically the same as $S(P, w)$ and hence for creating the sequence $\langle P_n, P_{n-1}, \dots, P_0 \rangle$. Then, by definition of PR, $P \in \text{PR}((\alpha = \alpha))$, where α is syntactically $S(P, w)$. \blacksquare

It remains to show that the inference rules will, in fact, be able to obtain proofs of the patterns in \mathcal{L}_1 . This will then prove that the theorem-prover is complete for \mathcal{L}_1 .

Theorem 3. All true formulas in \mathcal{L}_1 are provable by the theorem-prover.

Proof. By Lemma 8 it is sufficient to show that the true formulas of the forms (i), (ii) and (iii) are provable by the theorem-prover. When necessary, we will show the methods followed by the theorem-prover in diagrams. The form of the diagrams was explained and used in Section 5.

That all true formulas of the forms (i), (ii) and (iii) are provable is shown by induction on n , where $P_n \in PR(P)$, P the pattern of a formula in \mathcal{L}_1 . The base of the induction is $n = 0$. Suppose P is in the form (i). If $\alpha \in \mathcal{L}_V$, we can assume P is trivially provable. Suppose α is nonatomic. Then it is either of the form

- (a) $(\beta + \gamma)$,
- (b) $((\beta + \gamma) + \eta)$, or
- (c) $(\beta + (\gamma + \eta))$.

The proofs for (a), (b) and (c), as constructed by Method 1, are shown in Figure 2. In the case of forms (ii) and (iii), P_0 is a substitution instance of CA or AS and is provable by IRL, SUBSTITUTION.

The inductive hypothesis is that the theorem-prover will construct proofs for all patterns $A = P_{n-1} \in PR(P)$, where $\langle P_{n-1}, P_{n-2}, \dots, P_0 \rangle$ exists by virtue of A 's membership in $PR(P)$. It remains to show that the theorem-prover can find a proof for $G = P_n \in PR(P)$, where $\langle P_n, P_{n-1}, \dots, P_0 \rangle$ exists. There are three cases, one for each form of A . We consider, for each case, all possible one-step permutations or reassociations of A .

(i) A is in the form $(\alpha = \alpha)$, $\alpha \in \mathcal{L}_T$. α is either of the form (a), (b) or (c) given above. For (a), G can be $((\beta + \gamma) = (\gamma + \beta))$ or $((\gamma + \beta) = (\beta + \gamma))$, both provable by IRL, SUBSTITUTION. For (b), A is $((\beta + \gamma) + \eta) = ((\beta + \gamma) + \eta))$, G can be one of:

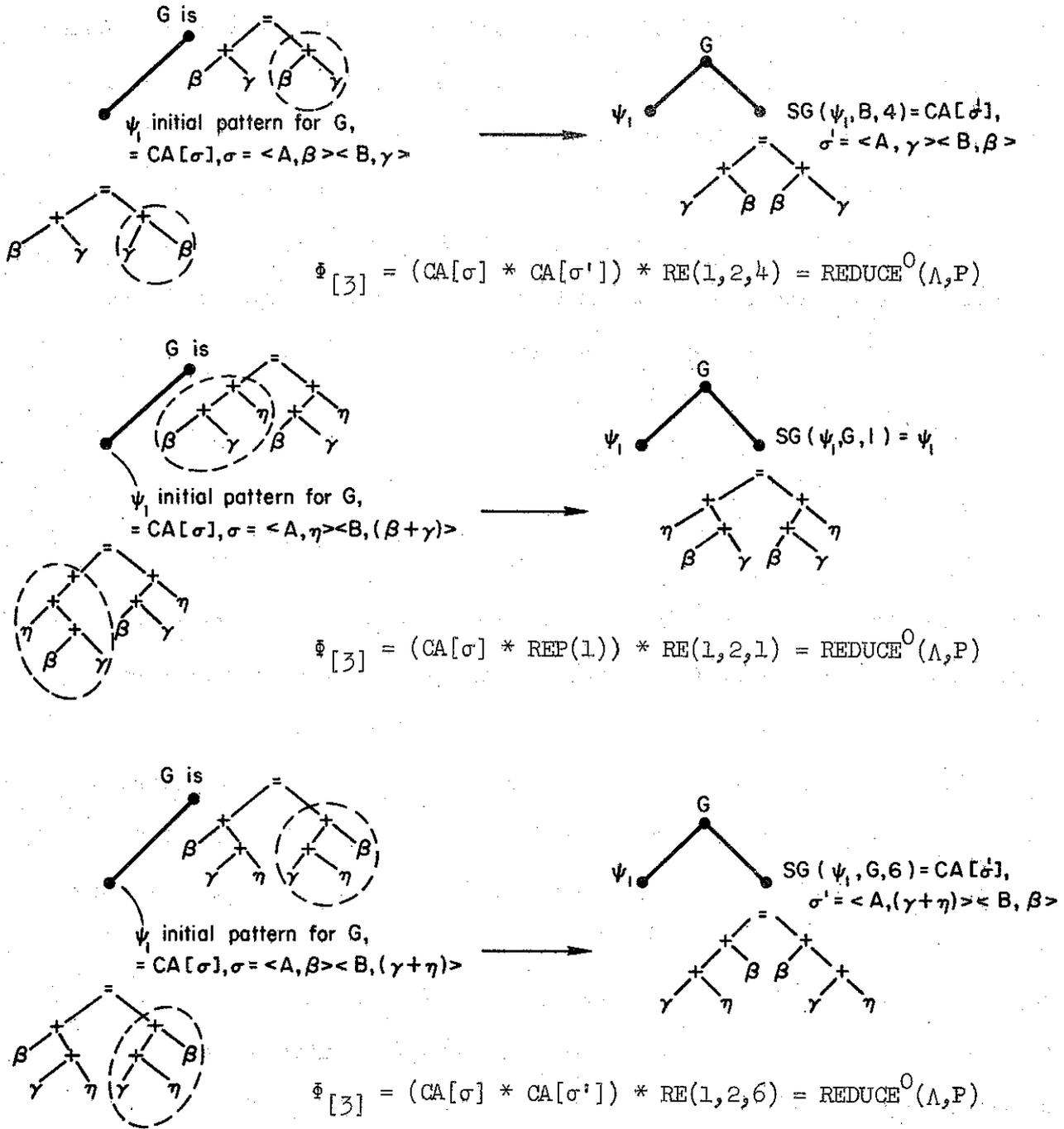


Fig. 2.--Proofs for Theorem 3, Case IIA.

1. $((\beta + (\gamma + \eta)) = ((\beta + \gamma) + \eta))$, using AS to replace P(A,1);
2. $((\eta + (\beta + \gamma)) = ((\beta + \gamma) + \eta))$, using CA to replace P(A,1);
3. $((\gamma + \beta) + \eta) = ((\beta + \gamma) + \eta)$, using CA to replace P(A,2);
4. $((\beta + \gamma) + \eta) = (\eta + (\beta + \gamma))$, using CA to replace S(A,6);
5. $((\beta + \gamma) + \eta) = (\beta + (\gamma + \eta))$, using AS to replace S(A,6);
6. $((\beta + \gamma) + \eta) = ((\gamma + \beta) + \eta)$, using CA to replace S(A,7).

For 1, 2, 4 and 5, the proofs are easily constructed by IRI and IR3. The SUBSET-TEST will replace $(\gamma + \beta)$ by $(\beta + \gamma)$ in 3 and 6. The proof for the resulting pattern, $((\beta + \gamma) + \eta) = ((\beta + \gamma) + \eta)$ was assumed provable by the inductive hypothesis.

For (c), A is $((\beta + (\gamma + \eta)) = (\beta + (\gamma + \eta)))$, G can be:

1. $((\gamma + \eta) + \beta) = (\beta + (\gamma + \eta))$, using CA to replace S(A,1);
2. $((\beta + (\eta + \gamma)) = (\beta + (\gamma + \eta)))$, using CA to replace S(A,3);
3. $((\beta + (\gamma + \eta)) = ((\gamma + \eta) + \beta))$, using CA to replace S(A,6);
4. $((\beta + (\gamma + \eta)) = (\beta + (\eta + \gamma)))$, using CA to replace S(A,8).

For 1 and 3, proofs are constructed by IRI and IR3. For 2 and 4, proofs are obtained from SUBSET-TEST and the inductive hypothesis as noted above.

(ii) A is in the form $((\beta + \gamma) = (\gamma + \beta))$, G can be

$((\gamma + \beta) = (\gamma + \beta))$ or $((\beta + \gamma) = (\beta + \gamma))$, shown to be provable by the diagrams in Figure 2.

(iii) A is in the form $((\beta + \gamma) + \eta) = (\beta + (\gamma + \eta))$, G can be:

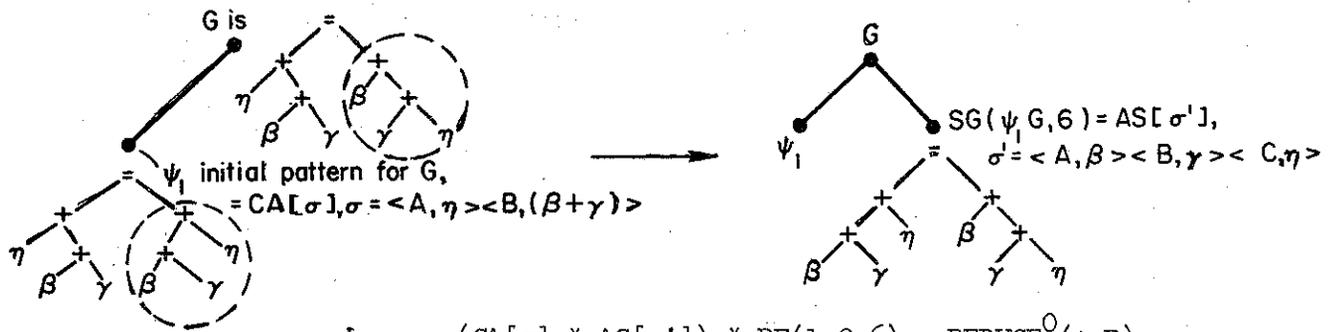
1. $((\eta + (\beta + \gamma)) = (\beta + (\gamma + \eta)))$, using CA to replace S(A,1);
2. $((\beta + (\gamma + \eta)) = (\beta + (\gamma + \eta)))$, using AS to replace S(A,1);
3. $((\gamma + \beta) + \eta) = (\beta + (\gamma + \eta))$, using CA to replace S(A,2);
4. $((\beta + \gamma) + \eta) = ((\gamma + \eta) + \beta)$, using CA to replace S(A,6);
5. $((\beta + \gamma) + \eta) = (\beta + (\eta + \gamma))$, using CA to replace S(A,7).

We assume 2 is provable; for the others, proofs in the form of diagrams are shown in Figure 3. By the induction, we have shown that the theorem-prover can find proofs for all true formulas in \mathcal{A}_1 . ▣

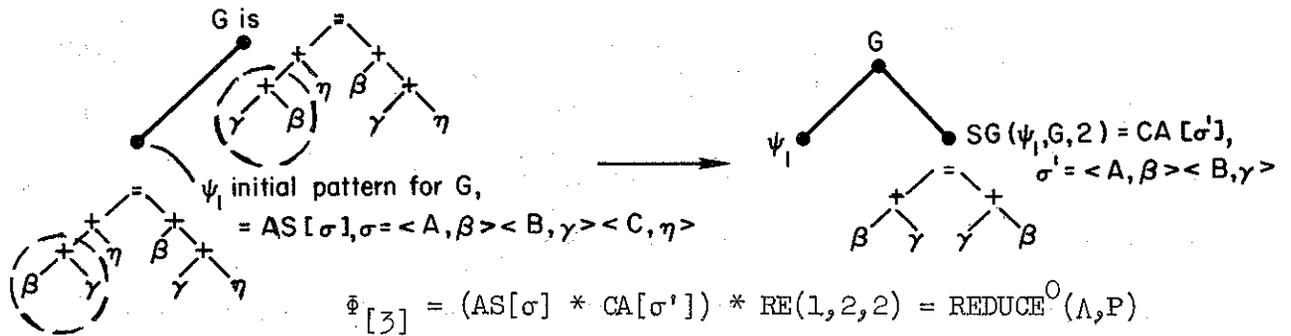
We can now add a particular identity element to \mathcal{A}_1 and call that system \mathcal{A}_2 . With \mathcal{A}_1 , the intended interpretation was addition over the positive integers. Suppose we now want to consider the identity 0, but do not want to consider any other identity elements in the system. (The reader may note at this point that we are building up to a system that is an Abelian group.) That is, define \mathcal{A}_2 to be a commutative monoid with the additional restriction that $(\forall x)(\forall y)(x + y = x \Rightarrow y = 0)$.

Theorem 4. All true formulas in \mathcal{A}_2 are provable by the theorem-prover.

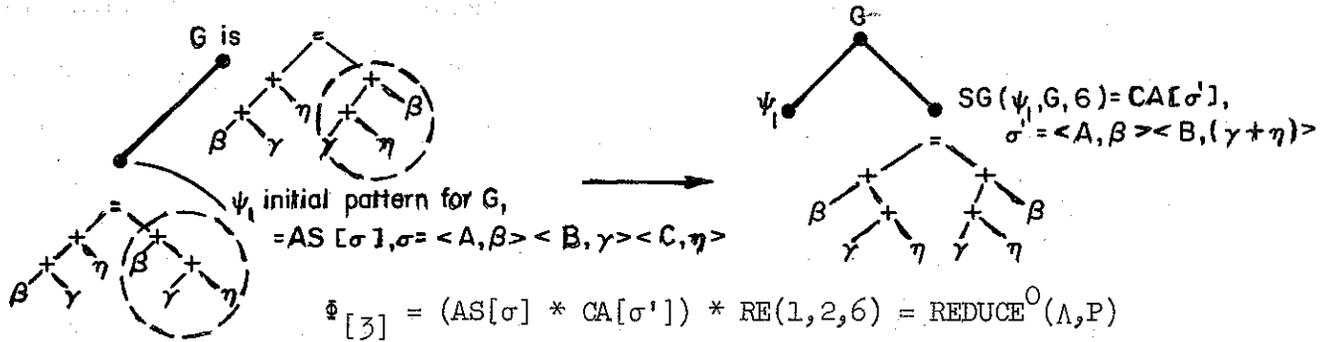
Proof. The proof is by induction on the number of excess occurrences of identity elements (difference between the number of zeros in either side of the identity). Clearly, if the formula we are dealing with has no leaves labeled by 0, we can construct the mapping f because without zero the true formula in \mathcal{A}_2 is in \mathcal{A}_1 . For the base of the induction, we suppose that we have the pattern P of a formula in \mathcal{A}_2 such that the number of occurrences of leaves in the left half of P is the same as the number of occurrences of leaves in the right half. Hence we can construct the isomorphism $f : S(P,1) \rightarrow S(P,1 + \#S(P,1))$ as presented in Lemma 7. Lemmas 7 and 8, and Theorem 3 will still hold. The theorem-prover can construct a proof of P if f can be constructed for P ; we showed f can always be constructed for P with zero excess identity element occurrences.



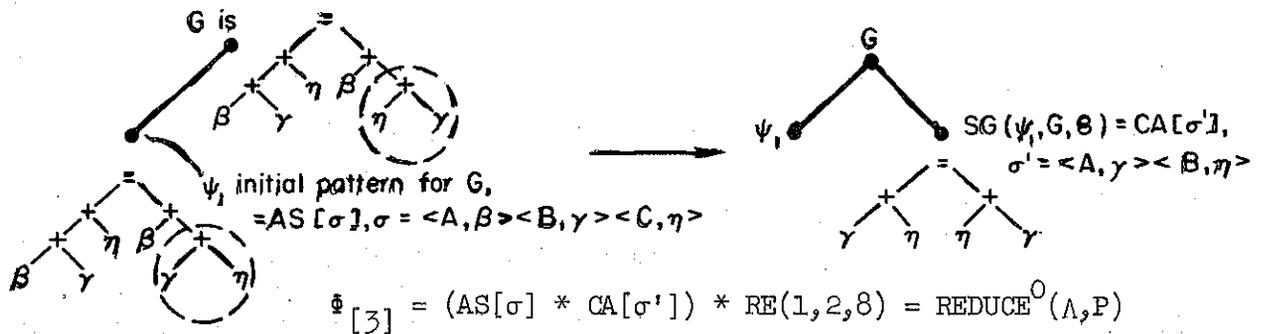
$$\Phi[3] = (CA[\sigma] * AS[\sigma']) * RE(1, 2, 6) = REDUCE^0(\Lambda, P)$$



$$\Phi[3] = (AS[\sigma] * CA[\sigma']) * RE(1, 2, 2) = REDUCE^0(\Lambda, P)$$



$$\Phi[3] = (AS[\sigma] * CA[\sigma']) * RE(1, 2, 6) = REDUCE^0(\Lambda, P)$$



$$\Phi[3] = (AS[\sigma] * CA[\sigma']) * RE(1, 2, 8) = REDUCE^0(\Lambda, P)$$

Fig. 3.--Proofs for Theorem 3, Case IIB.

Define the \mathcal{A}_1 -equivalent of a pattern P in \mathcal{A}_2 as the normal form of P using subtree replacement rules

$$E1. \quad \begin{array}{c} + \\ / \quad \backslash \\ \square \quad 0 \end{array} \rightarrow \square ,$$

and

$$E2. \quad \begin{array}{c} + \\ / \quad \backslash \\ 0 \quad \square \end{array} \rightarrow \square .$$

The normal form of $P \in \mathcal{A}_2$ is clearly a pattern in \mathcal{A}_1 . The point here is that the left-hand and right-hand sides of the normal form of P have the same value assignments as the left-hand and right-hand sides of P , so the normal form is also true. By Lemma 7, we can construct f for the normal form of P . The converse situation is also true since the addition of the identity element does not alter the value assignment of a term. Thus we can identify a true expression in \mathcal{A}_2 by forming the \mathcal{A}_1 -equivalent of \mathcal{A}_2 from rules E1 and E2 and determining if we can construct f for its \mathcal{A}_1 -equivalent.

As the inductive hypothesis we assume that the theorem-prover can construct a proof for a pattern $P \in \mathcal{A}_2$ in which we have $(n - 1)$ excess occurrences of identity elements for which we can construct the mapping f in the \mathcal{A}_1 -equivalent of P . To show that the theorem-prover can construct a proof for any P fulfilling these conditions, we assume P is in the form $(\alpha = \beta)$.

The excess occurrences of the identity element are in α or in β . Assume they are in α . An initial pattern I for P is $Z[\sigma]$, $\sigma = \langle A, \beta \rangle$. I is $((\beta + 0) = \beta)$. Suppose k belongs to the difference set of P and I . Then $SG(I, P, k)$ is the subgoal $((\beta + 0) = \alpha)$. The subgoal has $n - 1$

excess occurrences of the identity element. By the inductive hypothesis, the subgoal is provable. Suppose the proof is $\Phi[j] = I * \Phi[j-1]$. Then the goal P is provable by application of $RE(l,j,k)$, $IR2$.

If the excess occurrences of the identity element are in β , then Step 5 of the theorem-proving algorithm will search for a proof of $(\beta = \alpha)$ and then apply $IR3$, $COMMUTE EQUALS$. So the above analysis of P with $Z[\sigma]$, $\sigma = \langle A, \alpha \rangle$, as an initial pattern, can be carried out for this case. ▣

We now want to extend the class of formulas for which the theorem-prover can find a proof from \mathcal{L}_2 by admitting an inverse operation '-'. This is a unary operator such that $(\forall A)((A + (-A)) = 0)$. The addition of this assumption brings us to an Abelian group.

To extend our analysis, we extend the mapping f to include the cases in which the leaf is the descendent of a node labeled by the unary operator, '-'. The new mapping f' becomes:

- (i) If $f'(S(P,m)) = S(P,r)$ and $f'(S(P,m)) = S(P,s)$, then $r = s$;
- (ii) If $f'(S(P,m)) = S(P,r)$ and $f'(S(P,n)) = S(P,r)$, then $m = n$;
- (iii) If m and r are leaves that are not descendents of nodes labeled by '-', then $f'(S(P,m)) = S(P,r)$ if and only if $\ell(m) = \ell(r)$;
- (iv) If $\ell(m) = \ell(r) = '-'$, then $f'(S(P,m)) = S(P,r)$ if and only if it is possible to construct $f' : S(P,m+1) \rightarrow S(P,r+1)$.

(v) For every leaf m in $S(P,1)$, there exists a leaf r in $S(P,1 + \#S(P,1))$ such that $f'(S(P,m)) = S(P,r)$; for every leaf r in $S(P,1 + \#S(P,1))$, there exists a leaf m of $S(P,1)$ such that $f'(S(P,m)) = S(P,r)$.

Our first problem is to answer the question of whether a formula for which we can construct f' is necessarily true.

Lemma 9. Let $m, n \in S(P, 1)$ and $r, s \in S(P, 1 + \#S(P, 1))$, where none of $S(P, m)$, $S(P, n)$, $S(P, r)$ nor $S(P, s)$ is 0. Then the pattern of any formula in an Abelian group for which it is possible to construct the mapping f' is true.

Proof. Let M be a model for \mathcal{L} , $M = \langle D, R \rangle$, and g be a value assignment on D . We have already shown (Lemma 7 and Theorem 4) that a pattern in which we can construct f' based only on (i), (iii) and (v) is always true. It suffices here to show that formulas that include terms governed by the unary operator '-' are also true. That is, we have to show that, for subpatterns $S(P, m)$, $S(P, r)$ such that $f'(S(P, m)) = S(P, r)$ by (iv), $g'(S(P, m)) = g'(S(P, r))$. Then we will have that, for each term α in $S(P, 1)$ there is a unique corresponding term β in $S(P, 1 + \#S(P, 1))$ (and vice versa) such that $g'(\alpha) = g'(\beta)$ or α is zero. The rest of the proof will then follow in the same manner as in Lemma 7.

Suppose $f'(S(P, m)) = S(P, r)$ by (iv). By the recursive definition of f' , for any leaf n of $S(P, m)$, there is a leaf s of $S(P, r)$ such that either (a) n and s are leaves and $f'(S(P, n)) = S(P, s)$; (b) $\ell(n)$ is 0; or (c) n and s are descendants of nodes k and t , $\ell(k) = \ell(t) = '-'$ and $f' : S(P, k + 1) \rightarrow S(P, t + 1)$. Clearly the recursion stated in (c) only terminates when we have mapped every leaf of $S(P, m)$ not labeled by zero to a unique leaf of $S(P, r)$ such that the labels of the two leaves are the same (and we have also mapped every leaf of $S(P, r)$). But this is the same as saying that for every leaf α of $S(P, m)$ (every leaf γ of

$S(P,r)$), either the label is zero or there is a leaf β of $S(P,r)$ (η of $S(P,m)$) such that $g'(\alpha) = g'(\beta)$, ($g'(\gamma) = g'(\eta)$). This is so because $f'(S(P,\alpha)) = S(P,\beta)$ implies that $l(\alpha) = l(\beta)$, ($l(\gamma) = l(\eta)$). \square

Having already shown that the theorem-prover only constructs proofs for true formulas, we will now show that the true formulas in an Abelian group (for which the mapping f' exists) can be proved by the theorem-prover. This does not mean that all true formulas in an Abelian group can be proved by the theorem-prover. An example for which f' does not exist and for which the prover cannot construct a proof is given in Theorem 6.

Theorem 5. All expressions in an Abelian group for which it is possible to construct the mapping f' are provable by the theorem-prover.

Proof. Note that this extension to \mathcal{L}_2 does not include any additional restrictions on particular identities because there is only one general identity in an Abelian group and no special identity.

Suppose none of Steps 1-3 of the theorem-proving algorithm succeeds in finding a proof of P , the pattern for an expression in which f' can be constructed. Suppose also that Methods 1, 2, and 3 fail. Then the prover will try Method 4, the SUBSET-TEST. In this method, the prover examines every subpattern in the pattern in relation to each of the others, attempting to prove an equivalence of nonidentical subpatterns. Thus, for $f'(S(P,m)) = S(P,r)$, $l(m) = l(r) = '-'$, we know that the prover will try to find a proof for the subgoal $S(P,m+1) = S(P,r+1)$. Suppose there is no node w in either $S(P,m+1)$ or $S(P,r+1)$ such that $l(w) = '-'$.

Then, by Theorem 4, the theorem-prover can find a proof for the subgoal.

Having found this proof, Method 4 replaces $S(P, m + 1)$ by $S(P, r + 1)$.

We can then consider $S(P, m)$ and $S(P, r)$ to be the identical term, α , eliminating the minus sign from the form of P .

Since Method 4 searches in P for all subpatterns that are equivalent, it will eventually examine all such subgoals of the form $S(P, m + 1) = S(P, r + 1)$ and eliminate the minus sign from the form of P . Since the proof procedure is iterative, it will encounter subgoals of the form $(\gamma = \delta)$ such that $(-\gamma')$ is in γ , $(-\delta')$ is in δ , and γ' is the same as δ' because the procedure has already shown an equivalence between these subpatterns. Thus we can consider $(-\gamma')$ to be identical to $(-\delta')$. So we can extend the mapping f (in Lemma 7) to include $f(S(P, m)) = S(P, r)$, where m and r are not leaves and $\mathcal{L}(m) = \mathcal{L}(r) = '-'$, if and only if $S(P, m) = S(P, r)$. Thus, regardless of the number of such subpatterns $(-\gamma')$ in γ and $(-\delta')$ in δ , $f' : (-\gamma') \rightarrow (-\delta')$ will mean that $f : (-\gamma') \rightarrow (-\delta')$. It should then be easy to see that Theorem 4 can be extended to show that cases like $(\gamma = \delta)$ are provable.

Again, because the procedure is iterative, it will continue to examine P , eliminating the minus signs in this manner until it has constructed a $P' = \text{SUBSET}^n(\Lambda, P)$ such that P' is in the form of an expression in \mathcal{L}_2 . By Theorem 4, P' is provable and Method 4 succeeds. Therefore, P is provable. ▀

We make one further extension to the class of formulas for which we can show that the theorem-prover can always find a proof. Suppose the form of the pattern P is $(H \Rightarrow \chi)$ and χ is an identity $(\alpha = \beta)$ in which

we can construct the partial mapping $f' : \alpha \rightarrow \beta$. Then, using Method 2, CONDITIONAL-PROOF, we can prove P by (1) assuming H is true (IR6); (2) proving χ (known to be provable by Theorem 5); and (3) applying IR7 in order to form the conditional statement.

Recursively using Method 2, we can also find a proof for P in the form $(H \Rightarrow (\Gamma \Rightarrow \chi))$, where χ is in the form $(\alpha = \beta)$ and we can construct $f' : \alpha \rightarrow \beta$. The proof is constructed by the following steps.

1. $\Phi_1 = \text{AR}(H)$ [first attempt to use Method 4],
2. $\Phi_2 = \text{AR}(\Gamma)$ [first recursive use of Method 4],
3. $\Phi_{[j]}$ is a proof of χ given $\Phi_{[2]}$ [known to be provable by Theorem 5],
4. $\Phi_{j+1} = \text{CP}(2, j)$,
5. $\Phi_{j+2} = \text{CP}(1, j + 1)$, $\Psi_{j+2} = P$.

Clearly we can extend steps 1 and 2, and 4 and 5 in order to handle P in the form $(H \Rightarrow (\Gamma_1 \Rightarrow (\Gamma_2 \Rightarrow \dots (\Gamma_m \Rightarrow \chi) \dots)))$. Therefore, using Method 2 recursively, the theorem-prover can find a proof for any pattern in the form of a conditional if the rightmost consequent is in the form $(\alpha = \beta)$ and we can construct $f' : \alpha \rightarrow \beta$.

Our final task in this section is to prove that we cannot find proofs for all true formulas in an Abelian group. We do so by providing a counterexample, a true expression in an Abelian group for which the theorem-prover cannot construct a proof.

Theorem 6. Let α and $\beta \in \mathcal{L}_T$, α distinct from β . If neither α nor β can be reduced to 0, and G is $(((-(\alpha + \beta)) + \beta) = (-\alpha))$, then G is not provable.

Proof. We show that a proof for G cannot be obtained by the theorem-prover by demonstrating that none of Steps 1-5 of the algorithm succeed. We assume that the initial rule sequence is empty and that there are no premises. A set of axioms for the theory of Abelian groups was given in Section 1: $AX = \{CA, AS, AI, Z, N\}$. (We now consider the definitional axiom, N .)

Steps 1-3 clearly do not succeed. Method 2 fails because G is not in the form of a conditional. Method 3 fails because there are no axioms that are in the form of a conditional. The following is a list of all the subpatterns of G :

$$\alpha, \beta, (-\alpha), (\alpha + \beta), (-(\alpha + \beta)), ((-\alpha + \beta)) + \beta .$$

No two of these subpatterns are equivalent, except for the two sides of G which we do not consider in Method 4. No subgoal set up by this simplification method is a true formula and, by Theorem 2, none is provable by the theorem-prover. Therefore, $SUBSET^0(\Lambda, G) = SUBSET^1(\Lambda, G) = G$, and Method 4 does not succeed.

We want to determine whether G is provable by Method 1. The definition of initial pattern chooses only two initial patterns for G : (1) $CA[\sigma]$, $\sigma = \langle A, (-\alpha + \beta) \rangle \langle B, \beta \rangle$ and (2) $Z[\sigma']$, $\sigma' = \langle A, (-\alpha) \rangle$. A mapping fulfilling the conditions of f' can be constructed for each of these two initial patterns. Note that this is not the case for G ; we cannot construct f' for G . To show that the theorem-prover cannot construct a proof of G by Method 1, it is sufficient to show that Method 1 cannot change a pattern in which f' can be constructed into one in which f' cannot be constructed. Method 1 changes a pattern by replacing terms α in that pattern by terms β such that $\alpha = \beta$ is true. For example, if α is a term in the pattern

and there is an instance of an axiom that is $\alpha = \beta$, then Method 1 can replace α by β . But each instance of an axiom is true and we can construct $f' : \beta \rightarrow \alpha$. Suppose in the pattern we constructed f' such that $\alpha \rightarrow \gamma$. Then in the new pattern we map $\beta \rightarrow \gamma$; that is, we can construct f' in the new pattern.

Method 1 will not change an initial pattern in which we can construct f' into G in which f' cannot be constructed. So Method 1 fails.

The above analyses for G in Steps 1-4 are identical for the commuted form of G . Therefore, Step 5 does not succeed. No proof for G can be constructed by the theorem-prover. ▣

Although we do not have a general enough statement to warrant a separate lemma, it is significant to point out that we have shown in the proof of Theorem 6 that if Method 1 only chooses initial patterns for G in which f' can be constructed, then Method 1 will not provide a proof of G if f' cannot be constructed for G .

Theorem 7. There are $G \in \mathcal{L}$ such that G is not provable by the theorem-prover while G is true. (That is, the theorem-prover cannot find a proof for all true expressions in an Abelian group.)

Proof. Observe that G of Theorem 6 is the pattern for a formula in \mathcal{L} . G is clearly true in the intended interpretation, but, by Theorem 6, G is not provable by the theorem-prover. ▣

7. Some Heuristics Used by the Theorem-Prover

In the previous sections of this chapter we presented an idealization of the theorem-prover used in the research on response analysis presented in subsequent chapters. We want to turn to an informal discussion of how that idealization differs from the actual computer program realization of the theorem-prover. The difference lies only in a number of heuristic strategies used to find the proofs in less computation time. The results of Section 6 provide a decision procedure for determining whether or not the theorem-prover can definitely find a proof for a given expression. These results are not lost by inclusion of the heuristics we will now discuss. However, we will not offer proofs, here, of this last statement.

Definition 13, which specifies what is meant syntactically by an "initial pattern," suggests only one of several possible methods for choosing the first line of a derivation. We could have defined a much more inclusive choice method. For example, we could compute all the possible combinations of substituting variables in the axiom by each term in the goal expression. With this method, if the goal is $((-(A + B)) + B) = (-A)$ and the axiom is AI: $((A + (-A)) = 0)$, then possible initial patterns include AI[σ] where $\sigma = \langle A, -(A + B) \rangle$, $\sigma = \langle A, (A + B) \rangle$, $\sigma = \langle A, A \rangle$, $\sigma = \langle A, B \rangle$, or $\sigma = \langle A, (-A) \rangle$. Note that we only consider terms appearing in the goal since introducing new terms is in general unnecessary for solving elementary derivation problems. Such a method for choosing initial patterns would greatly increase the number of solution paths to search and, therefore, the amount of time the theorem-prover needs.

The actual computer-tutor is, as mentioned earlier, expected to operate within reasonable time constraints (reflecting the amount of time a student

should be expected to wait for the tutor to respond). Consequently, rather than providing a more inclusive choice method, we implemented one which cuts down the number of possible initial patterns to be tried by the theorem-prover. We call this method the FEATURE TEST; it is an adaptation of the Logic Theorist's similarity test.

The basis of the FEATURE TEST is to examine general properties of formulas to compare them for similarities in mathematical content as well as syntactic form. For example, in addition problems, the use of binary subtraction is mathematically the same as adding a negative number. (For this reason, the theorem-prover liberally replaces binary minus by "+-" whenever the program experiences difficulty finding a solution.) If an initial pattern is chosen by the FEATURE TEST, then the student can be told what similarities spawned the choice, and, furthermore, the student can be taught to search for and recognize these similarities.

To describe the FEATURE TEST algorithm, we need to introduce some further vocabulary. A feature, F^i , is a primitive element of the theorem-proving system. For example, we could, in the addition problems, let F^1 be the identity element, and F^2 the minus sign. In the set S , we store the items we want the FEATURE TEST to consider. S always includes (1) TV , the total number of variable places in the right-hand and the left-hand sides of the expression; (2) DV , the total number of distinct variables on each side of the expression; and (3) all F^i . We will suppose that S contains k elements.

$DATA$ is the set of initial patterns relative to G , the problem expression, as defined by Definition 13, Section 5. These patterns are chosen with respect to AX , where AX is the set of all axioms and theorems already

in the student's command language (i.e., already learned by the student), less any which the curriculum-writer has specified may not be used by the student in the present problem. In this manner, the tutor does not suggest using axioms or theorems not known to, or usable by, the student. The algorithm is then as follows.

With each element of DATA, associate a set of attribute-value pairs. The attributes are elements of S; the value of each attribute is computed by the following procedure.

Step 1. Set $TABLE = DATA \cup \{GOAL\}$.

Step 2. If TABLE is empty, then terminate; otherwise, set $N =$ first name on TABLE.

Step 3. Set $j = 1$.

Step 4. If $j > k$, then set $TABLE = TABLE - \{N\}$; go to Step 2.

Step 5. Compute the value of the feature S_j for the left-hand and then the right-hand sides of each pattern in DATA. The value is simply the numerical count of the number of occurrences of S_j on either side of the expression referred to by a name in DATA. Store this pair of values (V_L, V_R) as the value for the attribute S_j associated with the pattern in DATA.

Step 6. Set $j = j + 1$. Go to Step 4.

The names placed in the set INITIAL, according to the steps given below, will be the choices for initial patterns to be used in the REDUCE proof method. We let SET be a set of attribute-value pairs, (S_j, L_j) , where L_j is the list of each element of DATA such that the value paired with S_j for that pattern is the same as the value paired with S_j for the GOAL. Let m be the number of elements in DATA.

Step 1. Set $j = 1$.

Step 2. If $j > k$, then terminate; otherwise, set $L_j = \phi$,
set $i = 1$.

Step 3. If $i > m$, then pair L_j with the attribute S_j in SET;
set $j = j + 1$; go to Step 2.

Step 4. If both values V_l and V_r of the attribute S_j for the
 i th pattern in DATA are equal to the values of the attribute S_j for
GOAL, then set $L_j = L_j \cup \{i$ th element of DATA $\}$.

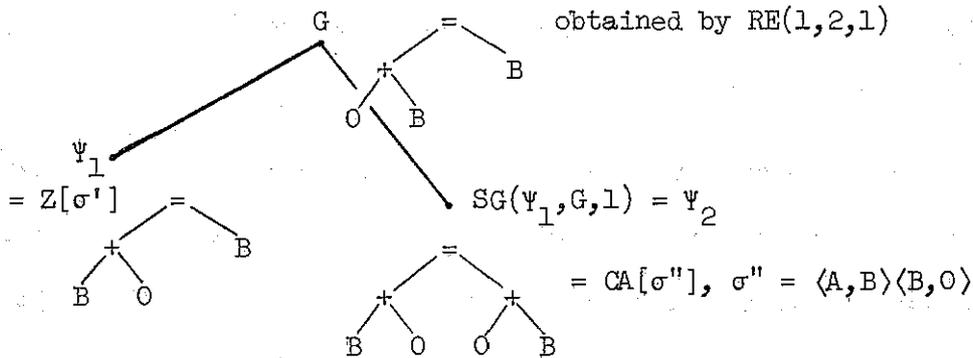
Step 5. Set $i = i + 1$; go to Step 3.

SET now associates with each attribute S_j a set of patterns whose
value for that feature is identical with that of the goal. These sets are
now combined in such a manner as to leave the best choice, that is, the one
with the most features identical with the goal, as the first element of the
ordered set INITIAL. The task is a simple one. For each combination of
 i S_j 's, form the intersection of L_j 's, where $i = k, k-1, \dots, 2$. Thus,
form $\binom{k}{k} + \binom{k}{k-1} + \dots + \binom{k}{2}$ intersections in all. As each intersection
is computed, form its union with the preceding set, while preserving the
order in which the new items are added to the set. Since the FEATURE TEST
is an attempt to avoid a breadth-first search (albeit an ordered one), we
say that each pattern in INITIAL must have at least two features in common
with the goal.

To compute INITIAL for the GOAL = $((O + B) = B)$, where
 $AX = \{CA, Z, AS, N, AI\}$ and $S = \{TV, DV, O, -\}$, we first compute the values
 V_l/V_r for each possible initial pattern for GOAL and for the GOAL
itself.

