AUTHOR Goldberg, Adele
TITLE A Generalized Instructional System for Elementary Mathematical Logic.
INSTITUTION Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY National Science Foundation, Washington, D.C.
REPORT NO TR-179
PUB DATE 11 Oct 71
NOTE 96p.; Psychology and Education Series

EDRS PRICE MF-$0.65 HC-$3.29
DESCRIPTORS Calculus; *Computer Assisted Instruction; *Computer Programs; *Mathematical Logic; *Mathematics Instruction

ABSTRACT

A computer-based instructional system for teaching the notion of mathematical proof is described. The system is capable of handling formalizations of the full predicate calculus with identity and, with minor work, definite description. Designed as an instructional device, the program is also the basis for a number of research projects involving the use of mechanical theorem-provers for teaching theorem-proving. The entire system is presented here in detail: the program as written in the LISP programing language for a PDP-10 computer. Instructions on how to use the system for research and teaching, block diagrams of key program routines, and example curriculums are included. Enough detail is provided so that versions in other languages for other computer systems may be programed from the information presented here. (Author/JY)

A GENERALIZED INSTRUCTIONAL SYSTEM FOR ELEMENTARY MATHEMATICAL LOGIC

BY

ADELE GOLDBERG

TECHNICAL REPORT NO. 179

OCTOBER 11, 1971

PSYCHOLOGY & EDUCATION SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CALIFORNIA

TECHNICAL REPORTS

PSYCHOLOGY SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

(Place of publication shown in parentheses; if published title is different from title of Technical Report, this is also shown in parentheses.)

(For reports no. 1 - 44, see Technical Report no. 125.)

50 R. C. Atkinson and R. C. Calfee. Mathematical learning theory. January 2, 1963. (In B. B. Wolman (Ed.), Scientific Psychology. New York: Basic Books, Inc., 1965. Pp. 254-275)

51 P. Suppes, E. Crothers, and R. Weir. Application of mathematical learning theory and linguistic analysis to vowel phoneme matching in Russian words. December 28, 1962.

52 R. C. Atkinson, R. Calfee, G. Sommer, W. Jeffrey and R. Shoemaker. A test of three models for stimulus compounding with children. January 29, 1963. (J. exp. Psychol., 1964, 67, 52-58)

53 E. Crothers. General Markov models for learning with inter-trial forgetting. April 8, 1963.

54 J. L. Myers and R. C. Atkinson. Choice behavior and reward structure. May 24, 1963. (Journal math. Psychol., 1964, 1, 170-203)

55 R. E. Robinson. A set-theoretical approach to empirical meaningfulness of measurement statements. June 10, 1963.

56 E. Crothers, R. Weir and P. Palmer. The role of transcription in the learning of the orthographic representations of Russian sounds. June 17, 1963.

57 P. Suppes. Problems of optimization in learning a list of simple items. July 22, 1963. (In Maynard W. Shelly, II and Glenn L. Bryan (Eds.), Human Judgments and Optimality. New York: Wiley. 1964. Pp. 116-126)

58 R. C. Atkinson and E. J. Crothers. Theoretical note: all-or-none learning and intertrial forgetting. July 24, 1963.

59 R. C. Calfee. Long-term behavior of rats under probabilistic reinforcement schedules. October 1, 1963.

60 R. C. Atkinson and E. J. Crothers. Tests of acquisition and retention, axioms for paired-associate learning. October 25, 1963. (A comparison of paired-associate learning models having different acquisition and retention axioms, J. math. Psychol., 1964, 1, 285-315)

61 W. J. McGill and J. Gibbon. The general-gamma distribution and reaction times. November 20, 1963. (J. math. Psychol., 1965, 2, 1-18)

62 M. F. Norman. Incremental learning on random trials. December 9, 1963. (J. math. Psychol., 1964, 1, 336-351)

63 P. Suppes. The development of mathematical concepts in children. February 25, 1964. (On the behavioral foundations of mathematical concepts. Monographs of the Society for Research in Child Development, 1965, 30, 60-96)

64 P. Suppes. Mathematical concept formation in children. April 10, 1964. (Amer. Psychologist, 1966, 21, 139-150)

65 R. C. Calfee, R. C. Atkinson, and T. Shelton, Jr. Mathematical models for verbal learning. August 21, 1964. (In N. Wiener and J. P. Schoda (Eds.), Cybernetics of the Nervous System: Progress in Brain Research. Amsterdam, The Netherlands: Elsevier Publishing Co., 1965. Pp. 333-349)

66 L. Keller, M. Cole, C. J. Burke, and W. K. Estes. Paired associate learning with differential rewards. August 20, 1964. (Reward and information values of trial outcomes in paired associate learning. (Psychol. Monogr., 1965, 79, 1-21)

67 M. F. Norman. A probabilistic model for free-responding. December 14, 1964.

68 W. K. Estes and H. A. Taylor. Visual detection in relation to display size and redundancy of critical elements. January 25, 1965, Revised 7-1-65. (Perception and Psychophysics, 1966, 1, 9-16)

69 P. Suppes and J. Donio. Foundations of stimulus-sampling theory for continuous-time processes. February 9, 1965. (J. math. Psychol., 1967, 4, 202-225)

70 R. C. Atkinson and R. A. Kinchla. A learning model for forced-choice detection experiments. February 10, 1965. (Br. J. math stat. Psychol., 1965, 18, 184-206)

71 E. J. Crothers. Presentation orders for items from different categories. March 10, 1965.

72 P. Suppes, G. Groen, and M. Schlag-Rey. Some models for response latency in paired-associates learning. May 5, 1965. (J. math. Psychol., 1966, 3, 99-128)

73 M. V. Levine. The generalization function in the probability learning experiment. June 3, 1965.

74 D. Hansen and T. S. Rodgers. An exploration of psycholinguistic units in initial reading. July 6, 1965.

75 B. C. Arnold. A correlated urn-scheme for a continuum of responses. July 20, 1965.

76 C. Izawa and W. K. Estes. Reinforcement-test sequences in paired-associate learning. August 1, 1965. (Psychol. Reports, 1966, 18, 879-919)

77 S. L. Blehart. Pattern discrimination learning with Rhesus monkeys. September 1, 1965. (Psychol. Reports, 1966, 19, 311-324)

78 J. L. Phillips and R. C. Atkinson. The effects of display size on short-term memory. August 31, 1965.

79 R. C. Atkinson and R. M. Shiffrin. Mathematical models for memory and learning. September 20, 1965.

80 P. Suppes. The psychological foundations of mathematics. October 25, 1965. (Colloques Internationaux du Centre National de la Recherche Scientifique. Editions du Centre National de la Recherche Scientifique. Paris: 1967. Pp. 213-242)

81 P. Suppes. Computer-assisted instruction in the schools: potentialities, problems, prospects. October 29, 1965.

82 R. A. Kinchla, J. Townsend, J. Yellott, Jr., and R. C. Atkinson. Influence of correlated visual cues on auditory signal detection. November 2, 1965. (Perception and Psychophysics, 1966, 1, 67-73)

83 P. Suppes, M. Jerman, and G. Groen. Arithmetic drills and review on a computer-based teletype. November 5, 1965. (Arithmetic Teacher, April 1966, 303-309.

84 P. Suppes and L. Hyman. Concept learning with non-verbal geometrical stimuli. November 15, 1968.

85 P. Holland. A variation on the minimum chi-square test. (J. math. Psychol., 1967, 3, 377-413).

86 P. Suppes. Accelerated program in elementary-school mathematics -- the second year. November 22, 1965. (Psychology in the Schools, 1966, 3, 294-307)

87 P. Lorenzen and F. Binford. Logic as a dialogical game. November 29, 1965.

88 L. Keller, W. J. Thomson, J. R. Tweedy, and R. C. Atkinson. The effects of reinforcement interval on the acquisition of paired-associate responses. December 10, 1965. (J. exp. Psychol., 1967, 73, 268-277)

89 J. I. Yellott, Jr. Some effects on noncontingent success in human probability learning. December 15, 1965.

90 P. Suppes and G. Groen. Some counting models for first-grade performance data on simple addition facts. January 14, 1966. (In J. M. Scandura (Ed.), Research in Mathematics Education. Washington, D. C.: NCTM, 1967. Pp. 35-43.

91 P. Suppes. Information processing and choice behavior. January 31, 1966.

92 G. Groen and R. C. Atkinson. Models for optimizing the learning process. February 11, 1966. (Psychol. Bulletin, 1966, 66, 309-320)

93 R. C. Atkinson and D. Hansen. Computer-assisted instruction in initial reading: Stanford project. March 17, 1966. (Reading Research Quarterly, 1966, 2, 5-25)

94 P. Suppes. Probabilistic inference and the concept of total evidence. March 23, 1966. (In J. Hintikka and P. Suppes (Eds.), Aspects of Inductive Logic. Amsterdam: North-Holland Publishing Co., 1966. Pp. 49-65.

95 P. Suppes. The axiomatic method in high-school mathematics. April 12, 1966. (The Role of Axiomatics and Problem Solving in Mathematics. The Conference Board of the Mathematical Sciences, Washington, D. C. Ginn and Co., 1966. Pp. 69-76.

(Continued on inside back cover)

A GENERALIZED INSTRUCTIONAL SYSTEM FOR ELEMENTARY MATHEMATICAL LOGIC

by

Adele Goldberg

TECHNICAL REPORT NO. 179

October 11, 1971

PSYCHOLOGY AND EDUCATION SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CALIFORNIA

Acknowledgments

The effort expended in developing this instructional system was shared equally by Tobin Barrozo and myself. I would like to thank him for his endless patience. Note that Tobin provided the annotation to Appendix V.

We would also like to thank Leon Henkin for suggesting the original idea for the program, and Patrick Suppes for his counsel.

The research in this report was supported by National Science Foundation Grant GJ-443.

Table of Contents

PART I. INTRODUCTION

This report describes a computer-based instructional system for teaching the notion of mathematical proof. The system is capable of handling formalizations of the full predicate calculus with identity (and, with minor work, definite description). Designed as an instructional device (a course in number theory is presently being prepared), the program is also the basis for a number of research projects involving the use of mechanical theorem-provers for teaching theorem-proving. The intention here is to present, in detail, the entire system: the program as written in the LISP programming language [McCarthy, 1963] for a PDP-10 computer. Instructions on how to use the system for both research and teaching, block diagrams of key program routines, and example curriculums are included. The purpose of this report is to provide enough detail so that versions in other languages for other computer systems may be programmed from the information offered here. As a result, subsequent sections are dense with descriptions of particular routines of the instructional program. While current research on interfacing various theorem-proving programs are mentioned in this paper, they will be reported on in more detail elsewhere.

The underlying foundation of first-order logic of this instructional system, like the Institute for Mathematical Studies in the Social Sciences (IMSSS) logic and algebra program [Suppes & Binford, 1965; Suppes, Jerman & Brian, 1968, Appendix I; Suppes & Ihrke, 1970; Suppes, 1971], is a natural deduction treatment. The main thrust of both programs is to let the student construct proofs or derivations. The student has a simple command language made up of mnemonics that name axioms and theorems, rules of inference, and proof procedures. The user can type commands that reference specific lines of the proof or derivation and that specify occurrences of symbols and terms within a line. If the commands are correct, the program generates new lines of the proofs or derivations. By these means, the student can generate any line regardless of its relevance to finding the problem solution. The freedom to use commands means that the program rejects all ideas of right and wrong proofs; the student can follow any one of a number of solution paths. But this freedom means that the program is not aware of what the student is doing. It may mean, aside from prestored hints, that the program cannot offer the user (the student) any help in completing the derivation. In an attempt to solve this problem, a mechanical theorem-prover that generates solutions of the algebra problems was developed. The theorem-prover is employed as a proof-analyzer that examines incomplete derivations done by students and gives the students hints or advice on how to utilize their own partial proofs to arrive at complete ones. Details of this theorem-prover will appear in Goldberg [1971].

The core of this instructional program is a set of routines that permit the user to construct proofs or derivations. The program is a natural successor to the IMSSS logic and algebra program, and many of its features result from experience obtained in schools running that program. Unlike the logic and algebra program, this instructional system can be characterized as more than a proof checker. It is a more powerful interpretive system in which an individual, be he student, teacher, or researcher, can develop and then study a nonlogical axiomatic theory along whatever lines he himself specifies. The program allows one to build the command language for constructing proofs; 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. The program, which "knows" only primitive rules of predicate calculus, consists of routines that 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 the command language he built himself. 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.

Both the command language and a curriculum can be constructed by a teacher. The curriculum can be a course, say, in number theory or elementary axiomatic geometry. The student follows the curriculum, but he always has the option of interrupting the linear sequence of problems and making up his own problems. The student might avail himself of this feature of the program when (a) he wants to redo a problem to assure himself that he understands the proof procedures being introduced; (b) the problems are too hard and he wants to try some easier ones before continuing with the teacher's problems; (c) the problems are too easy and he wants to try more challenging ones; or (d) a problem requires a subsidiary derivation. The student may prove the formula and name it as a lemma, thereby simplifying the original problem by using that lemma. Here, the distinction between a teacher and a student becomes a narrow one, because the student himself can enlarge (or initially specify) the command language or invent a "curriculum" for himself and his classmates.

The principal advantages of this program are twofold. First, the generality and flexibility of the program provides the user with a sort of experimental laboratory in which to explore new ideas (or investigate old ones) connected with formalized theories. The student with some of his own notions about constructing formal systems can readily test them in a systematic and unambiguous manner: the program acts as a tireless proof checker. Second, CAI programs are usually based on a curriculum that is organized around strictly frame-by-frame, or branch-on-a-predetermined-algorithm, strategy. This mode of teaching can sometimes be too restrictive and somewhat less informative. We hoped to provide a freer structure in order to give the more innovative students access to those computational facilities available to the teacher.

The interpretive tools of the system "understand" two languages. The first, call it C, is the set of commands used in constructing proofs or derivations and is defined in M, the second language. A block diagram of the program is presented in Figure 1. With M, the user is able to AXIOMATIZE a theory, i.e., to specify

----

Insert Figure 1 about here

(a) the class of nonlogical constants, and (b) a class of axioms. It is assumed that the logical system is always the foundation on which new theories are built, but the symbols representing the logical constants are specified by the user. The names of axioms and the well-formed formulas derivable from the axioms (theorems and lemmas) are members of C. Also, M enables the user to derive rules of inference from axioms and from established theorems and lemmas. These new rules are added to C. Thus, C is further augmented by the basic logical system, which is outlined in Part III.

After the command language for constructing derivations and proofs is specified, the program takes on the role of a proof checker. This is the DERIVE part of the program. In the DERIVE stage, the user types a command and expects the program to generate a line of a derivation. This may involve either requests for substitution instances of axioms or previously proved theorems, or references to a set of previous lines in order to infer a new one. The program accepts the command, checks to see that its syntax is correct, and, in attempting to carry out the command, checks to make sure that the application is proper.

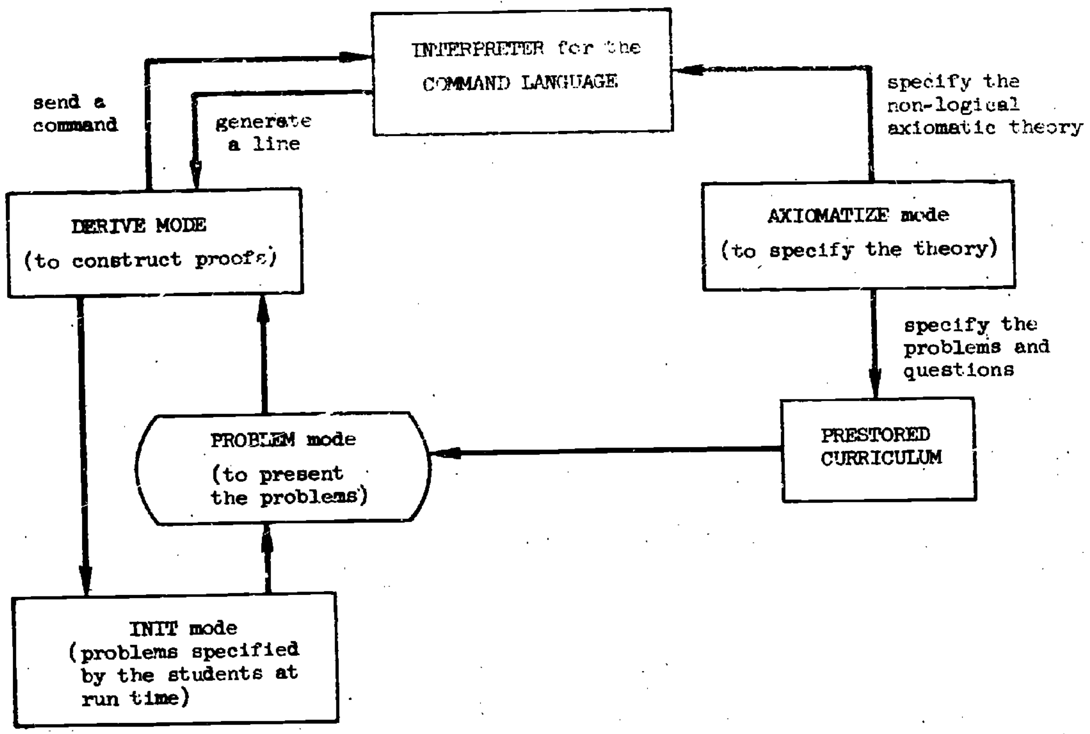

INTERPRETER for the COMMAND LANGUAGE
send a command
generate a line
specify the non-logical axiomatic theory
DERIVE MODE (to construct proofs)
AXIOMATIZE mode (to specify the theory)
specify the problems and questions
PROBLEM mode (to present the problems)
PRESTORED CURRICULUM
INIT mode (problems specified by the students at run time)

Fig. 1. Block diagram of the instructional system.

If the syntax of the command is not acceptable, the program has a simple method for giving syntactic error messages. The command cannot be executed if an application error is detected. In this case, the program has a method for computing the appropriate error message. This computation is necessary because, typically, the program interprets commands it knows only implicitly.

Figure 2 shows a block diagram of the routines monitoring the teacher- and student-user modes and presents a more detailed picture of the control structure

---

Insert Figure 2 about here

---

of the program. The history of the student's past performance is distinct from the collection of any data (that is, to collect exact user protocols). The profile consists of information on the last problem solved, the last theorem proved, rules and axioms specified by the teacher that the student has been taught and therefore is allowed to use, and the student's own derived rules of inference and lemmas he may have proven. This information is all part of the command language the student is building, and it can be retained for all future work.

The distinction between teacher and student is not a hard and fast one. Conceivably, an individual may act as both the teacher and the student, setting up a theory in which he wants to try to construct proofs. Or a student may try his skills on specifying a curriculum for his fellow classmates. It is possible that the teacher specifies the vocabulary and then gives the student a set of formulas. The student is told to choose no more than N formulas as axioms and then to prove the rest from these axioms, the primitive rules and procedures, and any new rules he derives. Although this is an instructional technique with which this instructional system has already been used, it will not be described here.

The remainder of this report is organized into three parts. In the first part, the components of a first-order theory and the methods for specifying it are explained. The reader who is mainly interested in how a user constructs proofs might well skip this background material. The second presents the structure of the command language, C, with explanations on how to use the proof procedures and primitive rules of inference, and how to derive new rules of inference. There is a problem, viz., the kinds of error analysis routines

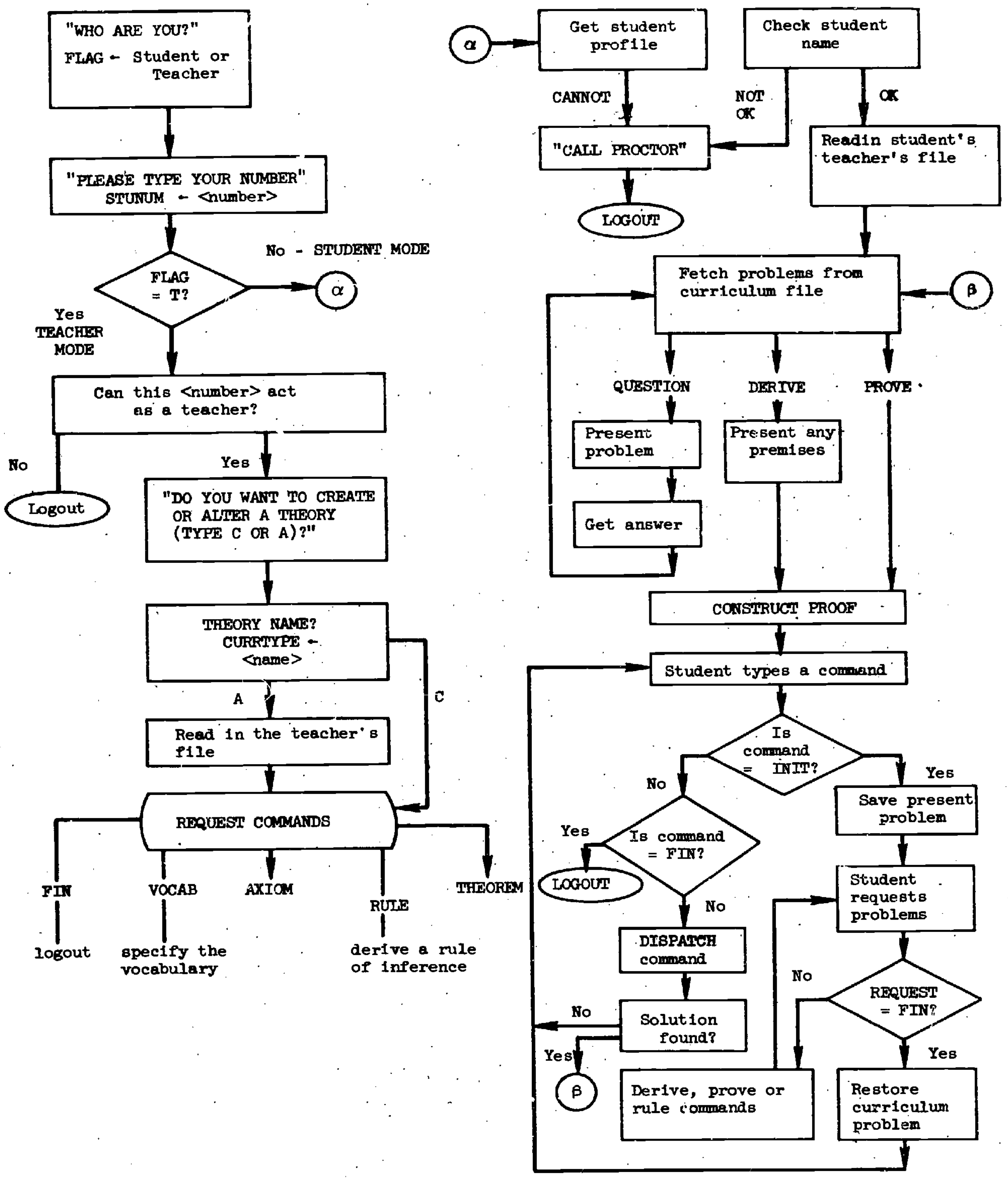


Fig. 2. Program structure.

required in order to effectively teach the student the command language. They are of two kinds--errors of command syntax and errors in applying the rules of inference, especially the derived rules. The error analysis routines are described in this second part. Finally, the language for specifying problems is defined with examples taken from elementary algebra.

PART II. SPECIFYING THE FORMAL SYSTEM

1. The Vocabulary

The program is limited to purely symbolic languages, where each identifier is a string of one or more alphabetic or special characters. The first step in developing a formal language is to define a meaningful expression. There are two kinds of expressions, terms and formulas. In order to give a precise characterization of a term or a formula, the user might specify the primitive vocabulary of the language, i.e., the user must unambiguously define each individual constant and variable, each operation symbol and each predicate letter. In addition, the list must include representations for the logical constants.

Associated with every symbol, except the individual variables, is a fixed DEGREE, and attached to every individual variable is a TYPE label.* These two elements are all that is needed to build a syntax-directed analysis routine to determine if a string constitutes a well-formed expression in the language. Every language includes three special characters that serve as delimitors--left parenthesis '(', right parenthesis ')' and comma ','. A check for precedence relationships is not included in the analysis. Unless expressions are explicitly grouped using the three special characters, the routine performs a simple left-to-right scan. Thus expressions of the form A x B + C are parsed as A x (B + C) and not as (A x B) + C, which is the usual parse if hierarchy tables are used.

A DEGREE is an ordered quadruple <i m n p> such that i and n are non-negative integers, and n and p are (a) any nonatomic element if the corresponding operation symbol, logical constant or predicate letter p represents a binary infix relation; (b) nonnegative integers otherwise. Formulas and nonatomic terms are formed with the aid of constants called "formula-makers" or "term-makers"

*With some modifications, this notion is borrowed from Kalish and Montague (1964).

according to the kind of expression they generate. In the associated DEGREE, i is either 0 or 1 according as the constant is a term-maker or formula-maker; m, n, and p, respectively, are the number of immediately following variables, number of terms and number of formulas that the constant demands. If $m \neq 0$, then the constant is a binding operator. The individual variables are atomic terms. Members of the set of integers are always considered terms, but if TYPE labels are to be considered, their TYPE label must be specified by the user (in the usual interpretation, the integers have type 'ALGEBRA' or 'ARITHMETIC'). The general mathematical characterization of terms and formulas is embodied in (1) through (8) below (Kalish & Montague, 1964, p. 272. Items (2)-(4) are their characterization of a term and a formula.)

1. The sequence of three dots, '...', is a <u>variable</u>.
2. Every variable is a term and every numeral is a <u>term</u>.
3. If δ is a constant of DEGREE <0 m n p>, $\alpha_1,\ldots,\alpha_m$ are immediately following variables, $\beta_1,\ldots,\beta_n$ are terms, $\sigma_1,\ldots,\sigma_p$ are formulas, and m, n, and p are nonnegative integers, then

$$\delta\alpha_1\ldots\alpha_m\beta_1\ldots\beta_n\sigma_1\ldots\sigma_p$$

is a <u>term</u>.
4. If δ is a constant of DEGREE <1 m n p>, $\alpha_1,\ldots,\alpha_m$ are immediately following variables, $\beta_1,\ldots,\beta_n$ are terms, $\sigma_1,\ldots,\sigma_p$ are formulas, and m, n, p are nonnegative integers, then

$$\delta\alpha_1\ldots\alpha_m\beta_1\ldots\beta_n\sigma_1\ldots\sigma_p$$

is a <u>formula</u>.

The following restrictions and additions to the characterization of terms and formulas handle formally what is usually considered informal notation conventions.

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 and n is nonatomic (we use n = '(2)'), then p must be zero and the constant δ represents a binary infix relation such that
   a. if δ is of DEGREE <0 0 (2) 0>, and β_1,β_2 are terms, then $\beta_1\delta\beta_2$ is a term;

b. if δ is of DEGREE <1 0 (2) 0>, and β_1, β_2 are terms, then $\beta_1 \delta \beta_2$ is a formula.

7. If m is zero and p is nonatomic (we use p = '(2)'), then n must be zero and the constant δ represents a binary infix relation such that if δ is of DEGREE <1 0 0 (2)> and σ_1, σ_2 are formulas, then $\sigma_1 \delta \sigma_2$ is a formula.

Hence, (1)-(7) is an exhaustive characterization of expressions in the language.*

If this characterization were, in fact, implemented on the computer, we could easily have as a well-formed formula in the language the expression FaG, where F has DEGREE <1 0 1 1> and G, <1 0 0 0>. By using the comma and the parentheses as delimitors or punctuation marks, we can write the expression F(a,G), a more readable format. Thus, an expression transcending first-order logic is acceptable. For our purposes this is undesirable. Admittedly, the parsing routine is a realization of (1)-(7). An eighth restriction is imposed:

8. n is zero if and only if p is nonzero.

So, with the eighth restriction included, any sequence that follows a constant and the possibly empty string of immediately following variables will be either a sequence of terms or one of formulas.

To complete the characterization of terms and formulas, the user must (a) give a list of symbols regarded as constants, together with their degrees; and (b) name the types associated with each individual variable. For example, in the case of quantification-free elementary algebra built on sentential logic, see Table 1. The constants in the table have their usual thematical

----

Insert Table 1 about here

----

interpretations, and, by the appended definition for binary relations, are written in the usual way. Note that multiple definition of the minus sign, '-', is acceptable. In fact, multiple definitions in general may be handled by careful recomputations in the cases when a particular definition is required.

The element associated with each individual variable is the TYPE. The purpose of type labeling is to restrict the use of individual variables in the expressions, e.g., it restricts the range of values of a variable to a particular universe of discourse. The TYPE is a list of labels where each label is used to

*A term-maker with degree <0 0 0 (2)> is meaningless.

TABLE 1

Elementary Algebra Symbols

Constant	Degree	Representation on the standard teletype
¬	<1 0 0 1>	NOT
&	<1 0 0 (2)>	&
V	<1 0 0 (2)>	OR
→	<1 0 0 (2)>	->
=	<1 0 (2) 0>	=
∀	<1 1 0 1>	A
∃	<1 1 0 1>	E
↔	<1 0 0 (2)>	IFF
+	<0 0 (2) 0>	+
-	<0 0 (2) 0>	-
-	<0 0 1 0>	-
/	<0 0 (2) 0>	/
x	<0 0 (2) 0>	x

name a list of individual variables. For example, consider the operation symbol '+' with DEGREE <0 0 (2) 0> and assume that '+' requires two algebraic terms. The user might name the TYPE of an algebraic term with the atom 'ALGEBRA' where ALGEBRA = (A B C D). The TYPE list, then, consists of the TYPE names for each term which the term-maker '+' demands. The term formed with '+' and the two terms is, in itself, a term that can have a TYPE name associated with it. This type is called the "computed type." The computed type is also stored on the TYPE list associated with '+'. Thus, the TYPE for '+' is a list consisting of three elements--three names, each of which specifies the TYPE of the argument expressions and well-formed terms. Given two algebraic terms, the operation symbol '+' forms an algebraic term.

A+B is a well-formed term because A and B are algebraic terms. Its computed type is ALGEBRA, so (A+B)+C is also well formed. Suppose the identity relation '=' with DEGREE <1 0 (2) 0> has TYPE = (ALGEBRA ALGEBRA NIL). This TYPE indicates that '=' demands two algebraic terms. The computed type is NIL, an atom that denotes the empty set or the 'don't care' type. In other words, NIL can represent any TYPE. Formulas, such as those formed by '=', always have TYPE NIL, and all formula-makers will have NIL as the computed type. Term-makers must be associated with a non-NIL computed type and so the atom T was chosen to denote the 'don't care' type for terms.

A=B is well formed. For this example, if BOOLE = (G H), then G=H is not well formed. However, if the TYPE associated with '=' were (NIL NIL NIL), it would not matter what the types of the two terms were and G=H would be well formed.

The list of all the types associated with the term-makers and formula-makers is called VARIABLES. If no type checking is desired, the user could presumably let VARIABLES be a list of all the individual variables and let all the types be NIL. This will not do however. The procedure for proper substitution for predicate letters (see discussion on page 22) is where problems would arise. The formula to be substituted for the predicate π may have a number of different free variables. Some of these variables may be replaced by corresponding arguments of π; the rest are nonsubstitutable parameters. How does the interpreter know the difference? Both kinds must be variables for the parser to recognize the expression generated by the operation of substitution.

So there must always be at least two lists. The first is always named 'PARAM' (for 'parameter') and contains a list of variables that are considered nonsubstitutable parameters in the procedure for proper substitution for predicates. All the others can be specially grouped under another TYPE name (such as BOOLE).

Some simplifications are possible.

1. If a variable can be in two groups, or when a TYPE is the union of two groups already named, the type checking is performed both directly (is the variable α a member of the list <TYPE>?) and indirectly (is the variable α a member of a list whose TYPE name is on the list <TYPE>?) Therefore, if PARAM are the special parameter-variables and VARS is a list of all the other variables, VARIABLES is the list (VARS PARAM). All the atomic terms are then recognized indirectly through the type name VARIABLES.
2. The TYPE associated with '+' is (ALGEBRA ALGEBRA ALGEBRA). Since all the members of the list are the same, the TYPE can be written as the single atom ALGEBRA.

By computing the associated types in the above manner, the program retains the recursive evaluation procedure for determining well formedness. This procedure depends on (a) the table of symbols; (b) associated DEGREES; (c) TYPE parameters; and (d) the list of individual variables. It is possible to multiply define symbols: if the parse fails in forming an expression with respect to one DEGREE, it will continue the search with any other DEGREE existing on the list for the constant in question. The result obtained by the parser is a representation of the expression in prefix-list notation. Henceforth, an expression in the prefix-list notation will be called the PATTERN for the expression. The PATTERN is the list form of the tree, e.g., (+ A B) for A+B. Further examples: (= (+ A B) C) is the PATTERN for A+B = C; (= (+ (+ A B) C) (+ A(+ B C))) is the PATTERN for (A + B) + C = A + (B + C).

That the details for setting up a well-defined vocabulary is tedious is acknowledged. The cumbersome method presented later for creating and altering the vocabulary will eventually be replaced in favor of routines for computing DEGREES and TYPES from user-entered definitions. At no time should a student have to go through the present process for specifying the vocabulary.

Specifying the vocabulary. As teacher, the user specifies the individual variables, and the table of logical constants, operation symbols and predicate letters within the constraints of (1)-(8) above. Under the defining procedures of the so-called TEACHER mode, the vocabulary can be created, altered by addition or deletion, or just viewed. This particular procedure is entered with the command VOCAB. The procedure is illustrated by the dialogue in Figure 3.* This is the first in a series of dialogues in which the teacher

---

Insert Figure 3 about here

---

specifies a first-order theory and the student receives and requests problems. (See Figure 16 and Appendix IV). Unless otherwise stated, sample proofs throughout this section depend on the vocabulary specified in Figure 3.

Observe that the teacher has one last option with regard to the form of the formulas printed out to the student. In many cases (notably all first-order theories), it is common mathematical practice to omit all universal quantifiers and their variables if the scope of the quantifiers is the entire formula. Upon request by the teacher, universal quantifiers that govern the entire formula can be suppressed.

2. Implementation of Proper Substitution

The instructional system includes routines that are realizations of the definitions of bondage, freedom, closure, proper substitution for free occurrences of variables and proper substitution for predicate letters.** The procedures required in order to carry out the two kinds of substitutions will be discussed. First, some definitions.

Definition 1. Dummy variables are additional variables not in the user-defined vocabulary of the system. The dummy of a variable is formed by concatenating the character '%' at the beginning and end of the variable. Thus A becomes %A%.

*The vocabulary and problems were chosen from exercises given by Professor Patrick Suppes to students taking an intermediate level course in logic.