	TV	DV	O	-
1. CA[σ], $\sigma = \langle A, 0 \rangle \langle B, B \rangle$	2/2	2/2	1/1	0/0
2. Z[σ'], $\sigma' = \langle A, B \rangle$	2/1	2/1	1/0	0/0
GOAL	2/1	2/1	1/0	0/0

Then $SET = \{(TV, (1))(DV, (1))(O, (1))(-, (1, 2))\}$. The result is clearly
 INITIAL = $\{Z[\sigma']\}$. The proof is quickly obtained as



Other theorem-proving heuristics include the COMMUTE-EQUALS rule in which we try to use Method 1, REDUCE, on the commuted form of the goal before trying the other methods (this eliminates Step 5 of the proving algorithm). Any subgoal tried but not solved during the attempt to solve the original goal is not repeated. ADD-ZERO, adding a zero to one side of the goal, attempts to compensate for information possibly lost by the FEATURE TEST because of the lack of an identity element in the goal. REDUCE is retried with the new goal. If it succeeds, the zero axiom is used to eliminate the additional zero. MACROS, following Slagle's standard transformation approach in his calculus program, attempts to recognize standard forms of goals. It makes use of common sequences of inference rules to eliminate terms from one side of the goal and to regroup parentheses.

Method 1, including the heuristics, is called MATCH in subsequent chapters. In step 1 of the proof method, we see if the goal, or its computed form, is already in the partial proof (INPROOFALREADY). Step 2 checks to see if the goal is a substitution instance of an axiom or theorem, while step 3 is the modified REDUCE procedure that uses the FEATURE TEST for choosing initial patterns. Steps 4-6 incorporate the three heuristics mentioned above.

Having examined the mathematical idealization of the theorem-prover and having discussed modifications of that idealization in terms of heuristic strategies for decreasing the computation time of the computer realization of the prover, we now turn to the prover's application as a tool for response analysis in a CAI system.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that every entry should be supported by a valid receipt or invoice. This ensures that the financial statements are reliable and can be audited without any discrepancies.

The second part of the document outlines the procedures for handling cash payments and receipts. It states that all cash transactions must be recorded in the cash book immediately after they occur. This helps in maintaining a clear and up-to-date record of the company's cash flow.

The third part of the document describes the process of reconciling bank statements with the company's records. It explains that the bank statement should be compared with the cash book entries to identify any differences. These differences are then investigated and explained, such as bank charges or errors in recording.

The fourth part of the document discusses the preparation of the profit and loss account. It shows how the various expenses and incomes are categorized and summed up to determine the net profit or loss for the period. This is a crucial step in assessing the financial performance of the business.

The fifth and final part of the document provides a summary of the key points discussed. It reiterates the importance of accuracy, regular reconciliation, and proper documentation in maintaining the financial records of the business.

CHAPTER V

THE THEOREM-PROVER AS PROOF-ANALYZER

In the preceding chapter, a system of inference rules and an algorithm for carrying out those rules, a theorem-prover, were specified. The objective of this chapter is to show how that theorem-prover can be altered so that it can assume the role of a proof-analyzer, that is, so that the theorem-prover's attempt to construct a proof will take into account work already done by the student towards finding a solution to the problem. Three essential changes must be made to the computer realization of the theorem-prover so that (a) more than one complete proof may be constructed, providing the computer-tutor a choice of proofs to discuss with the student; and (b) proof steps already given by the student will be referenced whenever they are useful in generating a complete proof. The three changes effectively assure that

1. the theorem-prover will not halt after finding one proof, but will continue its search;
2. whenever the theorem-prover decides to construct a subsidiary derivation, it will check to see if the student has already accomplished the task; and
3. if no proofs can be found by the theorem-prover, it will see if it can find a solution by reducing the difference between the problem expression and the last line generated by the student.

Change (1) is explored in detail in Section 1. Change (3) is simply accomplished by calling on Method 1, REDUCE (Φ), where Φ is the student's partial proof. The initial pattern is forced to be the student's last line and the problem expression is the goal. Change (2) requires preferred use of the REP inference rule over other means of obtaining a given expression. Redundant lines in the student's work create a problematic choice for the theorem-prover: which of the identical lines should be referenced? This is resolved by always choosing the line that was generated last. The idea is to try to capture the strategy on which the student was working just before requesting help.

Chapter IV demonstrated that the theorem-prover is able to discover proofs for expressions in the theory of Abelian groups and that these proofs resemble those which the students might be expected to discover. Sample proofs for each of twenty-one theorems on addition are shown in Appendix II. The theorems were taken from the curriculum of the Stanford logic-algebra program; Appendix II was reproduced from computer output generated by the theorem-prover. In this chapter, "theorem-prover" will denote the computer realization of the idealized prover described in Chapter IV with the heuristics of Section 7, Chapter IV.

The description of the instructional system on which the students work their problems is left to the next chapter. However, the question arises here as to how the theorem-prover communicates with that instructional system. It is sufficient to answer that (a) the instructional system and the theorem-prover share common storage area for saving proof lines, so the prover has immediate access to the student's work; and (b) both programs rely on similar pattern-manipulation rules. The theorem-prover produces information about possible completions of the student's partial proofs; this information is given to the computer-tutor

in a manner described later in this chapter. Using a simple example, $((O+B)=B)$, Section 2 examines how well the analyzer adapts to different information about the student's responses as the responses differ.

In the final sections of this chapter, the proof-analyzer is viewed from a user's standpoint. A case study of the interaction between a seventh grader, Lisa, and the computer-tutor was carried out in order to test the proof-analyzer on a fairly representative sample of problems taken from the Stanford curriculum. In terms of this study and the problems analyzed, it is possible to discuss the tutorial dialogues and to demonstrate the adaptive nature of the tutoring. To further illustrate the analysis process, partial solutions were then typed out, rather than the hints shown in the dialogue. Section 3 couples the sample dialogue and the set of completed solutions with annotations about individual derivation problems in order to highlight the computer's tutorial capabilities.

The ability to generate relevant dialogue aimed at helping a student complete a derivation depends on the information (the set of completed solutions) that the theorem-prover, acting as a proof-analyzer, can secure. Armed with this information, a dialogue routine can generate hints or advice directly from the theorem-prover's suggestions. This routine can further examine the student's work to see if axioms or theorems suggested by the theorem-prover have already been requested, but with different substitution sequences. It can compare the lines of the student's proof for redundancy, ensure that the theorem-prover itself has not duplicated work already done by the student, and so on.

If the theorem-prover does not find one or more solutions, the dialogue routine can draw on two theorem-prover heuristics in order to

generate a hint as to the next line of the proof: (a) call on the FEATURE TEST for a feasible way to start a solution; or (b), using the broader definition of initial pattern (as given in Section 7, Chapter IV), find an instance of an axiom or theorem, I, not found by (a).

As an example take the expression used in Chapter IV as a counterexample to prove the incompleteness of the set of inference rules which make up the theorem-prover: $((-(A+B))+B)=(-A)$. As shown in the proof of incompleteness (Theorems 6 and 7), the possible initial patterns are $CA[\sigma]$ and $Z[\sigma']$, where CA and Z are axioms, $\sigma = \langle A, -(A+B) \rangle \langle B, B \rangle$, and $\sigma' = \langle A, (-A) \rangle$. The FEATURE TEST rules out all the axioms. By explaining Definition 1, Chapter IV, to the student, the program can suggest that the student begin his proof with either $CA[\sigma]$ or $Z[\sigma']$. Although difficult to construct, there is, in fact, a valid proof which begins with $Z[\sigma']$. Using heuristic (b), the tutor can also find $AI[\sigma'']$, $\sigma'' = \langle A, (A+B) \rangle$, as an initial pattern. Again a proof can be constructed from this pattern. In this case, the tutor's ability to generate hints ends with the initial suggestions. In most cases, one of the above heuristics will indicate an initial statement to try. Good examples of a student's reaction to such hints appear as problems numbered 6 and 11 in the sample dialogue (Figure 12), and in the annotation to Figure 13.

Two possibilities now exist for further helping a student working on the above problem, $((-(A+B))+B)=(-A)$, depending on the given situation. One, the student may have requested help without typing any commands. The tutor can resort to discussing a stored proof. This is not a contradiction to the philosophy on which the proof-analyzer was based. It is just a practical device, used even for problems which the theorem-prover can solve. If the student did no work, the tutor need not expend time generating all possible

solutions. The solutions can be generated in advance of student interaction with the instructional system, and stored for later reference.

Second, the student might have constructed a partial proof which serves as a hint to the theorem-prover because the theorem-prover can try to reduce the difference between the student's last line and the problem expression. (In subsequent figures we call this heuristic DIFFERENCE test.) Suppose the student typed:

$$\underline{AI} \quad A+(-A)=0$$

$$A::\underline{A+B} \quad (1) \quad (A+B)+(-(A+B))=0$$

The theorem-prover will use the add equals rule, AE (based on a heuristic peculiar to addition problems), to add $(-A)$ to both sides of the identity. The program can then easily solve the problem. Thus the tutor would be able to help the student complete the partial solution.

The examples in the case study suggest adaptation to the student's work that goes beyond the level of (relevant) hints or advice. The computer-tutor can offer hints regarding which substitution sequences to use, which expressions to generate by subsidiary derivation in order to replace a term in one of the lines, and which rules to apply to which lines of the derivation. It can also explain how the substitutions and subgoals were determined. The tutor can suggest other solution approaches if others were discovered which, based on some criteria discussed later, seem worthwhile to point out to the student. In general, the tutor is able to provide useful suggestions derived from the theorem-prover's own strategies, rather than having to pass out answers or having to guide the student, step by step, to a particular solution determined by the analysis process. In the concluding section of this chapter, the behavior of the proof-analyzer is compared with that of human tutors to

determine how well this use of theorem-proving techniques works in providing information for a tutorial dialogue.

1. Finding One or More Solutions

The goal-subgoal search described in Chapters III and IV is commonly represented as a goal tree (Slagle, 1963). For the theorem-prover, the analogy is carried out by labeling the root of the tree as the problem expression, each descendent node as a subgoal, and a directed branch extending from subgoal A to subgoal B as a search method in which B might be useful in solving A. Each node "sprouts" new branches, that is, tries new methods, until one of the methods succeeds in finding a solution to the subgoal at that node.

As soon as a solution is found, the (method-) branch can be "pruned." For example, observe Figure 4, a tree representation of Definition 18, REPLACE⁰(ϕ, G), of Chapter IV. To solve the goal at node 1, form subgoal 1 at node 2. The subgoal is solved by the method represented by the branch from node 2 to node 7, henceforth denoted as (2-7). Suppose that the solution of this subgoal provides a solution for the goal at the next higher level of the tree (in this case, node 1). This is true only if the replacement inference rule (RE) forms a pattern P from the subgoal and some initial statement relative to the goal, such that P is identical to the goal. If a solution for the goal is obtained, then the subgoal node and its connecting branch can be "pruned" from the goal tree. So, in Figure 4(c), node 7 and branch (2-7) are removed; then node 2 and branch

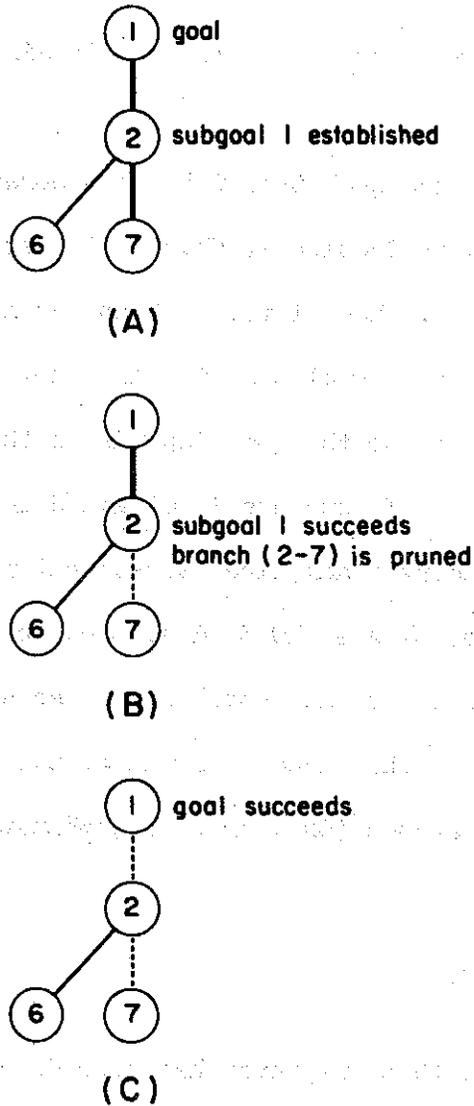


Fig. 4.--A goal tree.

(The dotted lines indicate that a branch has been pruned.)

(1-2) are pruned as well. Since there are no more subgoals to solve, the process halts and the original goal is solved. If P is not identical to the goal, another subgoal can be sprouted and the search continues. This analogy to trees is carried out in more detail in Nilsson (1971) and Slagle (1971).

Figure 5 is the goal tree for the theorem-prover. Methods and heuristics described in Section 7, Chapter IV, are shown as submethods of a branch called MATCH. Each level of branches mirrors the program's recursive behavior. This simple goal tree is also known as a disjunctive goal tree because the solution to the goal depends on the successful solution of only one of the methods. If success is attained for one method, indicated in Figure 5 by the heavy black line, subsequent methods can be ignored. For the SAINT program, Slagle (1963) developed the conjunctive-disjunctive (AND-OR) goal tree. At each node, it is necessary to specify whether the solution depends on the success of all of two or more branches (AND), one of two or more branches (OR), or some combination of the branches.

1.1 The COLLECTOR

So far, the theorem-prover has been described as a purely disjunctive goal tree. Suppose instead of pruning the tree whenever this happened, the program merely "plucked the fruit" of the branch and continued its search, acting as though that method had not succeeded. Suppose also that another branch is added, to be sprouted last by every subgoal node. Call this last branch the COLLECTOR. The COLLECTOR examines all the fruit which was

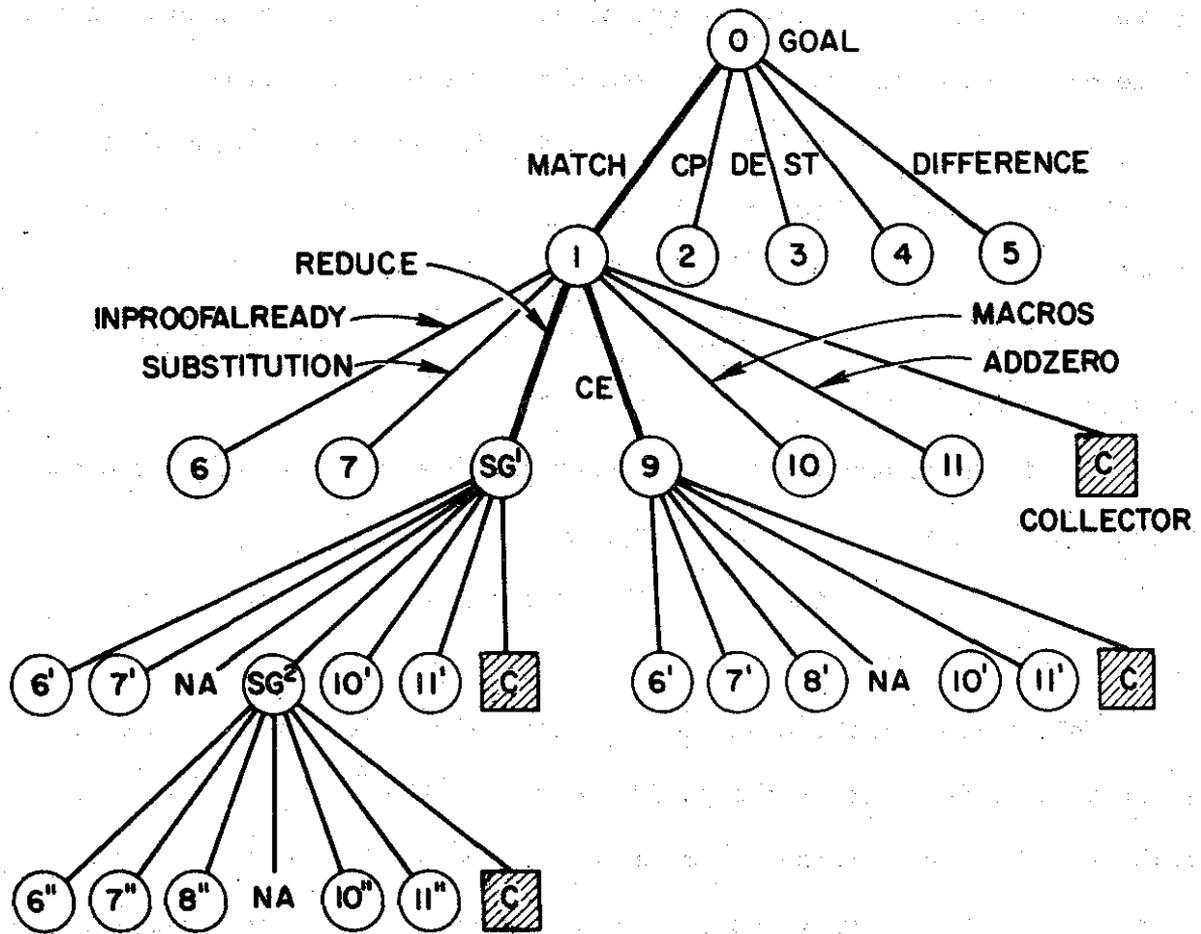


Fig. 5.--The goal tree for the tutor's theorem-prover. The numbers and primes indicate search methods and levels of recursion, respectively.

gathered, if any, and chooses the best one (according to criteria detailed below). The COLLECTOR's decision becomes the solution for the subgoal. In Figure 5, the darkened boxes represent the COLLECTOR. If subgoal 1 at node 1 could be solved by both methods (1-8) and (1-9), then it is up to the COLLECTOR to choose, from the two solutions, the one solution which will be given to method (0-1). This is essentially how the theorem-prover is able to delay termination and continue searching for more solutions.

The COLLECTOR's purpose is to examine the so-called "basket of fruit" and choose the best solution. The original criterion used for "best" was:

1. choose the solution which references the greatest number of student lines; and
2. if there are two or more solutions from 1, then choose the one which has the least number of steps; and
3. if, after 2, there is still more than one possible solution, choose any one of them.

This criterion was adjusted once and undoubtedly must be done so again as experience using the proof-analyzer indicates better decision methods. The first change was made as a result of the theorem-prover's response to the following situation: the theorem-prover was asked to suggest ways to complete the ten-line, partial derivation shown below.

DERIVE $(C+E)+B = C+(E+(4+2))$
 P (1) $B = 4+2$

AE
 $\therefore C+E$ (2) $B+(C+E) = (4+2)+(C+E)$

1CE1 (3) $4+2 = B$

2.3RE1 (4) $B+(C+E) = B+(C+E)$

$$\underline{AS} \quad (A+B)+C = A+(B+C)$$

A::C

B::E

C::B

$$(5) \quad (C+E)+B = C+(E+B)$$

5CA2

$$(6) \quad B+(C+E) = C+(E+B)$$

4.6RE2

$$(7) \quad B+(C+E) = C+(E+B)$$

7CA4

$$(8) \quad B+(C+E) = C+(B+E)$$

8.1RE2

$$(9) \quad B+(C+E) = C+((4+2)+E)$$

9CA5

$$(10) \quad B+(C+E) = C+(E+(4+2))$$

HELP

At this point, the theorem-prover took over and suggested two ways to complete the derivation. The first way will be discussed again in Section 3. That is,

$$5.1RE2 \quad (11) \quad (C+E)+B = C+(E+(4+2)) .$$

The second way indicated how to complete the proof already started:

$$7CE1 \quad (11) \quad C+(E+B) = B+(C+E)$$

$$5.11REL \quad (12) \quad (C+E)+B = B+(C+E)$$

$$12CE1 \quad (13) \quad B+(C+E) = (C+E)+B$$

$$10.13REL \quad (14) \quad (C+E)+B = C+(E+(4+2)) .$$

This second solution is wasteful in terms of the number of lines because line 13 is an instance of the CA axiom. The proof could be

$$CA \quad A+B = B+A$$

A::B

B::C+E

$$(11) \quad B+(C+E) = (C+E)+B$$

$$10.11REL \quad (12) \quad (C+E)+B = C+(E+(4+2))$$

The COLLECTOR had the choice, in fact, between the instance of CA and the above three lines (11-13), and chose the latter because it maximized the use of the student lines. The decision procedure was consequently augmented by the following:

4. The choice of (1) through (3) above is acceptable only if there is no solution which is a one-line solution--either an instance of an axiom, a line of the proof, or an application of the commute equals (CE) rule to a line already in the proof.

Note that there is no COLLECTOR at the top level. The final basket of solutions, S, represents solutions found by the different subroutines of MATCH. More solutions are added to S from the other procedures of the theorem-prover: CONDITIONAL PROOF, DETACHMENT, and SUBSET TEST. When the theorem-prover halts, the solutions in S capture several strategies for solving the same problem.

1.2 An Example

In Figure 3, the formula $O+B=B$ was used as an illustration of how the theorem-prover constructs a proof. The same problem is now offered to illustrate how several proofs for the formula are discovered by the modified theorem-prover. Suppose the student has already typed one line. The proof, at the time the student requests help, looks like:

```

PROVE  O+B=B
      CA  A+B=B+A
      A::O
      B::B      (1)  O+B = B+O
      HELP

```

There are actually several ways to construct a proof for this problem. As shown in Chapter IV, the theorem-prover finds a three-line proof which does not use line 1 above. The modified theorem-prover finds other possible solutions, two of which do use line 1.

An attempt is made here to trace the path followed by the search methods. The full goal tree generated is the one shown in Figure 5; it is built in the six stages illustrated in Figures 6 and 7. The legend for these figures is given in Table 1.

Since each of CONDITIONAL PROOF, DETACHMENT, and SUBSET TEST fail, and REDUCTION is not used because other solutions were obtainable which reference the last line, only the MATCH procedure is traced. The top goal is the problem statement: $((O+B) = B)$. INPROOFALREADY and SUBSTITUTION both fail. The result of a FEATURE TEST is a list containing only the axiom name Z. The difference set for the instance of Z and the goal is $\{1,2\}$. The first subgoal is therefore $((B+O) = (O+B))$. In Figure 5, (A) and (B), subtrees generated in trying to solve the subgoal are shown. Subgoal 1 is solved two ways:

1. 6(A) by SUBSTITUTION: it is an instance of the CA axiom; and
2. 6(B) by commuting the identity, CE, and solving the second subgoal: $((O+B) = (B+O))$.

Subgoal 2 is solved in two ways:

1. by INPROOFALREADY, because it is line 1, the line typed by the student; and
2. by SUBSTITUTION: it is an instance of the CA axiom.

The COLLECTOR chooses solution (1) for subgoal 2 because it is clearly the shortest solution using the greatest number of student lines. The solution 6(B) is then to commute line 1, i.e., 1CE1.

In Figure 6(C), the COLLECTOR must choose between solutions 6(A) and 6(B). The choice is the same as before, (1CE1), and is the solution selected for subgoal 1.

TABLE 1
LEGEND FOR FIGURES 6 AND 7

Symbol	Meaning
S	Method succeeded
F	Method failed
NA	Method not applicable
C(list)	The COLLECTOR's choice is (list)
1	INPROOFAFREADY check
2	SUBSTITUTION check
3	REDUCE method
4	COMMUTE EQUALS heuristic
5	ADD-ZERO heuristic
6	MACROS transformations

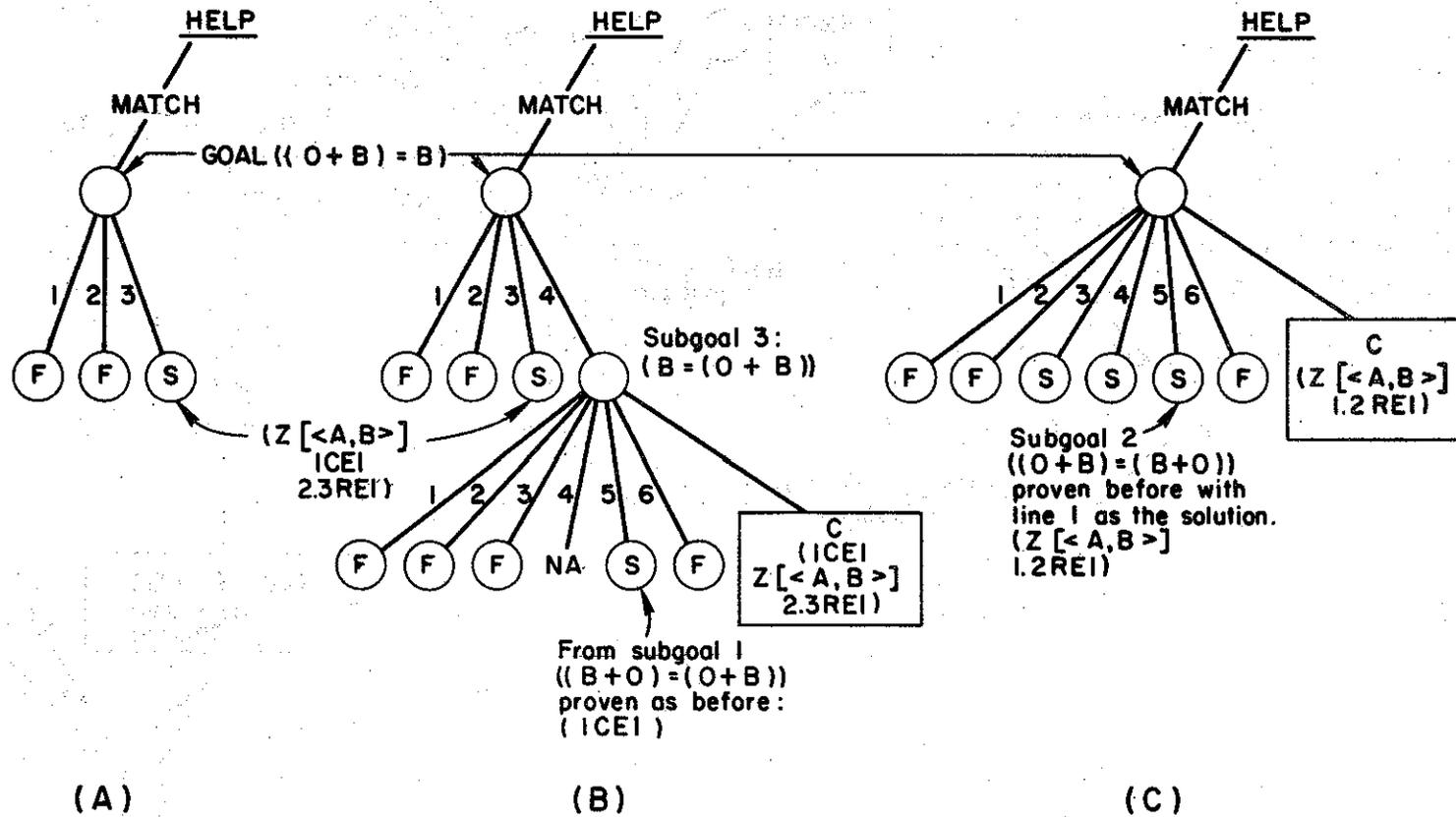


Fig. 7.--Finding solutions for $O+B=B$: Continued search.

In Figure 7(A), then, the instance of the Z axiom, $((B+0) = B)$, can be transformed by replacing $(B + 0)$ by $(0 + B)$. The result is a successful solution of the top goal.

The search for more solutions now continues. In Figure 7, (B) and (C), two more solutions are obtained. One is merely a permutation of the first. The other is solved by adding zero to the right-hand side of the goal; then the ADD-ZERO heuristic forms a subgoal equal to line 1. The heuristic assumes an initial application of the Z axiom, so the solution is:

$$\begin{array}{l} \underline{Z} \quad A+0=A \\ \underline{A}::\underline{B} \quad (1) \quad B+0 = B \\ \underline{1.2REL} \quad (2) \quad 0+B = B \end{array}$$

The COLLECTOR considers this the best solution. Because the search is over, this is also the solution which would be suggested to the student. Note that there is no actual COLLECTOR at this point (the top level of MATCH), so that all the proofs discovered are made available to the dialogue routine.

The following sections examine the kinds of things the computer-tutor would like to say to help the student complete his work. It will be important to keep in mind the main purpose of the instructional system, i.e., to teach the notion of mathematical proof. Although it may be possible to manipulate the lines of a partial proof in order to come up with a solution which completes the proof, the sequence of commands may not constitute an elegant proof. A simple example came up in the case study to be presented in Section 3. The student had just proven the first theorem: $0+A = A$, the problem examined in detail in this section. For the next problem, the student completed the following proof.

DERIVE $O+(A+B) = A+B$

Z $A+O=A$
 $A::A+B$ (1) $(A+B)+O = A+B$

CA $A+B=B+A$
 $A::A+B$
 $B::O$ (2) $(A+B)+O = O+(A+B)$

1,2REL (3) $O+(A+B) = A+B$

CORRECT...

The student then asked (orally) "is there a shorter way to do this?" The (human) tutor's response, based on the solutions known to the dialogue routine, was "yes, do you know what it is?" The student: "oh, yes... theorem 1." The theorem-prover can find all of the solutions given for $O+B=B$, with B replaced by $A+B$. This includes the one constructed by the student. But the student, and the theorem-prover, went to the trouble of re-proving a theorem. The theorem-prover, by SUBSTITUTION, also knows that a one-line instantiation of the theorem constitutes a proof. If the student is reminded about the theorem that was not used, he might remember to use it in the next appropriate problem. The student in the case study did.

2. Completing Partial Solutions

The theorem-prover terminates with a set of possible solutions to a problem. As demonstrated in Section 1, some of these solutions may include lines generated from commands typed by the student. A list of these lines and the total number of references to them are stored with the solution. Furthermore, each solution represents a different approach to constructing a derivation for the problem, approaches presented as the theorem-prover's set of strategies. Sometimes the solutions are identical,

which fact perhaps could be used to reinforce the tutor's willingness to discuss that solution. In searching for these solutions, the theorem-prover is careful not to use an axiom, theorem, or rule of inference which the student has not learned or was asked not to use. The theorem-prover is thereby certain to suggest only rules which the student is expected to know and use. In completing partial proofs, the theorem-prover flags some of the lines it generates with number codes and expressions. These flags are used to tell the dialogue routine which strategy (program procedure) was used in obtaining that line.

Two questions must be answered in this section. First, what partial proofs will the theorem-prover be able to successfully complete? Second, how well does the theorem-prover adapt; that is, does the theorem-prover modify the information it provides for a given problem when the information it receives, the partial solutions to that problem, differ?

To answer the first question, the preferred analysis method is to have the theorem-prover attempt to discover solutions on its own, referencing lines of the student's partial proof as often as possible. If the student has given any number of lines which can actually be used in constructing a complete proof, it is likely that the analyzer will incorporate those lines into its proofs. This is, of course, a conjecture, but one made with confidence from having tested the program on problems such as those shown in Section 3. Certainly the theorem-prover's ability to solve the problem does not decrease with the increased amount of information (given lines), a fact not true of all human tutors. In some cases, the theorem-prover's ability even increases; it can complete partial solutions it would otherwise be unable to produce. This occurs when the student introduces into the proof instances of axioms or theorems which the FEATURE TEST rules out.

The theorem-prover also tries to reduce the difference between the student's last line and the problem expression (essentially using the REDUCE method). Its purpose in doing this is to try to continue where the student left off, regardless of what path he used to obtain the last line.

If the curriculum is confined to expressions in which the mapping f of Lemma 7 (Chapter IV) can be constructed, then a definite statement can be made about the theorem-prover's ability to complete partial proofs. This is because the theorem-prover is known to be complete for this class of expressions (Theorems 2 and 5). It is also clear that if we want to construct a proof for $G \in \mathcal{L}$, the student has completed a partial proof whose last line is H , and there exists an isomorphism f between the leaves of $S(G,1)$ and the leaves of $S(H,1)$, then the REDUCE method can complete the proof. (The reader can convince himself that this is so by drawing the diagram:

$$\begin{array}{ccc} S(G,1) & \xrightarrow[\text{onto}]{1-1} & S(G,n) & \text{(Lemma 7)} \\ f \downarrow & & \downarrow g & \\ S(H,1) & \xrightarrow[\text{onto}]{1-1} & S(H,m) & \text{(Lemma 7)} \end{array}$$

The map g is clearly one-one, onto because isomorphism is an equivalence relation. So, by Theorem 5, subgoals $(S(G,1)=S(H,1))$ and $(S(G,n)=S(H,m))$ are provable.)

That teachers commonly do just as this reduction procedure suggests (i.e., start with the student's last line and try to reduce it to the problem expression) is a clear conclusion from the small study conducted with human tutors (presented in Section 4). It is interesting to note that the main reason the human tutors gave for choosing this method was the

desire to avoid discouraging the students, to avoid making them feel that their work cannot be used in a successful proof. By using the student's work, continuing to manipulate the lines they generated, teachers reinforce the student's confidence and willingness to try other problems. For expressions in an arbitrary commutative groupoid, this method can always be used if the last line meets the above description. That is, if the student has constructed a partial solution such that the number of occurrences of individual variables occurring in the last line of the proof is the same as in the problem expression, then the tutor can show the student how to permute and regroup these variables. The last line of the student's partial proof is a good choice, because it was the last thing the student did and therefore an easy line for the student to work on without becoming confused by lines appearing later in the proof.

In answer to the second question, how well does the theorem-prover adapt, two examples are given. The first reexamines the sample problem introduced in Section 1. Even this simple example shows the great variety of information that the theorem-prover can give the dialogue routine. The second is a comparison between using the stored hints approach for helping the student and using the theorem-prover. Section 3 contains more partial proofs and the various ways the theorem-prover is able to complete them.

2.1 Information about $O+B=B$

Figure 8, (A) through (E), are solutions for $O+B=B$. The actual internal representation of the data-base for 8(A) is shown as the value of the identifier STOREDPROOFS. (Note that 'GRINDEF' is a LISP printing

```

(DEFPROP STOREDPROOFS
(NIL ((INDEX . 3) (STDTLINES)
      (#SL . 0)
      (CHANGES)
      (FLAGS (1
              (14 (= (/+ 0 B) (/+ B 0))
                   (1 52 (= (/+ 0 B) (/+ B 0))))))
      NIL)
      (PROOF (3 ((1 2) (RE) (1)) (= (/+ 0 B) B))
              (2 ((NIL) (Z) ((B . %AZ))) (= (/+ B 0) B))
              (1 ((NIL) (CA) ((B . %BZ) (0 . %AZ)))
                  (= (/+ 0 B) (/+ B 0))))))
((INDEX . 4) (STDTLINES)
              (#SL . 0)
              (CHANGES)
              (FLAGS (1
                      (12 14
                       (= B (/+ 0 B))
                       (14 (= (/+ B 0) (/+ 0 B))
                            (1 50 (= (/+ B 0) (/+ 0 B))))))
                      NIL)
                      (PROOF (4 ((3) (CE) (1)) (= (/+ 0 B) B))
                              (3 ((1 2) (RE) (1)) (= B (/+ 0 B)))
                              (2 ((NIL) (Z) ((B . %AZ))) (= (/+ B 0) B))
                              (1 ((NIL) (CA) ((0 . %BZ) (B . %AZ)))
                                  (= (/+ B 0) (/+ 0 B))))))
((INDEX . 3) (STDTLINES)
              (#SL . 0)
              (CHANGES (G0016 . B))
              (FLAGS (1
                      (2
                       ((8 (/+ G0016 0) (/+ 0 B) . 2)
                        (14 (= (/+ G0016 0) (/+ 0 B)) (1))))))
                      NIL)
                      (PROOF (3 ((1 2) (RE) (1)) (= (/+ 0 B) B))
                              (2 ((NIL) (CA) ((B . %AZ) (0 . %BZ)))
                                  (= (/+ B 0) (/+ 0 B)))
                              (1 ((NIL) (Z) ((G0016 . %AZ)))
                                  (= (/+ G0016 0) G0016))))))
VALUE)
NIL
# *

```

A

Fig. 8.--Solutions for $0+B=B$: Adapting to new information.

In 8(A), the internal representation for solutions found in 8(B) is shown, indicating the flags which tell the tutor how the proofs were discovered. In 8(B) through 8(F), the student typed zero, one, or more proof lines before asking for help. By typing the command HELP, he calls on the tutor for a hint. Here, instead of a hint, the tutor typed the solutions generated by the theorem-prover.