**Standard terminology will be employed. The reader can consult Kalish and Montague as well as countless other books on logic. Definitions, unless to describe implementations, will not be repeated here.

```
*(START)
WHO ARE YOU (TYPE S OR T)?
*T$
PLEASE TYPE YOUR NUMBER.
*5$
DO YOU WANT TO CREATE OR ALTER A THEORY?(TYPE C OR A)
*C$
THEORY NAME?
*EX$

HI...
IN SETTING UP A FIRST-ORDER THEORY YOU MUST SPECIFY
THE VOCABULARY AND THE AXIOMS.  THEN YOU CAN CHOOSE
A SET OF WELL-FORMED FORMULAS AS THEOREMS, AND DERIVE
NEW RULES OF INFERENCE FROM THESE THEOREMS.  THE
COMMANDS ARE: VOCAB, AXIOM, THEOREM, AND RULE.  TYPE
FIN WHEN YOU ARE THROUGH.


:*VOCAB$
      YOU ARE SPECIFYING THE VOCABULARY FOR A
      FIRST-ORDER THEORY.  WE WILL BEGIN BY
      SETTING UP THE TABLE OF OPERATION SYMBOLS,
      PREDICATES, AND LOGICAL CONSTANTS.
      DO YOU WANT TO CREATE,ADD,DELETE,VIEW,
      OR FINISH? ( TYPE C,A,D, V, OR F)
*C$
FIRST YOU MUST SPECIFY THE LOGICAL CONSTANTS

NEGATION SIGN: *NOT$
CONJUNCTION SIGN: *&$
DISJUNCTION SIGN: *OR$
MATERIAL IMPLICATION: *THEN$
MATERIAL EQUIVALENCE: *IFF$
IDENTITY SIGN: *=$
UNIVERSAL QUANTIFIER: *A$
EXISTENTIAL QUANTIFIER: *E$


      NOW, SPECIFY THE NON-LOGICAL CONSTANTS. WHEN YOU ARE
      FINISHED, JUST TYPE ALTMODE INSTEAD OF A NEW SYMBOL.
SYMBOL: *B$
DEGREE: *(1 0 3 0)$
TYPELIST: *NIL$

SYMBOL: *R$
DEGREE: *(1 0 0 0)$
TYPELIST: *NIL$

SYMBOL: *F$
DEGREE: *(1 0 2 0)$
TYPELIST: *NIL$

SYMBOL: *P$
DEGREE: *(1 0 0 0)$
TYPELIST: *NIL$

SYMBOL: *F$
DEGREE: *(0 0 1 0)$
TYPELIST: *T$

SYMBOL: *N$
DEGREE: *(1 0 1 0)$
TYPELIST: *NIL$

SYMBOL: *1
      DO YOU WANT TO CREATE,ADD,DELETE,VIEW,
      OR FINISH? ( TYPE C,A,D, V, OR F)
*A$
SYMBOL: *B$
DEGREE: *(1 0 3 0)$
TYPELIST: *(POINT POINT POINT NIL)$
```

Fig. 3. Specifying a vocabulary.

Figure 3, continued.

```
SYMBOL: *$
     DO YOU WANT TO CREATE,ADD,DELETE,VIEW,
     OR FINISH? ( TYPE C,A,D, V, OR F)
*D$

     WHICH SYMBOL DO YOU WISH TO DELETE?
*B$
     I WILL TYPE OUT EACH DEGREE. TYPE Y IF YOU
     WANT TO DELETE IT. OTHERWISE TYPE ANYTHING.

(B 1 0 3 0 (POINT POINT POINT NIL)) *NO$

(B 1 0 3 0 NIL) *Y$
DONE

     WHICH SYMBOL DO YOU WISH TO DELETE?
*$

     DO YOU WANT TO CREATE,ADD,DELETE,VIEW,
     OR FINISH? ( TYPE C,A,D, V, OR F)
*F$

     DO YOU WANT TO CREATE,ALTER,OR VIEW
     THE VARIABLE LISTS OR ARE YOU
     FINISHED (TYPE C,A,V, OR F)?
*C$

     SPECIFY THE INDIVIDUAL VARIABLES
     BY PLACING THEM ON A LIST WITH AN
     ASSOCIATED TYPE NAME.  DO NOT USE
     THE NAME  -VARIABLES-
     END BY TYPING AN ALTMODE.

     FIRST INDICATE A TYPE  -PARAM-
     THIS IS THE LIST OF VARIABLES WHICH ARE
     CONSTANTS FOR THE PS PROCEDURE.  IT CAN
     BE AN EMPTY LIST.

 TYPE: PARAM

 LIST VARIABLES (WITH PARENTHESES): *(U V)$
 NOW ANY OTHERS?

 TYPE NAME: *POINT$
 LIST VARIABLES (WITH PARENTHESES): *(W X Y Z)$
 TYPE NAME: *ALGEBRA$
 LIST VARIABLES (WITH PARENTHESES): *(A B C D)$
 TYPE NAME: *$
 WHAT IS THE TYPE LABEL FOR THE NON-NEGATIVE
      INTEGERS?
 *ALGEBRA$

       DO YOU WANT TO CREATE,ALTER,OR VIEW
       THE VARIABLE LISTS OR ARE YOU
       FINISHED (TYPE C,A,V, OR F)?
 *V$
 ALGEBRA = (A B C D)
 POINT = (W X Y Z)
 PARAM = (U V)

       DO YOU WANT TO CREATE,ALTER,OR VIEW
       THE VARIABLE LISTS OR ARE YOU
       FINISHED (TYPE C,A,V, OR F)?
 *F$

      DO YOU WANT TO SUPPRESS PRINTING OF
 UNIVERSAL QUANTIFIERS WHERE THE SCOPE OF THE
 QUANTIFIER IS THE ENTIRE FORMULA?
 (TYPE Y, ELSE ANYTHING)
 *Y$

 :*FIN$
 T
 *
```

20

Definition 2. The dummy pattern of an expression is its prefix-list notation with each free occurrence of a variable replaced by its corresponding dummy. Thus 'A + B = B + A' is an expression whose dummy pattern is: (= (+ %A% %B%)(+ %B% %A%)).

Definition 3. The occurrence number of a symbol or term in an expression is determined by counting the occurrences of the symbol or term, starting at the left of the expression and scanning to the right.

Definition 4. The scope of a constant δ in a dummy pattern is given explicitly as any element, either a list or an atom, contained in the list whose first element is δ.

Definition 5. Let L be the list whose first element is a constant δ of DEGREE <i m n p> such that $m \neq 0$. Thus by 'L contains the variable (or term) α' is understood to mean that α is either an element of the list L or, recursively, there is a list L' such that L' is an element of L and L' contains α.

Definition 6. An occurrence of a variable α is bound only if there is an L of the sort described in definition 5 such that L contains α. Then the following determines which variable binding operator δ of the list L binds a variable α.

1. If the variable α is one of $\alpha_i, i=1,\ldots,m$ in the expression $\alpha_1 \ldots \alpha_m \beta_1 \ldots \beta_n \sigma_1 \ldots \sigma_p$, then α is a variable of quantification bound by that δ. In L, α_i is the (i+1)st element, δ is the first element.
2. If α does not satisfy (1), but α is within the scope of δ. If α is not also within a list L' such that L contains L' and the first element of L' is a binding operator, then δ binds α.

The system has two functions for testing bondage.

BOUND [IT STRING OCC]

BOUND asks if a particular occurrence (OCC) of a variable (IT) is bound in the expression (STRING). The value of the expression is *T* if the indicated occurrence of IT is not in STRING; it is NIL if there is such an occurrence, but it is not bound. If the occurrence is bound, the value of the functions is the occurrence number of the binding operator.

Since symbols can be multiply defined, it is possible that the functions which FIND ONE of the occurrences of a binding operator will, in fact, find an occurrence of a symbol identical with a binding operator, but one that is used as, say, a predicate. Therefore, the program has 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 operator. For example, let A have multiple degrees: <1 1 0 1> and <1 0 1 0>. Then the formula (A X(A X)) is well formed. 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 A has the DEGREE <1 1 0 0>, it would be.

In order to differentiate between these various cases, the program reparses the list L. 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 L is well formed, the first element of L is necessarily a binding operator.

BOUNDANY [IT STRING]

Is the variable (IT) bound anywhere in the expression (STRING)? The value is *T* if there is not any occurrence of the indicated variable in STRING, T if there is at least one bound occurrence, and NIL if there is at least one occurrence but no occurrence is bound. BOUNDANY is an iterated application of the function BOUND.

Definition 7. An occurrence of a variable α is free in the expression φ if it stands within φ but is not bound in φ. Three functions answer the questions about freedom.

FREE [IT STRING OCC]

Is the occurrence (OCC) of the variable (IT) in the expression (STRING) free? This function calls on BOUND and returns NIL if the value of BOUND is T or *T*; otherwise it returns T.

FREEEVERY [IT STRING]

Is every occurrence of the term (IT) free in the expression (STRING)? This function returns T if BOUNDANY returns the value NIL or *T* for all variables free in the term IT; otherwise it returns NIL.

FREEANYWHERE [IT STRING]

Is any occurrence of the variable (IT) free in the expression (STRING)? FREEANYWHERE returns NIL if no free occurrences of IT exist. Otherwise it returns the occurrence number of the first free occurrence of IT.

Definition 8. Two kinds of proper substitution--for free variables and for predicate letters--are made available by using the procedures for bondage and freedom of a variable in an expression.

Proper Substitution for Free Occurrences of Variables. Technically, a symbolic formula ψ comes from a symbolic formula φ by proper substitution of a symbolic term β for a variable α if ψ is like φ except for having free occurrences of β wherever φ has free occurrences of α. Implementation of proper substitution for free variables requires two steps.

1. Only replace an occurrence of α by β if the occurrence of α is free, Recall that α is free in φ if and only if it is not bound in φ. The function FREE performs this check.
2. Keep the (possibly nonatomic) term β free. For a term to be free, all free variables contained in the term must remain free after the substitution.

A flow diagram for the function PSVAR [α B φ] (proper substitution of variables) is presented in Figure 4.

Insert Figure 4 about here

Note on Figure 4: In testing for freedom and bondage, the function BOUND serves to "mark" the specified occurrence of the variable α in the expression by replacing α with the atom %α%. It then calls on the function BIND1 to determine if that marked location is within the scope of a binding operator and, moreover, if that binding operator governs α. To determine if the term β remains free, PSVAR bypasses the function BOUND because the location is already marked. PSVAR must call on BIND1 for each free variable in β.

Proper Substitutions for Predicate Letters. Recall that PARAM is a list of nonsubstitutable variables. Any variable in the language which is not a member of PARAM can, by proper substitution, be replaced by a well-formed term. For the following discussion, let PARAM = (W X Y Z). Let A,B,C be individual variables, and let the predicate letters F and G have degrees <1 0 1 0> and <1 0 2 0> respectively.

Consider the sentence of first-order logic:

$$\forall Z \exists X\ (F\ (X) \rightarrow \neg\ F\ (Z)).$$

23

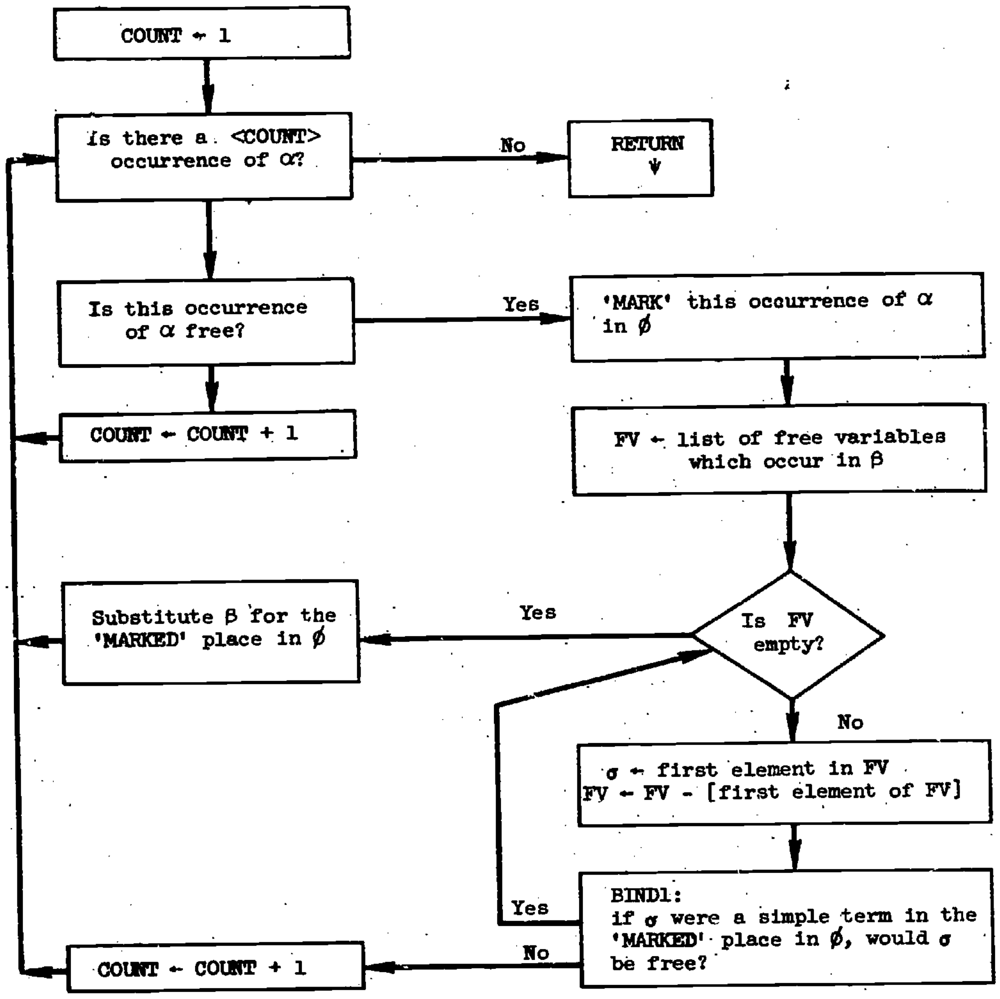

COUNT ← 1
Is there a <COUNT> occurrence of α?
No
RETURN
Is this occurrence of α free?
Yes
'MARK' this occurrence of α in ∅
COUNT ← COUNT + 1
FV ← list of free variables which occur in β
Substitute β for the 'MARKED' place in ∅
Yes
Is FV empty?
No
σ ← first element in FV
FV ← FV - [first element of FV]
Yes
BIND1: if σ were a simple term in the 'MARKED' place in ∅, would σ be free?
No
COUNT ← COUNT + 1

Fig. 4. PSVAR[α β ∅]
Propor:substitution of α for β in the expression ∅.

The proper substitution of the formula G(A,Y) for the one-place predicate F in the above sentence takes place as follows:

1. replace every occurrence of F by G(A,Y);
2. successively replace each variable (which is not an element of PARAM) of G(A,Y), in order, by the arguments of F. Here, A is replaced by X in the first occurrence of F; by Y in the second occurrence of F;
3. the resulting expression is:

$$\forall Z \exists X\ (G(X,Y) \rightarrow - G(Z,Y)).$$

There is one restriction imposed on this substitution operation. No variable that occurred in the formula G occurs in the sentence we started with. (A precise characterization of proper substitution for predicate letters appears in Kalish and Montague (1964, pp. 157ff).)

In the program, the function <u>PSPRED</u> [<u>χ η φ</u>] carries out the proper substitution of the predicate η by χ in the expression φ. The flow diagram of PSPRED is given in Figure 5. The routine first checks to see if χ and φ have arguments in common.

Insert Figure 5 about here

If they do, the function returns NIL. Otherwise, PSPRED computes the list of substitutable variables, L, and proceeds, for each occurrence of η and its arguments $\alpha_1,\ldots,\alpha_m$, to successively call on PSVAR [$\beta_i \epsilon L\ \alpha_i\ \chi$] in order to obtain the proper instance of χ which will replace the formula η. If there are no occurrences of η in φ, then PSPRED returns *T*.

The functions PSVAR and PSPRED, which carry out the two kinds of proper substitutions, are used in constructing derivations or proofs, i.e., they are used to compute a line which is an instance of an axiom or previously proven theorem. The next sections explain how these procedures are used by an individual in constructing a proof or derivation.

3. <u>Instantiation Procedures for Nonlogical Axiomatic Theories</u>

Each kind of substitution procedure (PSVAR and PSPRED) can be characterized in terms of its use in constructing proofs or derivations, i.e., for obtaining instances of an axiom or theorem.