DERIVE $(0 + B) = B$

:*HELPS

CA $A + B = B + A$
 B::B
 A::0 (1) $(0 + B) = (B + 0)$
 Z $A + 0 = A$
 A::B (2) $(B + 0) = B$
 1.2RE1 (3) $(0 + B) = B$

CA $A + B = B + A$
 B::0
 A::B (1) $(B + 0) = (0 + B)$
 Z $A + 0 = A$
 A::B (2) $(B + 0) = B$
 1.2RE1 (3) $B = (0 + B)$
 3CE1 (4) $(0 + B) = B$

Z $A + 0 = A$
 A::B (1) $(B + 0) = B$
 CA $A + B = B + A$
 A::B
 B::0 (2) $(B + 0) = (0 + B)$
 1.2RE1 (3) $(0 + B) = B$

B

DERIVE $(0 + B) = B$

:*CAS $A + B = B + A$
 A::*0\$
 B::*B\$ (1) $(0 + B) = (B + 0)$

:*HELPS

Z $A + 0 = A$
 A::B (2) $(B + 0) = B$
 1.2RE1 (3) $(0 + B) = B$

1CE1 (2) $(B + 0) = (0 + B)$
 Z $A + 0 = A$
 A::B (3) $(B + 0) = B$
 2.3RE1 (4) $B = (0 + B)$
 4CE1 (5) $(0 + B) = B$

Z $A + 0 = A$
 A::B (2) $(B + 0) = B$
 1CE1 (3) $(B + 0) = (0 + B)$
 2.3RE1 (4) $(0 + B) = B$

C

DERIVE $(0 + B) = B$

:*Z\$ $A + 0 = A$
 A::*B\$ (1) $(B + 0) = B$

:*HELPS

CA $A + B = B + A$
 B::B
 A::0 (2) $(0 + B) = (B + 0)$
 2.1RE1 (3) $(0 + B) = B$

CA $A + B = B + A$
 B::0
 A::B (2) $(B + 0) = (0 + B)$
 2.1RE1 (3) $B = (0 + B)$
 3CE1 (4) $(0 + B) = B$

Z $A + 0 = A$
 A::B (2) $(B + 0) = B$
 CA $A + B = B + A$
 A::B
 B::0 (3) $(B + 0) = (0 + B)$
 2.3RE1 (4) $(0 + B) = B$

D

DERIVE $(0 + B) = B$

:*CAS $A + B = B + A$
 A::*B\$
 B::*0\$ (1) $(B + 0) = (0 + B)$

:*Z\$ $A + 0 = A$
 A::*B\$ (2) $(B + 0) = B$

:*HELPS

2.1RE1 (3) $(0 + B) = B$

1CE1 (3) $(0 + B) = (B + 0)$
 3.2RE1 (4) $(0 + B) = B$

1.2RE1 (3) $B = (0 + B)$
 3CE1 (4) $(0 + B) = B$

E

```

DERIVE      (0 + B) = B

:*CAS      A + B = B + A
A::*BS
B::*0S     (1) (B + 0) = (0 + B)

:*AIS      A + (- A) = 0
A::*BS     (2) B + (- B) = 0

:*CE1S     (3) 0 = B + (- B)

:*HELPS

3AE
A::B       (4) (0 + B) = (B + - B) + B
AS (A + B) + C = A + (B + C)
A::B
B::- B
C::B       (5) ((B + - B) + B) = (B + (- B + B))
CA A + B = B + A
A::- B
B::B       (6) (- B + B) = (B + - B)
5.6RE1     (7) ((B + - B) + B) = (B + (B + - B))
AI A + (- A) = 0
A::B       (8) (B + - B) = 0
7.8RE2     (9) ((B + - B) + B) = (B + 0)
Z A + 0 = A
A::B       (10) (B + 0) = B
9.10RE1    (11) ((B + - B) + B) = B
4.11RE1    (12) (0 + B) = B

Z A + 0 = A
A::B       (4) (B + 0) = B
1.4RE1     (5) (0 + B) = B

1CE1       (4) (B + 0) = (0 + B)
Z A + 0 = A
A::B       (5) (B + 0) = B
4.5RE1     (6) B = (0 + B)
6CE1       (7) (0 + B) = B

Z A + 0 = A
A::B       (4) (B + 0) = B
1CE1       (5) (B + 0) = (0 + B)

```

F

Fig. 8, continued.

function.) The proofs are shown in their usual format, one identical to that available to the students using the instructional system (Chapter VI). To obtain these examples, some lines of a proof were typed in, followed by the HELP command. The dialogue routine then printed the stored proofs. Comments on these figures follow.

Figure 8, (A) and (B): The student typed no lines before typing the HELP command. These are the proofs shown in Section 1. The flags indicate that solutions were found by (a) setting up an immediate subgoal (flag 14) where the temporary variables in line 1 resulted from the FEATURE TEST's choice of the Z axiom; (b) commuting the goal (flag 12) and then adding zero to the left-hand side (flag 50); and (c) adding zero to the right-hand side of the goal (flag 52). Note that, in terms of the practicality of using a theorem-prover, it is a waste of time to regenerate proofs when the student has not tried something. In such cases, it is probably sufficient to store a hint or a proof.

Figure 8(C): Student typed

```
CA A+B=B+A
A::O
B::B (1) O+B = B+O
```

HELP

The proofs resemble those in Figure 8(A) except for the references to line 1.

Figure 8(D): Student typed

```
Z A+O=A
A::B (1) B+O=B
```

HELP

Again, the proofs are similar. The slip-up in recognizing line 1 in the third example is due to temporary variables that are not replaced until line 4.

Figure 8(E): Student typed

CA A+B=B+A
A::B
B::O (1) B+O=O+B

Z A+O=A
A::B (2) B+O=B

HELP

Note how the theorem-prover reacts to the extra information offered, effecting a reduction in the work it had to do.

Figure 8(F): Student typed

CA A+B=B+A
A::O
B::B (1) O+B=B+O

AI A+(-A)=O
A::B (2) B+(-B)=O

CEL (3) O=B+(-B)

HELP

This is an especially interesting proof because the student started to give a proof which has already been seen, but then the student apparently changed his approach. The last two lines deviate from the lines the theorem-prover generates on its own (as in Figure 8(B)). As expected, the theorem-prover catches the first line to suggest the three proofs already given. Using the AI axiom is also a plausible approach, and is completed by the theorem prover. In a sense, then, the theorem-prover was able to find out that the student attempted two different strategies. The dialogue routine can point out the usefulness of both approaches and then proceed to give hints to complete one of them. The choice would probably fall to the first set of solutions, since they are the more common and the shorter.

2.2 Comparison of the Theorem-Prover/Proof-Analyzer with the Stored-Hint Approach

Over the school year 1970-1971, the problems experienced by students and proctors taking the Stanford logic-algebra course were monitored.¹ From complaints and calls for help, it was obvious that the stored-hint approach to helping the students did not provide sufficient assistance. Several problems, despite hints amounting to direct answers, could not be done by a large number of the students. Moreover, many problems did not have hints at all. If the proctor at the school could not solve the problem, then the students were stuck. Some means to provide additional help was clearly necessary.

The fact that hints did not exist or were not understandable could be corrected by the curriculum writer. But even improved stored hints cannot overcome the possibility that the hints will tell the students to do something they have already done, or to start something new when their own work could be completed successfully. The curriculum writer has to come up with hints to help the students complete any possible solution; but he can just provide hints for only one possible solution. The hints in the logic-algebra program have been described as "irrelevant," "misleading," "not informative," or "too informative." Hints are too informative if they tell the student exactly what to do. The student has no chance to figure out the problem himself. Students learn by doing things themselves. They learn how to solve problems by figuring out how to solve them, not by following the precise dictates of the teacher, whether human or computerized. A tutor might first offer some reminder

¹Since I was responsible for the operation of the logic-algebra program, calls for help, complaints, and requests for special assistance were directed to me.

to the student which still gives him a chance to work the problem his own way. If he continued to have trouble, the tutor could gradually increase the amount of information offered. These were assumptions we used.

In order to provide a computer-tutor which knows how to help the students, we determined how to handle problems in algebra mechanically and how to use a theorem-proving program as a proof-analyzer. Use of the analysis methods permits the computer-tutor to adapt to the student's work rather than demanding that the students adapt to the tutor's pre-specified ideas. The theorem-prover is not an algorithm which always finds a solution, but rather a set of heuristics which represent good approaches to finding a solution. By staying away from the single algorithm approach, the computer-tutor is able to retain the student's freedom in constructing proofs without losing its ability to help him if he should ask.

Although the hints generated by the tutor are suggestive of the way the theorem-prover operates, the students do not have to use one particular method. The designers of the Fortran Deductive System (Quinlan and Hunt, 1968) claim that, because their algorithm programmed for a computer works so successfully, students of algebra should be taught to use it. It may be good to show them this one approach to use, but it is not good to channel students into a single approach to problem solving. Students, set to use a particular solution technique, will not know how to do a problem which does not benefit from that technique. The student's ability for "original thinking" may be stifled by the lack of encouragement or direction to do the same problem in new ways.

The stored hints in the Stanford logic-algebra curriculum exhibit this same problem. Take for example the solution of Theorem 8 (in Appendix II): $A+B=C \rightarrow A=C-B$. The standard approach for constructing derivations

for conditional statements is to start with a working premise (the command name is WP). Most students always used this heuristic and the stored hints encourage it. At the point at which this problem is given, the student already has proven Theorems 1 through 7 about addition; he knows the five axioms of an additive group. Two hints are stored with the problem. They are:

1. DID YOU ADD THE ANTECEDENT OF THE CONDITIONAL AS A WORKING PREMISE...
2. DERIVE $(A+B)+(-B) = C+(-B)$. HOW DOES THIS HELP YOU...

The computer-tutor, as illustrated later in the case study with the student Lisa (Section 3, this chapter), asked the student to try to use Theorem 7 in the proof of Theorem 8. Having completed this proof, Lisa expressed surprise to learn that a conditional statement could be derived without a working premise. She had been misled by the hints she had received. She had learned no appreciation for the numbering scheme of the theorems, ordered because of dependence relationships that could help the student find solutions. A theorem was like any other derivation problem; she did not learn to take advantage of new information as it became available. Some of Lisa's difficulties in doing the "Finding-Axioms" exercise (explained in Chapter I) seem reasonable. She did not understand how and why it was important to carefully order the proofs so that the availability of a theorem could help her do the next problem. The ability to do so was an advantage in successfully completing the exercise.

Note that, had Lisa started or completed a solution using WP and AE (add equals rule) as suggested in the stored hints, the theorem-prover could follow that strategy as well as the one using Theorem 7. When she completed the proof, the dialogue routine would point out the solution

using Theorem 7, acting on the criterion that Theorem 7 was recently proven. Lisa, by her own choice or that of the computer-tutor, could redo the problem.

One of the problems that causes a great deal of trouble in the Stanford curriculum is:

$$\text{DERIVE} \quad A+((-C)-D) = (A-C)-D \quad .$$

The two hints stored with the problem are:

1. START BY USING THE N AXIOM TO GET

$$(A-C)+(-D) = (A-C)-D.$$

2. TO FINISH YOU WILL NEED TO USE N, RE, AND AS.

The difficulty with the problem seems to be that the parentheses have to be moved over the minus sign. This cannot be done without first using the N axiom to change $A-C$ to $A+-C$. Once the association between terms is altered, the N axiom must be reapplied. Allowed to find a proof on its own, the theorem-prover uses a different approach from that suggested in the hints (because FEATURE TEST rules out the initial statement N); it uses the AS axiom and methodically replaces occurrences of $\langle \text{term 1} \rangle + \langle \text{term 2} \rangle$ by $\langle \text{term 1} \rangle - \langle \text{term 2} \rangle$ (see Figure 9(A)). No parentheses have to be moved around; no work has to be undone.

This problem, with different partial solutions and with the theorem-prover completing those partial solutions, is shown in Figure 9(A) through (I). Several solutions appear, some worked by Lisa. The format of the presentation is the same as that used in Figure 8. Lisa asked for help at the points indicated. The proof that starts with the N axiom, Figure 9(F), was worked by a proctor at an elementary school in East Palo Alto, California. He used a stored hint to get started, requested another hint after line 1. After line 4, no more hints were available

Fig. 9.--Solutions for $(A+(-C-D))=((A-C)-D)$: Completing different partial solutions.

The format of the figure is like that for Fig. 8. The computer presents the DERIVE problem, and the student types zero, one, or more lines before requesting HELP. Information typed by the student is always preceded by a colon and an asterisk (:*) and terminated by a dollar sign (\$). The examples are lettered A through I as follows:

- A. Solutions found by the theorem-prover: Lisa immediately requested help
- B. Lisa started a proof using LT (Logical Truth rule)
- C. Lisa continued after receiving help from the computer tutor
- D. She needed more help
- E. The LT rule was not allowed in this proof; a new solution was required
- F. With some help from the theorem-prover
- G. The proctor worked on the problem, trying a solution suggested by a stored hint; with some help, the theorem-prover can solve this problem using the N axiom
- H. Running out of stored hints, the proctor asked the tutor for help
- I. And then asked for more.

DERIVE (A + (- C - D)) = ((A - C) - D)

:*HELPS

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (1) ((A + - C) + - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A + - C
E::D (2) ((A + - C) + - D) = ((A + - C) - D)
1.2RE1 (3) ((A + - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A
B::C (4) (A + - C) = (A - C)
3.4RE1 (5) ((A - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A::- C
B::D (6) (- C + - D) = (- C - D)
5.6RE1 (7) ((A - C) - D) = (A + (- C - D))
7CE1 (8) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (1) ((A + - C) + - D) = (A + (- C + - D))
1CE1 (2) (A + (- C + - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A::- C
B::D (3) (- C + - D) = (- C - D)
2.3RE1 (4) (A + (- C - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A::A + - C
B::D (5) ((A + - C) + - D) = ((A + - C) - D)
4.5RE1 (6) (A + (- C - D)) = ((A + - C) - D)
N A + (- B) = A - B
A::A
B::C (7) (A + - C) = (A - C)
6.7RE1 (8) (A + (- C - D)) = ((A - C) - D)

```

:*

Fig. 9A

DERIVE (A + (- C - D)) = ((A - C) - D).

```

:*LTS
:*(A-C)-DS(1) ((A - C) - D) = ((A - C) - D)

:*NS A + (- B) = A - B
A::*AS
B::*CS (2) (A + - C) = (A - C)

:*CE1S (3) (A - C) = (A + - C)

:*1.3RE1S (4) ((A + - C) - D) = ((A - C) - D)

:*HELPS

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (5) ((A + - C) + - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A + - C
B::D (6) ((A + - C) + - D) = ((A + - C) - D)
5.6RE1 (7) ((A + - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A::- C
B::D (8) (- C + - D) = (- C - D)
7.8RE1 (9) ((A + - C) - D) = (A + (- C - D))
4.9RE1 (10) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (5) ((A + - C) + - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A + - C
B::D (6) ((A + - C) + - D) = ((A + - C) - D)
5.6RE1 (7) ((A + - C) - D) = (A + (- C + - D))
7.2RE1 (8) ((A - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A::- C
B::D (9) (- C + - D) = (- C - D)
8.9RE1 (10) ((A - C) - D) = (A + (- C - D))
10CE1 (11) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (5) ((A + - C) + - D) = (A + (- C + - D))
5CE1 (6) (A + (- C + - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A::- C
B::D (7) (- C + - D) = (- C - D)
6.7RE1 (8) (A + (- C - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A::A + - C
B::D (9) ((A + - C) + - D) = ((A + - C) - D)
8.9RE1 (10) (A + (- C - D)) = ((A + - C) - D)
10.2RE1 (11) (A + (- C - D)) = ((A - C) - D)

```

Fig. 9B

```

DERIVE      (A +(- C - D))=((A - C) - D)

:*LTS
:*(A-C)-D$(1) ((A - C) - D)=((A - C) - D)

:*NS  A +(- B)= A - B
A::*AS
B::*CS  (2) (A + - C)=(A - C)

:*CE1$  (3) (A - C)=(A + - C)

:*1.3RE1$ (4) ((A + - C) - D)=((A - C) - D)

:*NS  A +(- B)= A - B
A::*A+(-C)$
B::*D$  (5) ((A + - C) + - D)=((A + - C) - D)

:*CE1$  (6) ((A + - C) - D)=((A + - C) + - D)

:*4.6RE1$ (7) ((A + - C) + - D)=((A - C) - D)

:*HELPS

AS  (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D  (8) ((A + - C) + - D)=(A +(- C + - D))
N  A +(- B)= A - B
A::- C
B::D  (9) (- C + - D)=(- C - D)
8.9RE1 (10) ((A + - C) + - D)=(A +(- C - D))
7.10RE1 (11) (A +(- C - D))=((A - C) - D)

AS  (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D  (8) ((A + - C) + - D)=(A +(- C + - D))
8.5RE1 (9) ((A + - C) - D)=(A +(- C + - D))
9.2RE1 (10) ((A - C) - D)=(A +(- C + - D))
N  A +(- B)= A - B
A::- C
B::D  (11) (- C + - D)=(- C - D)
10.11RE1 (12) ((A - C) - D)=(A +(- C - D))
12CE1 (13) (A +(- C - D))=((A - C) - D)

AS  (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D  (8) ((A + - C) + - D)=(A +(- C + - D))
8CE1 (9) (A +(- C + - D))=((A + - C) + - D)
N  A +(- B)= A - B
A::- C
B::D  (10) (- C + - D)=(- C - D)
9.10RE1 (11) (A +(- C - D))=((A + - C) + - D)
11.5RE1 (12) (A +(- C - D))=((A + - C) - D)
12.2RE1 (13) (A +(- C - D))=((A - C) - D)

```

Fig. 9C

DERIVE (A + (- C - D)) = ((A - C) - D)

```

:*LTS
:*(A-C)-D$(1) ((A - C) - D) = ((A - C) - D)

:*NS A + (- B) = A - B
A: *AS
B: *CS (2) (A + - C) = (A - C)

:*CE1$ (3) (A - C) = (A + - C)

:*1.3RE1$ (4) ((A + - C) - D) = ((A - C) - D)

:*NS A + (- B) = A - B
A: *A+(-C)$
B: *D$ (5) ((A + - C) + - D) = ((A + - C) - D)

:*CE1$ (6) ((A + - C) - D) = ((A + - C) + - D)

:*4.6RE1$ (7) ((A + - C) + - D) = ((A - C) - D)

:*7AR1$ (8) (A + (- C + - D)) = ((A - C) - D)

:*HELPS

```

```

N A + (- B) = A - B
A: - C
B: D (9) (- C + - D) = (- C - D)
8.9RE1 (10) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A: A
B: - C
C: - D (9) ((A + - C) + - D) = (A + (- C + - D))
9.5RE1 (10) ((A + - C) - D) = (A + (- C + - D))
10.2RE1 (11) ((A - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A: - C
B: D (12) (- C + - D) = (- C - D)
11.12RE1 (13) ((A - C) - D) = (A + (- C - D))
13CE1 (14) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A: A
B: - C
C: - D (9) ((A + - C) + - D) = (A + (- C + - D))
9CE1 (10) (A + (- C + - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A: - C
B: D (11) (- C + - D) = (- C - D)
10.11RE1 (12) (A + (- C - D)) = ((A + - C) + - D)
12
.5RE1 (13) (A + (- C - D)) = ((A + - C) - D)
13.2RE1 (14) (A + (- C - D)) = ((A - C) - D)

```

Fig. 9D

```

DERIVE (A + (- C - D)) = ((A - C) - D)

:*CAS A + B = B + A
A::*AS
B::*(-C)-DS(1) (A + (- C - D)) = ((- C - D) + A)

:*NS A + (- B) = A - B
A::*-CS
B::*DS (2) (- C + - D) = (- C - D)

:*CE1S (3) (- C - D) = (- C + - D)

:*1.3RE2S (4) (A + (- C - D)) = ((- C + - D) + A)

:*HE,<,>LPS

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (5) ((A + - C) + - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A + - C
B::D (6) ((A + - C) + - D) = ((A + - C) - D)
5.6RE1 (7) ((A + - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A
B::C (8) (A + - C) = (A - C)
7.8RE1 (9) ((A - C) - D) = (A + (- C + - D))
9.2RE1 (10) ((A - C) - D) = (A + (- C - D))
10CE1 (11) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (5) ((A + - C) + - D) = (A + (- C + - D))
5CE1 (6) (A + (- C + - D)) = ((A + - C) + - D)
6.2RE1 (7) (A + (- C - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A::A + - C
B::D (8) ((A + - C) + - D) = ((A + - C) - D)
7.8RE1 (9) (A + (- C - D)) = ((A + - C) - D)
N A + (- B) = A - B
A::A
B::C (10) (A + - C) = (A - C)
9.10RE1 (11) (A + (- C - D)) = ((A - C) - D)

```

```

:*

```

Fig. 9E

```

DERIVE      (A +(- C - D))=((A - C) - D)

:*CAS      A + B = B + A
A::*AS
B::*(-C)-D5(1) (A +(- C - D))=(((- C - D) + A)

:*NS      A +(- B)= A - B
A::*-CS
B::*D5      (2) (- C + - D)=(- C - D)

:*CE15      (3) (- C - D)=(- C + - D)

:*1.3RE25      (4) (A +(- C - D))=(((- C + - D) + A)

:*CA35      (5) (A +(- C - D))=(A +(- C + - D))

:*HELPS

AS      (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D      (6) ((A + - C) + - D)=(A +(- C + - D))
N      A +(- B)= A - B
A::A + - C
B::D      (7) ((A + - C) + - D)=((A + - C) - D)
6.7RE1      (8) ((A + - C) - D)=(A +(- C + - D))
N      A +(- B)= A - B
A::A
B::C      (9) (A + - C)=(A - C)
8.9RE1      (10) ((A - C) - D)=(A +(- C + - D))
10CE1      (11) (A +(- C + - D))=((A - C) - D)
5.11RE1      (12) (A +(- C - D))=((A - C) - D)

AS      (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D      (6) ((A + - C) + - D)=(A +(- C + - D))
N      A +(- B)= A - B
A::A + - C
B::D      (7) ((A + - C) + - D)=((A + - C) - D)
6.7RE1      (8) ((A + - C) - D)=(A +(- C + - D))
N      A +(- B)= A - B
A::A
B::C      (9) (A + - C)=(A - C)
8.9RE1      (10) ((A - C) - D)=(A +(- C + - D))
10.2RE1      (11) ((A - C) - D)=(A +(- C - D))
11CE1      (12) (A +(- C - D))=((A - C) - D)

AS      (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D      (6) ((A + - C) + - D)=(A +(- C + - D))
6CE1      (7) (A +(- C + - D))=((A + - C) + - D)
7.2RE1      (8) (A +(- C - D))=((A + - C) + - D)
N      A +(- B)= A - B
A::A + - C
B::D      (9) ((A + - C) + - D)=((A + - C) - D)
8.9RE1      (10) (A +(- C - D))=((A + - C) - D)
N      A +(- B)= A - B
A::A
B::C      (11) (A + - C)=(A - C)
10.11RE1      (12) (A +(- C - D))=((A - C) - D)

:*

```

DERIVE (A + (- C - D)) = ((A - C) - D)

```

:*NS A + (- B) = A - B
A::*A-C$
B::*DS (1) ((A - C) + - D) = ((A - C) - D)

:*HELPS

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (2) ((A + - C) + - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A
B::C (3) (A + - C) = (A - C)
2.3RE1 (4) ((A - C) + - D) = (A + (- C + - D))
N A + (- B) = A - B
A::- C
B::D (5) (- C + - D) = (- C - D)
4.5RE1 (6) ((A - C) + - D) = (A + (- C - D))
1.6RE1 (7) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (2) ((A + - C) + - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A + - C
B::D (3) ((A + - C) + - D) = ((A + - C) - D)
2.3RE1 (4) ((A + - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A::A
B::C (5) (A + - C) = (A - C)
4.5RE1 (6) ((A - C) - D) = (A + (- C + - D))
N A + (- B) = A - B
A::- C
B::D (7) (- C + - D) = (- C - D)
6.7RE1 (8) ((A - C) - D) = (A + (- C - D))
8CE1 (9) (A + (- C - D)) = ((A - C) - D)

```

```

AS (A + B) + C = A + (B + C)
A::A
B::- C
C::- D (2) ((A + - C) + - D) = (A + (- C + - D))
2CE1 (3) (A + (- C + - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A::- C
B::D (4) (- C + - D) = (- C - D)
3.4RE1 (5) (A + (- C - D)) = ((A + - C) + - D)
N A + (- B) = A - B
A::A + - C
B::D (6) ((A + - C) + - D) = ((A + - C) - D)
5.6RE1 (7) (A + (- C - D)) = ((A + - C) - D)
N A + (- B) = A - B
A::A
B::C (8) (A + - C) = (A - C)
7.8RE1 (9) (A + (- C - D)) = ((A - C) - D)

```

DERIVE (A +(- C - D))=((A - C) - D)

:*NS A +(- B)= A - B
 A::*A-C\$
 B::*D\$ (1) ((A - C) + - D)=((A - C) - D)

:*NS A +(- B)= A - B
 A::*AS
 B::*CS (2) (A + - C)=(A - C)

:*2CE1\$ (3) (A - C)=(A + - C)

:*1.3RE1\$ (4) ((A + - C) + - D)=((A - C) - D)

:*HELPS

AS (A + B)+ C = A +(B + C)

A::A

B::- C

C::- D (5) ((A + - C) + - D)=(A +(- C + - D))

N A +(- B)= A - B

A::- C

B::D (6) (- C + - D)=(- C - D)

5.6RE1 (7) ((A + - C) + - D)=(A +(- C - D))

4.7RE1 (8) (A +(- C - D))=((A - C) - D)

AS (A + B)+ C = A +(B + C)

A::A

B::- C

C::- D (5) ((A + - C) + - D)=(A +(- C + - D))

N A +(- B)= A - B

A::A + - C

B::D (6) ((A + - C) + - D)=((A + - C) - D)

5.6RE1 (7) ((A + - C) - D)=(A +(- C + - D))

7.2RE1 (8) ((A - C) - D)=(A +(- C + - D))

N A +(- B)= A - B

A::- C

B::D (9) (- C + - D)=(- C - D)

8.9RE1 (10) ((A - C) - D)=(A +(- C - D))

10CE1 (11) (A +(- C - D))=((A - C) - D)

AS (A + B)+ C = A +(B + C)

A::A

B::- C

C::- D (5) ((A + - C) + - D)=(A +(- C + - D))

5CE1 (6) (A +(- C + - D))=((A + - C) + - D)

N A +(- B)= A - B

A::- C

B::D (7) (- C + - D)=(- C - D)

6.7RE1 (8) (A +(- C - D))=((A + - C) + - D)

N A +(- B)= A - B

A::A + - C

B::D (9) ((A + - C) + - D)=((A + - C) - D)

8.9RE1 (10) (A +(- C - D))=((A + - C) - D)

10.2RE1 (11) (A +(- C - D))=((A - C) - D)

:**

```

DERIVE      (A +(- C - D))=((A - C) - D)

:*NS      A +(- B)= A - B
A::*A-C$
B::*D$     (1) ((A - C) + - D)=((A - C) - D)

:*NS      A +(- B)= A - B
A::*A$
B::*C$     (2) (A + - C)=(A - C)

:*2CE1$   (3) (A - C)=(A + - C)

:*1.3RE1$ (4) ((A + - C) + - D)=((A - C) - D)

:*AR1$    (5) (A +(- C + - D))=((A - C) - D)

:*HELPS$

```

```

N      A +(- B)= A - B
A::- C
B::~D     (6) (- C + - D)=(- C - D)
5.6RE1   (7) (A +(- C - D))=((A - C) - D)

```

```

AS      (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D    (6) ((A + - C) + - D)=(A +(- C + - D))
N      A +(- B)= A - B
A::A + - C
B::~D     (7) ((A + - C) + - D)=((A + - C) - D)
6.7RE1   (8) ((A + - C) - D)=(A +(- C + - D))
8.2RE1   (9) ((A - C) - D)=(A +(- C + - D))
N      A +(- B)= A - B
A::- C
B::~D     (10) (- C + - D)=(- C - D)
9.10RE1  (11) ((A - C) - D)=(A +(- C - D))
11CE1    (12) (A +(- C - D))=((A - C) - D)

```

```

AS      (A + B)+ C = A +(B + C)
A::A
B::- C
C::- D    (6) ((A + - C) + - D)=(A +(- C + - D))
6CE1     (7) (A +(- C + - D))=((A + - C) + - D)
N      A +(- B)= A - B
A::- C
B::~D     (8) (- C + - D)=(- C - D)
7.8RE1   (9) (A +(- C - D))=((A + - C) + - D)
N      A +(- B)= A - B
A::A + - C
B::~D     (10) ((A + - C) + - D)=((A + - C) - D)
9.10RE1  (11) (A +(- C - D))=((A + - C) - D)
11.2RE1  (12) (A +(- C - D))=((A - C) - D)

```

(as shown in Figure 10). He had to call on outside assistance for the answer. He was given a suggestion, did line 5, and then called back. Observe that the theorem-prover can find a solution using the method suggested in the stored hints once the student has already done some work in that direction. The proctor's situation for each of the two methods, the stored hints and the theorem-prover/analyzer, is illustrated in Figures 10 and 11.

3. Lisa: A Case Study in Tutoring Elementary Algebra

Eleven-year-old Lisa had taken the Stanford logic-algebra course during her fifth and sixth grades at Walter Hays Elementary School, Palo Alto, California. Now in junior high school, she has also worked on the "Finding-Axioms" exercise explained in Chapter I. Lisa helped develop the dialogue for the computer-tutor by working on algebra problems taken from the Stanford curriculum. These problems were selected only because they included stored hints; it could thus be assumed that the curriculum writer thought some students would experience difficulty working these problems. Lisa was encouraged to request help whenever she wanted it. The Stanford program uses a timing device to decide when to offer a hint. A student could only receive a hint if the hint was first offered by the program, and then the student requested it by typing the command HINT. As Lisa was accustomed to having a hint offered before she could get it, she hesitated at first to initiate help sequences. Most of the problems on

221.13:

DERIVE: $A+((-C)-D)=(A-C)-D$

HINT TYPE HINT AND THE ENTER KEY IF YOU WANT A HINT.
 START BY USING N TO GET
 '(A-C)+(-D)=(A-C)-D'.

N $A+(-B)=A-B$

A:A-C

B:D

(1) $(A-C)+(-D)=(A-C)-D$

HINT TYPE HINT AND THE ENTER KEY IF YOU WANT A HINT.
 TO FINISH YOU WILL NEED
 TO USE N, RE, AND AR.

N $A+(-B)=A-B$

A:A

B:C

(2) $A+(-C)=A-C$

CE1

(3) $A-C=A+(-C)$

1.3RE1

(4) $(A+(-C))+(-D)=(A-C)-D$

HINT

HINT IS NOT A RULE

FIN

GOODBYE...

Fig. 10.--Tutoring using stored hints.

```

15
DERIVE      (A +(- C - D))=((A - C) - D)

:*NS$  A +(- B)= A - B
A::*A-C$
B::*D$    (1)  ((A - C) + - D)=((A - C) - D)

:*NS$  A +(- B)= A - B
A::*A$
B::*C$    (2)  (A + - C)=(A - C)

:*2CE1$   (3)  (A - C)=(A + - C)

:*1.3RE1$ (4)  ((A + - C) + - D)=((A - C) - D)

:*HELPS$  TRY TO APPLY THE AS AXIOM TO LINE 4.

:*4AR1$   (5)  (A +(- C + - D))=((A - C) - D)

:*HELPS$  SHOW THAT (-C)+(-D)=(-C)-D. DO YOU SEE HOW?
*NOS
          USE THE N AXIOM AND LET A BE -C AND
          B BE D.

:*NS$  A +(- B)= A - B
A::*-C$
B::*D$   (6)  (- C + - D)=(- C - D)

:*5.6RE1$ (7)  (A +(- C - D))=((A - C) - D)

CORRECT...
T
*
```

Fig. 11.--Tutoring using the theorem-prover to give hints.

which she worked were familiar because of her earlier experience with the curriculum, but of course Lisa did not always recall their solutions and so she could remain a subject for the study. The proofs shown in Figure 12 represent the problems on which Lisa experienced difficulty, either from the start or because she was asked to redo the problem using a different set of rules.¹

Each problem gives some indication of the kind of analysis the theorem-prover can carry out so that the instructional system can provide the tutorial dialogue. The computer-tutor starts with subtle hints or reminders that enable the student to come up with his own ideas. It is evident from the dialogues that Lisa did not always do precisely what the computer-tutor suggested. Yet the theorem-prover could reevaluate her work in order to respond to the new approach she was using. The same would be true for any student. Note that only if the student's method results in a very long or clumsy proof is she asked to return to the problem.

Curriculum modification and problem generation. The students observed using the Stanford program did not, for the most part, ask general questions about proof strategies. However, the intention of the program is not to teach them how to manipulate algebraic symbols, but to emphasize methods for discovering solutions. Thus the computer-tutor tries to point out a variety of possible solutions for a single problem. If there are other solutions to the problem, the theorem-prover

¹These problems are numbered 215.8, 217.15, 217.55, 221.8, 223.18, 223.19, 223.100, 223.101, 223.103, 224.23, 224.25, 224.29, 224.31 in the Stanford curriculum; copyright by the Board of Trustees of Leland Stanford, Jr., University, 1970.

*(START)
 WHO ARE YOU (TYPE S OR T)?
 *SS
 PLEASE TYPE YOUR NUMBER.
 *7S
 LISA PREBLE
 OK?
 *YESS

I
 DERIVE (A = 0) → ((A + B) = B)
 P (1) (0 + B) = B

:*HELPS TYPE THE ANTECEDENT OF THE
 CONDITIONAL YOU WANT TO DERIVE AS A
 WP. DO YOU SEE HOW?
 *NOS
 THE ANTECEDENT IS A=0.
 NOW DO YOU SEE?
 *YESS

:*WPS (2) *A=0S

:*HELPS DO YOU SEE HOW TO REPLACE 0 IN LINE 1
 BY A?
 *NOS
 TRY TO APPLY THE CE RULE TO LINE 2.
 THEN REPLACE THE 0 IN LINE 1 BY A.

:*2CE1S (3) 0 = A

:*1-3RE1S (4) (A + B) = B

:*2-4CP5 (5) (A = 0) → ((A + B) = B)

CORRECT...

Fig. 12.--A dialogue with Lisa.

2
 DERIVE ((C + E) + B) = (C + (E + (4 + 2)))
 P (1) B = (4 + 2)

*AES
 :C+ES (2) (B + (C + E)) = ((4 + 2) + (C + E))

*ICEIS (3) (4 + 2) = B

*2.3REIS (4) (B + (C + E)) = (B + (C + E))

*ASS (A + B) + C = A + (B + C)
 A::*CS
 B::*ES
 C::*BS (5) ((C + E) + B) = (C + (E + B))

*5CA2S (6) (B + (C + E)) = (C + (E + B))

*4.6RE2S (7) (B + (C + E)) = (C + (E + B))

*CA4S (8) (B + (C + E)) = (C + (B + E))

*8.1RE2S (9) (B + (C + E)) = (C + ((4 + 2) + E))

*CA5S (10) (B + (C + E)) = (C + (E + (4 + 2)))

*CAIS (11) ((C + E) + B) = (C + (E + (4 + 2)))

CORRECT...

BUT LISA. LOOK AT LINE 5. WHAT IS THE
 DIFFERENCE BETWEEN LINE 5 AND THE ANSWER?
 DO YOU SEE AN EASIER PROOF?

*YES

TELL ME THE NEXT COMMAND (AFTER LINE 5).

*5.1RE2S

YES, THAT'S CORRECT.

3

HERE'S THE CLINCHER FOR THIS LESSON..
 DO THIS ONE CAREFULLY ...

DERIVE ((A + B) + (C + D)) = (A + (B + (C + D)))

*CAS A + B = B + A
 A::*A+BS
 B::*C+DS (1) ((A + B) + (C + D)) = ((C + D) + (A + B))

*1AL2S (2) ((A + B) + (C + D)) = (((C + D) + A) + B)

*1CA4S (3) ((A + B) + (C + D)) = ((D + C) + (A + B))

*1CA5S (4) ((A + B) + (C + D)) = ((A + B) + (C + D))

*HELPS WHAT IS THE DIFFERENCE BETWEEN THE
 PARENTHESES AROUND THE TERMS IN LINE
 4 AND THOSE IN THE PROBLEM?

*4AR2S (5) ((A + B) + (C + D)) = (A + (B + (C + D)))

CORRECT...

CORRECT, BUT NOW TRY THE PROBLEM AGAIN.
 WHAT AXIOM LOOKS JUST LIKE THE PROBLEM?

DERIVE ((A + B) + (C + D)) = (A + (B + (C + D)))

*ASS (A + B) + C = A + (B + C)
 A::*AS
 B::*BS
 C::*C+DS (1) ((A + B) + (C + D)) = (A + (B + (C + D)))

CORRECT...

```

4
DERIVE      (9 - 8)=(- 8 + 9)

:*NS      A +(- B)= A - B
A:!*8$
B:!*9$      (1) (8 + - 9)=(8 - 9)

:*CE1$      (2) (8 - 9)=(8 + - 9)

:*HELPS      TRY A DIFFERENT SUBSTITUTION FOR THE N AXIOM.

:*NS      A +(- B)= A - B
A:!*9$
B:!*8$      (3) (9 + - 8)=(9 - 8)

:*CE1$      (4) (9 - 8)=(9 + - 8)

:*CA1$      (5) (9 - 8)=(- 8 + 9)

CORRECT...