<u>Simultaneous Universal Instantiation</u>. An axiomatic theory consists of two things: (a) the class of nonlogical constants, and (b) a class of axioms, which is any recursive class of formulas containing no nonlogical constants other than those in (a). The manner for defining the class of constants was already discussed

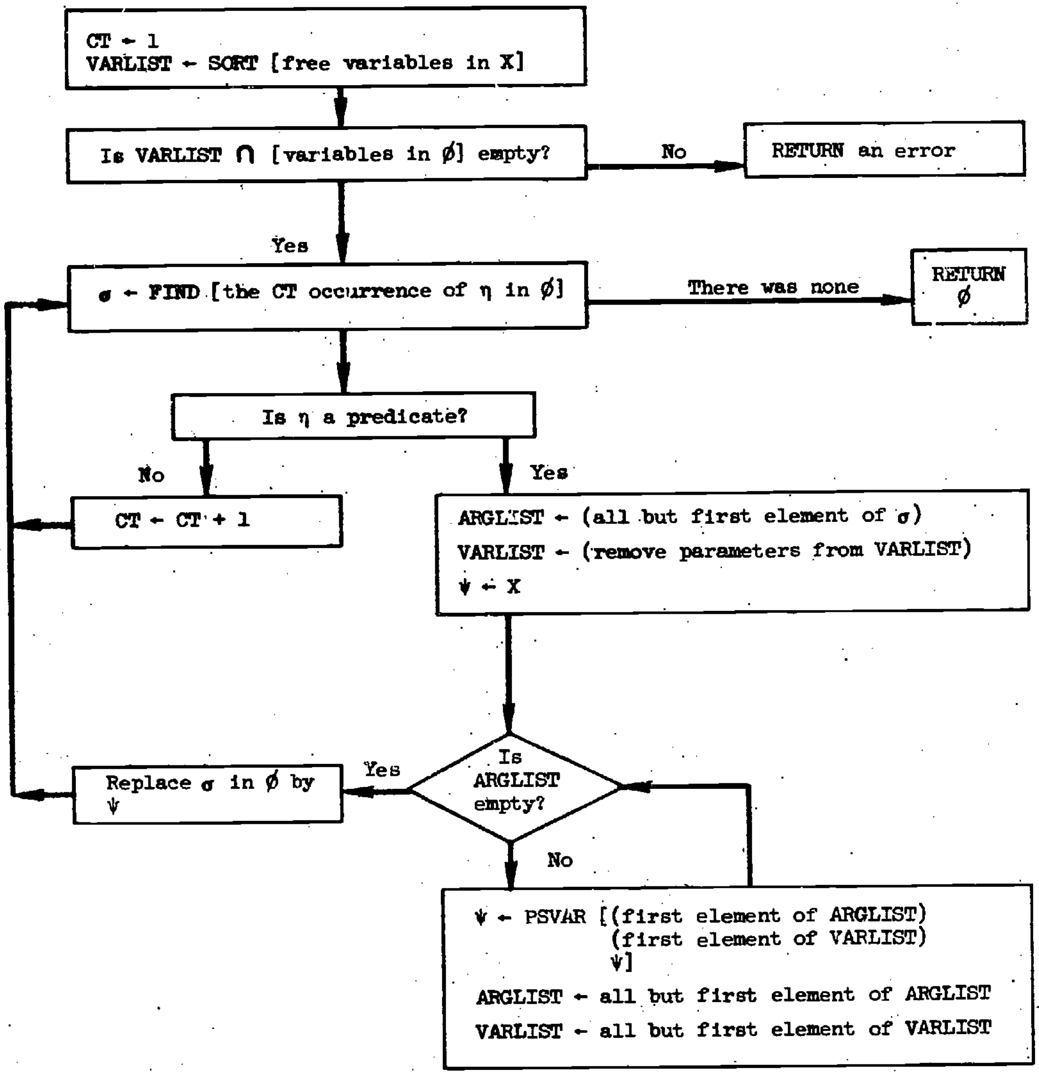

CT ← 1
VARLIST ← SORT [free variables in X]
Is VARLIST ∩ [variables in ∅] empty?
No
RETURN an error
Yes
σ ← FIND [the CT occurrence of η in ∅]
There was none
RETURN ∅
Is η a predicate?
No
Yes
CT ← CT + 1
ARGLIST ← (all but first element of σ)
VARLIST ← (remove parameters from VARLIST)
ψ ← X
Is ARGLIST empty?
Yes
Replace σ in ∅ by ψ
No
ψ ← PSVAR [(first element of ARGLIST) (first element of VARLIST) ψ]
ARGLIST ← all but first element of ARGLIST
VARLIST ← all but first element of VARLIST

Fig. 5. PSPRED[X η ∅]
Proper substitution of X for η in the expression ∅.

(page 7). The class of axioms is composed of a list of well-formed formulas. Each formula has a distinct name associated with it with which the user can refer to the axiom. Below is an example of a class of axioms for a Euclidean geometry in a one dimensional space. (The vocabulary was already specified where B is a three-place predicate denoting "betweenness" and the variables W, X, Y, and Z are points on a line.)

NAME	FORMULA
AXA	$B(X,Y,X) \rightarrow X = Y$
AXB	$B(X,Y,Z) \rightarrow B(Z,Y,X)$
AXC	$(B(X,Y,W)\ \&\ B(Y,Z,W)) \rightarrow B(X,Y,Z)$
AXD	$(B(X,Y,Z)$ OR $B(Y,Z,X))$ OR $B(Z,X,Y)$
AXE	$((\text{NOT } Y = Z)\ \&\ B(X,Y,Z))\ \&\ B(Y,Z,W)) \rightarrow B(X,Z,W)$

The command for this kind of instantiation is the name of the axiom or theorem. The program types out the formula associated with the name and then presents each distinct free variable V_i in the expression. The student types a well-formed term T_i. Each occurrence of the dummy V_i in the dummy pattern of that expression is replaced by T_i. The pattern is then typed out, in the usual format, as a new line of the proof or derivation.

As an example, take axiom AXA.* To obtain an instance of AXA, the sequence of commands is:

```
AXA      B(X,Y,X) → X = Y
X::W
Y::X     (1)  B(W,X,W) → W = X
```

The PS Procedure. 'PS' is the command name of the second kind of instantiation procedure. It is used to indicate that one wishes to take an instance of a formula by proper substitution for individual variables and predicates. The user indicates which formulas or terms are to be substituted for the predicates and the free variables, respectively, and gives the order in which the substitutions are to be made.

In the PS procedure, the actual value of the axiom or theorem, not the pattern, is referenced. The student types:

PS: <name of axiom or previously proved theorem>

If, for some reason, the axiom or theorem is not available, an error message is printed. Otherwise, the computer types a double colon indicating that the program is waiting for a response, and the user gives the sequence of substitutions by typing either the pair

*The convention is to underline all items typed by the user; all other information is typed by the computer program.

1. <variable> : <well-formed term>

or 2. <predicate letter> : <well-formed formula>

where <variable> and <predicate letter> all belong to the vocabulary of the theory. One or more such pairs could be typed. In fact, no pair can be typed and the result obtained will be the axiom or theorem itself.

The user indicates the end of the sequence by typing only the ALTMODE ($). Each substitution request is then applied, in the order in which it was typed, to the result of the previous substitution in the sequence. The first is, of course, applied to the axiom or theorem itself.

As an example, take the theorem TH10 to be $(\forall X(F\ X \rightarrow P)) \leftrightarrow (\exists X(F\ X \rightarrow P))$. If F is a one-place predicate and P is a zero-place predicate, the command sequence to generate line (n) might look like this.

```
     PS : TH10
     :: P :  ∀Z(G(Y,Z))
     :: F :  F(A,W)
     :: W :  Y
     ::                  (n)  (∀ X(F(X,Y) → ∀Z(G(Y,Z))))↔ (∃X(F(X,Y)) →
                              ∀ Z(G(Y,Z)))
```

Line (n) was obtained by

1. Proper substitution of $\forall$ Z G(Y,Z) for P in TH10 to give
   $(\forall X(F\ X \rightarrow \forall Z(G(Y,Z)))) \leftrightarrow (\exists X(F\ X)) \rightarrow (\forall Z(G(Y,Z)))$.
2. Proper substitution of F(A,W) for F in the result of (1):
   $(\forall X(F(X,W) \rightarrow \forall Z(G(Y,Z)))) \leftrightarrow ((\exists X(F(X,W))) \rightarrow (\forall Z(G(Y,Z))))$.
3. Proper substitution of Y for free occurrences of W to obtain the formula on line (n).

The command procedure PS merely checks to see which kind of substitution is desired and then calls on either the function of PSVAR (page 17) or the function PSPRED (page 17) to carry it out. However, once the new line is formed, the procedure reparses the formula in order to guarantee that all of the computed new terms agree with the corresponding TYPE parameters of the term-makers and formula-makers.

4. Axioms and Theorems

Below is a brief description of how the axioms and theorems are added to the command language when the user is in TEACHER mode, and how the formulas are processed so that the user can reference them as indicated in Section 3.

Adding Axioms to the System. To assist in describing the program procedures for adding axioms to the underlying nonaxiomatic logical system, some terminology is introduced or reiterated here. Each axiom has a name, some mneumonic made up of alphanumeric characters only. The formula φ is the value (VAL) of the name. If φ is parsed and its closure formed (CLOSED) such that φ has the form $\forall\alpha_1 \ldots \forall\alpha_n\psi$, then ψ is the matrix of φ. The dummy pattern is formed for the axiom by dropping the preceding string of universal quantifiers $\forall\alpha_1, \ldots, \forall\alpha_n$ and replacing each free occurrence of the individual variables $\alpha_1, \ldots, \alpha_n$ in the matrix by an alphabetic variant--the dummy. This transformed matrix, already in the prefix-list notation due to the parsing, is the dummy pattern (PATTERN) associated with the axiom name.

The REQUEST list is the list of variables of generalization $(\alpha_1, \ldots, \alpha_n)$, i.e., the variables occurring free in the matrix. Each element on the REQUEST list is in the vocabulary and, therefore, has a TYPE label associated with it. The list of these types, ordered in accordance with the REQUEST list, is called the TYPE list. Each of the five elements--VAL, CLOSED, dummy PATTERN, REQUEST list, TYPE list--is an attribute on a property list associated with the axiom name.

How to use the Axioms in Constructing Proofs. A line of a derivation of proof may be obtained by universal instantiation of an axiom as demonstrated in Section 3. When the user types the axiom name, the program replies with the value VAL associated with the axiom. (Note that the axiom as given by the teacher is always typed out or, at any rate, the representation preferred by the teacher is always typed out. In most cases, this would mean omitting any universal quantifiers whose scope is the whole formula.) Each member of the REQUEST list is then presented and followed by a double colon to indicate that the program expects a response from the user. The program, in each case, expects to receive a well-formed term of the same type as the corresponding element in the TYPE list. (If the TYPE is "T," any term is acceptable.) The acceptable terms are paired with the dummy of the variable from the REQUEST list in order to form a sequence of substitutions.

After all the elements of the REQUEST list have been presented, and the substitution sequence has been completely specified, the program carries out precisely, for each dummy-term pair, the proper substitution of the term for the dummy variable in the matrix of the axiom (that is, PSVAR [term dummy matrix].)

1. <variable> : <well-formed term>

or 2. <predicate letter> : <well-formed formula>

where <variable> and <predicate letter> all belong to the vocabulary of the theory. One or more such pairs could be typed. In fact, no pair can be typed and the result obtained will be the axiom or theorem itself.

The user indicates the end of the sequence by typing only the ALTMODE ($). Each substitution request is then applied, in the order in which it was typed, to the result of the previous substitution in the sequence. The first is, of course, applied to the axiom or theorem itself.

As an example, take the theorem TH10 to be $(\forall X(F\ X \rightarrow P)) \leftrightarrow (\exists X(F\ X \rightarrow P))$. If F is a one-place predicate and P is a zero-place predicate, the command sequence to generate line (n) might look like this.

```
PS : TH10
:: P :  ∀Z(G(Y,Z))
:: F :  F(A,W)
:: W :  Y
::                  (n)  (∀ X(F(X,Y) → ∀Z(G(Y,Z))))↔ (∃X(F(X,Y)) →
                          ∀ Z(G(Y,Z)))
```

Line (n) was obtained by

1. Proper substitution of $\forall Z\ G(Y,Z)$ for P in TH10 to give $(\forall X(F\ X \rightarrow \forall Z(G(Y,Z)))) \leftrightarrow (\exists X(F\ X)) \rightarrow (\forall Z(G(Y,Z)))$.
2. Proper substitution of $F(A,W)$ for F in the result of (1): $(\forall X(F(X,W) \rightarrow \forall Z(G(Y,Z)))) \leftrightarrow ((\exists X(F(X,W))) \rightarrow (\forall Z(G(Y,Z)))$.
3. Proper substitution of Y for free occurrences of W to obtain the formula on line (n).

The command procedure PS merely checks to see which kind of substitution is desired and then calls on either the function of PSVAR (page 17) or the function PSPRED (page 17) to carry it out. However, once the new line is formed, the procedure reparses the formula in order to guarantee that all of the computed new terms agree with the corresponding TYPE parameters of the term-makers and formula-makers.

4. Axioms and Theorems

Below is a brief description of how the axioms and theorems are added to the command language when the user is in TEACHER mode, and how the formulas are processed so that the user can reference them as indicated in Section 3.

Adding Axioms to the System. To assist in describing the program procedures for adding axioms to the underlying nonaxiomatic logical system, some terminology is introduced or reiterated here. Each axiom has a _name_, some mneumonic made up of alphanumeric characters only. The formula φ is the value (VAL) of the name. If φ is parsed and its closure formed (CLOSED) such that φ has the form $\forall\alpha_1\ldots\forall\alpha_n\psi$, then ψ is the _matrix_ of φ. The dummy pattern is formed for the axiom by dropping the preceding string of universal quantifiers $\forall\alpha_1,\ldots,\forall\alpha_n$ and replacing each free occurrence of the individual variables $\alpha_1,\ldots,\alpha_n$ in the matrix by an alphabetic variant--the dummy. This transformed matrix, already in the prefix-list notation due to the parsing, is the dummy pattern (PATTERN) associated with the axiom name.

The _REQUEST list_ is the list of variables of generalization $(\alpha_1,\ldots,\alpha_n)$, i.e., the variables occurring free in the matrix. Each element on the REQUEST list is in the vocabulary and, therefore, has a TYPE label associated with it. The list of these types, ordered in accordance with the REQUEST list, is called the _TYPE list_. Each of the five elements--VAL, CLOSED, dummy PATTERN, REQUEST list, TYPE list--is an attribute on a property list associated with the axiom name.

How to use the Axioms in Constructing Proofs. A line of a derivation of proof may be obtained by universal instantiation of an axiom as demonstrated in Section 3. When the user types the axiom name, the program replies with the value VAL associated with the axiom. (Note that the axiom as given by the teacher is always typed out or, at any rate, the representation preferred by the teacher is always typed out. In most cases, this would mean omitting any universal quantifiers whose scope is the whole formula.) Each member of the REQUEST list is then presented and followed by a double colon to indicate that the program expects a response from the user. The program, in each case, expects to receive a well-formed term of the same type as the corresponding element in the TYPE list. (If the TYPE is "T," any term is acceptable.) The acceptable terms are paired with the dummy of the variable from the REQUEST list in order to form a sequence of substitutions.

After all the elements of the REQUEST list have been presented, and the substitution sequence has been completely specified, the program carries out precisely, for each dummy-term pair, the proper substitution of the term for the dummy variable in the matrix of the axiom (that is, PSVAR [term dummy matrix].)

To continue the example given earlier, the property list of the axiom AXA is:

VAL	$B(X,Y,X) \rightarrow X = Y$
CLOSED	$\forall X(\forall Y(B(X,Y,X) \rightarrow X = Y))$
PATTERN	(→ (B %X% %Y% %X%)(= %X% %Y%))
REQUEST	(X Y)
TYPE	(POINT POINT)

Specifying an Axiom in the System. To specify an axiom, the teacher types and responds to the following sequence

```
AXIOM
NAME::  <name of the axiom>
WFF::   <a well-formed formula>
```

After seeing the name of the axiom and checking to see if it is distinct from any other name associated with formulas, the program types 'WFF::' in order to request the formula. The formula must be well formed. If it is not, the request is repeated. If the user changes his mind, he can type an ALTMODE to get out of the entire command sequence. This escape route is available for all command sequences in both the TEACHER and STUDENT modes of operation (see Figure 16).

If the formula is accepted, the attributes for the axiom are computed, the axiom name is added to the general AXIOMLIST and the program continues. Generally, in the TEACHER mode, the user specifies curriculum to be presented to other users, the "students." As such, the program differentiates between the general axiom list (the list of all the axioms specified by the teacher) and the list of axioms the student may use. An axiom is not placed on the student's axiom list (and thereby made available to that student for constructing proofs) until a command to do so is given in the curriculum (see page 63, on Defining Problems).

Adding Theorems to the System. A theorem is a formula of a theory derivable from the axioms of the theory alone. A theorem is processed in much the same way as an axiom. Once a theorem has been proved, it can be named and used in constructing other proofs or derivations. (The same is true of formulas chosen as lemmas by the student.)

A theorem is used in the same manner as axioms in the construction of proofs or derivations. The name of a theorem specified by the user in the TEACHER mode is always 'TH' concatenated with a positive integer. Thus the teacher can order and number the theorems. This is the only case in which a command name is not an alphabetic string. The program uses the numerical portion of the command to guarantee that the user is permitted only instances of theorems which he has already proven. Attached to a theorem name are the attributes VAL, CLOSED, dummy PATTERN, REQUEST list, and TYPE list. These attributes are computed and used just as for the axioms.

Specifying a Theorem in the System. The format of the command to specify a theorem if the user is in the TEACHER mode is:

THEOREM

NUMBER:: <positive integer>

WFF:: <well-formed formula>

The formula is processed exactly as that for an axiom. The name of the theorem automatically becomes 'TH<positive integer>.'

Formulas proven by the user while in STUDENT mode are named at the completion of the proof (see page 68, Requesting Problems at Run Time).

PART III. THE COMMAND LANGUAGE

1. Introduction

The result of using the language M to specify an axiomatic system is to build the command language C. The instructional program interprets a sequence of well-formed commands C_i belonging to C as an algorithm for constructing a derivation or proof of a formula in the specified system.

Two commands have already been given: (a) the name of an axiom or of an established theorem, and (b) the procedure PS. If the command takes the form of a simple mneumonic that was defined as the name of an axiom or theorem, the program carries out the procedure previously presented (page 22). These co
as well as those for the rules of inference and the special procedures (like PS), are "executed" by the program with the result that the program generates new lines of a derivation or proof. Line by line, the user types commands until the desired formula has been generated.