5
DERIVE      ((0 + A) + C)=(A + C)

:*THIS      0 + A = A
A:!*A+C$    (1) (0 +(A + C))=(A + C)

:*CAS      A + B = B + A
A:!*0$
B:!*A+$
          NOT A WELL-FORMED TERM
B:!*$
          TRY AGAIN

:*HELPS      DO YOU REMEMBER THE AS AXIOM?
*YESS

:*ASS      (A + B)+ C = A +(B + C)
A:!*0$
B:!*A$
C:!*C$      (2) ((0 + A) + C)=(0 +(A + C))

:*CAS      A + B = B + A
A:!*0$
B:!*A+C$    (3) (0 +(A + C))=((A + C) + 0)

:*2CE1$      (4) (0 +(A + C))=((0 + A) + C)

:*1.4RE1$    (5) ((0 + A) + C)=(A + C)

CORRECT...

```

Fig. 12, continued.

6
 DERIVE $((-(A + C) + A) + C) = 0$

!*HELPS THE LEFT-HAND SIDE OF WHAT AXIOM
 LOOKS LIKE THE LEFT-HAND SIDE
 OF THE PROBLEM?

!*NS $A + (-B) = A - B$
 A:!*A+CS
 B:!*-(A+C)\$ (1) $((A + C) + - (A + C)) = ((A + C) - (A + C))$

!*NS $A + (-B) = A - B$
 A:!*A+CS
 B:!*A+CS (2) $((A + C) + -(A + C)) = ((A + C) - (A + C))$

!*2CE1\$ (3) $((A + C) - (A + C)) = ((A + C) + -(A + C))$

!*2.3RE1\$ (4) $((A + C) + -(A + C)) = ((A + C) + -(A + C))$

!*HELPS TRY TO SHOW THAT $(A+C)+(-(A+C))=0$.

!*TH2\$ $(-A) + A = 0$
 A:!*A+CS (5) $(-(A + C) + (A + C)) = 0$

!*CA2\$ (6) $((A + C) + -(A + C)) = 0$

!*4.6RE2\$ (7) $((A + C) + -(A + C)) = 0$

!*AR2\$
 THERE ARE NOT 2 OCCURRENCES IN LINE 7 OF AN
 EXPRESSION TO WHICH THE RULE AR CAN BE APPLIED.

!*AR1\$ (8) $(A + (C + -(A + C))) = 0$

!*7CA2\$ (9) $(-(A + C) + (A + C)) = 0$

!*9AL1\$ (10) $((-(A + C) + A) + C) = 0$

CORRECT...
 LISA, LET'S DO THIS ONE AGAIN. I SUGGEST
 YOU START WITH THE AS AXIOM.

DERIVE $((-(A + C) + A) + C) = 0$

!*ASS $(A + B) + C = A + (B + C)$
 A:!*AS
 B:!*BS
 C:!*AS (1) $((A + B) + A) = (A + (B + A))$

!*DLL\$

!*ASS $(A + B) + C = A + (B + C)$
 A:!*-(A+C)\$
 B:!*AS
 C:!*CS (1) $((-(A + C) + A) + C) = (-(A + C) + (A + C))$

!*THIS $0 + A = A$
 A:!*S
 TRY AGAIN

!*TH2\$ $(-A) + A = 0$
 A:!*A+CS (2) $(-(A + C) + (A + C)) = 0$

!*1.2RE1\$ (3) $((-(A + C) + A) + C) = 0$

CORRECT...

Fig. 12, continued.

7

THIS IS THEOREM 5.

PROVE $0 = -0$

:*AIS $A + (-A) = 0$
 A::*AS (1) $(A + -A) = 0$

:*NS $A + (-B) = A - B$
 A::*AS
 B::*AS (2) $(A + -A) = (A - A)$

*1.2REIS (3) $(A - A) = 0$

*HELPS TRY TO SHOW THAT $0 + -0 = -0$.

:*ZS $A + 0 = A$
 A::*-OS (4) $(-0 + 0) = -0$

*HELPS DID YOU CARRY OUT THE FIRST SUGGESTION?

*YES\$

NO YOU DIDN'T. LOOK CAREFULLY AT LINE 4.

*CAIS (5) $(0 + -0) = -0$

*HELPS DO YOU SEE WHAT TO DO IF YOU SUBSTITUTE
0 FOR A IN THE A1 AXIOM?

*YES\$

OK, DO IT.

:*AIS $A + (-A) = 0$
 A::*OS (6) $(0 + -0) = 0$

*5.6REIS (7) $0 = -0$

CORRECT...

8

HERE IS THEOREM 6 FOR YOU TO PROVE.

PROVE $(A - 0) = A$

:*NS $A + (-B) = A - B$
 A::*AS
 B::*OS (1) $(A + -0) = (A - 0)$

:*THSS $0 = -0$
 (2) $0 = -0$

*CEIS (3) $-0 = 0$

*1.3REIS (4) $(A + 0) = (A - 0)$

:*ZS $A + 0 = A$
 A::*AS (5) $(A + 0) = A$

*4.5REIS (6) $A = (A - 0)$

*CEIS (7) $(A - 0) = A$

CORRECT... IF YOU STARTED WITH THE Z AXIOM, DO YOU SEE
WHAT TO DO?

*YES\$

OK.

Fig. 12, continued.

9
 DERIVE (5 - 0)=(5 + 0)
 :*INITS
 YOU CAN NOW REQUEST A DERIVE OR PROVE
 PROBLEM, OR DERIVE A NEW RULE OF INFERENCE.
 THE INTERRUPTED PROBLEM WILL BE RESTARTED
 IF YOU TYPE FIN .

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: A-0=A\$
 DERIVE (A - 0)= A

:*Z\$ A + 0 = A
 A::*AS (1) (A + 0)= A

:*TH5\$ 0 = - 0
 (2) 0 = - 0

:*1-2RE1\$ (3) (A + - 0)= A

:*3N1\$ (4) (A - 0)= A

CORRECT...

TYPE A DERIVE, PROVE OR RULE COMMAND

:*FINS

NOW REDO THE PROBLEM YOU INTERRUPTED

9
 DERIVE (5 - 0)=(5 + 0)

:*HELPS THE LEFT-HAND SIDE OF WHAT THEOREM
 LOOKS LIKE THE LEFT-HAND SIDE OF
 THE PROBLEM?

:*TH6\$ A - 0 = A
 A::*5\$ (1) (5 - 0)= 5

:*Z\$ A + 0 = A
 A::*CE1\$
 NOT A WELL-FORMED TERM
 A::*5\$ (2) (5 + 0)= 5

:*CE1\$ (3) 5 =(5 + 0)

:*1-3RE2\$ (4) (5 - 0)=(5 + 0)

CORRECT...

Fig. 12, continued.

10
 DERIVE $(0 - - 0) = 0$

:*TH6\$ $A - 0 = A$
 A:!*-0\$ (1) $(- 0 - 0) = - 0$

:*TH6\$ $A - 0 = A$
 A:!*0\$ (2) $(0 - 0) = 0$

:*HELPS DO YOU RECALL TH5?
 *YES
 TRY TO USE IT IN THE PROOF.

:*TH5\$ $0 = - 0$
 (3) $0 = - 0$

:*CE1\$ (4) $- 0 = 0$

:*2.4RE1\$
 THERE IS NOT A FREE OCCURRENCE OF $(- 0)$ IN LINE 2

:*HELPS APPLY TH5 TO LINE 2. SEE?
 *NOS
 REPLACE OCCURRENCE 2 OF 0 IN LINE 2 BY -0

:*2.3RE2\$ (5) $(0 - - 0) = 0$

CORRECT...

11
 NOW WE ARE READY TO PROVE THEOREM 7,
 WHICH IS CALLED THE CANCELLATION LAW FOR ADDITION.
 IT IS JUST LIKE WHAT WE HAVE BEEN DOING, BUT
 REQUIRES A CONDITIONAL PROOF TOO.

PROVE $((A + B) = (A + C)) \rightarrow (B = C)$

:*WPS (1) $*A+B=A+C$$

:*1CA1\$ (2) $(B + A) = (A + C)$

:*2CA2\$ (3) $(B + A) = (C + A)$

:*AES
 !:!*-A\$ (4) $((B + A) + - A) = ((C + A) + - A)$

:*4AR1\$ (5) $(B + (A + - A)) = ((C + A) + - A)$

:*A1\$ $A + (- A) = 0$
 A:!*A\$ (6) $(A + - A) = 0$

:*5.6RE1\$ (7) $(B + 0) = ((C + A) + - A)$

:*Z\$ $A + 0 = A$
 A:!*B\$ (8) $(B + 0) = B$

:*7.8RE1\$ (9) $B = ((C + A) + - A)$

:*AR1\$ (10) $B = (C + (A + - A))$

:*10.6RE1\$ (11) $B = (C + 0)$

:*Z\$ $A + 0 = A$
 A:!*C\$ (12) $(C + 0) = C$

:*11.12RE1\$(13) $B = C$

:*1.13CPS (14) $((A + B) = (A + C)) \rightarrow (B = C)$

CORRECT...

```

12
DERIVE      ((B + 0) = (B + C)) -> (C = 0)

:*WPS      (1)      *B+0=B+C$
:*CE1$     (2)      (B + C) = (B + 0)
:*DLS
:*I21$     (2)      B = (B + C)
:*CA1$     (3)      B = (C + B)
:*AES
:*+B$      (4)      (B + - B) = ((C + B) + - B)
:*AR1$     (5)      (B + - B) = (C + (B + - B))

:*A1$      A + (- A) = 0
A: :*B$     (6)      (B + - B) = 0
:*5.6RE1$ (7)      0 = (C + (B + - B))
:*7.6RE1$ (8)      0 = (C + 0)

:*Z$       A + 0 = A
A: :*C$     (9)      (C + 0) = C
:*8.9RE1$ (10)     0 = C
:*CE1$     (11)     C = 0
:*1.11CP$ (12)    ((B + 0) = (B + C)) -> (C = 0)

```

CORRECT...

LISA, DOES THIS PROOF RESEMBLE ANYTHING YOU
HAVE DONE BEFORE?

*YES\$

WHAT THEOREM IS IT LIKE?

*TH7\$

TRY THE PROOF AGAIN. THIS TIME, DO NOT USE WP.

```

DERIVE      ((B + 0) = (B + C)) -> (C = 0)

:*TH7$      A + B = A + C -> B = C
A: :*B$
B: :*0$
C: :*C$     (1)    ((B + 0) = (B + C)) -> (0 = C)

:*1CE2$     (2)    ((B + 0) = (B + C)) -> (C = 0)

```

CORRECT...

Fig. 12, continued.

```

13
DERIVE      A = (5 + - 3)
P           (1) (A + 3) = 5

:*HELPS    YOU WANT TO GET AN INSTANCE OF TH7
            SO THAT THE CONSEQUENT OF TH7 IS THE
            SAME AS THE PROBLEM.

:*TH7$     A + B = A + C -> B = C
A:!*A$
B:!*3$
C:!*5+(-3)$ (2) ((A + 3) = (A + (5 + - 3))) -> (3 = (5 + - 3))

:*X
HELPS      YOUR SUBSTITUTIONS FOR TH7 MIGHT BE CHANGED.
            TRY LETTING A BE 3 AND B BE A INSTEAD. THEN
            THE CONSEQUENT WILL BE EXACTLY WHAT YOU WANT.

:*DLL$

:*TH7$     A + B = A + C -> B = C
A:!*3$
B:!*A$
C:!*5+(-3)$ (2) ((3 + A) = (3 + (5 + - 3))) -> (A = (5 + - 3))

:*HELPS    TRY TO USE CA AND AR TO MOVE THE 3'S IN LINE 2.
            THEN YOU CAN ELIMINATE THE 3'S USING Z AND TH2.

:*CA2$     (3) ((3 + A) = ((5 + - 3) + 3)) -> (A = (5 + - 3))

:*AR3$
            THERE ARE NOT 3 OCCURRENCES IN LINE 3 OF AN
            EXPRESSION TO WHICH THE RULE AR CAN BE APPLIED.

:*AR1$     (4) ((3 + A) = (5 + (- 3 + 3))) -> (A = (5 + - 3))

:*R<R>TH2$ (- A) + A = 0
A:!*3$     (5) (- 3 + 3) = 0

:*4.SRE1$  (6) ((3 + A) = (5 + 0)) -> (A = (5 + - 3))

:*Z$      A + 0 = A
A:!*5$     (7) (5 + 0) = 5

:*6.7RE1$ (8) ((3 + A) = 5) -> (A = (5 + - 3))

:*CA1$     (9) ((A + 3) = 5) -> (A = (5 + - 3))

:*9.1AAS   (10) A = (5 + - 3)

CORRECT...

```

Fig. 12, continued.

14

HERE IS THEOREM 8 FOR YOU TO PROVE.

PROVE $((A + B) = C) \rightarrow (A = (C - B))$:*TH7\$ $A + B = A + C \rightarrow B = C$

A: :*BS

B: :*AS

C: :*C-B\$ (1) $((B + A) = (B + (C - B))) \rightarrow (A = (C - B))$:*1CA1\$ (2) $((A + B) = (B + (C - B))) \rightarrow (A = (C - B))$

:*HELPS REMEMBER THAT TO SUBTRACT A TERM IS THE SAME AS TO ADD THE NEGATIVE OF THE TERM.

:*HELPS USE N AXIOM TO CHANGE OCCURRENCE 1 OF C-B TO C+(-B).

:*NS $A + (-B) = A - B$

A: :*CS

B: :*BS (3) $(C + -B) = (C - B)$:*CE1\$ (4) $(C - B) = (C + -B)$:*2.4E<E>RE1\$ (5) $((A + B) = (B + (C + -B))) \rightarrow (A = (C - B))$:*HELPS YOU WANT TO CHANGE $B + (C + (-B))$ TO C.
USE CA, AR, AND TH2.:*CA2\$ (6) $((A + B) = ((C + -B) + B)) \rightarrow (A = (C - B))$:*AR1\$ (7) $((A + B) = (C + (-B + B))) \rightarrow (A = (C - B))$:*TH2\$ $(-A) + A = 0$ A: :*BS (8) $(-B + B) = 0$:*7.8RE1\$ (9) $((A + B) = (C + 0)) \rightarrow (A = (C - B))$:*9Z1\$ (10) $((A + B) = C) \rightarrow (A = (C - B))$

CORRECT...

Fig. 12, continued.

might find them. It is possible to point out shorter proofs, especially the one- or two-line solutions stemming from instantiations of a forgotten axiom or theorem. It is also possible to note that a problem can be solved from a recently proven theorem, and to encourage the student to make use of that theorem if he has not already done so. If the help sequence suggests a command the student does not know (although was taught), another solution can be chosen for discussion, or the computer-tutor can initiate a routine to reteach the command (as described in Chapter VI).

Ideas brought forth in the preceding paragraph can be classified under the title curriculum modification. Most CAI systems use this title to mean a static set of problems which are presented in different orders and/or with varying degrees of omissions. Lately, CAI systems have included problem generators, mechanical devices by which new problems, within the format and constraints of the subject and lesson to be given, are automatically determined. Uttal's (1969) program, described in Chapter II, serves as a good example of a system incorporating a problem generator. In teaching elementary proof construction, it is possible to generate new problems by using alphabetic variants of the expressions, replacing variables by complex terms, commuting identities, and reordering premise lines. A more significant technique is to require that the student redo the same problem under different constraints, that is, with the available list of axioms, theorems, and rules of inference changed. Such problem generation, which attempts to modify the course material as prespecified by the curriculum writer, depends on an ability to verify that the problem can indeed be solved within the new constraints--an ability afforded the instructional system by the use of the theorem-prover.

The dialogue routine. Before highlighting Figure 12, Lisa's work, some comment should be made on the dialogue routine itself. The statements made by the dialogue routine are constructed in a simple fashion, by filling in predetermined sentence patterns. At the first request for help in a given problem, a subtle hint is offered, and, if help is requested again, the hint becomes more informative, until at the third help request the student is explicitly told which command to use. The dialogue enters remedial loops whose purpose is to reteach commands, to suggest that the student redo a problem in a different way, and to generate a new problem of similar form. If the theorem-prover has found more than one solution to a problem, the dialogue chooses for discussion the one which uses the most lines generated by the student. However, that choice is altered if (a) the student has requested commands like the theorem-prover's except for the substitution sequence, (b) the student has used commands outside the permissible set, or (c) a solution exists which includes a rule which must be used while the earlier choice does not. If none of the theorem-prover's solutions uses the student's lines, the shortest admissible proof is chosen. In any case, the dialogue routine knows why the theorem-prover discovered the solution at all, due to the stored flags that indicate which heuristics and special procedures were used.

Figure 13 contains complete solutions discovered by the theorem-prover for the partial proofs constructed by Lisa and shown in Figure 12 as a dialogue from the user's viewpoint. The proofs in Figure 13 will be discussed, with references to Figure 12, in order to convey some additional information about the proof-analyzer. This study was carried out in two parts. First, Lisa was asked to work the problems, requesting help as she

```

DERIVE      (A = 0) -> ((A + B) = B)

:*PS       (1)  *0+B=B$

:*HELPS

WP          (2)      A = 0
2AE
A::B       (3)      (A + B) = (0 + B)
3.1RE1     (4)      (A + B) = B
2.4CP      (5)      (A = 0) -> ((A + B) = B)

```

```

:*WPS      (2)      *A=0$

:*HELPS

2AE
A::B       (3)      (A + B) = (0 + B)
3.1RE1     (4)      (A + B) = B
2.4CP      (5)      (A = 0) -> ((A + B) = B)

```

1A

```

DERIVE      (A = 0) -> ((A + B) = B)

:*PS       (1)  *0+B=B$

:*CAS      A + B = B + A
A::*AS
B::*B$     (2)  (A + B) = (B + A)

:*WPS      (3)      *A=0$

:*2.3RE2$ (4)      (A + B) = (B + 0)

:*HELPS

```

```

3COPY      (5)      A = 0
Z  A + 0 = A
A::B       (6)      (B + 0) = B
4.6RE1     (7)      (A + B) = B
3.7CP      (8)      (A = 0) -> ((A + B) = B)

```

```

3COPY      (5)      A = 0
5AE
A::B       (6)      (A + B) = (0 + B)
6.1RE1     (7)      (A + B) = B
3.7CP      (8)      (A = 0) -> ((A + B) = B)

```

**

1B

Fig. 13.--The theorem-prover's completions to the partial proofs in Fig. 12. (Format is the same as Figures 8 and 9.)

DERIVE $((C + E) + B) = (C + (E + (4 + 2)))$

*PS (1) *B=4+2\$

*HELPS

1AE

A::C + E (2) $(B + (C + E)) = ((4 + 2) + (C + E))$

2CA1 (3) $((C + E) + B) = ((4 + 2) + (C + E))$

CA $A + B = B + A$

A::4 + 2

B::C + E (4) $((4 + 2) + (C + E)) = ((C + E) + (4 + 2))$

AS $(A + B) + C = A + (B + C)$

A::C

B::E

C::4 + 2 (5) $((C + E) + (4 + 2)) = (C + (E + (4 + 2)))$

4.5RE1 (6) $((4 + 2) + (C + E)) = (C + (E + (4 + 2)))$

3.6RE1 (7) $((C + E) + B) = (C + (E + (4 + 2)))$

AS $(A + B) + C = A + (B + C)$

A::C

B::E

C::B (2) $((C + E) + B) = (C + (E + B))$

2.1RE2 (3) $((C + E) + B) = (C + (E + (4 + 2)))$

CA $A + B = B + A$

A::C + E

B::B (2) $((C + E) + B) = (B + (C + E))$

AS $(A + B) + C = A + (B + C)$

A::C

B::E

C::4 + 2 (3) $((C + E) + (4 + 2)) = (C + (E + (4 + 2)))$

CA $A + B = B + A$

A::C + E

B::4 + 2 (4) $((C + E) + (4 + 2)) = ((4 + 2) + (C + E))$

1CE1 (5) $(4 + 2) = B$

4.5RE2 (6) $((C + E) + (4 + 2)) = (B + (C + E))$

3.6RE1 (7) $(B + (C + E)) = (C + (E + (4 + 2)))$

2.7RE1 (8) $((C + E) + B) = (C + (E + (4 + 2)))$

.*

DERIVE ((A + B) + (C + D)) = (A + (B + (C + D)))

:*CAS A + B = B + A

A::*A+B\$

B::*C+D\$ (1) ((A + B) + (C + D)) = ((C + D) + (A + B))

:*HELPS

AS (A + B) + C = A + (B + C)

A::C

B::D

C::A + B (2) ((C + D) + (A + B)) = (C + (D + (A + B)))

AS (A + B) + C = A + (B + C)

A::A

B::B

C::C + D (3) ((A + B) + (C + D)) = (A + (B + (C + D)))

AS (A + B) + C = A + (B + C)

A::A + B

B::C

C::D (4) (((A + B) + C) + D) = ((A + B) + (C + D))

4CE1 (5) ((A + B) + (C + D)) = (((A + B) + C) + D)

5AR6 (6) ((A + B) + (C + D)) = ((A + B) + (C + D))

6CA5 (7) ((A + B) + (C + D)) = ((C + D) + (A + B))

7AR5 (8) ((A + B) + (C + D)) = (C + (D + (A + B)))

3.8RE1 (9) (C + (D + (A + B))) = (A + (B + (C + D)))

2.9RE1 (10) ((C + D) + (A + B)) = (A + (B + (C + D)))

1.10RE1 (11) ((A + B) + (C + D)) = (A + (B + (C + D)))

AS (A + B) + C = A + (B + C)

A::A

B::B

C::C + D (2) ((A + B) + (C + D)) = (A + (B + (C + D)))

3A

DERIVE ((A + B) + (C + D)) = (A + (B + (C + D)))

:*CAS A + B = B + A

A::*A+B\$

B::*C+D\$ (1) ((A + B) + (C + D)) = ((C + D) + (A + B))

:*1AL2\$ (2) ((A + B) + (C + D)) = (((C + D) + A) + B)

:*1CA4\$ (3) ((A + B) + (C + D)) = ((D + C) + (A + B))

:*1CA5\$ (4) ((A + B) + (C + D)) = ((A + B) + (C + D))

:*HELPS

AS (A + B) + C = A + (B + C)

A::A

B::B

C::C + D (5) ((A + B) + (C + D)) = (A + (B + (C + D)))

3B

Fig. 13, continued.

DERIVE $(9 - 8) = (-8 + 9)$

```

:*NS A + (- B) = A - B
A::*8$
B::*9$ (1) (8 + - 9) = (8 - 9)

:*CE1$ (2) (8 - 9) = (8 + - 9)

:*HELPS

```

```

CA A + B = B + A
A::- 8
B::9 (3) (- 8 + 9) = (9 + - 8)
N A + (- B) = A - B
A::9
B::8 (4) (9 + - 8) = (9 - 8)
3.4RE1 (5) (- 8 + 9) = (9 - 8)

```

```

N A + (- B) = A - B
A::9
B::- 8 (3) (9 + - 8) = (9 - 8)
CA A + B = B + A
A::9
B::- 8 (4) (9 + - 8) = (- 8 + 9)
3.4RE1 (5) (- 8 + 9) = (9 - 8)

```

:*

4

DERIVE $((0 + A) + C) = (A + C)$

```

:*THIS 0 + A = A
A::*A+C$ (1) (0 +(A + C)) = (A + C)

:*CAS A + B = B + A
A::*0$
B::*A+C$ (2) (0 +(A + C)) = ((A + C) + 0)

:*HELPS

```

```

AS (A + B) + C = A +(B + C)
A::A
B::C
C::0 (3) ((A + C) + 0) = (A +(C + 0))
3AL3 (4) ((A + C) + 0) = ((A + C) + 0)
4CA4 (5) ((A + C) + 0) = (0 +(A + C))
5AL3 (6) ((A + C) + 0) = ((0 + A) + C)
Z A + 0 = A
A::A + C (7) ((A + C) + 0) = (A + C)
6.7RE1 (8) (A + C) = ((0 + A) + C)
8CE1 (9) ((0 + A) + C) = (A + C)

```

:*

5

```

DERIVE      ((-(A + C) + A) + C) = 0

:*NS      A + (- B) = A - B
A: :*A+C$
B: :*-(A+C)$ (1) ((A + C) + -(A + C)) = ((A + C) - -(A + C))

:*NS      A + (- B) = A - B
A: :*A+C$
B: :*A+C$ (2) ((A + C) + -(A + C)) = ((A + C) - (A + C))

:*CE1$    (3) ((A + C) - (A + C)) = ((A + C) + -(A + C))

:*2.3RE1$ (4) ((A + C) + -(A + C)) = ((A + C) + -(A + C))

:*TH2$    (- A) + A = 0
A: :*A+C$ (5) (-(A + C) + (A + C)) = 0

:*CA2$    (6) ((A + C) + -(A + C)) = 0

:*4.6RE2$ (7) ((A + C) + -(A + C)) = 0

:*AS$     (A + B) + C = A + (B + C)
A: :*A+C$
B: :*A$
C: :*A$ (8) (((A + C) + A) + A) = ((A + C) + (A + A))

:*DLL$

:*AR1$    (8) (A + (C + -(A + C))) = 0

:*7CA2$   (9) (-(A + C) + (A + C)) = 0

:*9AL1$   (10) ((-(A + C) + A) + C) = 0

```

CORRECT...

6A

Fig. 13, continued.

DERIVE $((-(A + C) + A) + C) = 0$

```

:*TH2$ (- A) + A = 0
A::*A+C$ (1)  $((-(A + C) + (A + C)) = 0$ 
:*AS$ (A + B) + C = A + (B + C)
A::*A+C$
B::*A$
C::*C$ (2)  $((-(A + C) + A) + C) = ((-(A + C) + (A + C))$ 
:*DLS
:*AS$ (A + B) + C = A + (B + C)
A::*-(A+C)$
B::*A$
C::*C$ (2)  $((-(A + C) + A) + C) = ((-(A + C) + (A + C))$ 
:*CE1$ (3)  $((-(A + C) + (A + C)) = ((-(A + C) + A) + C)$ 
:*1.3RE1$ (4)  $((-(A + C) + A) + C) = 0$ 

```

CORRECT...

6B

DERIVE $((-(A + C) + A) + C) = 0$

```

:*AS$ (A + B) + C = A + (B + C)
A::*-(A+C)$
B::*A$
C::*C$ (1)  $((-(A + C) + A) + C) = ((-(A + C) + (A + C))$ 
:*TH1$ 0 + A = A
A::*$
TRY AGAIN
:*TH2$ (- A) + A = 0
A::*A+C$ (2)  $((-(A + C) + (A + C)) = 0$ 
:*1.2RE1$ (3)  $((-(A + C) + A) + C) = 0$ 

```

CORRECT...

6C

DERIVE (0 - A) = - A

```

:*LTS
:**AS (1) - A = - A

:*TH2S (- A) + A = 0
A:**AS (2) (- A + A) = 0

:*TH1S 0 + A = A
A:**AS (3) (0 + - A) = - A

:*CE1S (4) - A = (0 + - A)

:*4.1RE1S (5) - A = (0 + - A)

```

:*HELPS

```

N A + (- B) = A - B
A::0
B::A (6) (0 + - A) = (0 - A)
5.6RE1 (7) - A = (0 - A)
7CE1 (8) (0 - A) = - A

```

```

N A + (- B) = A - B
A::0
B::A (6) (0 + - A) = (0 - A)
3.6RE1 (7) (0 - A) = - A

```

:*

7

DERIVE 0 = - 0

```

:*AIS A + (- A) = 0
A:**AS (1) (A + - A) = 0

:*NS A + (- B) = A - B
A:**AS
B:**AS (2) (A + - A) = (A - A)

:*1.2RE1S (3) (A - A) = 0

:*HELPS

```

```

AI A + (- A) = 0
A::0 (4) (0 + - 0) = 0
Z A + 0 = A
A::- 0 (5) (- 0 + 0) = - 0
CA A + B = B + A
B::0
A::- 0 (6) (- 0 + 0) = (0 + - 0)
5.6RE1 (7) (0 + - 0) = - 0
4.7RE1 (8) - 0 = 0
8CE1 (9) 0 = - 0

```

8A

DERIVE 0 = - 0

:*HELPS

```

TH3 A - A = 0
A::0 (1) (0 - 0) = 0
TH4 0 - A = - A
A::0 (2) (0 - 0) = - 0
1.2RE1 (3) - 0 = 0
3CE1 (4) 0 = - 0

```

:*

8B

Fig. 13, continued.

DERIVE $(-5 + 5) = -0$

:*TH5\$ $0 = -0$
 (1) $0 = -0$

:*TH2\$ $(-A) + A = 0$
 A:*5\$ (2) $(-5 + 5) = 0$

:*CE1\$ (3) $0 = (-5 + 5)$

:*3.1RE1\$ (4) $-0 = (-5 + 5)$

:*HELPS

4.4RE1 (5) $(-5 + 5) = (-5 + 5)$

4CE1 (6) $(-5 + 5) = -0$

5.6RE2 (7) $(-5 + 5) = -0$

4CE1 (5) $(-5 + 5) = -0$

TH2 $(-A) + A = 0$

A:*5 (5) $(-5 + 5) = 0$

5.1RE1 (6) $(-5 + 5) = -0$

2.1RE1 (5) $(-5 + 5) = -0$

4CE1 (5) $(-5 + 5) = -0$

:*

9

DERIVE $(A - 0) = A$

:*HELPS

Z $A + 0 = A$

A:*A (1) $(A + 0) = A$

TH5 $0 = -0$

(2) $0 = -0$

1.2RE1 (3) $(A + -0) = A$

N $A + (-B) = A - B$

A:*A

B:*0 (4) $(A + -0) = (A - 0)$

3.4RE1 (5) $(A - 0) = A$

:*NS $A + (-B) = A - B$

A:*AS

B:*0\$ (1) $(A + -0) = (A - 0)$

:*TH5\$ $0 = -0$

(2) $0 = -0$

:*CE1\$ (3) $-0 = 0$

:*1.3RE1\$ (4) $(A + 0) = (A - 0)$

:*HELPS

Z $A + 0 = A$

A:*A (5) $(A + 0) = A$

4.5RE1 (6) $A = (A - 0)$

6CE1 (7) $(A - 0) = A$

Z $A + 0 = A$

A:*A (5) $(A + 0) = A$

5.2RE1 (6) $(A + -0) = A$

6.1RE1 (7) $(A - 0) = A$

:*

10

Fig. 13, continued.

DERIVE (5 - 0)=(5 + 0)

:*TH6\$ A - 0 = A
A::*5\$ (1) (5 - 0)= 5

:*HELPS

Z A + 0 = A
A::*5 (2) (5 + 0)= 5
2CE1 (3) 5 =(5 + 0)
1.3RE2 (4) (5 - 0)=(5 + 0)

!*

11

DERIVE (5 + 0)=(5 - 0)

:*HELPS

LT
A::*5 + - 0(1) (5 + - 0)=(5 + - 0)
TH5 0 = - 0
(2) 0 = - 0
2CE1 (3) - 0= 0
1.3RE1 (4) (5 + 0)=(5 + - 0)
N A +(- B)= A - B
A::*5
B::0 (5) (5 + - 0)=(5 - 0)
4.5RE1 (6) (5 + 0)=(5 - 0)

:*NS A +(- B)= A - B
A::*5\$
B::*0\$ (1) (5 + - 0)=(5 - 0)

:*HELPS

TH5 0 = - 0
(2) 0 = - 0
2CE1 (3) - 0= 0
1.3RE1 (4) (5 + 0)=(5 - 0)

!*

12

Fig. 13, continued.

```

DERIVE      (0 - - 0) = 0

:*TH6$     A - 0 = A
A::*-0$    (1) (- 0 - 0) = - 0

:*TH6$     A - 0 = A
A::*0$     (2) (0 - 0) = 0

:*TH5$     0 = - 0
           (3) 0 = - 0

:*CE1$     (4) - 0 = 0

:*2.4RE1$
THERE IS NOT A FREE OCCURRENCE OF (- 0) IN LINE 2

:*1.4RE1$  (5) (0 - 0) = - 0

:*1.4RE2$  (6) (- 0 - 0) = 0

:*HELPS

6.4RE1     (7) (0 - 0) = 0
7.3RE2     (8) (0 - - 0) = 0

:*
DERIVE      (0 - - 0) = 0

:*TH6$     A - 0 = A
A::*-0$    (1) (- 0 - 0) = - 0

:*TH6$     A - 0 = A
A::*0$     (2) (0 - 0) = 0

:*HELPS

TH5        0 = - 0
           (3) 0 = - 0
2.3RE2     (4) (0 - - 0) = 0

```

:*

13

```

DERIVE      ((8 + 0) = (8 + C)) -> (C = 0)

```

:*HELPS

```

TH7        A + B = A + C -> B = C
A::B
B::0
C::C
1CE2      (1) ((8 + 0) = (8 + C)) -> (0 = C)
          (2) ((8 + 0) = (8 + C)) -> (C = 0)

```

:*

14

DERIVE $A = (5 + - 3)$

!*PS (1) *A+3=5\$

!*HELPS

TH7 $A + B = A + C \rightarrow B = C$

B: A

C: $5 + - 3$

A: 3 (2) $((3 + A) = (3 + (5 + - 3))) \rightarrow (A = (5 + - 3))$

CA $A + B = B + A$

A: 3

B: A (3) $(3 + A) = (A + 3)$

2-3RE1 (4) $((A + 3) = (3 + (5 + - 3))) \rightarrow (A = (5 + - 3))$

CA $A + B = B + A$

B: $5 + - 3$

A: 3 (5) $(3 + (5 + - 3)) = ((5 + - 3) + 3)$

AS $(A + B) + C = A + (B + C)$

C: 3

B: $- 3$

A: 5 (6) $((5 + - 3) + 3) = (5 + (- 3 + 3))$

5-6RE1 (7) $(3 + (5 + - 3)) = (5 + (- 3 + 3))$

CA $A + B = B + A$

B: 3

A: $- 3$ (8) $(- 3 + 3) = (3 + - 3)$

7-8RE1 (9) $(3 + (5 + - 3)) = (5 + (3 + - 3))$

AI $A + (- A) = 0$

A: 3 (10) $(3 + - 3) = 0$

9-10RE1 (11) $(3 + (5 + - 3)) = (5 + 0)$

Z $A + 0 = A$

A: 5 (12) $(5 + 0) = 5$

11-12RE1 (13) $(3 + (5 + - 3)) = 5$

4-13RE1 (14) $((A + 3) = 5) \rightarrow (A = (5 + - 3))$

14-1AA (15) $A = (5 + - 3)$

1AE

A: $- 3$ (2) $((A + 3) + - 3) = (5 + - 3)$

2AR2 (3) $(A + (3 + - 3)) = (5 + - 3)$

AI $A + (- A) = 0$

A: 3 (4) $(3 + - 3) = 0$

3-4RE1 (5) $(A + 0) = (5 + - 3)$

Z $A + 0 = A$

A: A (6) $(A + 0) = A$

5-6RE1 (7) $A = (5 + - 3)$

!*
15

Fig. 13, continued.

DERIVE $((A + B) = C) \rightarrow (A = (C - B))$

!*HELPS

TH7 $A + B = A + C \rightarrow B = C$
 B: A
 C: $C - B$
 A: B (1) $((B + A) = (B + (C - B))) \rightarrow (A = (C - B))$
 CA $A + B = B + A$
 A: B
 B: A (2) $(B + A) = (A + B)$
 1.2RE1 (3) $((A + B) = (B + (C - B))) \rightarrow (A = (C - B))$
 N $A + (-B) = A - B$
 A: C
 B: B (4) $(C + -B) = (C - B)$
 4CE1 (5) $(C - B) = (C + -B)$
 3.5RE1 (6) $((A + B) = (B + (C + -B))) \rightarrow (A = (C - B))$
 CA $A + B = B + A$
 B: $C + -B$
 A: B (7) $(B + (C + -B)) = ((C + -B) + B)$
 AS $(A + B) + C = A + (B + C)$
 C: B
 B: $-B$
 A: C (8) $((C + -B) + B) = (C + (-B + B))$
 7.8RE1 (9) $(B + (C + -B)) = (C + (-B + B))$
 CA $A + B = B + A$
 B: B
 A: $-B$ (10) $(-B + B) = (B + -B)$
 9.10RE1 (11) $(B + (C + -B)) = (C + (B + -B))$
 AI $A + (-A) = 0$
 A: B (12) $(B + -B) = 0$
 11.12RE1 (13) $(B + (C + -B)) = (C + 0)$
 Z $A + 0 = A$
 A: C (14) $(C + 0) = C$
 13.14RE1 (15) $(B + (C + -B)) = C$
 6.15RE1 (16) $((A + B) = C) \rightarrow (A = (C - B))$

WP (1) $(A + B) = C$
 IAE
 A: $-B$ (2) $((A + B) + -B) = (C + -B)$
 2AR2 (3) $(A + (B + -B)) = (C + -B)$
 AI $A + (-A) = 0$
 A: B (4) $(B + -B) = 0$
 3.4RE1 (5) $(A + 0) = (C + -B)$
 Z $A + 0 = A$
 A: A (6) $(A + 0) = A$
 5.6RE1 (7) $A = (C + -B)$
 N $A + (-B) = A - B$
 A: C
 B: B (8) $(C + -B) = (C - B)$
 7.8RE1 (9) $A = (C - B)$
 1.9CP (10) $((A + B) = C) \rightarrow (A = (C - B))$

!*

needed it. The problems on which she experienced difficulty, and the tutoring she received, are reproduced as Figure 12. Second, each partial proof was reconstructed, only this time, when HELP was requested, the tutor printed the set of solutions discovered by the theorem-prover rather than hints or suggestions, as shown in Figure 13. Annotations to these solutions follow.

1. (a) The theorem-prover uses the command COPY when it references a line already in the proof, but prefers to group the lines in a consecutive block. (b) The dialogue must comment if the student enters a working premise (WP) other than the ones needed in the solution. Since the student will be reminded to delete the WP line or to discharge it with a CP command, some lines already in the proof may be lost.

2. This problem was presented earlier. Lisa solved it successfully. As shown in Figure 12, the theorem-prover analyzed a completed proof to determine if the student did a very long or clumsy one. Section 2 referred to this problem to illustrate that, if the theorem-prover were called on after line 10, a much shorter proof could be found.

3. If the student does some solution other than the one suggested by the theorem-prover, or the student's completed solution is very long compared to that already determined by the theorem-prover, the student could be asked to retry the same problem. As in Figure 12, the theorem-prover could provide some initial comment to get the student restarted.

4. Even when the theorem-prover finds a solution that does not reference the student's work, the dialogue routine can determine the similarity between the prover's results and lines already available; the similarity in this proof is the use of the N axiom with a reversed substitution sequence.

5. The theorem-prover missed an easy completion:

$$\underline{1AL1} \quad (3) \quad (0+A)+C = A+C .$$

The combination AS-CE in the theorem-prover's solutions implies the AL (associate left over addition) rule, whereas AS alone is the AR (associate right over addition) rule. The theorem-prover only uses these short forms for the axioms in sequences stemming from MACROS, the standard transformations. Note that this suggestive hint, rather than one which states a precise command to use, or a formula to generate, allows the student to do some more thinking. The result is that Lisa came up with a better proof than that found by the prover.

6. This example is important because it is a case in which the idealized theorem-prover fails. Lack of a proof, however, was compensated for by the FEATURE TEST heuristic. A hint is shown in Figure 12 for the problem: DERIVE $((-(A+B))+A)+C=0$. In order to give the hint, the tutor suggested that the student use Theorem 2, the only item returned by the FEATURE TEST. In this case, it turned out to be a very good guess, because a solution is:

$$\begin{array}{l} \text{TH2 } (-A)+A=0 \\ \text{A:}:(A+C) \quad (1) \quad (-(A+C))+A+C=0 \\ \underline{1AL1} \quad (2) \quad (-(A+C)+A)+C=0 \end{array}$$

Two more solutions attempted by Lisa are shown in Figure 12. For the second proof, the dialogue routine wanted to suggest that the student begin with the AS axiom because the left-hand side of the problem is an instance of the left-hand side of the axiom. Lisa interpreted the suggestion to mean she should use the N axiom. She proceeded to do a very clumsy proof, in which she got stuck and was helped by the theorem-prover. She did the proof again, now using the AS axiom.

7. (Theorem 4) Lisa used an odd proof, although it was one the theorem-prover could easily complete. The second solution is the proof usually given for the theorem.

8. Theorem 5 is shown with two possible solutions: one when the student uses only the axioms, the other when Theorems 1 through 4 are available. A theorem-prover heuristic could have been pointed out to the student in this example. This is a problem with the zero element as the only atomic term. Introducing new variables into the problem, as Lisa did, usually does not help in the solution search.

10. (Theorem 6) The single proof obtained by the theorem-prover is first shown. After Lisa generated four lines, the theorem-prover was capable of extending the number of proofs it discovers in order to take into account work done by her. In a reverse idea, she had essentially given a hint to the theorem-prover.

11. The theorem-prover cannot solve this on its own, although it can complete Lisa's work. Heuristics suggest starting with either TH6 or Z.

14. Note that the routine to set up a conditional proof (CP) and, in this case, use the elimination MACRO, is not a top-level heuristic. The COLLECTOR throws it out in favor of the two-line solution shown.

15. The theorem-prover discovers that the recently proven theorem, Theorem 7, can be used to solve this problem. Using AE rule results in a shorter proof unless short forms of the axioms are used, cutting down the number of lines by eight. (See Chapter VI for an explanation of short forms of the axioms.)

16. Theorem 8. Again, the theorem-prover comes up with a proof using Theorem 7.

Discussion of the proofs in Figure 13 are continued in the next section where a comparison is made, using this set of solutions, with work done by human tutors.

4. A Study of Experienced Human Tutors of Elementary Algebra

An effort has been made in this chapter to demonstrate, with examples of dialogues and completions of partial solutions and with comparisons to other mechanical analysis techniques, that the theorem-prover/proof-analyzer extracts information from the student's work which can be used to generate a tutorial dialogue. From this information, the computer-tutor can give advice which encourages the student to utilize his own partial solutions to arrive at a complete one. In this sense, the tutor is responsive to the problem-solving difficulties experienced by the student. In its suggestions, the tutor attempts to capture a knowledge of what solutions can be produced from lines already appearing in the proof and thus an awareness of what solution the student may have been thinking of when he started to work.

Much of the development of the proof-analyzer was based on intuitions about how human teachers perform the same tutorial task as that presented to the computer. In order to see how good some of these intuitions were, and to compare the behavior of the computer with that of experienced human tutors, a special study was carried out. The aim of the study was to find out (a) what kinds of suggestions or hints a human tutor makes when confronted with the task of helping a student complete a proof, and (b) how the tutor decides what particular suggestions to make. Two results were expected: the computer's ability to tutor is comparable to that of the

human teacher, and the computer-based analysis technique simulates the way a human teacher decides how to direct a student to discover solutions.

The study was qualitative and intended to provide further information to determine how well the computer-tutor for mathematical logic works. This section will explain how the investigation was conducted, and then we examine individual results, comparing them to the computer's behavior. If some general conclusion could be made from this study, it seems fair to say that the computer performs better than most of the human tutors questioned. The reasons for this statement are cited below.

Each of the five subjects in the study were experienced in tutoring students of the Stanford logic-algebra course; two were teaching assistants for an elementary logic course at Stanford, the others have worked with children in grades 3 through 6. Only one subject was an aide, trained to be a proctor in a local elementary school; the others were graduate students at Stanford in either the School of Education or the Department of Philosophy.

The experimental procedure followed is outlined below.

1. Each subject was given three derivation problems to solve, using a reference sheet made up of Appendix I and a list of logical rules. The expressions to be proven were the algebraic identities listed in Appendix IV, and the purpose of the exercise was to refresh each subject's memory of the axiomatic system used in the Stanford program.

2. The subjects were read the instructions and given a warm-up problem. Both are shown in Appendix IV. The warm-up exercise was used only to ensure that the subject understood the task he was to perform.

3. Essentially, the task was to write down a comment which the human tutor would give to a student who needed help in completing the

partial solutions shown to the tutor. This was the same task seen in Figure 13 as carried out by the computer-tutor. The subjects were given eight partial solutions taken from those in Figure 13 (and reproduced in Appendix IV), and asked to (a) write down an initial comment, (b) try to express the reason for making that comment, and (c) note any further suggestions they would consider giving to the student.

The overall view formed of the human tutor was that their ability to give helpful hints depended on their ability to solve the problems themselves. The variety of hints seen mirrored that shown in Figure 12, either a direct answer or a suggestion to reexamine a line or compare a line with the problem expression. An attempt was made in Table 2 to tabulate or code the human tutor's comments and reasons given for making the comments. But the table does not tell the entire story without further description. The five subjects can be separated into two groups: subjects A, B, and E; and subjects C and D.

The first group clearly attempted to solve the problem on their own, and to then compare these solutions with lines already in the proof. To quote tutor B: "In this case, I sketched a proof out in my mind... Then I looked at the partial proof to see if the student was following that outline." This seems more like the stored-proofs method, where the stored proof is regenerated each time by the human tutor. However, in some cases where the subjects provided more than one possible comment, it could be inferred that the tutors discovered a few solutions and chose one for the initial discussion, depending on its relationship to the student's work. In some cases, the tutors seemed to go over the student's work, line by line, questioning the student's motives and trying to see where they

TABLE 2

ANALYSIS OF COMMENTS MADE BY THE HUMAN TUTORS*

Attributes	Tutors				
	A	B	C	D	E
Type of hint changed for different partial solutions	yes	yes	no	no	yes
Tutor insisted on using only one method for helping the student, regardless of the partial solution	no	no	yes	yes	no
Comment encouraged use of student lines					
a. used the last line to suggest shortest proof		2/8	2/8	3/8	2/8
b. used the last line to suggest proof which was not the shortest one	2/8		1/8	5/8	
c. used lines other than the last one	3/8	1/8			4/8
Tutor tried to get the student to start a new proof although student's work could be completed	2/7	1/7			
Comment contained an error or was confusing	1/8			2/8	2/8
Comment was irrelevant because student had already done the suggested work	1/8				1/8
Hint let student try several ideas; hint was not an answer	1/8	3/8			2/8
Tutor solved the derivation problem before deciding on a comment					
a. worked problem before looking at the student's work	5/8	4/8			6/8
b. worked problem while looking at the student's work	2/8	2/8			

*The notation, n/m, represents the number (n) of problems, out of m possible problems, that fitted the attribute description in column 1.

diverged from the solutions thought up by the tutors. In only a few cases did the tutors immediately notice a property of the student's lines to comment upon without consciously searching for a solution. This was true of partial solution number 6.

The approach used by tutors in the first group was flexible; they did not encourage use of a single strategy for all the problems and they varied in their decision to encourage the student to continue his own work rather than to start over again with a new solution idea. The new ideas suggested were ones also found by the theorem-prover, but, because the computer-tutor saw that the student's proof could be completed, the computer-tutor chose to encourage the student to complete his work first. Then the new solution strategies were suggested as alternative methods for solving the problem.

The irrelevant or erroneous hints given by these human tutors were due to two causes:

1. the tutor did not notice some of the lines produced by the student (something the computer would never do); or
2. the tutor figured out part of a solution, decided it was a good strategy without carrying it out completely when in fact it was not a feasible approach, and then based the suggestion on an erroneous proof idea.

Tutor E came up with a "metatheoretical comment" (his words) for partial solutions 2 and 3, namely, that "a sum of terms can always be reordered and reassociated to any extent by using CA and AS enough times." It was shown that the theorem-prover could always find completions for problems like 2 and 3; it was pointed out that the student needs to learn to recognize how to obtain lines having the

correct number of occurrences of the individual variables before mastering permutation and regrouping skills. The second group is made up of two tutors who clearly employed a single method for deciding what to say to the student, regardless of the problem and the work done by the student on the problem. The method used by tutor D was to start with the student's last proof line and try to manipulate it to obtain the problem expression. This tutor fell into a trap purposely set when the problems for the study were chosen; in five out of the eight problems, shorter solutions could be obtained by considering lines other than the last. For the most part, the computer-tutor would discover these solutions without considering manipulations of the last line. Tutor D claimed that his approach always captured what the student was trying to do. But the errors he made, even suggesting that the student complete a partial solution which could not be completed (number 7), indicate that the tutorial process is not so simple.

Tutor C's problems were based on an inability to solve the derivation problems. This tutor always tried to find the line of the partial proof most like the problem expression, and never gave a direct hint unless he could find a solution himself, which was true of only three out of the eight problems. He stated that his tutoring approach was to use trial and error methods and to keep asking students questions which could explain what they were trying to do. If he failed to help them, he had to call another tutor, or have the student skip the problem. He never considered asking a neighboring student who may have finished the problem because the students worked on identical problems and, "since the students are very competitive, could not be expected to help one another."

The students were not discouraged from "racing" through the curriculum; this unskilled tutor could not divert the students' attention from merely manipulating expressions to examining the properties of the axiomatic system, a point emphasized by the computer-tutor. Moreover, this proctor's emphasis on competition was detrimental. The students became more concerned with finishing a lesson than with the content of the lesson, and more concerned with secreting their knowledge than with using it to help their classmates.

The students were not discouraged from "racing" through the curriculum; this unskilled tutor could not divert the students' attention from merely manipulating expressions to examining the properties of the axiomatic system, a point emphasized by the computer-tutor. Moreover, this proctor's emphasis on competition was detrimental. The students became more concerned with finishing a lesson than with the content of the lesson, and more concerned with secreting their knowledge than with using it to help their classmates.

The students were not discouraged from "racing" through the curriculum; this unskilled tutor could not divert the students' attention from merely manipulating expressions to examining the properties of the axiomatic system, a point emphasized by the computer-tutor. Moreover, this proctor's emphasis on competition was detrimental. The students became more concerned with finishing a lesson than with the content of the lesson, and more concerned with secreting their knowledge than with using it to help their classmates.

The students were not discouraged from "racing" through the curriculum; this unskilled tutor could not divert the students' attention from merely manipulating expressions to examining the properties of the axiomatic system, a point emphasized by the computer-tutor. Moreover, this proctor's emphasis on competition was detrimental. The students became more concerned with finishing a lesson than with the content of the lesson, and more concerned with secreting their knowledge than with using it to help their classmates.

CHAPTER VI

A GENERALIZED INSTRUCTIONAL SYSTEM FOR TEACHING ELEMENTARY MATHEMATICAL LOGIC

The image cast in the introductory chapters of an ideal tutor was one of sensitivity to the educational needs of the student. So far, our attention, and that of the computer-tutor, has been directed to the less skillful students, those who cannot discover a proof of some true expression in an elementary axiomatic theory. Strategies the computer-tutor uses for helping the student find a complete proof to a particular problem, as well as diverse solutions for that problem, have been discussed in terms of ideas any instructional system for teaching skills in proof construction might well incorporate.

We now turn to a description of the particular instructional framework in which research on the theorem-prover of Chapter IV was carried out. In particular, we turn to the methods introduced for helping a student while he is working on a problem, although the emphasis is still on how to use theorem-provers to teach about theorem-proving. This chapter examines how the computer-tutor can provide the student with an instructional environment in which to freely explore the subject area and to experiment with his newly acquired skills in proof construction. The system does not know a particular theory, its axioms and rules for constructing proofs, but rather it is made up of general routines for processing first-order theories from the syntactic structure of those theories alone.

Among the methods for helping a student that have already been discussed were those by which a student can: (a) have the freedom to work on a problem without interference from a teacher, that is, try any solution path regardless of whether it approaches a successful proof; (b) communicate with a proof-checker to verify immediately whether his solutions are correct; (c) receive advice and comments on his work which take into account only the material that the student knows; and (d) be encouraged to discover diverse solutions for a given problem. Methods to be highlighted in subsequent sections of this chapter are:

1. the student can specify a first-order theory, and build his own command language with which to construct proofs;
2. he can make up his own problem examples;
3. he receives immediate feedback on errors; and
4. he can ask about previously learned material.

Aside from the theorem-prover which is employed as a proof-analyzer for purposes of helping the students complete a solution (Chapters III-V), the instructional system uses two other kinds of theorem-provers. With these theorem-provers, the student can instruct the program to move from one point to another without explicitly carrying out the mediating steps. The first is in the form of shortcut rules for axioms and proved theorems. The second kind of theorem-prover is one which the student can call on to verify that a given line is a valid inference from a set of previous lines and instances of axioms and theorems. This is the instruction we call SHOW. The instructional system is written in the LISP programming language (McCarthy et al., 1962) for a DEC PDP-10 computer at IMSSS. It has been used at Stanford to teach students in an elementary course in logic and

by seventh-graders from a junior high school in Palo Alto, California, who worked on the Finding-Axioms exercise first described in Chapter I. The possible uses for this system are varied. A graduate student is presently using it to study axiomatizations of elementary geometry; another, a curriculum for number theory. The program is suitable for problem-solving tasks of varying levels of complexity.

In the fall term, 1972, 45 students were enrolled in a five-unit course at Stanford taught by the instructional program to be described in this chapter. For our own developmental purposes, not all of the features for deriving rules were made available to this class.¹

1. Basic Features of the Instructional System

This chapter presents the basic capabilities of the instructional system, emphasizing the methods by which the system processes a command language which it knows only implicitly. By "implicitly" is meant that the system is merely a framework with some tools which the user, a teacher or a student, uses to construct axiomatic theories. The framework is a natural deduction treatment of first-order logic, and the tools are procedures for specifying the vocabulary of the axioms and for deriving the

¹The program used is available upon request from the author or appropriate staff of the Institute for Mathematical Studies in the Social Sciences.

rules of inference. Together they are able to support formalizations of the full predicate calculus with identity (and, with minor work, definite description). Also available is the machinery for specifying a curriculum, i.e., descriptions of problems to be solved, names of theorems to be proved, and, in general, questions to be answered. The curriculum might be a course, say, in elementary number theory or axiomatic geometry. Students follow a prescribed sequence of problems and questions, but they always have the option to interrupt it. They then can initiate their own problems, prove lemmas, or derive new rules of inference.

The interpretive tools of the program "understand" two languages. The first language, call it C, is the set of commands used in constructing proofs or derivations. It is defined in A, the second language. With A, the user is able to AXIOMATIZE a theory, i.e., to specify (a) the class of nonlogical constants; and (b) a class of axioms, which is a recursive class of formulas containing no nonlogical constants other than those in class (a). The names of axioms and the well-formed formulas derivable from the axioms (theorems or lemmas) are members of C. A also enables the user to derive rules of inference from axioms and from established theorems and lemmas (as explained in Section 2). These new rules are added to C. C is further augmented by the basic logical system; specifically this includes programmed code for:

1. rules governing substitution (name of axiom or theorem, PS procedure);
2. proof procedures: assuming a working premise (WP), and obtaining conditional proof (CP) or indirect proof (IP);

3. primitive rules of inference: modus ponens (AA), quantifier rules (US, UG, ES, EG), laws of identity (IDS, IDC), rules to permit the replacement of a well-formed formula (or term) by an equivalent formula (or term) within a line of a derivation or proof;

4. generalized interchange rules: these permit an interchange on the basis of an axiom or proved theorem which has the form of an identity relation or a material biconditional;

5. miscellaneous rules: obtain the initiative to request problems (INIT); delete the last line (DLL); review the proof (REVIEW); call on a theorem-prover to show that a line is a valid inference from previous lines, axioms, and theorems (SHOW or special "shortcut" forms of theorems and axioms); and help the user complete his proof (HELP). (This last command has already been extensively discussed.)

These rules will be elaborated on later as the discussion of the instructional system demands. Goldberg (1971) contains a detailed exposition on all the rules of the primitive command language as well as on the language for specifying the curriculum, either that prestored on a peripheral device and presented in a linear fashion to the student, or that defined at the time the user actually attempts to solve the problems. Some additional problem types are described here.

Figure 14 is a block diagram of that system. In it we see the language A represented by "AXIOMATIZE mode." The system has two parts: one to understand and process the requests from A to learn new commands and the other to evaluate these commands as well as those enumerated above. In this second part, the program takes on the role of a proof-checker, as represented in the diagram by "DERIVE mode." In this mode

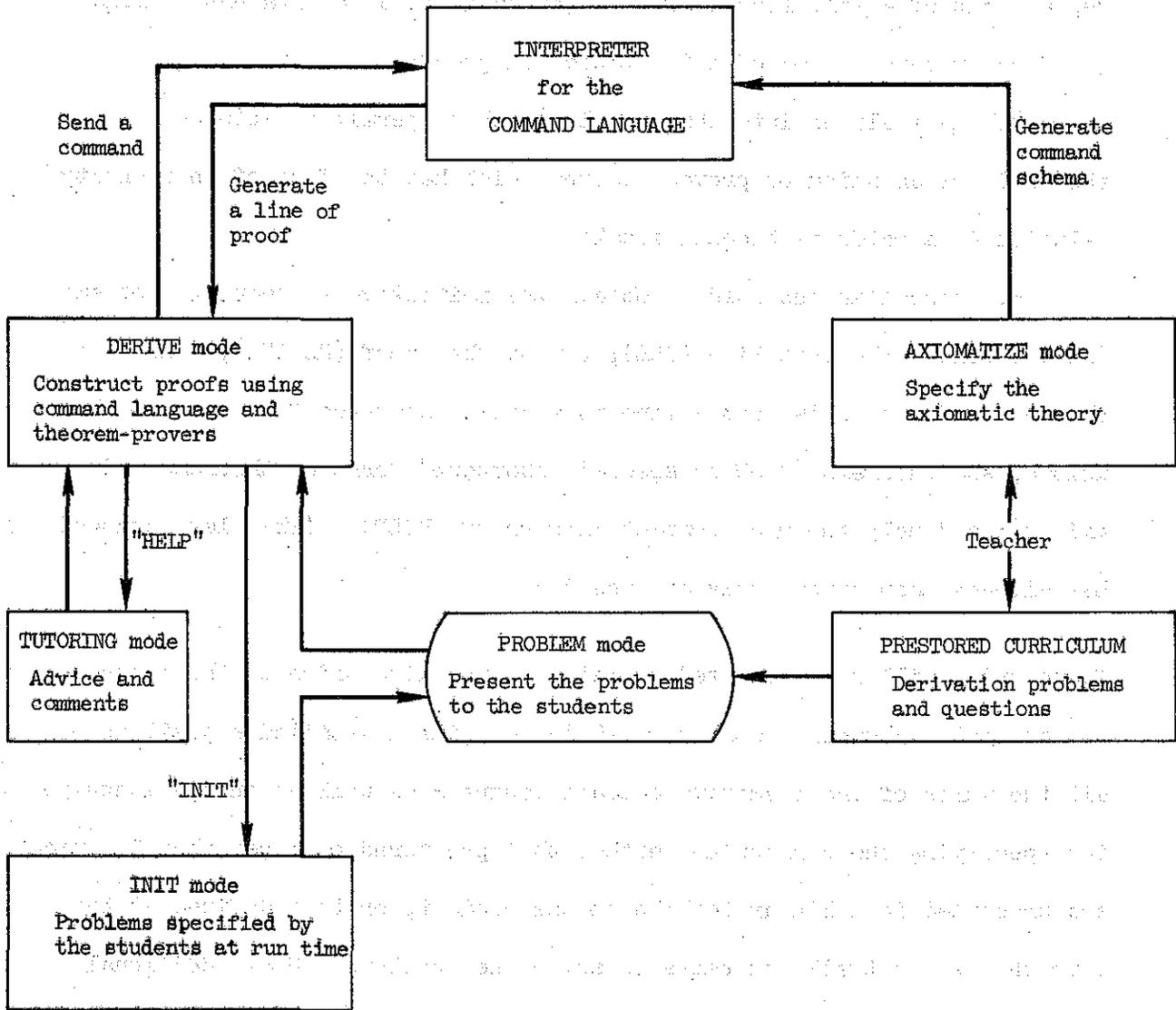


Fig. 14.--Block diagram of the instructional system.

of operation, the user or student constructs a proof or derivation of some true statement in the theory already axiomatized. Other types of problems fitted into the DERIVE mode will be outlined later. A derivation of a formula G in a theory is a sequence of steps or lines L_1, \dots, L_n , each justified in some way, such that L_n is G . In order to construct a proof or derivation, the student types a command, C_i , and expects the program to generate a new line, L_i . C_i is either a request for substitution instances of axioms or previously proved theorems, or a reference to a set of previous lines from which a new one is to be inferred. The proof-checker accepts the command, checks to see that the format is correct and that the intended application is proper. If the command is improperly formulated, the student receives a message to explain what error was committed. An example of this format for constructing proofs was given in Chapter I, Section 2; sample commands and proofs for expressions in a theory of Abelian groups were given in Chapter V. Although the words teacher and student are used to refer to the curriculum-writer and the problem-solver, respectively, the distinction is for convenience of this discussion only. As enumerated earlier in this chapter, as well as in Section 4 of Chapter I, the instructional system has applications beyond the usual "teacher-writes-lessons, student-takes-lessons" mode of instruction.

If the student is working on a prespecified curriculum, he can announce his intentions to make up his own problems by typing the command INIT. Although for the most part the student can use this instruction whenever he so chooses, an option is available to the teacher to block use of INIT. This feature is necessary if the teacher has designed problems which reference one another and which must be worked on consecutively. An

example of the INIT feature is shown in Figure 22 at the end of this chapter. The decision to let the student interrupt the teacher was made with the idea that the student could immediately test out any ideas he has about proof strategies; he would be able to ask the computer-tutor about formulas he thinks are derivable from the inference rules and would be able to carry out the derivation immediately. Other times when the student might avail himself of the INIT feature are (a) the teacher's problems are too difficult and the student wants to review easier problems he received earlier; (b) the teacher's problems are too easy and, to save himself from becoming bored, the student wants to make up more difficult ones; and (c) the teacher's problem can be more simply solved if the student proves and names a lemma which is used in proof constructions in the same manner as axioms and theorems. The student can also derive new rules of inference as explained in Section 2 of this chapter.

Given that the student is allowed to interrupt the teacher, when does the teacher interrupt the student? One situation where the teacher interrupts the student involves error messages on the syntax and application of rules in the command language. The decision to provide immediate feedback, that is, to interrupt the student whenever he makes an error in the form or use of a command, was explained in Chapter II, Section 2. Thus, each proof line generated is assured of being a valid inference from the set of premises and the rules of the logical system.

A second situation results from the lesson writer's ability to control the use of rules which can appear in the solution of a particular problem. The purpose of this feature of the instructional system is to give the teacher a vehicle for encouraging the student to try different proof strategies. By

preventing the student from using certain rules, or by demanding that the student use certain rules, the teacher can place constraints on the form of the solution.

The question here is whether or not to stop the student from requesting a command which has been disallowed, that is, to give the student an error message, such as `RULE IS NOT ALLOWED IN THIS PROBLEM`, whenever he types a prohibited command. If the computer-teacher interrupts the student with such a message, the student loses the main feature of the system: the ability to freely explore the command language. Perhaps the student was told not to use a command he has forgotten and so, reminded of its name, he decides to find out how it is used. If the computer-teacher refuses to generate proof lines or to give legitimate error messages about the command because the student was asked not to use it in the solution of the problem, then the student will not be able to relearn the command at a significant point in his instruction. Now suppose this command is crucial in the completed solution constructed by the student. The computer-teacher can only discover this fact after the student has finished the solution. So, our decision was to let the student request all the rules he has learned, freely exploring the language, and to let him complete his thoughts on how to solve the problem. Only when he has finished the problem outside the given restrictions will the computer-teacher interrupt the sequence of problems, correct the student, and ask him to redo the problem within the specified constraints.

What sorts of problems do we include in the instructional program (PROBLEM mode commands)?¹ As already stated, the main body of the system

¹Development of these problem types was based on ideas of Professor Patrick Suppes.

is a proof-checker and the basic problem is to construct a derivation (DERIVE or PROVE problems). In addition to using derivations for showing that an argument is valid, we can use it to show that an axiom is independent on other axioms or that an axiom is inconsistent with other axioms (derive the negation of the axiom in question). We can place other logical questions in terms of derivation problems. To show that an argument is invalid, we provide an interpretation in an algebraic domain and construct derivations to show each premise and the negation of the conclusion to be true. Axioms are consistent if there is at least one interpretation that makes each axiom true. So we construct derivations for this interpretation of each axiom. We can similarly show an axiom to be independent by providing an interpretation in which we construct derivations for the other axioms and for the negation of the axiom in question. Exercises on symbolizing English sentences can also be expressed in terms of a DERIVE problem. Given a sentence and suggested predicates and variables, the student provides his symbolization. If it is not identical to the stored answer, the student can construct a proof derivation to show his response is equivalent to the stored answer. In addition, we have Finding Axioms exercises (described in Chapter I), as well as simple questions expecting constructed responses in order to present some of the factual material.

Descriptions of the primitive codes for constructing proofs, as well as those codes automatically added to the command language when a theory is specified, are presented next. Section 3 will explore the

problem of providing error messages for invalid commands, especially for those commands learned by the instructional system. The problem of teaching the use of a command is dealt with in Section 4.

2. Formal Description of the Instructional System

Without burdening the reader with details of a fairly complicated program, this section describes data structures and syntax for the vocabulary and the command language, C. For the most part, the instructional system is a standard logical system,¹ and so the laws of identity and the sentential and quantifier rules will not be repeated here. They, as well as the other primitives of the command language C, and the commands in A, are summarized in Appendix V.

2.1 The Vocabulary

The instructional system is limited to purely symbolic languages, where each identifier is a string of one or more alphabetic or special characters. The data structure associates with every symbol, except the individual variables, a DEGREE, and attaches to every individual variable a TYPE label. The purpose of type-labeling is to restrict the use of individual variables in expressions, e.g., for purposes of restricting the range of values of a variable to a particular universe of discourse. Only these two elements, DEGREE and TYPE, are needed to build a table-driven

¹The logical system especially follows Kalish and Montague's system, optionally including their procedures for quantifier rules (Kalish and Montague, 1964).

parser (without checking for precedence relationships) to determine if a string constitutes a well-formed expression in the language. The output of the analysis routine is the expression rewritten as a pattern to facilitate manipulations of the expression.

A DEGREE is an ordered quadruple $\langle i m n p \rangle$ such that i and m are (atomic)¹ nonnegative integers, and n and p are either (a) the nonatomic element "(2)" if the corresponding operation symbol, logical constant, or predicate letter represents a binary infix relation, or (b) atomic nonnegative integers otherwise. Formulas and nonatomic terms are formed with the aid of symbols called "formula-makers" or "term-makers" according to the kind of expression they generate. In the associated DEGREE, i is either 0 if the symbol is a term-maker or 1 if it is a formula-maker; m , n , and p are respectively the number of immediately following variables, number of terms, and number of formulas that the constant demands. If $m \neq 0$, then the symbol is a binding operator. The general mathematical characterization of terms and formulas is embodied in (1) through (8) below.² The restriction and additions to the usual characterization formally handle what usually are considered informal notation conventions. Specification (8) prevents the expression from transcending first-order logic.

1. A variable is denoted either as an individual variable or as a sequence of three dots.

¹By "atomic" is meant that the element is a single, unparenthesized identifier. A nonatomic element, in LISP terminology, is an S-expression, a list of atoms and/or sublists.

²Characterizations (2) through (4) appear in Kalish and Montague (1964), p. 242.

2. Every variable and every number is a term.

3. If m , n , and p are nonnegative integers, δ is a constant of DEGREE $\langle 0 \ m \ n \ p \rangle$, $\alpha_1, \dots, \alpha_m$ are immediately following variables, β_1, \dots, β_n are terms, and $\gamma_1, \dots, \gamma_p$ are formulas, then

$$\delta \alpha_1 \dots \alpha_m \beta_1 \dots \beta_n \gamma_1 \dots \gamma_p$$

is a term.

4. If m , n , and p are nonnegative integers, δ is a constant of DEGREE $\langle 1 \ m \ n \ p \rangle$, $\alpha_1, \dots, \alpha_m$ are immediately following variables, β_1, \dots, β_n are terms, $\gamma_1, \dots, \gamma_p$ are formulas, then

$$\delta \alpha_1 \dots \alpha_m \beta_1 \dots \beta_n \gamma_1 \dots \gamma_p$$

is a formula.

5. If m is nonzero, n and p must be nonnegative integers.

Then the terms and formulas are defined as in 1-4.

6. If m is zero, then if n is (2), p must be zero, and the constant δ represents a binary infix relation such that if

(a) δ is of DEGREE $\langle 0 \ 0 \ (2) \ 0 \rangle$, and β_1, β_2 are terms, then $\beta_1 \delta \beta_2$ is a term;

(b) δ is of DEGREE $\langle 1 \ 0 \ (2) \ 0 \rangle$, and β_1, β_2 are terms, then $\beta_1 \delta \beta_2$ is a formula.

7. If m is zero, and if p is (2), then n must be zero and the constant δ represents a binary infix relation such that, if δ is of DEGREE $\langle 1 \ 0 \ 0 \ (2) \rangle$ and γ_1, γ_2 are formulas, then $\gamma_1 \delta \gamma_2$ is a formula.¹

¹The DEGREE $\langle 0 \ 0 \ 0 \ (2) \rangle$ should never occur in the vocabulary of a first-order system.

8. n is nonzero if and only if p is zero.

Table 3 is a list of the logical constants with their associated DEGREEs. Parentheses and commas serve as delimiters. Furthermore, symbols can be multiply defined. This introduces complications into the primitive routines, as will be pointed out later.

As the program stands at this writing, the routines for specifying the vocabulary require, on the user's part, knowledge of DEGREEs and TYPEs. This is too tedious for instructional use and will be replaced with routines for computing DEGREEs and TYPEs from definitions entered by the user. At no time is it intended that the students go through this process. So far, students have only been allowed to axiomatize systems for which the vocabulary has been specified in advance. The program is to be enlarged so that students will be able to specify their own vocabularies.

2.2 Proper Substitution: Instances of Axioms and Theorems

The instructional system includes routines which are realizations of the definitions of bondage, freedom, closure, proper substitution for free occurrences of variables, and proper substitution for predicate letters.

These routines are complicated by allowing multiple definitions. The procedures which search for occurrences of binding operators might, in fact, find an occurrence of a symbol identical with a binding operator, but one which is used as, say, a predicate. Therefore, the routines have the task of determining if the symbol is the initial element of the list L of a well-formed expression, and, if so, if it is being used as a binding

TABLE 3

THE LOGICAL CONSTANTS WITH THEIR ASSOCIATED DEGREES

Constant	Degree	Representation on standard teletype
\neg	$\langle 1\ 0\ 0\ 1 \rangle$	NOT
$\&$	$\langle 1\ 0\ 0\ (2) \rangle$	$\&$
\vee	$\langle 1\ 0\ 0\ (2) \rangle$	OR
\rightarrow	$\langle 1\ 0\ 0\ (2) \rangle$	\rightarrow
$=$	$\langle 1\ 0\ (2)\ 0 \rangle$	$=$
\forall	$\langle 1\ 1\ 0\ 1 \rangle$	A
\exists	$\langle 1\ 1\ 0\ 1 \rangle$	E
\leftrightarrow	$\langle 1\ 0\ 0\ (2) \rangle$	IFF

operator. For example, let A have the multiple degrees: $\langle 1\ 1\ 0\ 1 \rangle$ and $\langle 1\ 0\ 1\ 0 \rangle$, and let X be an individual variable. (Recall that if $m \neq 0$, then the constant is a binding operator.) Then the formula $(A\ X\ (A\ X))$ is well formed (and its prefix-list notation form is the same as the expression itself). The first occurrence of A is a binding operator, the second is a formula-maker. In the sublist $(A\ X)$, X is not, of course, bound by the formula-maker A . But if the DEGREE $\langle 1\ 1\ 0\ 0 \rangle$ were assigned to A , it would be:

In order to differentiate between the various cases, the quantifier routines reparse the list or the sublist under consideration. The parse is initially limited to a symbol table consisting entirely of degrees for binding operators. It is then expanded to the original symbol table in order to complete the analysis. If the list or sublist is well formed, the first element is necessarily a binding operator.

Axioms and previously proved theorems can be used in constructing a proof in either of two ways, each embodying an instantiation procedure: (a) simultaneous universal specification over a list of individual variables, denoted by σ ; and (b) proper substitution for individual variables and predicate letters. In the command language, the second procedure is obtained by calling on the primitive rule PS, while the first is requested by typing the name of the axiom, theorem, or lemma. Each kind of instantiation can be characterized in terms of its use in constructing proofs.

Name of an axiom or theorem. The command for the first kind of instantiation (simultaneous) is the name of the axiom or theorem. After the student types the name, the program types out the expression associated with the name. This expression is the value of the axiom, with all universal

quantifiers whose scope is the entire formula suppressed if the teacher so chooses. The program then presents each distinct free variable V_i in the expression, and the student types a well-formed term T_i to be paired with V_i . The list of pairs becomes the substitution σ which, when applied to the pattern for the axiom or theorem, generates the line of the proof.

As an example, let CA be the Commute Addition axiom for an additive group. To obtain an instance of CA, the student types:

CA $A+B=B+A$
 A: C+A
 B: B (1) $(C+A)+B=B+(C+A)$

Here, the substitution σ is $\langle A, (+ C A) \rangle, \langle B, B \rangle$. The new line is then typed to the student.

PS. PS is the command name of the second kind of instantiation procedure. It is a precise realization of proper substitution for free occurrences of variables and predicate letters.¹ In the PS procedure, the value of the axiom or theorem, with all quantifiers retained, is referenced. The student types: PS: <name of axiom, theorem, or lemma>; then the student must give a sequence of substitutions by typing either the pair

(a) $\langle \text{variable} \rangle : \langle \text{well-formed term} \rangle$, to obtain the substitution of the free occurrences of $\langle \text{variable} \rangle$ by $\langle \text{well-formed term} \rangle$; or

¹The specifications of the vocabulary include a list of non-substitutable parameters which are required by the PS procedure. In the example the list is (W X Y Z).

(b) <predicate letter>: <well-formed formula>.

One or more pairs can be typed. Each substitution request is applied, in order, to the result of the previous substitution in the sequence, starting with the axiom or theorem itself. Note that if no pair is typed, the result obtained is the axiom or theorem itself.

As an example, take the theorem TH10 to be $(\forall X(F X \rightarrow P)) \leftrightarrow (\exists X(F X) \rightarrow P)$. F is a one-place predicate and P is a zero-place predicate. Then the command sequence of three substitution pairs for generating line (n) might look like the following.¹

PS: TH10

:: P: $\forall Z(G(Y, Z))$

:: F: $F(A, W)$

:: W: Y

:: (n) $\forall X(F(X, Y) \rightarrow \forall Z G(Y, Z)) \leftrightarrow$

$((\exists X F(X, Y)) \rightarrow \forall Z G(Y, Z))$

Another example appears in Figure 16.

2.3 The Command Language

A list of primitive rules in the command language appears in the introduction to this chapter. Several of these rules will be elaborated on here. Figure 15 establishes the syntax for a well-formed command. In general, a command name is preceded by a sequence of line references and

¹The specifications of the vocabulary include a list of non-substitutable parameters which are required by the PS procedure. In the example this list is (W X Y Z).