The command language consists of code for the rules governing substitution (as already explained), sentential rules and quantifier rules that reflect the basic logical system (see especially Appendix II for a summary of these sentential rules), laws of identity, and rules for definite descriptions. Specifically these are:

1. <u>Proof procedures</u>: conditional proof (CP), indirect proof (IP), and universal derivation (UG);
2. <u>Six primitive rules of inference</u>: modus ponens (AA), three rules for quantification (US, ES, EG), two rules for the logic of identity (IDC, IDS), as well as two interchange 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;
3. <u>Generalized interchange rules</u>: these permit an interchange on the basis of an axiom of theorem. that has the form of a generalized identity or material bi-conditional;
4. <u>Derived rules of inference</u>; and
5. <u>Miscellaneous rules</u>: delete the last line (DLL), enter a line (ENT), INITiative to request problems is given to the student, SHOW that a line is a valid inference to make, and HELP the user complete his proof.

The general syntax for the command language is:

```
<command>              ::= <line references> <command name> <occurrence references>:
                           <other information> $ | <line references> <command name>
                           <occurrence references> $

<line references>      ::= <line number> | <sequence of line numbers> |∅

<sequence of line numbers>
                       ::= <sequence of line numbers> . <line number> | <line number>

<occurrence references>
                       ::= <occurrence number> |<sequence of occurrence numbers> |∅

<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 closed off)

<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 | UG
<primitive rule of inference>	::= RE | RQ | AA | IDC | IDS | ES | US | EG
<miscellaneous codes>	::= DLL | ENT | INIT | SHOW | HELP
<derived rule of inference>	::= string of alphabetic characters (other than those already reserved)
<other information>	::= <well-formed expression> | <name> | <variable>

If the execution of a command code requires one or more further responses from the user, the interpreter always types a double colon to indicate that it is ready to accept the response.

After the user types a command, the interpreter performs a syntactic analysis of it. If any syntax error is located, the command is ignored and an appropriate error message is printed. Detecting syntax errors is straightforward, and the benefit of concise error messages should be clear. These messages can reteach a particular command's format. By typing the command name and then successively making corrections in the syntax as directed by the error messages, the user could learn to properly format the command.

This initial analysis of a command is sufficiently standard so that message forms can be stored and retrieved from a peripheral device and the interpreter can compute appropriate insertions. Three main syntactic analysis routines make use of these messages. They serve to

1. read in the command;
2. dispatch the command to the appropriate processing routine; and
3. check the syntax of the command in terms of the line and occurrence references.

These three routines are presented as block diagrams in Figure 6. Each circled number designates an error return from the routine: the error message (as numbered in Figure 7) is completed and printed.

Insert Figures 6 and 7 about here

The rest of this section contains a detailed description of each of the five elements of the command language listed above. Included in each discussion are definitions, program procedures for learning and carrying out the commands, together with some examples.

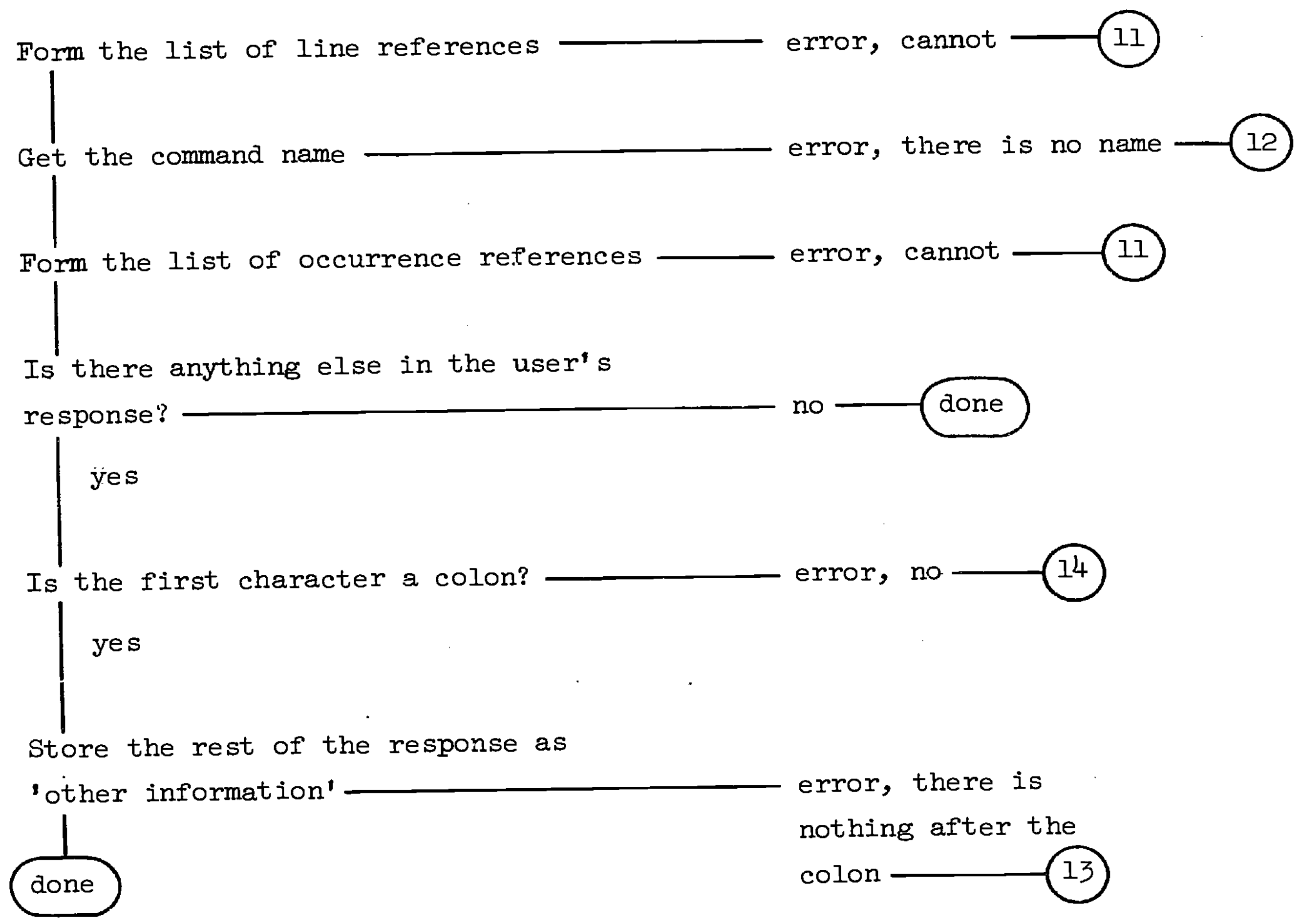

Form the list of line references
error, cannot
11
Get the command name
error, there is no name
12
Form the list of occurrence references
error, cannot
11
Is there anything else in the user's response?
no
done
yes
Is the first character a colon?
error, no
14
yes
Store the rest of the response as 'other information'
error, there is nothing after the colon
13
done

Figure 6. Syntax Analysis--
(a) Read in the Command

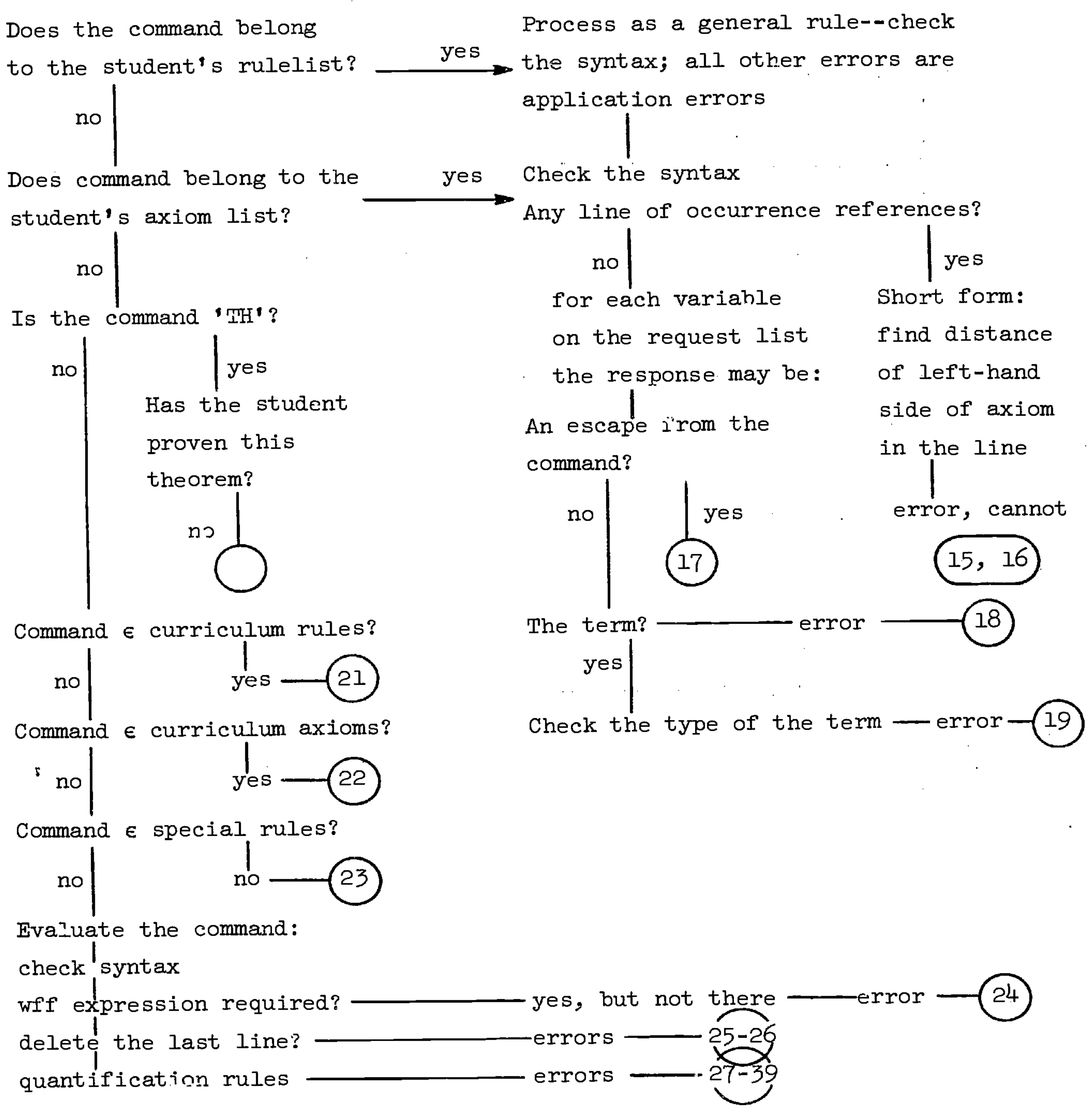

Does the command belong to the student's rulelist?
yes
Process as a general rule--check the syntax; all other errors are application errors
no
Does command belong to the student's axiom list?
yes
Check the syntax
Any line of occurrence references?
no
yes
for each variable on the request list the response may be:
Short form: find distance of left-hand side of axiom in the line
Is the command 'TH'?
no
yes
Has the student proven this theorem?
no
An escape from the command?
no
yes
17
error, cannot
15, 16
The term?
error
18
yes
Check the type of the term
error
19
Command ϵ curriculum rules?
no
yes
21
Command ϵ curriculum axioms?
no
yes
22
Command ϵ special rules?
no
no
23
Evaluate the command:
check syntax
wff expression required?
yes, but not there
error
24
delete the last line?
errors
25-26
quantification rules
errors
27-39

Figure 6. Syntax Analysis--
(b) Dispatch the Command to the Appropriate Processing Routine

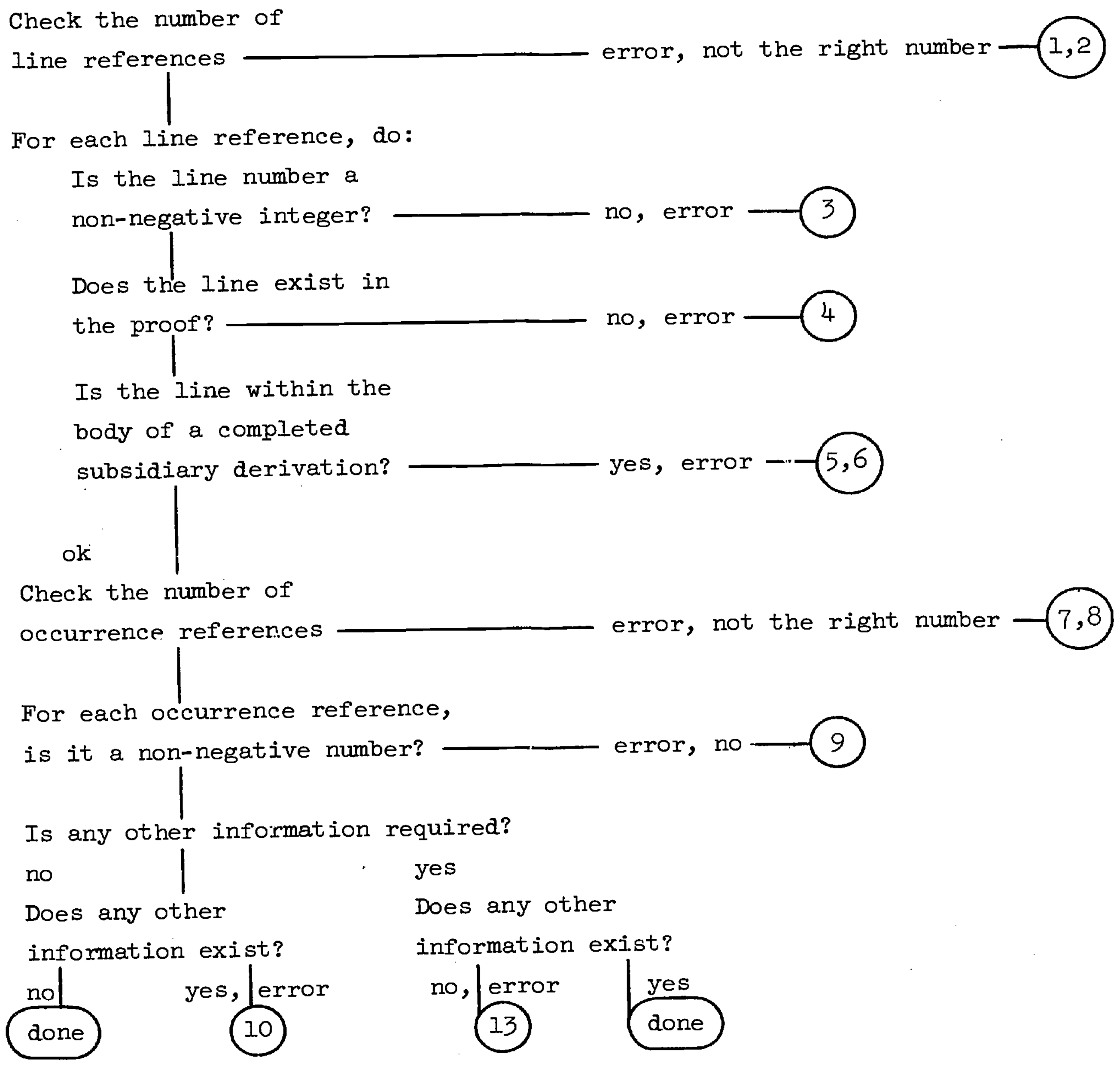

Check the number of line references
error, not the right number
1,2
For each line reference, do:
Is the line number a non-negative integer?
no, error
3
Does the line exist in the proof?
no, error
4
Is the line within the body of a completed subsidiary derivation?
yes, error
5,6
ok
Check the number of occurrence references
error, not the right number
7,8
For each occurrence reference, is it a non-negative number?
error, no
9
Is any other information required?
no
yes
Does any other information exist?
Does any other information exist?
no
done
yes, error
10
no, error
13
yes
done

Figure 6. Syntax Analysis--
(c) Check the Syntax of the Command

1. <command name> REQUIRES ONE LINE NUMBER.
2. <command name> REQUIRES <number of premises> LINE NUMBER.
3. THE LINE REFERENCE MUST BE A NUMBER.
4. THERE IS NO LINE <line number>.
5. YOU MAY NOT USE LINE <line number>. LINE <line number> DEPENDS ON THE WORKING PREMISE LINE <line at which the working premise was introduced> WHICH IS NO LONGER AVAILABLE.
6. YOU MAY NOT USE LINE <line number>. LINE <line number> DEPENDS ON THE ASSUMPTION FOR UNIVERSAL GENERALIZATION MADE AT LINE <the gen command precedes this line>.
7. <command name> REQUIRES AN OCCURRENCE NUMBER.
8. <command name> REQUIRES <number of occurrences> OCCURRENCE NUMBERS.
9. THE OCCURRENCE NUMBER MUST BE A NUMBER.
10. <command name> DOES NOT EXPECT A COLON WITH AN EXPRESSION OR SYMBOL TO FOLLOW.
11. A NUMBER MUST FOLLOW A PERIOD.
12. NO COMMAND REQUESTED?
13. AN EXPRESSION, SYMBOL OR NAME MUST FOLLOW A COLON IN THE COMMAND.
14. THE FORMAT IS INCORRECT. AN OCCURRENCE NUMBER OR A COLON MUST FOLLOW THE COMMAND NAME.
15. THERE IS NO OCCURRENCE IN LINE <line number> OF A TERM TO WHICH THE RULE <command name> CAN BE APPLIED.
16. THERE ARE NOT <number of occurrences> OCCURRENCES IN LINE <line number> OF A TERM TO WHICH THE RULE <command name> CAN BE APPLIED.
17. TRY AGAIN.
18. NOT A WELL-FORMED TERM.
19. THE TYPE OF THE TERM MUST BE <type name>.
20. YOU HAVE NOT PROVEN THEOREM <theorem number>.
21. YOU MAY NOT USE RULE <command name> IN THIS PROBLEM.
22. YOU MAY NOT USE THE <command name> AXIOM.
23. <command name> IS NOT A RULE.
24. <user's response> IS NOT A WELL-FORMED EXPRESSION.
25. THERE IS NO LINE TO DELETE.
26. LINE <line number> IS A PREMISE AND CANNOT BE DELETED.
27. <term of instantiation> CANNOT BE USED AS A VARIABLE OF INSTANTIATION. YOU ALREADY KNOW SOMETHING ABOUT <term of instantiation>.
28. <variable> WAS INTRODUCED AS AN ARBITRARY INDIVIDUAL FOR THE PURPOSE OF UNIVERSAL DERIVATION.
29. <variable> IS ALREADY MENTIONED IN LINE <line number>.
30. <term> IS NOT A VARIABLE OF INSTANTIATION.
31. <term> IS NOT A VARIABLE OF GENERALIZATION.
32. <term> OCCURS FREE IN LINE <line number>.
33. LINE <line number> DOES NOT OCCUR AFTER THE LAST GEN WAS REQUESTED.
34. <variable> WAS THE LAST VARIABLE OF GENERALIZATION REQUESTED.
35. THERE IS NOT A FREE OCCURRENCE OF <term> IN LINE <line number>.
36. THERE ARE NOT <number of occurrences> FREE OCCURRENCES OF <term> IN LINE <line number>.
37. THE VARIABLE OF GENERALIZATION MUST NOT OCCUR FREE IN THE TERM.
38. AT LEAST ONE OF THE OCCURRENCES OF <alpha> ON WHICH YOU ARE GENERALIZING IS BOUND IN LINE <line number>.
39. IMPROPER APPLICATION OF <command name>.

Figure 7. Syntactic Error Message Forms.

2. Proof Procedures

To facilitate the construction of proofs and derivations three derivation or proof procedures are available. They constitute the heart of a natural deduction formulation of first-order logic. Two of these, the conditional proof (CP) and indirect proof (IP), depend on the introduction of a working premise (WP). The third is universal derivation (UG) in which the user establishes a universal statement. UG (for "Universal Generalization") depends (as explained below) on the user's announcing his intention to generalize a particular individual variable (this is done with the command GEN).

The command language is similar to a programming language; each command is an instruction to the interpretive system. A well-formed command, executed by that system, constitutes an application of a rule to construct an expression in the first-order theory. This defines the construction in terms of purely mechanical manipulations of expressions.

Associated with the notion of executing a sequential list of instructions that belong to a (programming) language is the concept of subroutines. A subroutine is a complicated command that has a name, a possibly empty set of formal parameters and a body. The body is a sequence of commands that manipulate the formal parameters. Moreover, each routine is delimited by identifiers that indicate the beginning and end of the subroutine. Such routines illustrate the idea of block structuring, or grouping, of commands. In the usual sense, a block structure is a means of defining the scope of identifiers. Variables, arrays and definitions may be declared at the head of a block and have no significance outside this block. The importance of a block comes from the fact that blocks may be nested, i.e., the beginning of a new block may be declared within the body of another block. Definitions at the head of a block have meaning only within that block and any that it encloses.

Clearly the command sequences for obtaining instances of axioms and previously proved theorems may be viewed as a block or subsidiary routine. The command name is the name of the routine as well as the beginning delimitor, the substitutable variables are the parameters and ALTMODE is the ending delimitor. But this might be carrying an analogy too far. The three proof procedures described below serve as sharper analogies to programming with subroutines. In the sequel, consider the command mneumonics CP, IP and UG as the names of subroutines of a programming language.

WP. For CP and IP, the beginning of a block is denoted by the simple command WP. WP itself is an instruction to the program, first, to set up the mechanisms for specifying the beginning of a proof procedure, and second, to allow the user to enter a premise of his own into the derivation. This is a working premise; the lines of the derivation are conditional upon it. The new premise can be thought of as a formal parameter since it only has meaning within the block. Once the block is closed (boxed off or completed) by the ending delimitor (in this case, the command containing the name of the proof procedure), the premise entered by the WP rule can no longer be referenced as a line of the proof.

The body of the block is a sequence of arbitrary but finite-length lines that occur between the delimiting lines, i.e., those generated by the WP and the CP or IP commands. Once a procedure is completed, the block is closed off and no line in the body may be referenced as a line of the proof.

Analogous to the notion of the nesting of subroutines, several working premises may be requested in (not necessarily contiguous) succession. The program retains the order of this series of WP requests so that only the last WP entered may be closed, i.e., referenced in a CP or IP command. Moreover, although the user may be able to generate the correct expression for solving the derivation problem, he has not solved the problem unless all the WP's have been closed. In other words, like a well-formed computer program, every beginning of a routine must have a well-defined end. To help the user keep track of the block structure, the program indents several spaces for each open WP (each level of incompleted nesting).

The command format for WP is simply the mnemonic:

WP (i) <well-formed formula>

The program types the number of the new line and the user enters a well-formed formula as a premise.

CP. The user can construct or generate a conditional statement by introducing the antecedent of that statement as a working premise, by working out a derivation of the consequent in the usual manner, but making it conditional upon the entered premise, and by then using the CP rule to finally derive the conditional statement. The format for the command is

<line a>. <line b> CP

where <line a> refers to the last working premise introduced and <line b> refers to any line of the derivation. Of course, <line a> and <line b> can refer to the same line. The formula on <line a> becomes the antecedent and that on <line b> the consequent of the conditional statement formed. CP indicates that the particular WP is now closed. The delimiting line is <line b>. If <line b> comes before <line a> in the derivation sequence, then the body of the block is empty and only <line a> becomes unavailable.

As an example, a proof of $R \rightarrow (R \text{ OR } P)$ is provided in Figure 8.

Insert Figure 8 about here

IP. In order to derive a formula φ, the user may introduce the negation of φ as a working premise (WP). He then attempts to construct ψ and (NOT ψ) as two lines of the proof or derivation, thus establishing the logical truth of the negation of the working premise. This is known as a proof by contradiction, or reductio ad absurdum.

The IP command requires a sequence of three line numbers to precede it as follows:

<line a> . <line b> . <line c> IP

where <line a> refers to the last working premise entered and <line b> is the denial of <line c>. The program generates as a new line the denial of the premise on <line a>. An example of a proof using the indirect derivation procedure is given in Figure 9.

Insert Figure 9 about here

GEN and UG. Universal derivation, or generalization (UG), permits the user to establish a universal statement, i.e., one of the form $\forall\alpha\varphi$, as a line of the derivation or proof. The beginning of the sequence of commands for obtaining this universal statement is denoted by the command GEN. In using this command, the user indicates which variable, α, he wants to introduce as an arbitrary individual for the purpose of universal derivation. The command format is:

GEN: $<\alpha>$