```

<command> ::= <line references> <command name> <occurrence references> :
            <other information> <enter key> |
            <line references> <command name> <occurrence references>
            <enter key>
<line references>
            ::= <line number> | <sequence of line numbers> |  $\phi$ 
<sequence of line numbers>
            ::= <sequence of line numbers> . <line number> | <line number>
<occurrence references>
            ::= <occurrence number> | <sequence of occurrence numbers> |  $\phi$ 
<sequence of occurrence numbers>
            ::= <sequence of occurrence numbers> . <occurrence number> |
            <occurrence number>
<line number> ::= nonnegative integer--the number of a line already generated
                and not within a closed subsidiary derivation
<occurrence number>
            ::= nonnegative integer
<command name> ::= <axiom name> | TH<nonnegative integer> | PS | WP | GEN |
                <proof procedure> | <primitive rule of inference>
                <derived rule of inference> | <miscellaneous codes>
<proof procedure>
            ::= CP | IP
<primitive rule of inference>
            ::= RE | RQ | AA | IDC | IDS | ES | US | EG | UG
<miscellaneous codes>
            ::= DLL | ENT | INIT | SHOW | HELP | REVIEW
<derived rule of inference>
            ::= string of alphabetic characters other than those above
<other information>
            ::= <well-formed formula> | <name> | <variable>

```

Fig. 15.--Syntax for the Command Language. Names within metabrackets are nonterminal symbols, all others are terminal symbols or denote terminal symbols. The vertical bar is read "OR".

followed by an occurrence reference, i.e., the number (counting left to right) of the exact occurrence of some symbol, expression, or instance of an expression in one of the lines referenced.

The Replace Equals Rule (RE). RE (Inference Rule 2 of Chapter IV) permits the replacement of a well-formed term by an equivalent term within a line of the derivation. Figure 16 is an example of a complicated proof, constructed by using some rules of logic which are listed in Table 4. It also shows that the RE rule is derivable in the basic instructional system.

Generalized Interchange Rules. The interpreter accepts special commands as shortcuts for using an axiom or theorem if that axiom or theorem is of the form $\alpha = \beta$ or $\phi \leftrightarrow \psi$, where α, β are terms, and ϕ, ψ are formulas. The interpretation is based on the RE and RQ (replace an equivalent formula for a formula in the line) rules.

Suppose a derivation contains the line

$$(1) \quad A+(B+0)=(A+((-C)+0))+0 .$$

As frequently occurs in proof constructions requiring pattern manipulations, the student will want to alter a term or a formula contained in a line of the derivation. In the example above, he may want to replace the occurrence of the term $(-C)+0$ with the term $(-C)$. This requires an application of the Z axiom, followed by the RE rule. The next steps of the derivation would be:

$$\underline{Z} \quad A+0=A$$

$$A:: \underline{-C} \quad (2) \quad (-C)+0=-C$$

$$\underline{1.2RE1} \quad (3) \quad A+(B+0)=(A+(-C))+0$$

TABLE 4

RULES OF LOGIC USED IN FIGURE 16*

Command Name	Example
AA Affirm the Antecedent	(1) $P \rightarrow Q$ (2) P <u>1.2AA</u> (3) Q
IDC Law of Identity	(4) $Y = (Z \times Y) - Y$ <u>1IDC1.2</u> <u>::X:Y</u> (5) $\forall X(X = Y \rightarrow (X = (Z \times Y) - Y))$
CE Commute Equal Sign	<u>4CE1</u> (6) $(Z \times Y) - Y = Y$
WP Working Premise	<u>WP</u> (7) Q
CP Conditional Proof	<u>7.2CP</u> (8) $Q \rightarrow P$
FC Form the Conjunction	<u>1.8FC</u> (9) $(P \rightarrow Q) \& (Q \rightarrow P)$
LC Left Conjunct	<u>9LC</u> (10) $P \rightarrow Q$
LB Law of Biconditional	<u>9LB</u> (11) $P \leftrightarrow Q$
US Universal Specification	<u>5US</u> <u>X::Z</u> (12) $Z = Y \rightarrow (Z = (Z \times Z) - Y)$
UG Universal Generalization	(13) $V = Y$ <u>13UG:V</u> (14) $\forall V (V = Y)$

*A more detailed presentation of these rules appears in Appendix V.

TYPE A DERIVE, PROVE OR RULE COMMAND

```

:*PROVE: A X(A Y((X=Y)->(N X IFF N Y)))$
PROVE      A X(A Y((X = Y) ->(N X IFF N Y)))

:*WPS      (1)      *X=Y$

:*WPS      (2)      *N(X)$

:*2IDC1$
::*Y:X$    (3)      A Y((Y = X) -> N Y)

:*3US$
:Y::*Y$    (4)      (Y = X)-> N Y

:*1CE1$    (5)      Y = X

:*4.5AA$   (6)      N Y

:*2.6CP$   (7)      N X-> N Y

:*WPS      (8)      *N(Y)$

:*8IDC1$
::*X:Y$    (9)      A X((X = Y) -> N X)

:*9US$
:X::*X$    (10)     (X = Y)-> N X

:*10.1AA$  (11)     N X

:*8.11CP$  (12)     N Y-> N X

:*7.12FC$  (13)     (N X -> N Y)&(N Y -> N X)

:*13LB$    (14)     N X IFF N Y

:*1.14CP$  (15)     (X = Y)->(N X IFF N Y)

:*15UG:Y$  (16)     A Y((X = Y) ->(N X IFF N Y))

:*16UG:X$  (17)     A X(A Y((X = Y) ->(N X IFF N Y)))

CORRECT...
NAME: *LEIBNIZ$

```

Fig. 16.--Proof of Euclid's Postulate. We first prove Leibniz' Indiscernability of Identicals and give it the name LEIBNIZ. A form of LEIBNIZ can be taken as definitive of identity within second-order logic. The second derivation is a proof of Euclid's Postulate. THA is the reflexivity of the identity. Note the use of the PS procedure.

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: A Z (A Y((Z=Y)-> ((F Z)=(F Y))))\$
 DERIVE A Z(A Y((Z = Y) ->(F Z = F Y)))

:*WPS (1) *Z=Y\$

:*THAS A X(X = X)
 X:*F(Z)\$ (2) F Z= F Z

:*PS:LEIBNIZ\$ A X(A Y((X = Y) ->(N X IFF N Y)))
 :*N:(F(Z)=F(A))\$
 :*\$

(3) A X(A Y((X = Y) ->((F Z = F X) IFF(F Z = F Y))))

:*3US\$
 X:*Z\$ (4) A Y((Z = Y) ->((F Z = F Z) IFF(F Z = F Y)))

:*4US\$
 Y:*Y\$ (5) (Z = Y)->((F Z = F Z) IFF(F Z = F Y))

:*5.1AAS (6) (F Z = F Z)IFF(F Z = F Y)

:*6LB\$ (7) ((F Z = F Z) ->(F Z = F Y))&((F Z = F Y) ->(F Z = F Z))

:*7LC\$ (8) (F Z = F Z)->(F Z = F Y)

:*8.2AAS (9) F Z= F Y

:*1.9CPS (10) (Z = Y)->(F Z = F Y)

:*10UG:Y\$ (11) A Y((Z = Y) ->(F Z = F Y))

:*11UG:Z\$ (12) A Z(A Y((Z = Y) ->(F Z = F Y)))

CORRECT...

Fig. 16, continued.

A shortcut method for obtaining the formula on line 3 is to use the command: 1 Z 2. Since the command name is an axiom or established theorem, and since the formula associated with the name is either an identity or a material biconditional, the program automatically carries out the two-step procedure shown above. Note that the program, not the user, determines the substitution sequence for obtaining the proper instantiation of the Z axiom.

To summarize, the shortcut use of axioms and theorems is a method by which the axioms and theorems are used as straightforward rewrite rules (much the way rewrite rules are used in the FDS and GPS problem-solving programs). Application of replace equals is eliminated and the student no longer has to specify the sequence of substitution pairs.

Short forms are automatically provided for all axioms and theorems, specified or proven by the user of the system, if the formula associated with the axiom or theorem is of the form $\alpha = \beta$ or $\phi \leftrightarrow \psi$, where α, β are terms, and ϕ, ψ are formulas. In using the shortcut, the student must, of course, determine the correct occurrence reference.

Derived Rules of Inference. Methods have been provided for specifying the formal system. Axioms when specified, and lemmas and theorems when proved, are automatically entered into the command language with instantiation procedures and with shortcut applications (as explained above). An even more flexible framework is achieved by providing a means for deriving new rules of inference. To every theorem of logic there is a corresponding derived rule; they are indispensable from the standpoint of decreasing the number of steps necessary for a proof or derivation. There are two kinds of derived rules. One is merely a commuted form of the shortcut commands

presented in the previous part of this section. Here, instead of replacing an instance of α by β (or ϕ by ψ), the program searches for the proper occurrence of β (or ψ) and replaces it with the corresponding instance of α (or ϕ). As an example, take the axiom AS, associativity axiom for addition: $(A + B) + C = A + (B + C)$. The short form is usually called AR (associate right over addition). A new rule, AL (associate left over addition) is obtainable by requesting a derived rule of inference based on AS. Associating left means to replace the instance of the schema $(A + (B + C))$ by the corresponding instance of $(A + B) + C$ in the proof line. Observe the derivation:

$$\begin{array}{ll}
 \underline{\text{AS}} & (A + B) + C = A + (B + C) \\
 \text{A:} & 4 \\
 \text{B:} & 3 \\
 \text{C:} & 2 \\
 & (1) \quad (4 + 3) + 2 = 4 + (3 + 2) \\
 \underline{\text{LAR1}} & (2) \quad 4 + (3 + 2) = 4 + (3 + 2) \\
 \underline{\text{2AL2}} & (3) \quad 4 + (3 + 2) = (4 + 3) + 2 \\
 \underline{\text{2CA3}} & (4) \quad 4 + (3 + 2) = (3 + 4) + 2
 \end{array}$$

The AR command brings about a regrouping of the parentheses, that is, $(4 + 3) + 2$ was replaced by $4 + (3 + 2)$. AL2 requests that the second occurrence of $4 + (3 + 2)$ be replaced by $(4 + 3) + 2$. (The result of the two commands is the same as the result of the command 1CEL.)

The use of the second kind of derived rule is effectively an iteration of proper substitution, applications of the rule to form a conjunction (FC), and modus ponens (AA). The theorem is transformed into a conditional statement P such that the consequent of P is not conditional. This makes use of the Deduction Theorem (see Mendelson, 1964, p. 61). The premises, the patterns of each conjunct of the antecedent of P, are

patterns for the lines which the new rule will reference. The number of required line references is equal to the number of premises. The result of using this new rule of inference is the proper instance of the pattern of the consequent of P. How this type of command is processed is discussed in Section 3 on error messages because the process corresponds to the method used for computing error messages for the derived rules.

An example will help illustrate the algorithm. "Form a Conjunction" (FC) is a rule of logic which lets us combine two lines of a derivation or proof into a conjunction. Given the formulas P and Q, we can infer the expression (P & Q). The theorem from which FC is derived is: $P \rightarrow (Q \rightarrow (P \& Q))$. By the Deduction Theorem, $P \rightarrow (Q \rightarrow (P \& Q))$ is equivalent to $(P \& Q) \rightarrow (P \& Q)$. The two premises, the conjuncts of the antecedent, are P and Q. The pattern of the consequent, (P & Q), is stored as the conclusion (CONCL) of the rule. Now, suppose a derivation has lines

$$(1) \quad A = B$$

$$(2) \quad A + B = C$$

and the command is 1.2FC. Then, for $P[\sigma]$ to be (A=B), σ must be $\langle\langle P, (A = B) \rangle\rangle$. Similarly, for line 2, $Q[\sigma']$ to be (A + B = C), σ' is $\langle Q, (A + B = C) \rangle$. Combining σ and σ' gives the substitution set which, when applied to the value of CONCL, generates the new line (A = B) & (A + B = C). The command has the correct number of reference lines and no occurrence numbers. Also, there is no pair $\langle V_i, T_i \rangle$ in σ such that V_i is also in σ' but, for the pair $\langle V_i, T_j \rangle$ in σ' , $T_i \neq T_j$.

There are several other attributes which may optionally appear on the property list of a rule. The fourth option must be entered by direct

editing of the property list associated with the rule name. They are:

1. REQ and TYPE. A rule may require the student to enter a particular type of well-formed term which is paired with a free variable belonging to the value of CONCL. In deriving a new rule from an expression containing a free variable, the program assumes that the variable can be replaced by any well-formed term. An example is the add equals rule (AE), which may be obtained from the open formula: $\forall A \forall B(A = B \rightarrow A + C = B + C)$. The property list of AE contains

```

PREMISE ((A=B))
CONCL (A + REQ = B + REQ)
NOP 1
REQ T (for "term")
TYPE ALGEBRA

```

where REQ denotes the free variable C in the value of CONCL. If REQ is to be replaced by a term, then the term must have the same type label ((as that specified under the attribute TYPE.

2. OR. The main connective of the single premise of AE can also be one of the inequality signs. The options, <, >, and =, could be specified by listing each separately as a premise on the PREMISE list beginning with the atom OR. Such a list states that the rule has more than one form which can be indicated by the same rule name. The value of CONCL must be a list of patterns corresponding to each optional set of premises. Whichever set of premises matches the referenced lines generates the desired substitution, σ , and then σ is applied to the corresponding item on CONCL. For AE the change is:

```

PREMISE (OR ((A = B))((A < B))((A > B)))
CONCL ((A + REQ = B + REQ)(A + REQ < B + REQ)(A + REQ > B + REQ)) .

```

3. %%. As long as the only difference between sets of premises is the main connective, use %%, a special atom, to indicate the optional

list of main connectives. Then $%%$ is stored on the property list as the attribute whose value is the list of main connectives. AE is now

```

PREMISE ((A %% B))
NOP      1
%%      (= < >)
CONCL   ((A + REQ) %% (B + REQ))
REQ     T
TYPE    ALGEBRA

```

4. RESTRICT. This attribute is used to cover the situations when special restrictions on the use of a rule must be specified. The value of RESTRICT is an executable LISP lambda-expression. As an example, take the divide equals rule (DE), in which the user is not allowed to divide by zero. The property list of DE might be:

```

PREMISE ((A = B))
NOP      1
REQ      T
TYPE     ALGEBRA
CONCL    (A/REQ = B/REQ)
RESTRICT (COND ((EQ REQ 0)(ERR*(QUOTE "YOU MAY NOT DIVIDE BY ZERO.")))
          (T T))

```

That is, RESTRICT tests to see if the term requested, REQ, equals zero, and, if so, prints an error message. Note that if the main connective of the premise were changed to $%%$ to include the inequalities, RESTRICT would have to be extended to account for negative values of REQ.

To use the shortcut forms for theorems in the form of a conditional, $(\phi \rightarrow \psi)$, the user references the name of the theorem and an ordered list of proof lines whose conjunction is a substitution instance of the antecedent of the formula. The program can then generate the corresponding instance of the consequent. If substitutable variables occurring in the consequent do

(...)

not occur in the antecedent, the program will ask the student to complete the desired substitution.

As a simple example, take the following derivation:

PROVE $S(X,Y,W,T) = S(Y,X,W,T)$
PS:AXA (1) $S(X,Y,Y,X)=X$
1THA
 U: W
 R: T (2) $S(X,Y,W,T)=S(Y,X,W,T)$

In the above proof, AXA is the axiom, $S(X,Y,Y,X)=X$, and THA is a theorem, $S(X,Y,W,T)=T \rightarrow S(X,Y,U,R)=S(W,T,U,R)$. When the user typed 1THA, the computer determined whether line 1 was a substitution instance of the antecedent of the theorem. If it had not been an instance of the antecedent, an error message would have been typed by the computer. The computer then requested the user to type a term to be paired with each variable that appears in the consequent but not in the antecedent of THA (i.e., U and R). Then the computer generated the appropriate instance of the consequent of THA as the new line of the proof.

A longer proof, to which we shall refer again in the next section, appears below. B is a three-place predicate meaning "betweenness," where $B(X,Y,Z)$ states that point Y stands between points X and Z on the straight line they define. Suppose we have two axioms

AXA: $((\text{NOT } X=Y) \& B(W,X,Y)) \& B(X,Y,Z) \rightarrow B(W,X,Z)$, and

AXB: $B(X,Y,Z) \rightarrow B(Z,Y,X)$.

Then the proof looks like:

PROVE : (((NOT Y=Z) & B(X,Y,Z)) & B(Y,Z,W)) → B(X,Z,W)

WP (1) (((NOT Y=Z) & B(X,Y,Z)) & B(Y,Z,W))

LLC (2) (NOT Y=Z) & B(X,Y,Z)

2LC (3) NOT Y=Z

2RC (4) B(X,Y,Z)

1RC (5) B(Y,Z,W)

3CE1 (6) NOT Z=Y

4AXB (7) B(Z,Y,X)

5AXB (8) B(W,Z,Y)

6.8.7AXA (9) B(W,Z,X)

9AXB (10) B(X,Z,W)

1.10CP (11) (((NOT Y=Z) & B(X,Y,Z)) & B(Y,Z,W)) → B(X,Z,W)

The SHOW Command. The automatic availability of short cuts for all axioms and proved theorems, and the ability to derive new rules, means that the program provides the student with miniature theorem-provers. Their purpose is to cut down the number of proof-steps the student must produce and to eliminate routine steps from the student's explicit focus of concern. As students of mathematics are introduced to more complex problems, they tend to produce less rigorous proofs, yet ones clearly valid to the trained eye. A simulation of such a "trained eye" can be realized by giving the student access to an instruction by which he can communicate with a mechanical theorem-prover.¹ Such a theorem-prover need not do the work

¹It is important at this point not to confuse this discussion on a mechanical theorem-prover with that already given on the proof-analyzer. The two theorem-provers are considered distinct for purposes of this chapter.

the same way the student does; its purpose is to verify whether or not a desired new line is in fact a (trivial) deduction from the set of formulas cited by the student. Before revealing how the simulation was carried out and whether initial results seem promising, an illustration of the idea is offered.

The problem rendered on page 229 and again in Figure 17 was given to a number of college students. Those who solved it came up with proofs ranging somewhere between seventeen and seventy lines. The length of the proof shown earlier was reduced further by using a SHOW command to skip sequences and commands for forming and separating conjunctions.

The SHOW command calls on a mechanical theorem-prover to verify that a line typed in by the student is a valid inference. The student, in the illustration, makes the claim that line (4) is a valid inference from lines (1) and (3), and from an instance of axiom AXB. The theorem-prover decides the claim is justified, and the new line is accepted. If the formula on line (4) could not be proved by the theorem-prover, it does not necessarily follow that it is indeed not derivable. This only suggests that the formula does not answer some constraint within which the theorem-prover operates, perhaps some criterion for triviality. Note that the student, not the theorem-prover, is required to specify the proper substitution sequence for the axiom. Contrary to the usual aims of development and use of mechanical theorem-provers, the interest here is not in having the prover do complex analyses. That is left to the student. The theorem-prover is only to be used to cut out trivial manipulations; the determination of proper instantiation often is not trivial.

```

PROVE      (((NOT Y=Z)& B(X,Y,Z))& B(Y,Z,W))-> B(X,Z,W)

:*WP$     (1)      ((NOT Y=Z)& B(X,Y,Z))& B(Y,Z,W)$
:*1RC$    (2)      B(Y,Z,W)
:*2AXB$   (3)      B(W,Z,Y)
:*SHOW$   (4)      ((NOT Z=Y)& B(W,Z,Y))& B(Z,Y,X)$
FROM LINES OF THE DERIVATION?
:.*1,3$
FROM AXIOMS OR THEOREMS?
:.*AXB$
OK? *Y$
LINE 4 IS OK

:*4AXA$   (5)      B(W,Z,X)
:*5AXB$   (6)      B(X,Z,W)

:*1.6CP$  (7)      (((NOT Y=Z)& B(X,Y,Z))& B(Y,Z,W))-> B(X,Z,W)

```

Fig. 17.--A proof using the SHOW command.

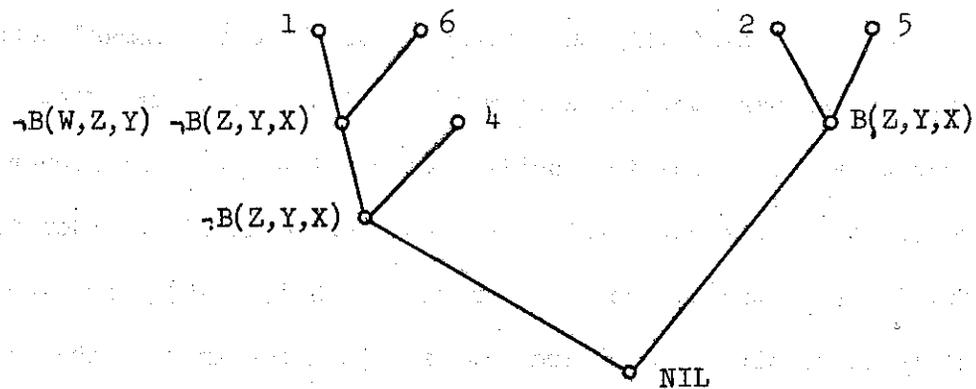
To provide a SHOW command, the instructional system was connected to a theorem-prover¹ based upon the Resolution Principle (Robinson, 1965), a refutation scheme by which a statement is proved true by showing that the conjunction of its negation with all known true statements in the logical system at hand is inconsistent. Successful interface of the prover with the instructional system only requires computation of the set of clauses, statements in prenex normal form, on which the theorem-prover performs its resolutions. The set of clauses for the above problem is exhibited in Figure 18 as axioms 1-5, the negation of the problem statement as axiom 6, and a solution is given as the tree of resolvents.

Although basically this approach to the SHOW command works, numerous difficulties were met in experimental endeavors. Some were apparently due to the interactive nature of the theorem-prover being used, and to the refinement strategies, routines required in order to improve the efficiency of the search for a proof (Luckham, 1970). For the most part, published results seem to indicate a high dependency on the part of the program for human intervention in order to add new lemmas or to change the set of strategies. This need for intervention, while often suitable for research, is not acceptable for the teaching purposes outlined above. One stumbling block seems to be the determination of those settings of the search strategies which will, in fact, permit a solution to be found, within the time and space constraints set by the teacher. Since it is not desirable to let the students interact with the theorem-prover, one set of parameters for the strategies for a given group of problems has to be used,

¹The program was written by John Allen as reported in Allen and Luckham (1970).

Axioms

1. $\neg E(Y,Z)$
2. $B(X,Y,Z)$
3. $B(Y,Z,W)$
4. $B(W,Z,Y)$
5. $\neg B(X,Y,Z) \quad B(Z,Y,X)$
6. $E(Z,Y) \quad \neg B(W,Z,Y) \quad \neg B(Z,Y,X)$

Resolution TreeFig. 18. Proof by Resolution Principle

anticipating the kinds of problems the students will try. Optionally, internal routines could compute the parameters.

The situation is not as problematic as the previous comments indicate. We are only interested in carrying out ground resolution, examining perhaps four or five initial clauses. Our definition of "trivial" also simplifies the problem. "Trivial" is determined by the number of clauses generated by the theorem-prover in its search for a solution. It is also the amount of time the prover must spend in searching for that solution. Thus an acceptable response to the student from the SHOW command is:

THE DEDUCTION IS NONTRIVIAL.

PLEASE CARRY OUT MORE DETAILS YOURSELF.

3. Error-analysis for Commands--Syntax and Application

This section demonstrates how error messages are automatically provided when either the syntactic formulation or the intended application of a command is invalid. The question of how to use the same routines described here for reteaching the use of a command is the topic of Section 4. Both sections explain formally how the instructional system provides help to the student while he is working on a problem and without his directly requesting such help.

The benefits of providing concise and explanatory error messages in an instructional system are clear. Such messages serve to teach the student how to use the commands by promptly pointing out his mistakes. Correcting syntax errors is, in most cases, immediate. The student should be able to construct the correct format for a command by iterative corrections of each error the program announces. The need for such strict command formats may be obscure, but lack of free form is not solely for the computer's convenience. The

students are taught a language with which to discuss derivations and asked to obey the rules of that language. This is definitely within the domain of mathematical training.

Application errors occur when incorrect relationships between and within lines of the derivation are detected. The error messages explain these structural relationships and thus help clarify the valid use of a command. Although each rule is not explicitly built into the system, we can construct application error messages by using the same routines which carry out the rule and by referencing the names of the symbols belonging to the vocabulary. The recursive analysis is explained later in this section.

Now, suppose a student requests help in completing a proof and the tutor suggests, say, that he try to form the antecedent of line i and apply modus ponens. Perhaps the antecedent of line i is a conjunction. If the student has not mastered the FC or the AA rule, the advice is useless. Suppose a problem in the curriculum is:

YOU'LL HAVE TO USE AR TWICE TO GET THIS ONE.

DERIVE $A + (3 + 4) = B + (3 + 4)$

(1) $(A + 3) + 4 = (B + 3) + 4$

and the student wonders "what's AR?" Without (automatic) error messages and teaching sequences, the student cannot turn to the computer-tutor for an answer. Concise error messages should actually be a means of reteaching the commands. Furthermore, the same routines used to form the error messages can be used to teach the commands upon the student's request (Section 4), or when the computer-tutor realizes that the student does not know the command.

The syntax for a command was presented in Figure 15. We shall not dwell on how to match inserts with templates for error messages on invalid syntax. There are several examples in Figures 12, 19, 20, and 21. A command which

calls for the application of a derived rule of inference is carried out by a matching process which attempts to determine whether two patterns are substitution instances of one another. Recall that three items appear on the property list of the name of a rule: a list of patterns for each premise, P_i , for $i \geq 1$; a pattern for the conclusion, R ; and the number of premises, NOP . The matching compares each line referenced, L_i , with each premise, P_i , for $i = 1, \dots, NOP$, in order to determine whether L_i is a substitution instance of P_i and whether that substitution is consistent with the ones already examined (i.e., for all $1 \leq j < i$). If this process succeeds, a substitution sequence is obtained which, when applied to R , generates a new line of the proof or derivation. Thus, the analysis of a derived rule depends on the ability to determine whether or not the pattern for an expression is a substitution instance of the pattern for a premise of the rule.

Optional attributes on the property list of the rule name, REQ , $TYPE$, OR , and $RESTRICT$, must also be considered. These complicate the matching process. The OR option is particularly cumbersome since, if the match does not succeed for one set of premises, the search must continue trying to find a match, at the same time remembering the reason why the earlier set did not match. In the sample output (Figure 19), the error message for the law of the biconditional, LB , demonstrates "remembering" multiple reasons for match failures.

There are four cases in which the matching process may fail: (a) L_i is not the correct statement type (main connective improper), (b) a sublist of L_i is not the correct statement type, (c) L_i is not an instance of P_i , or (d) there exists a V_j identical to V_k , but the corresponding T_j is not equal to T_k . Matching is a recursive routine which, as it compares each element

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: ((A=B) & (B=C))-> (A=C)\$

DERIVE ((A = B) &(B = C))->(A = C)

:*WPS (1) *A=BS

:*1AES

:*PS

NOT A WELL-FORMED TERM

:*CS (2) (A + C) = (B + C)

:*1DLLS

:*WPS (1) *(A=B)&(B=C)\$

:*1AES

THE MAIN CONNECTIVE OF LINE 1 MUST BE ONE OF: = < >

:*1LCS (2) A = B

:*1RCS (3) B = C

:*3.2.<.>RE1\$

THERE IS NOT A FREE OCCURRENCE OF A IN LINE 3

:*2.3RE1\$ (4) A = C

:*1.4CPS (5) ((A = B) &(B = C))->(A = C)

CORRECT...

TYPE A DERIVE, PROVE OR RULE COMMAND

:*FINS

T

*

Fig. 19.--Sample error messages.

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: P->R\$

DERIVE P -> R

:*P\$ (1) *(P& Q)-> R\$

:*P\$ (2) *P -> Q\$

:*1WP\$

WP REQUIRES 0 LINE NUMBERS

:*WP1\$

WP REQUIRES 0 OCCURRENCE NUMBERS

:*WPS

(3)

***R\$**

:*2DLL\$

**LINE 2 IS A PREMISE AND CANNOT BE DELETED.
EVERY LINE AFTER IT HAS BEEN DELETED**

:*WPS

(3)

***P\$**

:*3.2AAS

LINE 3 MUST BE A CONDITIONAL

:*2.3AAS

(4)

Q

:*3.4FCS

(5)

P & Q

:*5.2\$

NO COMMAND REQUESTED?

:*5.3CP\$

LINE 5 IS NOT A WORKING PREMISE

:*3.5CPP\$

CPP IS NOT A RULE

:*3.5CP\$ (6) P ->(P & Q)

:*1.5AAS

YOU MAY NOT USE LINE 5.

**LINE 5 DEPENDS ON THE WORKING
PREMISE IN LINE 3 WHICH IS NO
LONGER AVAILABLE.**

:*1.6HSS

ANTECEDENT OF LINE 6 MUST BE THE CONSEQUENT OF LINE 1

:*6.1HSS (7) P -> R

CORRECT...

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: R IFF P\$

DERIVE R IFF P

:*P\$ (1) *P & R\$

:*P\$ (2) *P -> Q\$

:*P\$ (3) *R -> P\$

:*1HS2\$ HS REQUIRES 2 LINE NUMBERS

:*1.2HS2\$ HS REQUIRES 0 OCCURRENCE NUMBERS

:*1.2HS\$ LINE 1 MUST BE A CONDITIONAL

:*2.3HS\$ ANTECEDENT OF LINE 3 MUST BE THE CONSEQUENT OF LINE 2

:*WP\$ (4) *P\$

:*1RC\$ (5) R

:*4.5CP\$ (6) P -> R

:*3.1FC\$ (7) (R -> P)&(P & R)

:*8LB\$ THERE IS NO LINE 8

:*7LB\$ RULE LB HAS 2 FORMS. NONE WERE SATISFIED BECAUSE EITHER: RIGHT CONJUNCT OF LINE 7 MUST BE A CONDITIONAL OR LINE 7 MUST BE A BICONDITIONAL.

:*DLL\$

:*3.6FC\$ (7) (R -> P)&(P -> R)

:*7LB\$ (8) R IFF P

CORRECT...

Fig. 19, continued.

of L_i with the element in the corresponding position of P_i , builds the error message. The match starts with the message stating the line number. Each constant or position of P_i has a name. As the program recursively examines each sublist of P_i , stopping when the sublist is an atom, the name of the constant or its position is appended to the message being built. When a substitution pair $\langle V_i, T_i \rangle$ is formed, the message M_i is a precise description of T_i . The triplet $\langle V_i, T_i, M_i \rangle$ is stored on a substitution list, S . Now, when the recursion finds a new V_j , the program can look on the list S to see if it already knows something about V_j . If so, and if T_j is not the same as the stored T_i , then an error has been found. M_j states what is T_j and M_i states what T_j ought to be. The message stating the error in application is then: M_j must be (the) M_i .

The derivations in Figure 19 point out the kinds of statements that can be generated, depending on whether the error was a mismatch of terms (as above), a mismatch of constants or main connectives, or whether the OR option failed for all possible sets of premises. If a well-formed term or formula is expected, the message merely states that the expression is improper.

Below is an annotated trace of the matching process when the derivation contains lines:

- (1) $A = B \rightarrow B = C$
- (2) $A = B$
- (3) $B = A$

The program will try to apply the AA rule. AA is not a derived rule in the system, but it is processed as though it were. Because it is a familiar rule, it is used for the illustration. The property list for AA is

```

PREMISE ((P → Q)P)
NOP      2
CONCL   Q

```

The student types the command 1.2AA, which, syntactically, is a valid command. Line 1 must match the first premise ($P \rightarrow Q$).

$$1. \quad V = P \rightarrow Q \quad T = (A = B \rightarrow B = C) \quad M = (\text{LINE 1})$$

The main connective of V and T are the same, so LINE 1 is a correct statement type. S is empty.

$$2. \quad V_1 = P \quad T_1 = (A = B) \quad M_1 = (\text{ANTECEDENT OF LINE 1})$$

S now equals $((P, (= A B), (\text{ANTECEDENT OF LINE 1})))$.

$$3. \quad V_2 = Q \quad T_2 = (B = C) \quad M_2 = (\text{CONSEQUENT OF LINE 1})$$

Q is not in S , so we add the new triplet, $(Q, (B = C), (\text{CONSEQUENT OF LINE 1}))$ to S . We continue to try to match the second line and the second premise.

$$4. \quad V_3 = P \quad T_3 = (A = B) \quad M_3 = (\text{LINE 2})$$

P is already in S with $T_1 = (A = B)$. T_1 is the same as T_3 so the match is consistent.

We apply X to the value of CONCL in order to obtain the new line: $B=C$.

What occurs when the command is 1.3AA? Traces 1-3 remain the same, but we now have:

$$4'. \quad V_3 = P \quad T_3 = (B = A) \quad M_3 = (\text{LINE 3})$$

T_3 is not the same as the term already saved on S for P . M_1 is retrieved and the error message is

LINE 3 MUST BE THE ANTECEDENT OF LINE 1.

M_3

M_1

4. Generating a Dialogue to Teach a Command

Observe the derivation in Figure 19 again. In the derivation of the sentence $R \text{ IFF } P$, the student made several attempts to use the HS

rule (hypothetical syllogism). After two attempts to type the command with the correct number of line and occurrence references, the student continued to receive messages pertaining to application errors. A correct HS command which the student could have typed is: 3.2HS (n) $R \rightarrow Q$.

The student's freedom to explore the use of a rule is a feature of the instructional system. He should not be willing to give up working on a problem only because he tires of receiving error messages. He should be able to ask the computer, as he would a human tutor, to (re)explain the use of the command. A tutorial dialogue aimed at reteaching commands can be realized by utilizing the procedures for analyzing format errors and errors of application given in Section 3. A question mark (?) is added to the command syntax.¹ A question mark, appearing anywhere in a command, indicates to the computer that the student needs information about the command name. The purpose of freely placing the question mark is to allow the student to tell the computer-tutor how much he already knows about the command.

The following discussion on the reteaching of commands is limited to the commands which result from derived rules of inference. The remaining types of commands present straightforward problems of teaching because they are programmed into the system or belong to a standard class of commands; teaching their correct applications can be accomplished by algorithms programmed for each primitive rule and for the axioms, theorems, and lemmas in general. However, derived rules of inference, because they are learned

¹I would like to thank David Levine for suggesting this addition to the instructional system, and for helping in some of the programming.

by the instructional system, require computation of messages in a fashion similar to that put forth in Section 3.

Teaching about derived rules takes on three aspects.

1. Informing the student about the correct syntax. By ignoring the question mark, and processing the rest of the command that the student typed, it is possible to find out how much syntax he already knows. The program then proceeds to suggest what is lacking to lead him to complete a syntactically correct command. Note that this task is simplified by the fact that commands for derived rules only consist of a sequence of line references followed by the name of the existing rule (see Figure 20).

2. Explaining, for each premise line referenced by the rule, the required statement type and any relationships within and among the premises. This is similar to the earlier problem of providing error messages for improper applications of rules. The routines are the same, with the modification that the program ignores the T_i (the term in the student's line), and lets the partial message M_i describe the individual variable V_i . The teaching sequence for the LB rule is shown in Figure 21. The monologue on the relationships between and within the patterns associated with LB is quite wordy. This example does, however, fully capture the recursive nature of the analysis routines.

3. Giving examples. The computer-tutor wants to know if there are any lines already in the student's work that can be used to illustrate the command. It can then show the student how to construct a command that will apply the rule to those lines. If no lines can be used as an example, the program generates simple instances of the necessary premises.

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: Q\$

DERIVE Q

:*P\$ (1) *P -> Q\$

:*P\$ (2) *Q OR P\$

:*?\$

WHICH COMMAND DO YOU NEED HELP WITH?

*FC\$

FC REQUIRES 2 LINE NUMBERS.

HERE'S AN EXAMPLE: 1.2 FC

MORE?*N\$

WOULD YOU LIKE AN EXAMPLE?*N\$

:*3..4FC\$

A NUMBER MUST FOLLOW A PERIOD

:*3..4?FC\$

FC REQUIRES 2 LINE NUMBERS.

LINE NUMBERS ARE SEPARATED BY SINGLE PERIODS

LIKE THIS: 3.4 FC

MORE?*N\$

WOULD YOU LIKE AN EXAMPLE?*N\$

:*3.2HS?\$

WITHOUT ?, YOU TYPED THE CORRECT FORMAT.

MORE?*N\$

WOULD YOU LIKE AN EXAMPLE?*N\$

:*DD1?\$

DD REQUIRES 2 LINE NUMBERS.

HERE'S AN EXAMPLE: 1.2 DD

THE COMMAND REQUIRES 0 OCCURRENCE NUMBERS

MORE?*N\$

WOULD YOU LIKE AN EXAMPLE?*N\$

:*FIN\$

Fig. 20.--Teaching the commands; Syntax.

DERIVE P \rightarrow P
P (1) P IFF Q

!*LB?S
LB REQUIRES 1 LINE NUMBER.
HERE'S AN EXAMPLE: 1 LB

MORE?*YS
LB HAS 2 FORMS.

IN FORM 1:

LINE 1 MUST BE A BICONDITIONAL
THE NEW LINE WILL BE A CONJUNCTION
LEFT CONJUNCT OF THE NEW LINE WILL BE A CONDITIONAL
ANTECEDENT OF THE LEFT CONJUNCT OF THE NEW LINE WILL BE THE
LHS OF LINE 1
CONSEQUENT OF THE LEFT CONJUNCT OF THE NEW LINE WILL BE THE
RHS OF LINE 1
RIGHT CONJUNCT OF THE NEW LINE WILL BE A CONDITIONAL
ANTECEDENT OF THE RIGHT CONJUNCT OF THE NEW LINE WILL BE THE
RHS OF LINE 1 AND CONSEQUENT OF THE LEFT
CONJUNCT OF THE NEW LINE
CONSEQUENT OF THE RIGHT CONJUNCT OF THE NEW LINE WILL BE THE
LHS OF LINE 1 AND ANTECEDENT OF THE LEFT
CONJUNCT OF THE NEW LINE

IN FORM 2:

LINE 1 MUST BE A CONJUNCTION
LEFT CONJUNCT OF LINE 1 MUST BE A CONDITIONAL
RIGHT CONJUNCT OF LINE 1 MUST BE A CONDITIONAL
ANTECEDENT OF THE RIGHT CONJUNCT OF LINE 1 MUST BE THE
CONSEQUENT OF THE LEFT CONJUNCT OF LINE 1
CONSEQUENT OF THE RIGHT CONJUNCT OF LINE 1 MUST BE THE
ANTECEDENT OF THE LEFT CONJUNCT OF LINE 1
THE NEW LINE WILL BE A BICONDITIONAL
LHS OF THE NEW LINE WILL BE THE ANTECEDENT OF THE LEFT
CONJUNCT OF LINE 1 AND CONSEQUENT OF THE RIGHT
CONJUNCT OF LINE 1
RHS OF THE NEW LINE WILL BE THE CONSEQUENT OF THE LEFT
CONJUNCT OF LINE 1 AND ANTECEDENT OF THE RIGHT
CONJUNCT OF LINE 1

WOULD YOU LIKE AN EXAMPLE?*YS

FOR FORM 1
THERE IS AN EXAMPLE OF THE LB RULE IN THE LINES OF YOUR DERIVATION.
A CORRECT COMMAND IS:
1LB

THE NEW LINE WOULD BE: (P \rightarrow Q)&(Q \rightarrow P)

FOR FORM 2
IF YOUR DERIVATION CONTAINS LINES:
(2) (P \rightarrow Q)&(Q \rightarrow P)

THEN A CORRECT COMMAND WOULD BE:
2LB

AND THE NEW LINE WILL BE:
P IFF Q

!*

Note that the teaching process, like the analysis of applications of a command, is complicated by the optional properties which can be associated with a rule name. Since the instructional system only knows about a derived rule from the information on the property list associated with the name of that rule, and, for the most part, the program itself computed the property lists, the program's ability to provide error messages and teaching sequences represents a saving and service to the teacher.

This section, and the description of the instructional system, is completed with five sample derivations (Figure 22) in which the student is taught how to use some of the commands available to him. In problem five, the student uses the INIT command to obtain the initiative to request his own problems. In this case, he proves a lemma that is useful in solving problem five. In problem four, the student learns about the DS rule. In the process of giving an example from the student's own work, the student also receives a hint on how to continue the derivation.

Another feature of the system is highlighted in the second problem: the teacher may require that the student use or not use particular command(s) when constructing a derivation or proof. Since the decision was made earlier not to interrupt the student's work, the computer-tutor waits until the derivation is completed before examining it and checking to see that the derivation was done within the set constraints. The program computes a list of lines and commands which contributed materially to his solution by chaining backward through the line references, beginning with the last line generated. Lines not in this list are called "irrelevant" and are ignored. If rules which must be in the list are absent, or rules which may not be in the list are present, the student is told so. He is

1
DO NOT USE CONDITIONAL PROOF.

DERIVE P \rightarrow P
P (1) P IFF Q

:*1LB?S

WITHOUT ?, YOU TYPED THE CORRECT FORMAT.
MORE?*NS
WOULD YOU LIKE AN EXAMPLE?*YS

FOR FORM 1
THERE IS AN EXAMPLE OF THE LB RULE IN THE LINES OF YOUR DERIVATION.
A CORRECT COMMAND IS:
1LB

THE NEW LINE WOULD BE: (P \rightarrow Q)&(Q \rightarrow P)

FOR FORM 2
IF YOUR DERIVATION CONTAINS LINES:
(2) (P \rightarrow Q)&(Q \rightarrow P)

THEN A CORRECT COMMAND WOULD BE:
2LB

AND THE NEW LINE WILL BE:
P IFF Q

:*1LBS (2) (P \rightarrow Q)&(Q \rightarrow P)

:*2?HSS
HS REQUIRES 2 LINE NUMBERS.
HERE'S AN EXAMPLE: 1.2 HS

MORE?*YS
LINE 1 MUST BE A CONDITIONAL
LINE 2 MUST BE A CONDITIONAL
ANTECEDENT OF LINE 2 MUST BE THE CONSEQUENT OF LINE 1

THE NEW LINE WILL BE A CONDITIONAL
ANTECEDENT OF THE NEW LINE WILL BE THE ANTECEDENT OF LINE 1
CONSEQUENT OF THE NEW LINE WILL BE THE CONSEQUENT OF LINE 2

WOULD YOU LIKE AN EXAMPLE?*NS

:*2LCS (3) P \rightarrow Q

:*2RCS (4) Q \rightarrow P

:*3.4HSS (5) P \rightarrow P

CORRECT...

Fig. 22. -- Teaching the commands: Giving examples.

2

USE THE DC RULE (DENY THE CONSEQUENT)
IN THIS PROBLEM.

DERIVE NOT P
P (1) Q & (NOT S)
P (2) P -> S

:*WPS (3) *PS

:*2.3AA5 (4) S

:*1RC5 (5) NOT S

:*3.4.5IP5 (6) NOT P

CORRECT...

YOU MUST USE THE DC RULE. TRY THE PROOF AGAIN.

DERIVE NOT P
P (1) Q & (NOT S)
P (2) P -> S

:*DC?S

DC REQUIRES 2 LINE NUMBERS.

HERE'S AN EXAMPLE: 1.2 DC

MORE?*YS

LINE 1 MUST BE A CONDITIONAL

LINE 2 MUST BE A NEGATION

NEGATED FORMULA OF LINE 2 MUST BE THE CONSEQUENT OF LINE 1

THE NEW LINE WILL BE A NEGATION

NEGATED FORMULA OF THE NEW LINE WILL BE THE ANTECEDENT OF LINE

1

WOULD YOU LIKE AN EXAMPLE?*YS

IF YOUR DERIVATION CONTAINS LINES:

(3) P -> Q

(4) NOT Q

THEN A CORRECT COMMAND WOULD BE:

3.4DC

AND THE NEW LINE WILL BE:

NOT P

:*1RC5 (3) NOT S

:*2.3DC5 (4) NOT P

CORRECT...

Fig. 22, continued.

3
 DERIVE (A < 0)->((A + B) < B)
 P (1) (0 + B)= B

:*WPS (2) *A < 0\$

:*AE?\$

AE REQUIRES 1 LINE NUMBER.
 HERE'S AN EXAMPLE: 1 AE

MORE?*YS

THE MAIN CONNECTIVE OF LINE 1 MUST BE ONE OF: = < >

THE MAIN CONNECTIVE OF THE NEW LINE WILL BE THE MAIN
 CONNECTIVE OF LINE 1

LHS OF THE NEW LINE WILL BE A SUM

LHS OF THE LHS OF THE NEW LINE WILL BE THE LHS OF LINE 1

RHS OF THE LHS OF THE NEW LINE WILL BE A TERM YOU TYPE

RHS OF THE NEW LINE WILL BE A SUM

LHS OF THE RHS OF THE NEW LINE WILL BE THE RHS OF LINE 1

RHS OF THE RHS OF THE NEW LINE WILL BE A TERM YOU TYPE

WOULD YOU LIKE AN EXAMPLE?*YS

THERE IS AN EXAMPLE OF THE AE RULE IN THE LINES OF YOUR DERIVATION.
 A CORRECT COMMAND IS:

2AE

:: <WELL-FORMED TERM>

THE NEW LINE WOULD BE: (A + <TERM>)<(0 + <TERM>)

WOULD YOU LIKE ANOTHER EXAMPLE?*YS

A CORRECT COMMAND IS:

1AE

:: <WELL-FORMED TERM>

THE NEW LINE WOULD BE: ((0 + B) + <TERM>)=(B + <TERM>)

:*2AES

::*BS (3) (A + B)<(0 + B)

:*3.1RE1\$ (4) (A + B)< B

:*2.4CP\$ (5) (A < 0)->((A + B) < B)

CORRECT...

Fig. 22, continued.

4
USE THE FORM A DISJUNCTION (FD) RULE.

DERIVE (NOT R)->(NOT Q)
P (1) NOT P
P (2) S ->(NOT Q)
P (3) (NOT R)-> P

:*WPS (4) *NOT RS

:*?FDS
FD REQUIRES 1 LINE NUMBER.
HERE'S AN EXAMPLE: 1 FD

MORE?*YS

THE NEW LINE WILL BE A DISJUNCTION
LEFT DISJUNCT OF THE NEW LINE WILL BE LINE 1
RIGHT DISJUNCT OF THE NEW LINE WILL BE A SENTENCE YOU TYPE

WOULD YOU LIKE AN EXAMPLE?*NS

:*4FDS
: : *S\$ (5) (NOT R) OR S

:*3.2.5DSS
LINE 3 MUST BE A DISJUNCTION

:*DS?\$
DS REQUIRES 3 LINE NUMBERS.
HERE'S AN EXAMPLE: 1.2.3 DS

MORE?*YS

LINE 1 MUST BE A DISJUNCTION
LINE 2 MUST BE A CONDITIONAL
ANTECEDENT OF LINE 2 MUST BE THE LEFT DISJUNCT OF LINE 1
LINE 3 MUST BE A CONDITIONAL
ANTECEDENT OF LINE 3 MUST BE THE RIGHT DISJUNCT OF LINE 1

THE NEW LINE WILL BE A DISJUNCTION
LEFT DISJUNCT OF THE NEW LINE WILL BE THE CONSEQUENT OF LINE 2
RIGHT DISJUNCT OF THE NEW LINE WILL BE THE CONSEQUENT OF LINE

3

WOULD YOU LIKE AN EXAMPLE?*YS
THERE IS AN EXAMPLE OF THE DS RULE IN THE LINES OF YOUR DERIVATION.
A CORRECT COMMAND IS:
5.3.2DS

THE NEW LINE WOULD BE: P OR(NOT Q)

:*5.3.2DSS (6) P OR(NOT Q)

:*6.1DDS (7) NOT Q

:*4.7CPS (8) (NOT R)->(NOT Q)

CORRECT...