```
DERIVE     R ->(R  OR P)

:*WP$      (1)       *R$

:*1FD$
::*P$      (2)       R OR P

:*1.2CP$   (3)  R ->(R  OR P)

CORRECT...
```

Fig. 8. Conditional proof.

```
DERIVE     R OR(NOT R)

:*WP$      (1)       *NOT (R OR (NOT R))$

:*WP$      (2)            *R$

:*WP$      (3)                 *R$

:*3FD$
::*NOT R$  (4)                 R OR(NOT R)

:*3.4.1IP$ (5)            NOT R

:*2.2.5IP$ (6)       NOT R

:*6FD$
::*R$      (7)       (NOT R)OR R

:*7CD1$    (8)       R OR(NOT R)

:*1.1.8IP$ (9)  R OR(NOT R)

CORRECT...
```

Fig. 9. Indirect proof.

```
DERIVE     A Y(F Y -> F Y)

:*P$       (1)  *F X$

:*GEN:X$
     X CANNOT BE USED AS A VARIABLE OF
      GENERALIZATION; IT IS NOT AN ARBITRARY INDIVIDUAL.

:*GEN:Y$
OK

:*WP$      (2)            *F Y$

:*2.2CP$   (3)        F Y-> F Y

:*3UG:Y$   (4)  A Y(F Y -> F Y)

CORRECT...
```

Fig. 10. Universal derivation.

The notion of nesting GEN's is like that of the WP's so the discussion on block structures will not be repeated except to note that GEN's and WP's can be nested within one another, i.e., one or more working premises may be entered after the GEN command is accepted. However, the universal derivation introduced by a GEN request cannot be completed unless all such working premises are closed. Conversely, GEN commands that occur after a working premise is entered must be closed before the external WP can be.

This derivation procedure is intended to capture the proof technique in which one demonstrates that <u>every</u> object has a certain property by showing that an <u>arbitrary</u> object from the universe of discourse has the property. Consequently, the interpreter must ensure that the object α indicated for this purpose is actually an arbitrary one. This is done by determining if α occurs free in any antecedent line φ_i (FREEANYWHERE[α φ_i]). An "antecedent line" is defined as any line occurring in the proof which has not already been closed off by one of the pairs WP-CP, WP-IP, or GEN-UG. If the α is accepted, the program types 'OK'.

GEN does not generate a line. The main reason for its use in the generalization procedure is to prevent the user from performing a number of unnecessary steps; for example, without GEN, the user may discover several steps later that he cannot generalize over a variable as he had intended. Furthermore, by introducing α into the proof, the user is prevented from using α as a variable of specification in ES (existential specification rule, page 40), which violates the restriction on the latter rule. This GEN announcement is one method used to reduce the amount of irrelevant work and at the same time emphasizes the care needed in introducing new variables.

After the user has constructed the matrix ψ of the desired universally quantified formula, he types the command

<u><line a> UG: <α></u>

where <line a> refers to the line containing ψ. This line must occur after the last GEN and the <α> must be the last variable of generalization specified. Thus, the new line is formed. A simple example of GEN-UG is given in Figure 10.

Insert Figure 10 about here

Note that GEN serves as a delimitor so that if the user types an improper UG command an appropriate error message is given.

3. Primitive Rules of Inference

Modus Ponendo Ponens (AA). By the classical rule of modus ponens, a symbolic sentence φ may be inferred from the symbolic sentence $\psi \rightarrow \varphi$ and ψ. If a derivation consists of lines

(1) $P \rightarrow Q$

(2) P

then, the command 1.2AA generates the new line: (3) Q. The mneumonic AA stands for "affirm the antecedent," which suggests the semantical analysis of the rule.

The AA command is processed in the same manner as a derived rule of inference. This procedure is discussed in detail in the section on derived rules of inference (page 47ff.). Rules of inference for sentential logic include Form a Conjunct (FC), Form a Disjunct (FD), Double Negation (DN), and Deny the Consequent (DC), to name a few familiar ones.

Rule of Universal Specification (US). The implementation of the three rules for quantification theory reflect the standard characterizations of these rules. In the following, let α be a variable, β a term, and φ and ψ formulas.

The principle of specification permits the deduction of some symbolic formula ψ from a universal statement $\forall\alpha\varphi$. The rule of universal specification US (sometimes referred to as universal instantiation) states that ψ comes from φ by PSVAR[β α φ] for some term β. The term of instantiation is β.

The intuitive content of this rule may be expressed in the slogan "what is true of everything is true of any given thing." The only restriction on β is that it can be properly substituted for the indicated universal quantification variable. The procedure for carrying out the rule must also ensure that the computed type for β is consistent with that for α. The command format for US is

<line number> US

<α> :: <β>

where the line reference must contain a universal statement. The interpreter types the variable of generalization α found on the line and then a double colon to request the user enter β. The new line is ψ where ψ = PSVAR[β α φ]. A simple example of this command is:

(1) $\forall$ X(F X $\rightarrow\exists$ X(G(X,Y)))

<u>1 US</u>

X::<u>Z</u> (2) F(Z)$\rightarrow\exists$ X(G(X,Y))

<u>Rule of Existential Generalization</u> (EG). The rule of existential generalization (EG) permits an inference from some expression ψ to an existentially quantified statement $\exists\alpha\,\varphi$, where, for some term β, β contained in ψ, ψ = PSVAR[$\alpha\ \beta\ \varphi$]

The formula ψ may contain more than one occurrence of β and it may not be the case that the user wants to generalize over all such occurrences. In typing the command, the user must specify a sequence of occurrence references for each occurrence of a β in ψ that is to be considered.

In order to apply this generalization process FREE[$\beta\,\psi$ OCC] must equal T for each referenced occurrence OCC of β. Moreover, α cannot occur free in ψ(FREEANYWHERE [$\alpha\ \psi$] = NIL). Otherwise, in substituting α for β in ψ to obtain φ and in forming the existential statement $\exists\alpha\,\varphi$, ψ will not be obtainable by PSVAR[$\beta\ \alpha\ \varphi$].

The format for the EG command is

<u><line number> EG <sequence of occurrence references></u>

<u><α> : <β></u>

The user states on which line to find the ψ and specifies each occurrence of β to be considered in the generalization procedure. The program then requests the user to type the variable of generalization α followed by a colon followed by the term β. For example:

(1) G(X,Y) $\rightarrow$ F Y

<u>1 EG 1.2</u>

:: <u>Z:Y</u> (2) $\exists$Z(G(X,Z) $\rightarrow$ F(Z))

<u>1 EG 1</u>

:: <u>Z:Y</u> (3) $\exists$Z(G(X,Z) $\rightarrow$ F(Y))

but the request

<u>1 EG 1.2</u>

:: <u>X:Y</u> could not be carried out since this would generate the expression $\exists$X(G(X,X) $\rightarrow$ F(X)) in which the second occurrence of X is no longer free. Consequently, substituting Y for X in this expression does not yield the original formula of line 1.

Rule of Existential Specification (ES). In order to infer that what is true of something is true of a particular thing, the user calls on the rule of existential specification (or instantiation) ES. This rule lets us deduce from the existential statement $\exists \alpha \varphi$ the formula ψ, where ψ = PSVAR[$\beta \alpha \varphi$] for some variable β. Since it is possible to generate fallacies in identifying variables, a restriction is placed on the variable β: β must be new to the derivation; it must not have occurred in any previous line. (This includes nonantecedent lines and is somewhat overly cautious.) Included in the meaning of "occurred in any previous line" is the restriction that the user must not have stated an intent to universally generalize over β. Like the US rule, the interpreter must ensure that the computed type for β is consistent with that for α. The command format for ES is

<line number> ES
<α> :: <β>

where the line referenced must contain an existential statement. The program types the variable of generalization α on the next line and requests the user type β. If β has occurred in any previous line, an error message is printed. Otherwise, ψ is computed from φ by PSVAR[$\beta \alpha \varphi$].

An example of the command sequence is:

```
                (1)  ∃Y(F Y → G X)
                (2)  ∃Y(G Y)
                (3)  G X
2 ES
Y::Z            (4)  G Z        (Z does not occur in lines 1, 2, or 3.)
2 ES
Y::X                            (The error message tells where X was
                                 first encountered.)
   X CANNOT BE USED AS A VARIABLE OF INSTANTIATION.
   YOU ALREADY KNOW SOMETHING ABOUT X.
   X IS ALREADY MENTIONED IN LINE 1.
```

47

The Logic of Identity (IDC and IDS). Two new primitive rules of inference are needed to obtain a formulation of the first-order predicate calculus with identity. (These rules occur in Kalish and Montague, 1864, p. 220.) No attempt is made here to justify the choice of these two primitive rules from among the several possible ones other than to demonstrate that, with these two rules at our disposal, the interchange laws (RE and RQ), the law of symmetry (CE), the law of reflexivity (LT) and the transitivity of identity (TR) are derivable as rules of inference. Of these five rules, only RE and RQ are described in detail. Use of CE as a rule of inference should be clear from the discussion on derived rules of inference and the property lists given in Appendix II.

The rules have been named IDC and IDS, with the 'ID' part standing for 'identity'. As shown below, the command format for the first rule, IDC, is similar to that for EG and represents a form of conditionalizing for identity. Likewise, IDS is a form of specification where the term of instantiation is already specified within the formula itself.

The command format for IDC is:

<line number> IDC <sequence of occurrence references>

:: <α> : <β>

where <line number> points to the formula ψ. In order to apply this rule for each occurrence reference OCC, FREE[$\beta\ \psi$ OCC] must equal T. The user specifies the variable of generalization α and the term β such that α is not contained in β. Then φ is derived from ψ by replacing each referenced free occurrence of β by the variable α. Since ψ must be obtainable by PSVAR[$\beta\,\alpha\,\varphi$], the rule can be used only if α does not occur free anywhere in ψ.

IDC permits the following possible sequence of lines to occur as part of a possible derivation.

P	(1)	F(A)	premise
P	(2)	X=X	premise
1 IDC 1			
:: B:A	(3)	$\forall$B(B=A $\rightarrow$ F(B))	
2 IDC 2			
:: Z:X	(4)	$\forall$Z(Z=X $\rightarrow$ X=Z)	

The command format for IDS is

<line a> IDS

where <line a> points to a formula of the form $\forall\alpha(\alpha=\beta\rightarrow\varphi)$. The new line generated is ψ where ψ comes from φ by proper substitution of β for α. Figures 11, 12, and 13 establish that the interpreter, with IDS and IDC as primitive rules of inference, captures first-order logic with identity.

Insert Figures 11, 12, 13 about here

The proof of the law of reflexivity ($\forall X(X=X)$) is shown in Figure 11. If this law is named as a theorem, say LT, then it may be used to prove CE, the symmetry law (Figure 12). And that identity obeys the transitivity law, TR, is proved in Figure 13. The three formulas (LT, CE, and TR) are the first-order axioms sometimes taken as definitive of identity.

Replace Equals Rule (RE). It can be shown from the rules of identity that

$$X = Y \rightarrow (F(X) = F(Y))$$

where F is any 1-place operation symbol (see Figure 14). This is Euclid's

---

Insert Figure 14 about here

---

postulate that corresponds to saying, if equals are substituted for equals, the result is equal. As a derived rule of inference, we named this pattern RE.

Let α,β be terms, and let φ be a well-formed formula of the theory. Let line 1 of a derivation contain the formula φ, and let line 2 have a formula of the form $\alpha=\beta$. Then the replace equals rule (RE) says that if there is an occurrence of α in φ then ψ (a new line of the derivation) is obtainable from φ by replacing the occurrence of α by β.

For example, if the derivation has lines

(1) $C + (0 + A) = A + C$

(2) $0 + A = A + 0$

then, by the RE rule, where α = 0+A, replacing α by β = A+0 generates the new line: C + (A + 0) = A + C.

Suppose there is more than one occurrence of α in φ. Let n = the number of such occurrences and $1 \leq k \leq n$. The kth occurrence of α in φ is determined by scanning φ from left to right searching for exact pattern matches with α, and counting until the kth such pattern match is found. Then the general RE

```
PROVE       A X(X = X)

:*GEN:X$
OK

:*GEN:Y$
OK

:*WP$       (1)                    *Y=X$

:*1.1CP$    (2)               (Y = X)->(Y = X)

:*2UG:Y$    (3)          A Y((Y = X) ->(Y = X))

:*3IDS$     (4)          X = X

:*4UG:X$    (5)    A X(X = X)

CORRECT...
NAME: *LT$
```

Fig. 11. Reflexivity of identity.

```
PROVE       A X(A Y((X = Y) ->(Y = X)))

:*GEN:X$
OK

:*GEN:Y$
OK

:*LT$
::*Y$       (1)              Y = Y

:*1IDC2$
::*X:Y$     (2)              A X((X = Y) ->(Y = X))

:*2US$
X::*X$      (3)              (X = Y)->(Y = X)

:*3UG:Y$    (4)        A Y((X = Y) ->(Y = X))

:*4UG:X$    (5)  A X(A Y((X = Y) ->(Y = X)))
```

Fig. 12. Law of symmetry for identity.

```
PROVE       A X(A Y(A Z(((X = Y) &(Y = Z)) ->(X = Z))))

:*GEN:X$
OK

:*GEN:Y$
OK

:*GEN:Z$
OK

:*WP$       (1)                        *(X=Y) & (Y=Z)$

:*1RC$      (2)                        Y = Z

:*1LC$      (3)                        X = Y

:*2IDC1$
::*X:Y$     (4)                        A X((X = Y) ->(X = Z))

:*4US$
X::*X$      (5)                        (X = Y)->(X = Z)

:*5.3AA$    (6)                        X = Z

:*1.6CP$    (7)                   ((X = Y) &(Y = Z))->(X = Z)

:*7UG:Z$    (8)             A Z(((X = Y) &(Y = Z)) ->(X = Z))

:*8UG:Y$    (9)        A Y(A Z(((X = Y) &(Y = Z)) ->(X = Z)))

:*9UG:X$    (10)  A X(A Y(A Z(((X = Y) &(Y = Z)) ->(X = Z))))

CORRECT...
NAME: *TR$
```

Fig. 13. Law of transitivity for identity.

```
PROVE       A X(A Y((X = Y) ->(N X IFF N Y)))

:*GEN:X$
OK

:*GEN:Y$
OK

:*WP$       (1)                  *X=Y$

:*WP$       (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

:*WP$       (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 XIFF 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$

DERIVE      A Z(A Y((Z = Y) ->(F Z = F Y)))


:*GEN:Z$
OK

:*GEN:Y$
OK

:*WP$       (1)                  *Z=Y$

:*THA$   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))

:*5.1AA$    (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)

:*8.2AA$    (9)                  F Z= F Y

:*1.9CP$    (10)             (Z = Y)->(F Z = F Y)

:*UG:Y$     (11)        A Y((Z = Y) ->(F Z = F Y))

:*UG:Z$     (12)  A Z(A Y((Z = Y) ->(F Z = F Y)))

CORRECT...
```

Fig. 14. 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 reflexitivity of the identity. Note the use of the PS procedure.

rule says that ψ is obtainable from φ by replacing the kth occurrence of α in φ by β.

The format for the RE rule is

<u><line a> . <line b> RE <occurrence reference></u>

In other words, <line b> must refer to a line on which there is an identity formula, and <line a> must have at least k occurrences of the left-hand side of this identity formula. Then the kth occurrence, K can be replaced by the right-hand side of the identity only if (a) FREE[α φ K] = T; and (b) FREE[β PSVAR [β α φ] = T. This, of course, amounts to PSVAR[β α <line a>].

In the first example, <occurrence reference> = 1. The command is <u>1.2RE1</u>. As another example, let the two lines of a derivation be:

(2) $A+(B+C) = (A+(B+C)) + 0$

(4) $A+(B+C) = 6$.

Then, <u>2.4RE2</u> generates the new line: (5) $A+(B+C) = 6+0$. If the command had been <u>2.4RE3</u> (if <occurrence reference> >2), then the rule would not apply and an error message would be typed.

<u>Replace Equivalents Rule</u> (RQ). The replace equivalents rule, RQ, the second of the interchange rules, is similar to RE. Let line 1 of a derivation contain the formula ψ, and let line 2 have the formula of the form φ IFF φ'. The RQ rule may be described as follows: if there is an occurrence of φ in ψ, then ψ' (a new line) may be inferred from ψ by replacing the occurrence of φ by φ'. Furthermore, like RE, the RQ rule must consider which occurrence of φ to replace. The format for the RQ rules is:

<u><line a> . <line b> RQ <occurrence reference></u>

For example,

(1) $(\forall X\ F\ X\ \&\ \exists Y\ G\ Y) \rightarrow \forall X\ F\ X$

(2) $(\forall X\ F\ X)$ IFF $(\forall Y\ F\ Y)$

<u>1.2RQ2</u>

(3) $(\forall X\ F\ X\ \&\ \exists Y\ G\ Y) \rightarrow \forall Y\ F\ Y$

The second line referenced must contain a material bi-conditional, otherwise an error message is printed.

4. <u>Generalized Interchange Rules</u>

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 φ IFF ψ. The interpretation is based on the RE and RQ rules. The format for all such short forms is:

<u><line number> <name of axiom or established theorem> <occurrence reference></u>

Suppose a derivation in elementary algebra contains the line:

$$(1)\quad A + B = A + ((-C) + (B + C))$$

Frequently in proof constructions requiring pattern manipulations, the student may 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 $(-C) + (B + C)$ with the term $(B + C) + (-C)$. This requires an application of the CA axiom: $A + B = B + A$, followed by the RE rule. The steps of the derivation would be:

```
CA       A + B = B + A
A::-C
B::B+C   (2)  (-C) + (B + C) = (B + C) + (-C)
1.2RE1   (3)  A + B = A + ((B + C) + (-C))
```

The shortcut method for obtaining the formula on line 3 is to use the command: <u>1CA3</u>. Since the command name is an axiom of 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 δ for obtaining the proper instantiation of the axiom CA.

To illustrate the procedure further, and especially to show why the occurrence reference is 3, consider again line (1) above. There are four different possible applications of the CA axiom, i.e., instances of the left-hand side of the pattern for CA. Scanning left to right, they are

1. $A + B = B + A$
2. $A + ((-C) + (B + C)) = ((-C) + (B + C)) + A$
3. $(-C) + (B + C) = (B + C) + (-C)$
4. $B + C = C + B$

Since line 2 of the example corresponds to application 3, the occurrence number must be 3.

In summary, this shortcut use of axioms and theorems is always permitted if the formula associated with the axiom or theorem is of the form $\alpha = \beta$ or φ IFF ψ. In using the shortcut, the student must, of course, determine the correct occurrence reference.

5. 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 in Section 4). An even more flexible framework is achieved by providing a means for deriving new rules of inference--the so-called "derived rules."

To every theorem of logic there is a corresponding derived rule which is indispensible from the standpoint of decreasing the number of steps necessary for a proof or derivation. The use of a derived rule is effectively an iteration of proper substitution, application of the rules to form a conjunction (FC) and affirm the antecedent (AA). The instructional program contains two algorithms for deriving new rules: one for theorems of the form $\varphi \rightarrow \psi$, and the other for theorems and axioms of the form $\alpha = \beta$ and φ IFF ψ.

The second kind of derived rule is merely a commuted form of the shortcut commands we presented in Section 4. 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 associate axiom AS for an additive group: (A + B) + C = A + (B + C). The shortform is usually called AR (associate right). A new rule, AL (associate left) is obtained 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%). Observe the derivation:

```
DERIVE  4 + (3 + 2) = (4 + 2) + 3
AS      (A + B) + C = A + (B + C)
A.:4
B::3
C::2    (1)  (4 + 3) + 2 = 4 + (3 + 2)
1AR1    (2)  4 + (3 + 2) = 4 + (3 + 2)
2CA4    (3)  4 + (3 + 2) = 4 + (2 + 3)
3AL2    (4)  4 + (3 + 2) = (4 + 2) + 3
```

To process the rule, the program computes the information

PREMISE	((+ %A%(+ %B% %C%)))
CONCL	(+ (+ %A% %B%) %C%)
NOP	1
OCCUR	1

These attribute-value pairs are stored on the property list of AL.

In the first type of derived rule, the pattern of the theorem is transformed into a conditional statement P such that the consequent is not a conditional. Then the premises, i.e., the patterns of the conjuncts of the antecedent of P, are patterns for the lines which the rule must reference. The number of line references is the number of conjuncts. The result of using the rule of inference is the proper instance of the pattern of the consequent of P. The algorithm for obtaining P is as follows.

Let the theorem be of the form $\delta\varphi$ where δ is a string of universal quantifiers with their variables, namely, of the form $\forall\alpha_1,\ldots,\forall\alpha_n$ where each α_i,i=1,...,n, is an individual variable; φ is a conditional statement. Then a rule is derived from the (closure of the) theorem by the following algorithm:

1. φ' is obtained from φ by iterated proper substitution of α_i by $\%\alpha_i\%$ in φ, for i=1,...,n.
2. φ'' is obtained from φ' by replacing each occurrence of a predicate letter by its dummy. By this replacement process, the predicates are recognizable as substitutable elements of the expression.
3. φ'' is now a pattern for the theorem. Any symbols not replaced by dummies are constants that must appear in the lines referenced in the rule command.
4. φ'' is in the form $\Delta\rightarrow\psi$, where ψ itself may be a conditional. In order to compute attribute-value pairs used in processing the rule, φ'' must be in the form of a conditional whose consequent is not a conditional statement.

   By repeated application of the Deduction Theorem (Mendelson, 1964, p. 61), if $\vdash \eta \rightarrow (\mathcal{E}\rightarrow\delta)$ then $\vdash (\eta \,\&\, \mathcal{E})\rightarrow\delta$, φ'', is transformed into the desired form, $\alpha\rightarrow\beta$, where β is not a conditional.

55

5. Let N denote the name of the new rule. Then β is placed on the property list of N under the attribute name CONCL.
6. The list of premises is formed from the conjuncts of the antecedent α. The premises are placed on N's property list under the attribute PREMISE. Each element of the list is a pattern which, in processing the rule, will be matched with a line in the derivation ("Matched" in the sense of determining how to instantiate the pattern in order to obtain the line.)
7. NOP is the number of conjuncts (or the length) of the PREMISE list. It is, specifically, the number of reference lines which will be expected in the command.

The command format is:

<u><sequence of NOP line numbers> <rule name></u>

Each line number refers to a line of the derivation that must match a corresponding pattern on the PREMISE list. There are, of course, NOP of these ("match" here means "is an instance of"). The sequence of line references must be ordered with respect to the order of the elements on the PREMISE list.

In summary, the theorem is transformed into a conditional statement such that the consequent is not a conditional. Then three attributes are placed on the property list associated with the rule name:

PREMISE	<list of the conjuncts of the antecedent in their dummy PATTERN notation>
NOP	<the number of conjuncts>
CONCL	<the consequent in its dummy PATTERN notation>

Two examples are offered in order to illustrate the derived rule procedure.

NAME: <u>FC</u>

Form a conjunction is a rule of logic that 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 is $P \to (Q \to (P \& Q))$. The first premise is P. The consequent of this theorem, a conditional statement, is replaced by its consequent P& Q and Q becomes the second premise of the derived rule pattern. Now the consequent is not a conditional and is stored as the conclusion (CONCL). On the property list of FC, where %P% and %Q% are dummy names for the formulas P and Q, respectively, we store:

```
PREMISE   (%P% %Q%)
NOP       2
CONCL     (& %P% %Q%)
```

Now, if a derivation has lines

(1) $A = B$

(2) $A + B = C$

then the command <u>1.2FC</u> generates the line $(A = B)$ & $(A + B = C)$. The command has the correct number of line references and no occurrence numbers, which is characteristic of all rules of this nature. The new line is obtained by forming a conjunction with the lines 1, 2 as the conjuncts.

The second example is the rule AA which, as was mentioned earlier, is processed the same as the derived rule of inference <u>although it is a primitive rule in the system</u>. The theorem is $((P \rightarrow R) \& P) \rightarrow R$. By the above algorithm, the computed attributes are:

```
PREMISE   ((→ %P% %R%) %P%)
NOP       2
CONCL     %R%
```

Commands that call for the application of a rule of inference are executed by a matching process that attempts to determine whether each line referenced in the command is an instance of a corresponding dummy pattern on the PREMISE list. For each line referenced, a routine initially receives the formula on that line, the corresponding dummy pattern from the PREMISE list, and the message (LINE <line number>). By recursive calls, with the elements of each expression as arguments, i.e., the lists or atoms within the expression lists (T_i, V_i), the routine builds a message that is the name of the location of the T_i in the initial formula. The routine examines the atomic element (variable or predicate), or the first element (main connective of a term or a formula) of the list if it is nonatomic, in order to form a sequence of substitutions by which the referenced lines of the derivation may be obtained from the premises. Once this sequence is computed, the corresponding instance of the conclusion pattern (CONCL) is generated as a new line of the derivation.

For example, if the derivation contains lines:

(1) $A = B \rightarrow B = C$

(2) $A = B$

(3) $B = A$

and the user types

<u>1.2AA</u>

then the interp[illegible]r will call on the checking or matching routine with the first premise in the PREMISE list associated with AA and the formula on the first line referenced. Below is a trace of this procedure:

1. T: (→ %P% %R%) V: (→ (= A B)(= B C)) M: (LINE 1)

   At this point, the main connective of T, the pattern of the first premise, is the same as V, the line of the derivation. M describes V.

2. T1: %P% V1: (= A B) M1: (ANTECEDENT OF LINE 1)

   %P% is paired with (= A B) and M1.

3. T2: %R% V2: (= B C) M2: (CONSEQUENT OF LINE 2)

   %R% is paired with (= B C) and the message M2.

   Saving the above two pairs, the program continues the process with the second premise and references the formula on the second line.

4. T3: %P% V3: (= A B) M3: (LINE 2)

%P% was already paired with (= A B) which is identical with V3, so no inconsistent pairing has been located. There are no more line references. By substituting into the value of CONCL with respect to the substitution pairs determined in 1-4, the new line: (B = C) is generated.

The command <u>1.3AA</u> is processed as above except, in 3, V3' is (= B A). Since this is not identical with the value already paired with %P%, an error message is formed from M1 and M3:

LINE 2 MUST BE THE ANTECEDENT OF LINE 1.

The derivation in Figure 15 demonstrates other message types. The teaching sequence that we can provide for all derived rules of inference, using the same matching routine traced above, show explicitly how messages are recursively computed.

Insert Figure 15 about here

```
DERIVE     Q

:*PS       (1)  *P & QS

:*PS       (2)  *R -> PS

:*PS       (3)  *Q  ->PS

:*PS       (4)  *P -> QS

:*1.2HSS
     LINE 1 MUST BE A CONDITIONAL

:*2.3HSS
     ANTECEDENT OF LINE 3 MUST BE THE CONSEQUENT OF LINE 2

:*?S
     WHICH COMMAND DO YOU NEED HELP WITH?
*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?*YS
THERE IS AN EXAMPLE OF THE HS RULE IN THE LINES OF YOUR DERIVATION.
A CORRECT COMMAND IS:
2.4HS

THE NEW LINE WOULD BE: R -> Q
WOULD YOU LIKE ANOTHER EXAMPLE?*YS
A CORRECT COMMAND IS:
3.4HS

THE NEW LINE WOULD BE: Q -> Q
WOULD YOU LIKE ANOTHER EXAMPLE?*YS
A CORRECT COMMAND IS:
4.3HS

THE NEW LINE WOULD BE: P -> F

:*2AES
     AE IS NOT A RULE
```

```
:*2AES
     THE MAIN CONNECTIVE OF LINE 2 MUST BE ONE OF: & < >

:*1LBS
     RULE LB HAS 2 FORMS.
     NONE WERE SATISFIED BECAUSE EITHER:
     LEFT CONJUNCT OF LINE 1 MUST BE A CONDITIONAL OR
     LINE 1 MUST BE A BICON[illegible]NAL.