```

5
DERIVE Q
P (1) R -> Q
P (2) S -> Q
P (3) S OR R

```

:*INIT\$

YOU CAN NOW REQUEST A DERIVE OR PROVE PROBLEM, OR DERIVE A NEW RULE OF INFERENCE.

THE INTERRUPTED PROBLEM WILL BE RESTARTED IF YOU TYPE FIN .

TYPE A DERIVE, PROVE OR RULE COMMAND

```

:*PROVE: (Q OR Q) -> QS
PROVE (Q OR Q)-> Q

```

:*WPS (1) *Q OR QS

:*WPS (2) *NOT QS

:*1.2DDS (3) Q

:*2.2.3IPS (4) Q

:*1.4CPS (5) (Q OR Q)-> Q

CORRECT...

NAME: *LEMMAS

TYPE A DERIVE, PROVE OR RULE COMMAND

:*FIN\$

NOW REDO THE PROBLEM YOU INTERRUPTED

```

5
DERIVE Q
P (1) R -> Q
P (2) S -> Q
P (3) S OR R

```

:*3.2.1DSS (4) Q OR Q

```

:*LEMMAS (Q OR Q)-> Q
(5) (Q OR Q)-> Q

```

:*5.4AAS (6) Q

CORRECT...

LESSON OVER...

GOODBYE...ADELE

T
*

then asked to try the problem again. Note that a required rule may not have been in the list because the student did not know how to use it and did know how to do the problem without it. Thus he can type a question mark to ask the computer-tutor for help.

1. The first part of the document is a letter from the
author to the editor of the journal. The letter
describes the author's interest in the subject
of the article and the reasons for writing it.
The author also mentions that the article is
based on his own research and that he has
checked the accuracy of the information.

CHAPTER VII

SUMMARY AND FINAL COMMENTS

The announced intention of this dissertation was to study the computer's ability to assume a tutorial role; the aim was to provide the student of a computer-based teacher with the dynamic and personalized interaction afforded him by an attentive human tutor. The importance of understanding the extent and limit of the computer's ability to provide individualized instruction is evident in light of the surge of interest and activity in using computers in actual classrooms.

The research was carried out by identifying characteristics that distinguish the tutorial role, and by providing a concrete example of the computer's ability to teach elementary mathematical logic. To determine how well the computer-teacher for logic performs, several studies were carried out: case studies of students using the instructional system, comparisons with other methods for tutoring logic, and a comparison with the performance of experienced human tutors. The basic principle underlying the investigation was that knowledgeable computer-tutors, tutors that know how to do the work the students are expected to do, could provide individualized attention to the bright students as well as to the less skillful students. The emphasis in developing the sample computer-tutor was on providing a response-sensitive teaching system; one that can adapt to the student's educational requirements, especially when the student experiences difficulty

in carrying out the tasks set for him. The instructional system for mathematical logic, with a theorem-prover acting as a proof-analyzer, is able to capture the flexible and adaptive nature of human tutoring. So, the response strategies and techniques that form the basis of the system are of significant interest in formulating computer-tutors for other curricula.

The instructional system was designed to increase the active participation of the students with the computer-teacher. It embodies methods by which a student can (a) experiment freely with his new skills in proof construction; (b) receive immediate feedback on errors; (c) ask about previously learned material; and (d) make up his own examples. These examples can either be derivation problems requested within the theory prespecified by a curriculum writer, or the student's own axiomatization specified within a formalization of the full predicate calculus with identity. In this manner, the student can test out his own notions of what is meant by a nonlogical axiomatic theory. Inherent in the teaching system is the theme of free exploration of the subject matter, offering the student an opportunity to practice, in varied contexts, his newly learned skills. Nevertheless, the system retains tutorial capabilities for providing feedback to confirm correct knowledge and to correct errors, and to reteach the inference rules of the theory presently under consideration by the student. Such feedback always derives from material already learned, or specified, by the student.

A unique feature of a good human tutor is the ability to give advice and suggestions to a student who requests help in completing the task set for him. He does not make just any comment to the student;

rather, he examines the work already done by the student in order to extract information from which he can make remarks relevant to the student's individual needs. In developing a computer-tutor for teaching proof construction, a central question was how to analyze the proof steps already constructed by the student in order to obtain information about how to complete the student's proof. The computer-tutor for logic employs a theorem-prover as the proof-analyzer. The theorem-prover is capable of doing the proofs the students must do, and is able to take the student's proof steps into account in its search for a solution. In Chapters IV and V, the set of inference rules used by the theorem-prover and some mathematical results about the generality and appropriateness of the rules were offered.

The original intent in using the theorem-prover was to obtain a set of possible ways to complete the student's work. It would then be possible to devise strategies for deciding which completion the tutor should talk about, what the tutor should say to the student, and how the tutor might encourage the student to find other solutions discovered by the theorem-prover. In developing the theorem-prover, the manner in which the set of solutions was determined by the program was not as important a consideration as knowing that the responses made to the student were correct. However, in order to come up with hints the student could understand and use, it became necessary to consider how the theorem-prover actually operated. If the theorem-prover made use of search techniques which simulated those thought to be used by good students, then the tutor could simply give hints suggesting that the student try to use the search methods already successfully used by the theorem-prover.

The computer-tutor, in employing a theorem-prover as a proof-analyzer, attempts to provide the kind of tutorial dialogue that human tutors ought to provide. In the version reported on here of the dialogue, the tutor does not take into account errors made by the student in previous problems. However, it does try to emphasize the use of recently learned proof procedures and inference rules in order to have the student practice using new solution methods. Routines which organize and save information about the student's past performance in the curriculum can be easily implemented. To what extent, then does the computer-based tutor fail to perform as well as a human teacher might? The answer to this question is based on the ability of the human teacher to leave the present domain of discourse and to borrow freely from general sources of knowledge. The human teacher can let the student ask general questions, and can devise illustrations from other subject areas in order to help the student understand the answer to his query. The human teacher is not as restricted, as is the present computer-tutor, in formulating the tutorial dialogue, or in allowing interruptions from the student which could be useful in inferring problems the student may be experiencing. These sorts of abstractions, or generalizations, were not dealt with in the computer-tutor for elementary mathematical logic.

However, as the comparison of the behavior of the computer-tutor with that of experienced human tutors demonstrated, the program performs as well as, and in some cases better than, a human tutor. In particular, the computer-tutor never makes a mistake, never tells the student to do something which will not lead to a valid proof, and never fails to notice work done by the student. These are all errors that the human tutors

committed. The computer-tutor, like other computer-based teachers, is able to handle several students simultaneously, and still give each student the attention afforded him by a private tutor. The computer is patient, and does not become irritated with the student. One of the human tutors admitted that after a while he becomes fatigued and very mechanical in his answers to a student. (The word "mechanical" here suggests that the tutor resorted to an equivalent of a stored-hint approach to helping the student.) The knowledgeable computer-tutor for mathematical logic is willing to try a variety of proof strategies, is able to discover whether the student has started more than one proof that can be successfully completed, and is truly sensitive to different information available in the student's partial or erroneous work.

Further exploration in several areas will be needed in order to realize improvements in the computer-tutor. For one, as pointed out before, the actual generation of the tutorial dialogue is uninteresting from a language processing point of view. The tutorial dialogue should be changed to actually construct the comments. Moreover, it could be extended to operate as a question-answering routine, as suggested by the work of Simmons (1970) and Carbonell (1970). The theorem-prover would compute the information about the student's work, as it does now, and then this information would be used as the data from which the answers to student questions are retrieved. It would be of further interest to be able to prove that each answer is, in fact, correct; it is of both mathematical and practical concern to determine (that is, to prove) that the error messages and computed teaching sequences of the present instructional system do what we are assuming they do on the basis of our testing experience.

Second, of especial import for improving the power of the instructional system is the development of methods for providing a more satisfactory SHOW command (Chapter VI, Section 2.3). The methods can either be new strategies for a resolution-based theorem-prover and techniques for internally deciding which set of strategies to use, or new, heuristic theorem-provers to perform the verification task for individual theories. Curricula of a more complex nature than that proffered in preceding chapters has yet to be implemented in a CAI environment. Successful implementation of such courses might rely on general facilities which allow the student to specify a theory and explore the implications of the primitives chosen, but success will rely more on those features which dictate the sorts of proofs the students can attempt. These features include the miniature theorem-provers which enable the student to skip trivial steps, large-scale provers capable of verifying nonrigorous proofs (SHOW command), and the routines able to do the student's work and thereby act as the proof analyzer.

The design of the computer-tutor for mathematical logic suggests a style for CAI research which necessitates developing teaching systems that depend more on the actual computer program and less on a curriculum-writer armed with a multiple-choice/constructed-response author language. System statistics from our operational experiences show that the instructional program spends 80 to 90 percent of its time computing, contradicting the usual assumption that CAI programs are tied to input/output processes. Use of such an instructional program requires a reevaluation of the computer's operating system. The ability to understand a command language specified by a teacher, or by a student, added flexibility to the structure of the teaching system and increased the variability of the content of the course. The

theorem-prover/proof-analyzer described in Chapter IV is useful mainly in the context of teaching proof construction. The theorem-prover is applicable for teaching about Abelian groups--whether it can easily be extended is yet to be determined. For new fields of study other than proof construction, the course author would have to examine the particular tasks of the subject and determine what sort of program is needed to carry out the response analysis. However, many features of the instructional system which have been emphasized in this dissertation would strengthen the framework of other computer-based teaching systems; more notably, the integral role of an ideal tutor, one which can effectively take into account the student's work in determining relevant help sequences, and one which provides a flexible environment in which the students are encouraged to explore the structure of the subject area.

REFERENCES

- Abrahams, P. Machine verification of mathematical proof. (Doctoral dissertation, MIT) Cambridge, Mass.: Microreproduction Lab, 1963. NS371.
- Allen, J., & Luckham, D. An interactive theorem-proving program. In B. Meltzer & D. Michie (Eds.), Machine intelligence 5. New York: American Elsevier, 1970, 321-336.
- Atkinson, R. C., & Wilson, H. A. Computer-assisted instruction: A book of readings. New York: Academic Press, 1969.
- Bruner, J. S. Towards a theory of instruction. New York: Norton, 1966.
- Bushnell, D. D., & Allen, D. D. The computer in American education. New York: Wiley, 1967.
- Carbonell, J. R. Mixed-initiative man-computer instruction. Technical Report No. 1971, May, 1970, Bolt, Beranek & Newman, ONR Contract No. N00014-69-0233.
- Coulson, J. E. (Ed.) Programmed learning and computer-based instruction. Proceedings of the conference on application of digital computers to automated instruction, Oct. 10-12, 1961. New York: Wiley, 1962.
- Crowder, N. A. Intrinsic and extrinsic programming. In J. E. Coulson (Ed.), Proceedings of the conference on application of digital computers to automated instruction, Oct. 10-12, 1961. New York: Wiley, 1962, 58-66.
- Easley, J. A., Gelder, H., & Golden, W. A PIATO program for instruction and data collection in mathematical problem solving. CSL Report R-185, 1964, University of Illinois Coordinated Science Laboratory.

- Gelernter, H. Realization of a geometry theorem proving machine. In E. A. Feigenbaum & J. Feldman (Eds.), Computers and thought. New York: McGraw-Hill, 1963, 134-152.
- Goldberg, A. A generalized instructional system for teaching elementary mathematical logic. Technical Report No. 179, October 11, 1971, Stanford University, Institute for Mathematical Studies in the Social Sciences.
- Goldberg, A., & Suppes, P. A computer-assisted instruction program for exercises on finding axioms. Educational Studies in Mathematics, 1972 (in press).
- Green, C. C. The applications of theorem proving to question-answering systems. Memo No. AIM-96, 1969, Stanford University, Artificial Intelligence Project.
- Hickey, A. E. (Ed.) CAI: A survey of the literature. Newbury, Mass.: Entelek, 1968.
- Kalish, D. Semantics. In P. Edwards (Ed.), The encyclopedia of philosophy. Vol. 7. New York: Macmillan and the Free Press, 1967, 348-358.
- Kalish, D., & Montague, R. Logic: Techniques of formal reasoning. New York: Harcourt Brace & World, 1964.
- Luckham, D. Refinement theorems in resolution. In M. Laudet (Ed.), Proceedings IRIA symposium on automatic demonstration, Versailles, France, December 16-21, 1968. New York: Springer-Verlag, 1970, 163-188.
- Mauer, W. D. Computer experiments in finite algebra. Communications of the ACM, 1966, 9, 598-603.

- McCarthy, J. Programs with common sense. Proceedings symposium on mechanization of thought processes, National Physics Laboratory, Teddington, England. London: Her Majesty's Stationery Office, 1959, 77-84.
- McCarthy, J., Abrahams, P., Edwards, D. J., Hart, T. P., & Levin, M. I. LISP 1.5 programmer's manual. Cambridge, Mass.: MIT Press, 1962.
- Mendelson, E. Introduction to mathematical logic. Princeton: Van Nostrand, 1964.
- Minsky, M. (Ed.) Semantic information processing. Cambridge, Mass.: MIT Press, 1968.
- Newell, A., Shaw, J., & Simon, H. Empirical explorations of the logic theory machine. In E. A. Feigenbaum & J. Feldman (Eds.), Computers and thought. New York: McGraw-Hill, 1963, 109-133.
- Newell, A., & Simon, H. Computers in psychology. In R. D. Luce, R. R. Bush, & E. H. Galanter (Eds.), Handbook of mathematical psychology. Vol. 1. New York: Wiley, 1963, 361-428. (a)
- Newell, A., & Simon, H. GPS: A program that simulates human thought. In E. A. Feigenbaum & J. Feldman (Eds.), Computers and thought. New York: McGraw-Hill, 1963, 279-293. (b)
- Nilsson, N. Problem-solving methods in artificial intelligence. New York: McGraw-Hill, 1971.
- Norton, L. ADEPT--A heuristic program for proving theorems of group theory. (Doctoral dissertation, MIT) Cambridge, Mass.: October, 1966. MAC-TR-33.
- Papert, S. Teaching children thinking. Memo No. 249, July, 1971, MIT, Artificial Intelligence Group.
- Polya, G. How to solve it. (2nd ed.) New York: Doubleday, 1957.

Pressey, S. A simple apparatus which gives tests and scores and teaches.

School and Society, 1926, 23, 373-376.

Quillian, M. R. Semantic memory. Scientific Report No. 2, October, 1968,

Bolt, Beranek & Newman, Contract AF19 (628)-5065.

Quillian, M. R. The teachable language comprehender: A simulation program

and theory of language. Communications of the ACM, 1969, 12, 459-476.

Quinlan, J. R., & Hunt, E. B. A formalization of the general problem solver.

SWRL Scientific Report 8, November 4, 1968. Los Angeles. (a)

Quinlan, J. R., & Hunt, E. B. A formal deductive problem-solving system.

JACM, 1968, 15, 625-646. (b)

Robinson, J. A. A machine oriented logic based on the resolution principle.

JACM, 1965, 12, 23-41.

Rosen, B. Subtree replacement systems. Report 2-71, February, 1971, Harvard

University, Center for Research in Computing Technology.

Siklossy, L. Computer tutors that know what they teach. AFIPS conference

proceedings, 1970. Vol. 37. Washington, D. C.: Spartan Books, 1970.

Simmons, R. F. Natural language question-answering systems: 1969.

Communications of the ACM, 1970, 13, 15-30.

Skinner, B. F. Teaching machines. Science, 1958, 128, 969-977.

Skinner, B. F. The technology of teaching. New York: Meredith, 1968.

Slagle, J. A heuristic program that solves integration problems in

freshman calculus. In E. A. Feigenbaum & J. Feldman (Eds.),

Computers and thought. New York: McGraw-Hill, 1963, 191-203.

Slagle, J. Artificial intelligence: The heuristic programming approach.

New York: McGraw-Hill, 1971.

- Stolurow, L. H. Computer-based instruction: Psychological aspects and systems conception of instruction. Technical Report No. 4, December, 1967, Harvard Computing Center.
- Suppes, P. Computer-assisted instruction at Stanford. In Man and computer (Proceedings of international conference, Bordeaux 1970). Basel: Karger, 1972, 298-330.
- Suppes, P., & Binford, F. Experimental teaching of mathematical logic in the elementary school. The Arithmetic Teacher, 1965, 12, 187-195.
- Suppes, P., & Crothers, E. Experiments in second language learning. New York: Academic Press, 1967.
- Suppes, P., & Hill, S. First course in mathematical logic. New York: Blaisdell, 1964.
- Suppes, P., & Ihrke, C. Accelerated program in elementary-school mathematics--the fourth year. Psychology in the Schools, 1970, 7, 111-126.
- Szmielew, W. Arithmetical properties of Abelian groups. (Doctoral dissertation, University of California, Berkeley) Berkeley, Calif.: 1950.
- Tarski, A. A decision method for elementary algebra and geometry. Santa Monica, Calif.: The Rand Corp., 1948 (Revised: 1951).
- Tarski, A., Mostowski, A., & Robinson, R. Undecidable theories. Amsterdam: North-Holland, 1953.
- Uttal, W., Pasich, T., Rogers, M., & Hieronymous, R. Generative computer assisted instruction. Communications No. 243, January, 1969, University of Michigan, Mental Health Institute.
- van der Waerden, B. L. Modern algebra. New York: Ungar, 1947.
- Wexler, J. A generative, remedial and query system for teaching by computer. Unpublished doctoral dissertation, University of Wisconsin, March 21, 1970.

Zinn, K., & McClintock, S. A guide to the literature on interactive use of computers for instruction. January, 1970. University of Michigan, CRLT, Project CLUE. Distributed by ERIC Clearinghouse on Media and Technology.

APPENDIX I

AXIOMS AND THEOREMS ON ADDITION

Axioms

CA	Commute Addition	$A+B=B+A$
AS	Associate Addition	$(A+B)+C=A+(B+C)$
Z	Zero Axiom	$A+0=A$
AI	Additive Inverse	$A+(-A)=0$
N	Negative Number Axiom	$A+(-B)=A-B$

Theorems

1. $0+A=A$	12. $A+B=A \rightarrow B=0$
2. $(-A)+A=0$	13. $-(-A) = A$
3. $A-A=0$	14. $(-(A+B))+B=-A$
4. $0-A=-A$	15. $-(A+B)=(-A)-B$
5. $0=-0$	16. $(-A)-B=(-B)-A$
6. $A-0=A$	17. $-(A-B)=B-A$
7. $A+B=A+C \rightarrow B=C$	18. $(A-B)-C=A+((-B)-C)$
8. $A+B=C \rightarrow A=C-B$	19. $(A-B)-C=A-(B+C)$
9. $A=C-B \rightarrow A+B=C$	20. $A+(B-A)=B$
10. $A+B=0 \rightarrow A=-B$	21. $A-(A+B)=-B$
11. $A=-B \rightarrow A+B=0$	22. $(A-B)+(B-C)=A-C$

APPENDIX II

PROOFS FOR THEOREMS ON ADDITION

The following proofs (for theorems 1-22 of Appendix I, as well as five problems solved by the SUBSET TEST rule) were generated by the theorem-proving program. The format for these proofs is the same as that available to users of the instructional system described in Chapter VI. The program was initially given five axioms about addition. As each new expression was proven and assigned a name (usually the letters "TH" followed by a number) the prover added the new theorem to its list of axiom- and theorem-schemata.

An illustration of the format used in this appendix to present each set of solutions in the form of a dialogue between a user and the theorem-prover is shown below. The user only types information preceded by a colon and an asterisk (i.e., :*); all else is generated by the theorem-prover. The ENTER key, \$, indicates that the user's response is completed.

TYPE A DERIVE, PROVE, OR RULE COMMAND

:*PROVE: B+O=B\$

PROVE B+O=B

:*HELP\$

[solution]

[solution]

:

:*ENT\$ (1) B+O=B

CORRECT...

NAME:*LEMMA\$

The computer asks the user to decide what kind of problem he wants to try.

The user requests a derivation problem.

The computer indicates that the expression is well formed.

The user calls on the theorem-prover to solve the problem.

The theorem-prover offers several solutions.

The user now enters the expression as a line of the proof.

The instructional system thinks that the user has successfully solved the problem and lets the user give the expression a name before going on to the next problem.

APPENDIX II, continued.

Proofs for Theorems about Addition*

Theorem 1

TYPE A DERIVE, PROVE OR RULE COMMAND

```

**PROVE: 0+A=A
PROVE (0 + A)= A

```

**HELPS

```

CA A + B = B + A
A:0
B:A (1) (0 + A)=(A + 0)
Z A + 0 = A
A:A (2) (A + 0)= A
1-2RE1 (3) (0 + A)= A

```

```

CA A + B = B + A
A:A
B:0 (1) (A + 0)=(0 + A)
Z A + 0 = A
A:A (2) (A + 0)= A
1-2RE1 (3) A=(0 + A)
3CE1 (4) (0 + A)= A

```

```

Z A + 0 = A
A:A (1) (A + 0)= A
CA A + B = B + A
B:0
A:A (2) (A + 0)=(0 + A)
1-2RE1 (3) (0 + A)= A

```

**ENTS (1) *0+A=A

```

CORRECT...
NAME: *TH1$

```

Theorem 3

TYPE A DERIVE, PROVE OR RULE COMMAND

```

**PROVE: A-A=0
PROVE (A - A)= 0

```

**HELPS

```

AI A + (- A)= 0
A:A (1) (A + - A)= 0
ICE1 (2) 0=(A + - A)
N A + (- B)= A - B
A:A
B:A (3) (A + - A)=(A - A)
2-3RE1 (4) 0=(A - A)
4CE1 (5) (A - A)= 0

```

```

AI A + (- A)= 0
A:A (1) (A + - A)= 0
N A + (- B)= A - B
A:A
B:A (2) (A + - A)=(A - A)
1-2RE1 (3) (A - A)= 0

```

**ENTS (1) *A-A=0

```

CORRECT...
NAME: *TH3$

```

Theorem 2

TYPE A DERIVE, PROVE OR RULE COMMAND

```

**PROVE: (-A)+A=0
PROVE (- A + A)= 0

```

**HELPS

```

AI A + (- A)= 0
A:A (1) (A + - A)= 0
CA A + B = B + A
A:A
B:- A (2) (A + - A)=(- A + A)
1-2RE1 (3) (- A + A)= 0
3CE1 (4) 0=(- A + A)
4CE1 (5) (- A + A)= 0

```

```

AI A + (- A)= 0
A:A (1) (A + - A)= 0
CA A + B = B + A
A:A
B:- A (2) (A + - A)=(- A + A)
1-2RE1 (3) (- A + A)= 0

```

```

AI A + (- A)= 0
A:A (1) (A + - A)= 0
CA A + B = B + A
A:A
B:- A (2) (A + - A)=(- A + A)
1-2RE1 (3) (- A + A)= 0

```

**ENTS (1) *(-A)+A=0

```

CORRECT...
NAME: *TH2$

```

Theorem 4

TYPE A DERIVE, PROVE OR RULE COMMAND

```

**PROVE: 0-A=-A
PROVE (0 - A)= - A

```

**HELPS

```

TH1 0 + A = A
A:- A (1) (0 + - A)= - A
ICE1 (2) - A=(0 + - A)
N A + (- B)= A - B
A:0
B:A (3) (0 + - A)=(0 - A)
2-3RE1 (4) - A=(0 - A)
4CE1 (5) (0 - A)= - A

```

```

TH1 0 + A = A
A:- A (1) (0 + - A)= - A
N A + (- B)= A - B
A:0
B:A (2) (0 + - A)=(0 - A)
1-2RE1 (3) (0 - A)= - A

```

**ENTS (1) *0-A=-A

```

CORRECT...
NAME: *TH4$

```

*Proofs were generated by the theorem-prover in the same format as in Figures 8 and 9.

Theorem 5

TYPE A DERIVE, PROVE OR RULE COMMAND

*PROVE: 0=-0S
PROVE 0 = - 0

*HELPS

TH3 A - A = 0
A110 (1) (0 - 0) = 0
TH4 0 - A = - A
A110 (2) (0 - 0) = - 0
1-3RE1 (3) - 0 = 0
3CE1 (4) 0 = - 0

TH3 A - A = 0
A110 (1) (0 - 0) = 0
TH4 0 - A = - A
A110 (2) (0 - 0) = - 0
1-3RE1 (3) - 0 = 0
3CE1 (4) 0 = - 0

*ENTS (1) *0=-0S

CORRECT...
NAME: *TH5S

Theorem 6

TYPE A DERIVE, PROVE OR RULE COMMAND

*PROVE: A-0=A5
PROVE (A + 0) = A

*HELPS

Z A + 0 = A
A11A (1) (A + 0) = A
TH5 0 = - 0
(2) 0 = - 0
1-3RE1 (3) (A + - 0) = A
N A + (- B) = A - B
A11A
B110 (4) (A + - 0) = (A - 0)
3-4RE1 (5) (A - 0) = A

*ENTS (1) *A-0=-4->A5

CORRECT...
NAME: *TH6S

Theorem 7

TYPE A DERIVE, PROVE OR RULE COMMAND

*PROVE: (A+B=A+C)->(B=C)S
PROVE ((A + B) = (A + C)) -> (B = C)

*HELPS

UP (1) (A + B) = (A + C)
CA A + B = B + A
A11A
B110 (2) (A + B) = (B + A)
1-3RE1 (3) (B + A) = (A + C)
3AE
A11A (4) ((B + A) + - A) = ((A + C) + - A)
4ARS (5) (B + (A + - A)) = ((A + C) + - A)
A1 A + (- A) = 0
A11A (6) (A + - A) = 0
5-6RE1 (7) (B + 0) = ((A + C) + - A)
Z A + 0 = A
A11B (8) (B + 0) = B
7-8RE1 (9) B = ((A + C) + - A)
CA A + B = B + A
B11A
A11A + C (10) ((A + C) + - A) = (- A + (A + C))
CA A + B = B + A
B11A + C
A11A (11) (- A + (A + C)) = ((A + C) + - A)
10-11RE1 (12) ((A + C) + - A) = ((A + C) + - A)
CA A + B = B + A
B11C
A11A (13) (A + C) = (C + A)
12-13RE1 (14) ((A + C) + - A) = ((C + A) + - A)
AS (A + B) + C = A + (B + C)
C11A A
B11A
A11C (15) ((C + A) + - A) = (C + (A + - A))
14-15RE1 (16) ((A + C) + - A) = (C + (A + - A))
A1 A + (- A) = 0
A11A (17) (A + - A) = 0
16-17RE1 (18) ((A + C) + - A) = (C + 0)
Z A + 0 = A
A11C (19) (C + 0) = C
18-19RE1 (20) ((A + C) + - A) = C
9-20RE1 (21) B = C
1-21CP (22) ((A + B) = (A + C)) -> (B = C)

*ENTS (1) *(A+B=A+C)->(B=C)S

CORRECT...
NAME: *TH7S

Theorem 8

TYPE A DERIVE, PROVE OR RULE COMMAND

```

**PROVE: (A+B=C)-->(A=C-B)$
PROVE      ((A + B) = C)-->(A = (C - B))

**HELPS

TH7  (A + B = A + C)-->(B = C)
B**A
C**C - B
A**B      (1) ((B + A) = (B + (C - B)))-->(A = (C - B))
CA  A + B = B + A
A**B
B**A      (2) (B + A) = (A + B)
1-2RE1  (3) ((A + B) = (B + (C - B)))-->(A = (C - B))
N  A + (- B) = A - B
A**C
B**B      (4) (C + - B) = (C - B)
4CE1    (5) (C - B) = (C + - B)
3-5RE1  (6) ((A + B) = (B + (C + - B)))-->(A = (C - B))
CA  A + B = B + A
B**C + - B
A**B      (7) (B + (C + - B)) = ((C + - B) + B)
AS  (A + B) + C = A + (B + C)
C**B
B** - B
A**C      (8) ((C + - B) + B) = (C + (- B + B))
7-8RE1  (9) (B + (C + - B)) = (C + (- B + B))
CA  A + B = B + A
B**B
A** - B   (10) (- B + B) = (B + - B)
9-10RE1  (11) (B + (C + - B)) = (C + (B + - B))
AI  A + (- A) = 0
A**B     (12) (B + - B) = 0
11-12RE1 (13) (B + (C + - B)) = (C + 0)
Z  A + 0 = A
A**C     (14) (C + 0) = C
13-14RE1 (15) (B + (C + - B)) = C
6-15RE1  (16) ((A + B) = C)-->(A = (C - B))
    
```

```

WP      (1)      (A + B) = C
IAE
A** - B  (2)      ((A + B) + - B) = (C + - B)
2AR2    (3)      (A + (B + - B)) = (C + - B)
AI  A + (- A) = 0
A**B     (4)      (B + - B) = 0
3-4RE1  (5)      (A + 0) = (C + - B)
Z  A + 0 = A
A**A     (6)      (A + 0) = A
5-6RE1  (7)      A = (C + - B)
N  A + (- B) = A - B
A**C
B**B     (8)      (C + - B) = (C - B)
7-8RE1  (9)      A = (C - B)
1-9CP   (10)     ((A + B) = C)-->(A = (C - B))
    
```

**ENTS (1) *(A+B=C)-->(A=C-B)\$

CORRECT...
NAME: *TH8\$

Theorem 9

TYPE A DERIVE, PROVE OR RULE COMMAND

```

**PROVE: (A=C-B)-->(A+B=C)$
PROVE      (A = (C - B))-->((A + B) = C)

**HELPS

WP      (1)      A = (C - B)
N  A + (- B) = A - B
A**C
B**B     (2)      (C + - B) = (C - B)
2CE1    (3)      (C - B) = (C + - B)
1-3RE1  (4)      A = (C + - B)
4AE
A**B     (5)      (A + B) = ((C + - B) + B)
AS  (A + B) + C = A + (B + C)
C**B
B** - B
A**C     (6)      ((C + - B) + B) = (C + (- B + B))
CA  A + B = B + A
B**B
A** - B  (7)      (- B + B) = (B + - B)
6-7RE1  (8)      ((C + - B) + B) = (C + (B + - B))
AI  A + (- A) = 0
A**B     (9)      (B + - B) = 0
8-9RE1  (10)     ((C + - B) + B) = (C + 0)
Z  A + 0 = A
A**C     (11)     (C + 0) = C
10-11RE1 (12)    ((C + - B) + B) = C
9-12RE1  (13)    (A + B) = C
1-13CP  (14)    (A = (C - B))-->((A + B) = C)
    
```

**ENTS (1) *(A=C-B)-->(A+B=C)\$

CORRECT...
NAME: *TH9\$

Theorem 10

TYPE A DERIVE, PROVE OR RULE COMMAND

```

**PROVE: (A+B=0)-->(A=-B)$
PROVE      ((A + B) = 0)-->(A = - B)

**HELPS
    
```

```

TH7  (A + B = A + C)-->(B = C)
B**A
C** - B
A**B      (1) ((B + A) = (B + + B))-->(A = - B)
CA  A + B = B + A
A**B
B**A      (2) (B + A) = (A + B)
1-2RE1  (3) ((A + B) = (B + + B))-->(A = - B)
AI  A + (- A) = 0
A**B     (4) (B + - B) = 0
3-4RE1  (5) ((A + B) = 0)-->(A = - B)
    
```

```

WP      (1)      (A + B) = 0
IAE
A** - B  (2)      ((A + B) + - B) = (0 + - B)
2AR2    (3)      (A + (B + - B)) = (0 + - B)
AI  A + (- A) = 0
A**B     (4)      (B + - B) = 0
3-4RE1  (5)      (A + 0) = (0 + - B)
Z  A + 0 = A
A**A     (6)      (A + 0) = A
5-6RE1  (7)      A = (0 + - B)
TH1  0 + A = A
A** - B  (8)      (0 + - B) = - B
7-8RE1  (9)      A = - B
1-9CP   (10)     ((A + B) = 0)-->(A = - B)
    
```

**ENTS (1) *(A+B=0)-->(A=-B)\$

CORRECT...
NAME: *TH10\$

Theorem 11

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: (A=-B)->(A+B=0)\$
 PROVE (A = - B)->((A + B) = 0)

!*HELPS

TH9 (A = C - B)->(A + B = C)
 A1:A
 B1:B
 C1:0
 (1) (A = (C - B))->((A + B) = C)
 TH4 0 - A = - A
 A1:B (2) (0 - B) = - B
 1-2RE1 (3) (A = - B)->((A + B) = 0)

WP (1) A = - B
 IAE
 A1:B (2) (A + B) = (- B + B)
 TH2 (- A) + A = 0
 A1:B (3) (- B + B) = 0
 2-3RE1 (4) (A + B) = 0
 1-4CP (5) (A = - B)->((A + B) = 0)

!*ENTS (1) *(A=-B)->(A+B=0)\$

CORRECT...
 NAME: *TH11

Theorem 13

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: --A=AS
 PROVE - - A = A

!*HELPS

TH10 (A + B = 0)->(A = - B)
 A1:A
 B1:- A (1) ((A + - A) = 0)->(A = - - A)
 AI A + (- A) = 0
 A1:A (2) (A + - A) = 0
 1-2AA (3) A = - - A
 3CE1 (4) - - A = A

!*ENTS (1) *--A=AS

CORRECT...
 NAME: *TH13

Theorem 12

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: (A+B=A)->(B=0)\$
 PROVE ((A + B) = A)->(B = 0)

!*HELPS

TH7 (A + B = A + C)->(B = C)
 B1:B
 C1:0
 A1:A (1) ((A + B) = (A + 0))->(B = 0)
 Z A + 0 = A
 A1:A (2) (A + 0) = A
 1-2RE1 (3) ((A + B) = A)->(B = 0)

WP (1) (A + B) = A
 CA A + B = B + A
 A1:A
 B1:B (2) (A + B) = (B + A)
 1-2RE1 (3) (B + A) = A
 3AE
 A1:- A (4) ((B + A) + - A) = (A + - A)
 4AR2 (5) (B + (A + - A)) = (A + - A)
 AI A + (- A) = 0
 A1:A (6) (A + - A) = 0
 5-6RE1 (7) (B + 0) = (A + - A)
 Z A + 0 = A
 A1:B (8) (B + 0) = B
 7-8RE1 (9) B = (A + - A)
 9-6RE1 (10) B = 0
 1-10CP (11) ((A + B) = A)->(B = 0)

!*ENTS (1) *(A+B=A)->(B=0)\$

CORRECT...
 NAME: *TH12

Theorem 14

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: -(A+B)=A->B=-A\$
 PROVE -((A + B) + B) = - A

!*FINS

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: -(A+B)+B=-A\$
 PROVE (-(A + B) + B) = - A

!*ENTS (1) *-(A+B)+B=-A\$

CORRECT...
 NAME: *TH14

Theorem 15

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: $\neg(A+B)=(\neg A)-B$
 PROVE $\neg(A+B)=(\neg A)-B$

!*HELPS

THIS $(A+B=C) \rightarrow (A=C-B)$
 ASS- $(A+B)$
 C11- A
 B11B (1) $((\neg(A+B)+B) = -A) \rightarrow (\neg(A+B) = (\neg A - B))$
 TH14 $(\neg(A+B)+B = -A)$
 A11A
 B11B (2) $(\neg(A+B)+B) = -A$
 I12AA (3) $\neg(A+B) = (\neg A - B)$

!*ENTS (1) $\neg(A+B)=(\neg A)-B$

CORRECT...
 NAME: *THIS

Theorem 17

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: $\neg(A-B)=B-A$
 PROVE $\neg(A-B)=(B-A)$

!*HELPS

TH16 $(\neg A)-B = (\neg B)-A$
 A11A
 B11B (1) $(\neg A - B) = (\neg B - A)$
 N A $+(\neg B) = A - B$
 A11B
 B11A (2) $(B + -A) = (B - A)$
 TH13 $-A = A$
 A11B (3) $- = B - B$
 3CE1 (4) $B = - - B$
 2-4RE1 (5) $(B + -A) = (\neg B - A)$
 5CE1 (6) $(\neg B - A) = (B + -A)$
 1-6RE1 (7) $(\neg A - B) = (B + -A)$
 TH15 $\neg(A+B) = (\neg A)-B$
 B11B
 A11A (8) $\neg(A + -B) = (\neg A - B)$
 5CE1 (9) $(\neg A - B) = \neg(A + B)$
 7-9RE1 (10) $\neg(A + -B) = (B - A)$
 10-8RE1 (11) $\neg(A + -B) = (B - A)$
 N A $+(\neg B) = A - B$
 A11A
 B11B (12) $(A + -B) = (A - B)$
 11-12RE1 (13) $\neg(A - B) = (B - A)$

!*ENTS (1) $\neg(A-B)=(B-A)$

CORRECT...
 NAME: *TH17

Theorem 16

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: $(\neg A)-B = (\neg B)-A$
 PROVE $(\neg A - B) = (\neg B - A)$

!*HELPS

THIS $(A+B=C) \rightarrow (A=C-B)$
 ASS- A - B
 C11- B
 B11A (1) $((\neg(A-B)+A) = -B) \rightarrow ((\neg(A-B) = (\neg B - A))$
 TH14 $(\neg(A+B)+B = -A)$
 A11B
 B11A (2) $(\neg(B+A)+A) = -B$
 TH15 $\neg(A+B) = (\neg A)-B$
 A11B
 B11A (3) $\neg(B+A) = (\neg B - A)$
 CA A + B = B + A
 A11- B
 B11- A (4) $(\neg B + -A) = (\neg A + -B)$
 N A $+(\neg B) = A - B$
 A11- B
 B11A (5) $(\neg B + -A) = (\neg B - A)$
 4-5RE1 (6) $(\neg B - A) = (\neg A + -B)$
 3-6RE1 (7) $(\neg B + A) = (\neg A + -B)$
 2-7RE1 (8) $((\neg A + -B) + A) = -B$
 N A $+(\neg B) = A - B$
 A11- A
 B11B (9) $(\neg A + -B) = (\neg A - B)$
 8-9RE1 (10) $((\neg A - B) + A) = -B$
 1-10AA (11) $(\neg A - B) = (\neg B - A)$

N A $+(\neg B) = A - B$
 A11- B
 B11A (1) $(\neg B + -A) = (\neg B - A)$
 TH15 $\neg(A+B) = (\neg A)-B$
 A11A
 B11B (2) $\neg(A+B) = (\neg A - B)$
 CA A + B = B + A
 A11- A
 B11- B (3) $(\neg A + -B) = (\neg B + -A)$
 N A $+(\neg B) = A - B$
 A11- A
 B11B (4) $(\neg A + -B) = (\neg A - B)$
 3-4RE1 (5) $(\neg A - B) = (\neg B + -A)$
 2-5RE1 (6) $\neg(A+B) = (\neg B - A)$
 2-6RE1 (7) $(\neg B + -A) = (\neg A - B)$
 1-7RE1 (8) $(\neg A - B) = (\neg B - A)$

N A $+(\neg B) = A - B$
 A11- B
 B11A (1) $(\neg B + -A) = (\neg B - A)$
 CA A + B = B + A
 A11- B
 B11- A (2) $(\neg B + -A) = (\neg A + -B)$
 N A $+(\neg B) = A - B$
 A11- A
 B11B (3) $(\neg A + -B) = (\neg A - B)$
 2-3RE1 (4) $(\neg B + -A) = (\neg A - B)$
 1-4RE1 (5) $(\neg A - B) = (\neg B - A)$
 *ENTS (1) $\neg(A-B) = (\neg B)-A$

CORRECT...
 NAME: *TH16

Theorem 18

TYPE A DERIVE, PROVE OR RULE COMMAND

!PROVE: (A-B)-C=A+((-B)-C)
 PROVE ((A - B) - C)=(A + (- B - C))

!HELPS

AS (A + B) + C = A + (B + C)
 AS: A
 BS: B
 CS: C (1) ((A + B) + C)=(A + (B + C))
 N A + (- B) = A - B
 AS: A + - B
 BS: C (2) ((A + - B) + C)=(A + (- B + C))
 1-3REI (3) ((A + - B) - C)=(A + (- B + - C))
 N A + (- B) = A - B
 AS: A
 BS: B (4) (A + B) = (A - B)
 3-4REI (5) ((A - B) - C)=(A + (- B - C))
 N A + (- B) = A - B
 AS: B
 BS: C (6) (- B + - C) = (- B - C)
 5-6REI (7) ((A - B) - C) = (A + (- B - C))

!ENTS (1) *(A-B)-C=A+((-B)-C)

CORRECT...
 NAME: *THIS

Theorem 20

TYPE A DERIVE, PROVE OR RULE COMMAND

!PROVE: A+(B-A)=B
 PROVE (A + (B - A)) = B

!HELPS

CA A + B = B + A
 BS: B + A
 AS: A (1) (A + (B + - A)) = ((B + - A) + A)
 AS (A + B) + C = A + (B + C)
 CS: A
 BS: - A
 AS: B (2) ((B + - A) + A) = (B + (- A + A))
 1-2REI (3) (A + (B + - A)) = (B + (- A + A))
 CA A + B = B + A
 BS: A
 AS: - A (4) (- A + A) = (A + - A)
 3-4REI (5) (A + (B + - A)) = (B + (A + - A))
 AS: A + (- A) = 0
 AS: A (6) (A + - A) = 0
 5-6REI (7) (A + (B + - A)) = (B + 0)
 AS: A + 0 = A
 AS: B (8) (B + 0) = B
 7-8REI (9) (A + (B + - A)) = B
 N A + (- B) = A - B
 AS: B
 BS: A (10) (B + - A) = (B - A)
 9-10REI (11) (A + (B - A)) = B

!ENTS (1) *A+(B-A)=B

CORRECT...
 NAME: *THS05

Theorem 19

TYPE A DERIVE, PROVE OR RULE COMMAND

!PROVE: (A-B)-C=A-(B+C)
 PROVE ((A - B) - C)=(A - (B + C))

!HELPS

THIS (A - B) - C = A + ((- B) - C)
 AS: A
 BS: B
 CS: C (1) ((A - B) - C) = (A + (- B - C))
 AS (A + B) + C = A + (B + C)
 AS: A
 BS: B
 CS: C (2) ((A + B) + C) = (A + (B + C))
 BS: C (3) (A + (- B + C)) = ((A + - B) + C)
 N A + (- B) = A - B
 AS: - B
 BS: C (4) (- B + C) = (- B + C)
 3-4REI (5) (A + (- B + C)) = ((A + - B) + C)
 1-5REI (6) ((A - B) - C) = ((A + - B) + - C)
 1-6REI (7) ((A + - B) + - C) = (A + (- B - C))
 THIS -(A + B) = (- A) - B
 BS: C
 AS: B (8) -(B + C) = (- B - C)
 BS: C (9) (- B - C) = -(B + C)
 7-9REI (10) ((A + - B) + - C) = (A + -(B + C))
 N A + (- B) = A - B
 AS: A + - B
 BS: C (11) ((A + - B) + - C) = ((A + - B) - C)
 10-11REI (12) ((A + - B) - C) = (A + -(B + C))
 N A + (- B) = A - B
 AS: A
 BS: B (13) (A + - B) = (A - B)
 12-13REI (14) ((A - B) - C) = (A + -(B + C))
 N A + (- B) = A - B
 AS: A
 BS: B + C (15) (A + -(B + C)) = (A - (B + C))
 14-15REI (16) ((A - B) - C) = (A - (B + C))

!ENTS (1) *(A-B)-C=A-(B+C)

CORRECT...
 NAME: *THIS

Theorem 21

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: A-(A+B)=-B
 PROVE (A-(A+B))=-B

!*HELPS

```

AS (A+B)+C = A+(B+C)
C:=B
B:=A
A:=A (1) ((A+(-A))+(-B))=(A+(-A+(-B)))
ICEI (2) (A+(-A+(-B)))=(A+(-A))+(-B)
CA A+B = B+A
A:=A+(-A)
B:=B (3) ((A+(-A))+(-B))=(-B+(A+(-A)))
2-3REI (4) (A+(-A+(-B)))=(-B+(A+(-A)))
AI A+(-A)=0
A:=A (5) (A+(-A))=0
4-5REI (6) (A+(-A+(-B)))=(-B+0)
Z A+0 = A
A:=B (7) (-B+0)=-B
6-7REI (8) (A+(-A+(-B)))=-B
N A+(-B)=A-B
A:=A
B:=B (9) (-A+(-B))=(-A-B)
8-9REI (10) (A+(-A-B))=-B
THIS -(A+B)=(-A)-B
B:=B
A:=A (11) -(A+B)=(-A-B)
11CEI (12) (-A-B)=-A-B
10-12REI (13) (A+(-A-B))=-B
N A+(-B)=A-B
A:=A
B:=A+B (14) (A+(-A+B))=(A-(A+B))
13-14REI (15) (A-(A+B))=-B
    
```

!*ENTS (1) *(A-(A+B))=-B

CORRECT...

NAME: *THS15

Theorem 22

TYPE A DERIVE, PROVE OR RULE COMMAND

!*PROVE: (A-B)+(B-C)=A-C
 PROVE ((A-B)+(B-C))=(A-C)

!*HELPS

```

LT
A:=A+(-B)+(B+(-C))
(1) ((A+(-B)+(B+(-C)))=(A+(-B)+(B+(-C)))
IARS (2) ((A+(-B)+(B+(-C)))=(A+(-B)+(B+(-C)))
CA A+B = B+A
B:=B+(-C)
A:=B (3) (-B+(B+(-C)))=(-B+(-C)+(-B))
CA A+B = B+A
B:=C
A:=B (4) (B+(-C))=(-C+B)
3-4REI (5) (-B+(B+(-C)))=(-C+B)+(-B)
AS (A+B)+C = A+(B+C)
C:=B
B:=B
A:=C (6) ((-C+B)+(-B))=(-C+(B+(-B)))
5-6REI (7) (-B+(B+(-C)))=(-C+(B+(-B)))
AI A+(-A)=0
A:=B (8) (B+(-B))=0
7-8REI (9) (-B+(B+(-C)))=(-C+0)
Z A+0 = A
A:=C (10) (-C+0)=-C
9-10REI (11) (-B+(B+(-C)))=-C
8-11REI (12) ((A+(-B)+(B+(-C)))=(A+(-C)))
N A+(-B)=A-B
A:=A
B:=B (13) (A+(-B))=(A-B)
12-13REI (14) ((A-B)+(B+(-C)))=(A+(-C))
N A+(-B)=A-B
A:=B
B:=C (15) (B+(-C))=(B-C)
14-15REI (16) ((A+(-B)+(B+(-C)))=(A+(-C)))
N A+(-B)=A-B
A:=A
B:=C (17) (A+(-C))=(A-C)
16-17REI (18) ((A-B)+(B-C))=(A-C)
    
```

!*ENTS (1) *(A-B)+(B-C)=A-C

CORRECT...

NAME: *THS25

Problem A

TYPE A DERIVE, PROVE OR RULE COMMAND

```
*DERIVE: (A+B)-(B+A)=0$
DERIVE ((A + B) -(B + A))= 0
```

*HELPS

```
AI A +(- A)= 0
A1:A + B (1) ((A + B) + -(A + B))= 0
N A +(- B)= A - B
B1:A + B
A1:A + B (2) ((A + B) + -(A + B))=((A + B) -(A + B))
1-2RE1 (3) ((A + B) -(A + B))= 0
CA A + B = B + A
A1:A
B1:B (4) (A + B)=(B + A)
3-4RE2 (5) ((A + B) -(B + A))= 0
```

*ENTS (1) *(A+B)-(B+A)=0\$

CORRECT...

Problem C

TYPE A DERIVE, PROVE OR RULE COMMAND

```
*DERIVE: B+(A+C)=A+(B+C)$
DERIVE (B +(A + C))=(A +(B + C))
```

*HELPS

```
AS (A + B)+ C = A +(B + C)
A1:A
B1:B
C1:C (1) ((A + B) + C)=(A +(B + C))
AS (A + B)+ C = A +(B + C)
A1:B
B1:A
C1:C (2) ((B + A) + C)=(B +(A + C))
AS (A + B)+ C = A +(B + C)
A1:B
B1:A
C1:C (3) ((B + A) + C)=(B +(A + C))
3AL3 (4) ((B + A) + C)=(B + A) + C)
4CA3 (5) ((B + A) + C)=(A + B) + C)
5ARA (6) ((B + A) + C)=(A +(B + C))
2-6RE1 (7) (A +(B + C))=(B +(A + C))
1-7RE1 (8) ((A + B) + C)=(B +(A + C))
1-8RE1 (9) (B +(A + C))=(A +(B + C))
```

LT

```
A1:A +(B + C)
(1) (A +(B + C))=(A +(B + C))
2CA1 (2) ((B + C) + A)=(A +(B + C))
2ARB (3) (B +(C + A))=(A +(B + C))
CA A + B = B + A
A1:C
B1:A (4) (C + A)=(A + C)
3-4RE1 (5) (B +(A + C))=(A +(B + C))
```

*ENTS (1) *B+(A+C)=A+(B+C)\$

CORRECT...

Problem B

TYPE A DERIVE, PROVE OR RULE COMMAND

```
*DERIVE: (A+(-A))+(B+(-B))=0$
DERIVE ((A + - A) +(B + - B))= 0
```

*HELPS

```
LT
A1:(A + - A)+(B + - B)
(1) ((A + - A) +(B + - B))=((A + - A) +(B + - B))
AI A +(- A)= 0
A1:A (2) (A + - A)= 0
1-2RE2 (3) ((A + - A) +(B + - B))=(0 +(B + - B))
AI A +(- A)= 0
A1:B (4) (B + - B)= 0
3-4RE2 (5) ((A + - A) +(B + - B))=(0 + 0)
Z A + 0 = A
A1:0 (6) (0 + 0)= 0
5-6RE1 (7) ((A + - A) +(B + - B))= 0
```

*ENTS (1) *(A+(-A))+(B+(-B))=0\$

CORRECT...

Problem D

TYPE A DERIVE, PROVE OR RULE COMMAND

```
*DERIVE: (A+B)+(C+D)=(D+C)+(B+A)$
DERIVE ((A + B) +(C + D))=((D + C) +(B + A))
```

*HELPS

```
LT
A1:(D + C)+(B + A)
(1) ((D + C) +(B + A))=((D + C) +(B + A))
CA A + B = B + A
A1:B
B1:A (2) (B + A)=(A + B)
1-2RE1 (3) ((D + C) +(A + B))=((D + C) +(B + A))
CA A + B = B + A
A1:D + C
B1:A + B (4) ((D + C) +(A + B))=((A + B) +(D + C))
CA A + B = B + A
A1:D
B1:C (5) (D + C)=(C + D)
4-5RE2 (6) ((D + C) +(A + B))=((A + B) +(C + D))
3-6RE1 (7) ((A + B) +(C + D))=((D + C) +(B + A))
```

*ENTS (1) *A<A>(A+B)+(C+D)=(D+C)+(B+A)\$

CORRECT...

TYPE A DERIVE, PROVE OR RULE COMMAND

*FINS

T

*

APPENDIX III

THE THEOREM-PROVING ALGORITHM

The theorem-proving algorithm given in this appendix constructs the rule sequence $\Phi_{[k]}$ such that $\Phi_{[k]}$ is a proof or derivation of the pattern of some well-formed expression E ; if no such $\Phi_{[k]}$ can be found, the algorithm returns the value ϕ . Although not written specifically in each appropriate step, we assume that the pattern sequence $\Psi_{[k]}$ is constructed at the same time as Ψ_k .

The subroutine, SUBGOAL, and the main routine, PROVER, are presented with no further explanation since it is assumed that the interested reader is familiar with the step-by-step execution of an ALGOL-like program. We clearly took some liberty in writing English statements to describe several steps.

The algorithm refers to the following global variables with their corresponding values.

1. AX is the set of patterns for the axioms of the theory.
2. PREMISE is the set of patterns that may be chosen as assumptions.
3. SUB is the set of attempted goals (subgoals). Once added to SUB, no goal is deleted from SUB since, if a goal in SUB is already proved, the algorithm should find it in the pattern sequence in steps 1 of SUBGOAL and 2 of PROVER.
4. LIMIT is an integer, the upper limit on the allowed depth of recursive calls of PROVER.

The variables j , k , n , IN, I, G, and SG are local variables saved with respect to each routine called.

PROVER($\Phi_{[n]}, G$)

Step 1. Set $SUB = SUB \cup \{G\}$.

Step 2. If there exists an $i \leq n$ such that $\Psi_i = G$, then

RETURN $\Phi_{[n]} * REP(i)$.

Step 3. If $\lambda(0)$ is = and there exists an $i \leq n$ such that $S(\Psi_i, 1)$

$= S(G, 1 + \#S(G, 1))$ and $S(\Psi_i, 1 + \#S(\Psi_i, 1)) = S(G, 1)$, then

RETURN $\Phi_{[n]} * CE(i)$.

Step 4. If there exists a $P \in AX$ and a substitution sequence σ such

that $P[\sigma]$ is well-defined and $G = P[\sigma]$, then

RETURN $\Phi_{[n]} * SUB(\sigma, P)$.

Step 5. If there exists a $P \in PREMISE$ such that $P = G$, then

RETURN $\Phi_{[n]} * AR(P)$.

Step 6. Set level = 0.

6.1 Set $IN = \{I \mid I \text{ is an initial pattern for } G\}$.

6.2 If $IN = \phi$, then go to step 7.

6.3 Set $I = \text{first element of } IN$.

Set $IN = IN - \{I\}$.

Set $\Phi_{[n+1]}$ such that $\Phi_{[n+1]}$ is obtained from $\Phi_{[n]}$ by IRL,

IR3, IR5, or IR6 where $\Psi_{n+1} = I$.

6.4 Set $D = D(I, G)$.

Set $j = \min(\#G, \#I) + 1$.

6.5 Set $j = j - 1$.

6.6 If $j \leq 0$, then go to step 6.2.

6.7 If $j \notin D$ or $SG(I, G, j) \in SUB$, then go to step 6.5.

6.8 Set $\Phi_{[k]} = SUBGOAL(\Phi_{[n+1]}, SG(I, G, j))$.

- 6.9 If $\Phi_{[k]} = \phi$, then go to step 6.5.
- 6.10 Set $\Phi_{[k+1]} = \Phi_{[k]} * RE(n+1, k, j)$.
- 6.11 If $\Psi_{k+1} = G$, then RETURN $\Phi_{[k+1]}$.
- 6.12 Set $I = \Psi_{k+1}$.
- 6.13 Go to step 6.4.
- Step 7. If $\mathcal{L}(0) \neq \Rightarrow$, then go to step 8.
- 7.1 Set $\Phi_{[n+1]} = \Phi_{[n]} * AR(S(G, 1))$.
- 7.2 Set $\Phi_{[k]} = PROVER(\Phi_{[n+1]}, S(G, 1 + \#S(G, 1)))$.
- 7.3 If $\Phi_{[k]} = \phi$, then go to step 8.
- 7.4 RETURN $\Phi_{[k]} * CP(n+1, k)$.
- Step 8. Set $DE = \{I \mid \mathcal{L}(0) \text{ is } \Rightarrow; \text{ and } S(I, 1 + S(I, 1)) \text{ is an initial pattern for } G; \text{ and } I \in \text{PREMISE, or } I = \Psi_i, i \leq n; \text{ or } I = P[\sigma], P \in AX, P[\sigma] \text{ well-defined}\}$.
- 8.1 If $DE = \phi$, then go to step 9.
- 8.2 Set $I = \text{first element of } DE$.
- Set $DE = DE - \{I\}$.
- Set $\Phi_{[n+1]}$ such that $\Phi_{[n+1]}$ comes from $\Phi_{[n]}$ by IR1, IR5, or IR6 where $\Psi_{n+1} = I$.
- Set $\Phi_{[k]} = PROVER(\Phi_{[n+1]} + S(I, 1))$.
- 8.3 If $\Phi_{[k]} = \phi$, then go to step 8.1.
- 8.4 Set $\Phi_{[k+1]} = \Phi_{[k]} * DE(n+1, k)$.
- 8.5 If $\Psi_{k+1} = G$, then RETURN $\Phi_{[k+1]}$.
- 8.6 Set $\Phi_{[j]} = PROVER(\Phi_{[k+1]}, G)$.
- 8.7 If $\Phi_{[j]} = \phi$, then go to step 8.1.
- 8.8 RETURN $\Phi_{[j]}$.

- Step 9. Set $G' = G$.
- 9.1 Set $j = 1$.
 - 9.2 If $j > \#G'$, then go to step 9.14.
 - 9.3 Set $k = 1$.
 - 9.4 If $k > \#G'$, then go to step 9.10.
 - 9.5 If $S(G',j)$ is identical to $S(G',k)$, then go to step 9.8.
 - 9.6 Set $\phi_{[m]} = \text{SUBGOAL}(\wedge, (S(G',j) = S(G',k)))$.
 - 9.7 If $\phi_{[m]} \neq \phi$, then go to step 9.12.
 - 9.8 Set $k = k + 1$.
 - 9.9 Go to step 9.4.
 - 9.10 Set $j = j + 1$.
 - 9.11 Go to step 9.2.
 - 9.12 Set $G' = G'[j, S(G',k)]$.
 - 9.13 Go to step 9.1.
 - 9.14 If G' is identical to G , then RETURN ϕ .
 - 9.15 Set $\phi_{[j]} = \text{PROVER}(\phi_{[n]}, G')$.
 - 9.16 If $\phi_{[j]} = \phi$, then RETURN ϕ .
 - 9.17 Set $I = G$.
 - 9.18 Go to step 6.4.

SUBGOAL($\Phi_{[n]}$, SG)

- Step 1. If there exists an $i \leq n$ such that $\Psi_i = SG$, then
 RETURN $\Phi_{[n]} * \text{REP}(i)$.
- Step 2. If $\ell(0)$ is = and there exists an $i \leq n$ such that $S(\Psi_i, 1) = S(SG, 1 + \#S(SG, 1))$ and $S(\Psi_i, 1 + \#S(\Psi_i, 1)) = S(SG, 1)$, then
 RETURN $\Phi_{[n]} * \text{CE}(i)$.
- Step 3. If there exists a $P \in AX$ and a substitution sequence σ such that $P[\sigma]$ is well-defined and $SG = P[\sigma]$, then
 RETURN $\Phi_{[n]} * \text{SUB}(\sigma, P)$.
- Step 4. If there exists a $P \in \text{PREMISE}$ such that $P = SG$, then
 RETURN $\Phi_{[n]} * \text{AR}(P)$.
- Step 5. If $\text{level} \geq \text{limit}$ or $SG \in \text{SUB}$, then go to step 17.
- Step 6. Set $\text{level} = \text{level} + 1$.
 Set $\text{SUB} = \text{SUB} \cup \{SG\}$.
 Set $\text{IN} = \{I \mid I \text{ is an initial pattern for } SG\}$.
- Step 7. If $\text{IN} = \emptyset$, then go to step 17.
- Step 8. Set $I = \text{first element of } \text{IN}$.
 Set $\text{IN} = \text{IN} - \{I\}$.
 Set $\Phi_{[n+1]}$ such that $\Phi_{[n+1]}$ obtained from $\Phi_{[n]}$ by IR1, IR3, IR5, or IR6 where $\Psi_{n+1} = I$.
- Step 9. Set $D = D(I, SG)$.
 Set $j = \min(\#SG, \#I) + 1$.
- Step 10. Set $j = j - 1$.
- Step 11. If $j \leq 0$, then go to step 7.
- Step 12. If $j \notin D$ or $SG(I, SG, j) \in \text{SUB}$, then go to step 10.

- Step 11. If $j \leq 0$, then go to step 7.
- Step 12. If $j \notin D$ or $SG(I, SG, j) \in SUB$, then go to step 10.
- Step 13. Set $\phi_{[k]} = SUBGOAL(\phi_{[n+1]}, SG(I, G, j))$.
- Step 14. If $\phi_{[k]} = \phi$ or $\psi_k \neq SG(I, SG, j)$, then go to step 10.
- Step 15. Set $level = level - 1$.
- Step 16. RETURN $\phi_{[k]} * RE(n + 1, k, j)$.
- Step 17. Set $level = level - 1$.
- Step 18. RETURN ϕ .

APPENDIX IV

MATERIAL FOR THE STUDY OF EXPERIENCED HUMAN TUTORS

Practice Exercises

1. DERIVE $(8+0 = 8+C) \rightarrow (C = 0)$

2. DERIVE $((-(A+C))+A)+C = 0$

3. DERIVE $A = 5+ -3$

P (1) $A+3 = 5$

Instructions

Suppose you are tutoring a student on the logic program. At a certain point, the student asks your assistance on a problem he got stuck with, either before having typed in any lines, or after a few lines. Your response to him may be one of several types.

a. You may make an encouraging remark, such as: "Well, can't you think of something?" or "I'm sure you can see what to do now."

b. You may ask him to give you some information about what he was doing, such as: "What made you type line x?" or "Have you used axiom y?" or "Why did you start with line z?"

c. You may actually make a helpful suggestion, such as: "Try using rule x" or "Note the similarity between y and z."

You will receive 8 problems, each of which has a (possibly empty) set of lines already in the proof. A student was given the derivation problem. He generated all proof lines shown on the page. Your task is to help the student successfully complete the problem, from the point at which you see the command HELP. Please write down what you would say to the student as

a first comment or hint. Be concise; for example, write down a comment which you would not mind having a teletypewriter type out to the student. I am not interested in remarks of types a or b, only in remarks of type c, i.e., what would be the first piece of information you would offer the student. Also, write down everything you think of which helped you decide what to say. Try to answer such questions as: What would you say to help the student? Why did you say it? What did you do in order to make your decision?

Please do not hesitate to write down any tedious details you think has something to do with making your decision. If there is some comment or question you think you would say to the student after he has responded to your first comment, go ahead and give it.

Warm-up Exercise

Student knows theorems 1-5 as well as the axioms.

DERIVE $A-0 = A$

N $A+-B=A-B$

A::A

B::0 (1) $A+(-0) = A-0$

TH5 $0=-0$

(2) $0=-0$

CEL (3) $-0=0$

1.3REL (4) $A+0 = A-0$

HELP

Partial Solutions

1. Student knows only the axioms

DERIVE $(A=0)$ $(A+B=B)$

P (1) $0+B = B$

CA $A+B=B+A$

A::A

B::B (2) $A+B = B+A$

WP (3) $A = 0$

2.3RE2 (4) $A+B = B+0$

HELP

2. Student knows only the axioms

```

DERIVE (C+E)+B = C+(E+(4+2))
P (1) B = 4+2
IAE
A:: C+E (2) B+(C+E) = (4+2)+(C+E)
HELP

```

3. Student knows only the axioms

```

DERIVE (A+B)+(C+D) = A+(B+(C+D))
CA A+B=B+A
A:: A+B
B:: C+D (1) (A+B)+(C+D) = (C+D)+(A+B)
HELP

```

4. Student knows the axioms and theorems 1-2

```

DERIVE 0-A = -A
IT
:: -A (1) -A = -A
TH2 (-A)+A = 0
A:: A (2) (-A)+A = 0
TH1 0+A = A
A:: -A (3) 0+(-A) = (-A)
3CEL (4) (-A) = 0+(-A)
4.1REL (5) (-A) = 0+(-A)
HELP

```

5. Student knows the axioms and theorems 1-4

```

DERIVE 0 = -0
AI A+(-A)=0
A:: A (1) A+(-A)=0
N A+(-B)=A-B
A:: A
B:: A (2) A+(-A) = A-A
1.2REL (3) A-A = 0
HELP

```

6. Student knows axioms and theorems 1-5

```

DERIVE (-5)+5 = (-0)
TH5 0=-0
(1) 0=-0
TH2 (-A)+A = 0
A:: 5 (2) (-5)+5 = 0
CEL (3) 0 = (-5)+5
3.1REL (4) (-0) = (-5)+5
HELP

```

7. Student knows the axioms and theorems 1-6

```

DERIVE 5+0 = 5-0
N A+(-B) = A-B
A::5
B::0 (1) 5+(-0) = 5-0
TH6 A-0 = A
A::5 (2) 5-0 = 5
HELP

```

8. Student knows the axioms and theorems 1-6

```

DERIVE 0-(-0) = 0
TH6 A-0=A
A::-0 (1) (-0)-0 = (-0)
TH6 A-0=A
A::0 (2) 0-0 = 0
TH5 0=-0
(3) 0=-0
3CE1 (4) (-0)=0
1.4RE1 (5) 0-0 = (-0)
1.4RE2 (6) (-0)-0 = 0
HELP

```

APPENDIX V

SUMMARY OF THE COMMAND LANGUAGE PRESENTED IN CHAPTER VI

Our convention will be to underline items typed by the user; all other information is typed by the program. Each user command is terminated by an ENTER key which types as a dollar sign (\$). In most cases, we will not show the ENTER key. Note that the user can type an empty answer (ENTER key) in order to escape from any command sequence.

In the following, α is an individual variable, β is a term, and ϕ, ψ, ψ' are well-formed formulas (WFF).

INSTANCES OF AXIOMS AND ESTABLISHED THEOREMS

1. Proper substitution of a term for a variable.

<axiom, lemma, or theorem name>
<variable>:: <term of instantiation>
.
.
<variable>:: <term of instantiation>

The substitution sequence continues for each universally quantified variable whose scope is the entire formula in the axiom, lemma, or theorem. Substitution is carried out simultaneously.

2. Proper substitution for predicate letters and of terms for variables.

PS : <name of axiom, lemma, or theorem>
:: <variable> : <well-formed term>
:: <predicate letter> : <WFF>
.
.
.

The substitution sequence continues until the user types the ENTER key without one of the two possible substitution pairs. The substitutions are carried out iteratively.

PROOF PROCEDURES

1. WP Working premise
WP (i) $\langle \underline{WFF} \rangle$

2. CP Conditional proof
WP (i) ϕ
 (j) ψ
i.jCP (n) $\phi \rightarrow \psi$

3. IP Indirect proof (reductio ad absurdum)
WP (i) ϕ
 (j) ψ
 (k) NOT ψ
i.j.kIP (n) NOT ϕ

4. Introduce a variable for universal generalization
GEN: $\langle \text{variable} \rangle$
 OK If the variable does not occur free in any antecedent lines, the program types 'OK'; otherwise, an error message is given.

5. UG Universal generalization
GEN: α or, alternatively:

OK	(i) $\phi(\alpha)$	
	(i) $\phi(\alpha)$	$\frac{iUG}{:\underline{\alpha}}$
$\frac{iUG}{:\underline{\alpha}}$	(n) $\forall\alpha\phi(\alpha)$	(n) $\forall\alpha\phi(\alpha)$

This version of UG need only
check the last GEN introduced.

This version of UG requires
checking for α free in any
antecedent line.

PRIMITIVE RULES OF INFERENCE

1. AA Affirm the antecedent (Modus Ponens)

(i) $\phi \rightarrow \psi$

(j) ϕ

$\frac{i.jAA}{\psi}$ (n) ψ

2. Quantification rules

ES Existential specification

(i) $\exists\alpha\phi(\alpha)$

$\frac{iES}{\alpha:\beta}$

(n) $\phi(\beta)$ where β must be new (or have been
introduced by us).

EG Existential generalization

(i) $\phi(\beta)$

$\frac{iEG}{\alpha:\beta}$

(n) $\exists\alpha\phi(\alpha)$ where substitute α for each free occurrence
of β referenced.

US Universal specification

(i) $\forall\alpha\phi(\alpha)$

iUS $\alpha :: \beta \quad (n) \quad \phi(\beta)$

3. Logic of identity

IDC and IDS

(i) $\phi(\beta)$ iIDC(sequence of occurrence numbers) $\alpha : \beta \quad (m) \quad \forall \alpha (\alpha = \beta \rightarrow \phi(\alpha))$ where substitute α for each occurrence of β referenced.m IDS (n) $\phi(\beta)$

4. Interchange rules RE and RQ

(i) $\phi(\alpha)$ (j) $\alpha = \beta$ i.jRE(occurrence number)(n) $\phi(\beta)$ where replace β for the free occurrence of α referenced.(o) $\phi(\psi)$ (p) $\psi \rightarrow \psi'$ i.jRQ(occurrence number)(q) $\phi(\psi')$ where replace ψ for the occurrence of ψ' referenced.

5. Generalized interchange rules--short forms of axioms and theorems.

<line number> <axiom, lemma, or theorem name> <occurrence number><sequence of line numbers> <axiom, lemma, or theorem name>

MISCELLANEOUS COMMANDS

1. Delete the last line DLL

<line number> DLL

All lines other than premises, beginning with \langle line number \rangle and continuing to the last line generated, are deleted. If IP and CP lines are deleted, the subsidiary derivation is no longer considered closed.

2. REVIEW the derivation problem

REVIEW

This command is a request to have the partially completed proof re-typed, letting the student get a clean copy of his work.

3. Use of mechanical theorem-provers

SHOW is described in Chapter VI.

HELP is explained in Chapters I-V.

4. Obtain the initiative to request problems at runtime.

INIT

The student can now type one of the following commands:

a. derive problems

DERIVE: \langle WFF \rangle

P (1) \langle WFF \rangle

The student can enter any number of

P (2) \langle WFF \rangle

premise lines, continuing until he

enters a command different from P or DLI.

b. prove problems

PROVE: \langle WFF \rangle

NAME:: \langle alphabetic string \rangle

After the proof is completed, the student may assign a name to be associated with the WFF. This name becomes part of the student's available command language.

- c. derive a new rule of inference

RULE : <name of a new rule>

FROM: <axiom, lemma, or theorem name>

The new rule is derived from the axiom, lemma, or theorem.

If the rule exists already, the student can add another form of the rule or delete the existing forms.

- d. Do a finding axioms exercise

FA

The student can type REPORT to learn how much of the problem has been completed, which expressions were named as axioms, which were proved as theorems, and which expressions entered into each proof. Typing DELETE deletes an axiom or theorem name.

DERIVED RULES OF INFERENCE

The general format for a derived rule of inference is:

<sequence of NOP line numbers> <name of the rule>.

The line numbers refer to lines of the derivation or proof which must match the corresponding premises of the rule. The premises are saved as patterns (under the name PATTERN) on the property list of each rule. RESTRICT options are written as LISP S-expressions. The procedure which processes derived rules evaluates these S-expressions in order to check for restrictions on the values of the variables or the requested expressions (REQ), or to (re-)compute substitution pairs for the substitution list, MAINSUB.