:*3?LBS

WITHOUT ?, YOU TYPED THE CORRECT FORMAT.
MORE?*NS
WOULD YOU LIKE AN EXAMPLE?*YS

FOR FORM 1
IF YOUR DERIVATION CONTAINS LINES:
(5)  S IFF R

THEN A CORRECT COMMAND WOULD BE:
5LB

AND THE NEW LINE WILL BE:
(S -> R)&(R -> S)


FOR FORM 2
THERE IS AN EXAMPLE OF THE LB RULE IN THE LINES OF YOUR DERIVATION.
A CORRECT COMMAND IS:
4LB

THE NEW LINE WOULD BE: Q IFF P

:*WPS      (5)       *A=BS

:*5DES
::*0S
     YOU MAY NOT DIVIDE BY 0

:*FD?S
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

:*1RCS     (6)     Q

:*5DLLS

:*1RCS     (5)  Q

CORRECT...
```

Fig. 15. Reaching the command language.

Optional Attributes for Derived Rules. Several other attributes may optionally appear on the property list of a rule. As yet, they are not entered via the above algorithm, but rather by direct editing of the property list. They are:

1. REQ and TYPE. A rule may require the student to enter a particular type of well-formed term (or formula). If a new rule is derived from an expression containing a free variable (or predicate), the free variable (or predicate) can be replaced by a well-formed term (or formula). (The inclusion of bound predicates, of course, transcends first-order logic.) 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%))
   NOP       1
   REQ       T (for 'term')
   TYPE      ALGEBRA
   CONCL     (= ( +%A% REQ)( +%B% REQ)).
   ```

2. OR. The main connective of the single premise of AE can also be one of the inequality signs. The options, >, <, and =, are specified by listing each premise on a PREMISE list that begins with the atom OR. CONCL must be a list of patterns corresponding to each optional set of premises. Whichever set matches the referenced lines generates the desired substitution list. 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))).
   ```

   Observe the error message and the teaching sequence for the LB rule in Figure 15 for a sample of how the OR option affects the analysis routines.

3. %%%. As long as the only difference between sets of premises is the main connective, the special character %%% in a premise indicates the optional list of main connectives. Then %%% is stored on the property list as the attribute whose value is the list of main connectives. Again, observe the handling of the AE rule in Figure 15.

```
     PREMISE    ((%%% %A% %B%))
     %%%        ( = >  < )
     CONCL      (%%% ( + %A% REQ)( + %B% REQ))
```

4. RESTRICT. This attribute is used when special restrictions on the use of a rule must be specified. The value of RESTRICT is an executable LISP S-expression. As an example, take the Divide Equals rule (DE) in which the user may not 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 O)(ERR (QUOTE "YOU MAY NOT
                                        DIVIDE BY ZERO"))))
                (T T))
```

Note that if the main connective were changed to %%% so as to include the inequalities RESTRICT would have to be extended to account for negative values of REQ.

How to Specify New Rules of Inference. Derived rules of inference whose property lists require only a list of PREMISES (no options) and a CONCL are easily generated by the interpreter. The user, in either STUDENT or TEACHER modes types

RULE

The program first requests

NAME: <name of the new rule>, then

FROM: <axiom or theorem from which to derive the rule>.

The name cannot be a reserved name, i.e., one of the procedure names or a primitive rule of inference. It must be an alphabetic string of characters with arbitrary length greater than one. The axiom or the theorem referenced must be a (possibly quantified) conditional statement, an identity statement, or a bi-conditional as described above. After the derivation of the rule is completed, the program will type 'OK'. In what follows, let δ be a string of universal quantifiers with the corresponding variables of generalization. Algorithmically what takes place is the following.

1. Determine if the user is allowed to reference the axiom or theorem. In TEACHER mode, this only requires checking to see whether the axiom or theorem exists. In STUDENT mode, as defined later, it requires seeing whether the student actually knows the axiom or the theorem.
2. Let φ be the value of CLOSED. Is the main connective of the matrix of φ an identity or a bi-conditional statement? If neither, go to step 4.
3. Commands of the form <line><name><occurrence> are automatically generated for axioms and theorems with matrices of the form $\alpha = \beta$ and φ IFF ψ. Procedure completed.
4. If the matrix of φ is not a conditional, no rule can be derived. Procedure completed with an error message.
5. Otherwise, carry out the procedure outlined on page 48 for deriving rules of inference.

Figure 16 is the second in the series of dialogues that began with Figure 3. Here, the teacher adds the axioms, two theorems, and some of the rules to be used in constructing derivations of expressions of elementary algebra. Figure 17 lists the properties of these newly specified axioms and theorems. It ends with two derivations for the same formula, one using the specified theorem and one using the rule derived from that theorem. A complete list of derived rules of inference for the sentential calculus is given in Appendix II.

Insert Figures 16 and 17 about here

```
*(START)
WHO ARE YOU (TYPE S OR T)?
*T$
PLEASE TYFE YOUR NUMBER.
*3$
DO YOU WANT TO CREATE OR ALTER A THEORY?(TYPE C OR A)
*A$
THEORY NAME?
*EX$

HI...
IN SETTING UP A FIRST-ORDER THEORY YOU MUST SPECIFY
THE VOCABULARY AND THE AXIOMS.  THEN YOU CAN CHOOSE
A SET OF WELL-FORMED FORMULAS AS THEOREMS, AND DERIVE
NEW RULES OF INFERENCE FROM THESE THEOREMS.  THE
COMMANDS ARE: VOCAB, AXIOM, THEORZM, AND RULE.  TYPE
FIN WHEN YOU ARE THROUGH.


 :*AXIOM$
 NAME: *AXA$
 WFF: *(B(X,Y,X))->(X=Y)$

 :*AA<A>XIOM$
 NAME: *AXB$
 WFF: *B(X,Y,Z) -> B(Z,Y,X)$

 :*AXIOM$
 NAME: *AXC$
 WFF: *(B(X,Y,W) & B(Y,Z,W)) -> B(X,Y,Z)$

  :*AXIOM$
  NAME: *AXD$
  WFF: *(B(X,Y,Z) OR B(Y,Z,X)) OR B(Z,X,Y)$

  :*AXIOM$
  NAME: *AXE$
  WFF: *(((NOT (Y=Z)) & B(X,Y,Z)) & B(Y,Z,W)) -> B(X,Z,W)$

  :*THEOREM$
  THEOREM NUMBER: *1$
  WFF: *B(X,YY,<,Y>,Z) -> ((B(Y,X,Z))-> (X=Y))$

  :*RULE$
  NAME: *SP$
  FROM: *TH1$

  :*FIN$
  I
  *
```

Fig. 16. Specifying axioms.

63

```
*(PROPERTYLIST AXA)

VAL        (B(X,Y,X))->(X = Y)
CLOSED     (A X(A Y(->(B %X% %Y% %X%)(= %X% %Y%))))
PATTERN    (->(B %X% %Y% %X%)(= %X% %Y%))
REQUEST    X Y
TYPE       POINT POINT

T
*(PROPERTYLIST AXB)

VAL        B(X,Y,Z)-> B(Z,Y,X)
CLOSED     (A X(A Y(A Z(->(B %X% %Y% %Z%)(B %Z% %Y% %X%)))))
PATTERN    (->(B %X% %Y% %Z%)(B %Z% %Y% %X%))
REQUEST    X Y Z
TYPE       POINT POINT POINT

T
*(PROPERTYLIST AXC)

VAL        (B(X,Y,W)& B(Y,Z,W))-> B(X,Y,Z)
CLOSED     (A X(A Y(A W(A Z(->(&(B %X% %Y% %W%)(B %Y% %Z% %W%))(B %X%
                   %Y% %Z%))))))
PATTERN    (->(&(B %X% %Y% %W%)(B %Y% %Z% %W%))(B %X% %Y% %Z%))
REQUEST    W X Y Z
TYPE       POINT POINT POINT POINT

T
*(PROPERTYLIST AXD)

VAL        (B(X,Y,Z)OR B(Y,Z,X))OR B(Z,X,Y)
CLOSED     (A X(A Y(A Z(OR(OR(B %X% %Y% %Z%)(B %Y% %Z% %X%))(B %Z%
                   %X% %Y%)))))
PATTERN    (OR(OR(B %X% %Y% %Z%)(B %Y% %Z% %X%))(B %Z% %X% %Y%))
REQUEST    X Y Z
TYPE       POINT POINT POINT

T
*(PROPERTYLIST AXE)

VAL        (((NOT(Y = Z))& B(X,Y,Z))& B(Y,Z,W))-> B(X,Z,W)
CLOSED     (A Y(A Z(A X(A W(->(&(&(NOT(= %Y% %Z%))(B %X% %Y% %Z%))(B
                   %Y% %Z% %W%))(B %X% %Z% %W%))))))
PATTERN    (->(&(&(NOT(= %Y% %Z%))(B %X% %Y% %Z%))(B %Y% %Z% %W%))(B
                   %X% %Z% %W%))
REQUEST    W X Y Z
TYPE       POINT POINT POINT POINT

T
*(PROPERTYLIST TH1)

VAL        B(X,Y,Z)->((B(Y,X,Z))->(X = Y))
CLOSED     (A X(A Y(A Z(->(B %X% %Y% %Z%)(->(B %Y% %X% %Z%)(= %X% %Y%
                   )))))))
PATTERN    (->(B %X% %Y% %Z%)(->(B %Y% %X% %Z%)(= %X% %Y%)))
REQUEST    X Y Z
TYPE       POINT POINT POINT

T
*(PROPERTYLIST     SF)

PREMISE    (B %X% %Y% %Z%)(B %Y% %X% %Z%)
NOP        2
CONCL      (= %X% %Y%)

T
*
```

```
*(PROBLEM)

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: Y=Z$
DERIVE     Y = Z

:*P$        (1)  *B(Y,Z,X)$

:*P$        (2)  *B(8<8>Z,Y,X)$

:*THIS   B(X,Y,Z)->((B(Y,X,Z))->(X = Y))
X::*Y$
Y::*Z$
Z::*X$      (3)  B(Y,Z,X)->(B(Z,Y,X) ->(Y = Z))

:*3.1AA$    (4)  B(Z,Y,X)->(Y = Z)

:*4.2AA$    (5)  Y = Z

CORRECT...

TYPE A DERIVE, PROVE OR RULE COMMAND

:*DERIVE: Y=Z$
DERIVE     Y = Z

:*P$        (1)  *B(Y,Z,,X)<)X,>X)$

:*P$        (2)  *B(Z,Y,X)$

:*1.2SP$    (3)  Y = Z

CORRECT...

TYPE A DERIVE, PROVE OR RULE COMMAND

:*FIN$
T
*
```

Fig. 17. Property lists for axioms in Fig. 16 and example derivations.

6. Miscellaneous Commands

Five mneumonics are reserved as special commands of the command language. Each will be mentioned, although not always elaborated on. The description of the instructional program will be completed with a section on defining problems.

Enter a Line (ENT). This rule should usually not be available to a student. Like WP, ENT expects the user to type a well-formed formula. Unlike WP, nothing is being asserted. No proof procedure is beginning so no procedure must be closed. If the line typed is the problem expression itself, then the program thinks the problem is solved.

While the ENT rule may be used to type in expressions to determine whether they are well formed, it usually is used to debug the curriculum without constructing an entire proof, or to quickly finish a problem that has given the user too much trouble. By requesting this problem at another time, he can retry it when he feels better prepared to do so. At line n., the format is

ENT (n.) <well-formed formula>

Delete the Last Line (DLL). The user conceivably could request a working premise or a GEN to indicate the beginning of a derivation procedure and then could decide, any number of lines later, that the strategy he chose was incorrect. By the rules of construction, however, he either must close off the procedure or delete the command that denoted the beginning. The delete-the-last-line rule (DLL) lets the user delete the last line from consideration. Another use for DLL is to delete a sequence of irrelevant lines that are only confusing the user's perception of the problem.

DLL alone permits the deletion of the last line of the derivation or proof. If the last line is a working premise, then the line is removed as a conditional premise. If a GEN request occurs after the last line, then that request is eliminated.

The format is <line number> DLL

The DLL rule lets the user delete more than one line at a time. All the lines of the derivation or proof, beginning with <line number>, are deleted. If the derivation already has n lines and the command given is j DLL where $j \leq n$, then the lines $j, j+1, \ldots, n$ are deleted. The derivation under consideration is now $j-1$ lines long. The only restriction on this rule is that no premise entered as part of the statement of the problem can be deleted.

INIT. The INIT command, if permitted by the curriculum writer for the DERIVE or PROVE problem being presented, defers solution of the problem until the student decides he is ready to continue. By typing INIT, the student announces his desire for the initiative to request his own DERIVE or PROVE problems, or to derive a new RULE of inference (see page 70 on defining problems).

When the student types the usual terminating command, FIN, the system returns to the problem originally interrupted. The problem is presented anew. If the student, while he had the initiative, specified lemmas or rules, he could now use them to solve his current and future problems. Appendix IV, problem 5, contains an example of the INIT command.

7. Inferfacing Mechanical Theorem Provers to the Instructional System

SHOW. The instructional system under discussion was designed primarily for teaching the construction of derivations and proofs. The system's command language leaves most of the typing of newly generated lines to the computer and relieves some of the tediousness in the student's work. Supposedly, this only leaves him with the task of thinking out the stages of the proof. But trivia is also tedious. No mathematician, when constructing a complex proof, relishes trifling with the laws of commutativity, or simple term eliminations, i.e., obtaining those results normally introduced by 'obviously' or 'clearly' in mathematical discourse. As students of mathematics are introduced to more intricate problems, they too tend to produce less rigorous proofs, yet ones clearly valid to the trained eye. We would like to emulate this behavior--to provide some mechanism by which a student can say to the computer "This line is obviously a valid inference from the work which I have already done, and from instances of such and such axioms and theorems."

We simulate the ability of the 'trained eye' by giving the student access to a SHOW command by which he communicates with a mechanical theorem-prover. The theorem-prover decides whether a new desired line is in fact a trivial deduction from the set of formulas cited. Before revealing how the simulation was carried out and whether our initial results seem promising, we offer an illustration.

In the proof in Figure 18, B is a three-place predicate denoting "betweenness,"

Insert Figure 18 about here

```
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)

 :*AXB      B(X,Y,Z)->B(Z,Y,X)
 X::*Y
 Y::*Z
 Z::*W      (3)        B(Y,Z,W)-> B(W,Z,Y)

 :*3.2AA    (4)        B(W,Z,Y)

 :*SHOW     (5)        ((NOT Z=Y)& B(W,Z,Y))& B(Z,Y,X)
 FROM LINES OF THE DERIVATION?
 ::*1,4
 FROM AXIOMS OR THEOREMS?
 ::*AXB      B(X,Y,Z) -> B(Z,Y,X)
 X::*X
 Y::*Y
 Z::*Z
 ::
 OK? *Y
 LINE 5 IS OK

 :*AXE      (((NOT Y = Z)& B(X,Y,Z))& B(Y,Z,W))-> B(X,Y,W)
 W::*X
 X::*W
 Y::*Z
 Z::*Y      (6)        (((NOT Z = Y)& B(W,Z,Y))& B(Z,Y,X))-> B(W,Z,X)

 :*6.5AA    (7)        B(W,Z,X)

 :*AXB      B(X,Y,Z)-> B(Z,Y,X)
 X::*W
 Y::*Z
 Z::*Y      (8)        B(W,Z,X)-> B(X,Z,W)

 :*8.7AA    (9)        B(X,Z,W)

 :*1.9CP    (10)  (((NOT Y = Z)& B(X,Y,Z))& B(Y,Z,W))-> B(X,Z,W)

 CORRECT...
```

Fig. 18. A proof using the SHOW command.

where B(X,Y,Z) states that point Y stands between points X and Z on the straight line they define. The axioms were listed earlier (page 22). The problem rendered below was given to a number of college students. Those who solved it came up with proofs ranging somewhere between 40 and 70 lines. One student discovered a 17-line solution that was shortened to 10 by using a SHOW command to skip trivial sequences and commands for forming and separating conjunctions.

The student makes the claim that line 5 is a trivial deduction from lines 1 and 4, and from a particular instance of axiom AXB. The theorem-prover decides the claim is justified, and the new line is accepted. The fact that the formula on line 5 might not be proved by the theorem-prover does not necessarily imply that it is not derivable. It only suggests that the formula does not answer our criterion for triviality. Note that the student, <u>not</u> 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 prover does not have to perform complex analyses. That is left to the student. While the theorem-prover is used only to remove trivial manipulations, the determination of proper instantiation often is not trivial.

For the SHOW routine, the instructional system was connected to a theorem-prover based upon the <u>Resolution Principle</u> [Robinson, 1965 and J. Allen & D. Luckman, 1970], a refutation scheme by which a statement is proved true by showing that the conjuction 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 requires computation of the set of clauses on which the theorem-prover performs its resolutions. In order to compute these clauses, the program must form the Skolem transformation of each line of the derivation and of each instance of an axiom or theorem which the student claims should be considered by the prover in its deduction process. Computation of the Skolem transformation requires obtaining the prenex normal form of the formula by a routine that is also used in determining the closure for a formula. (Appendix I contains an algorithm for computing the Skolem transformation of a formula.) The set of clauses for the above problem is exhibited in Figure 19 as axioms 1-5, the negation of the problem statement as axiom 6, and the solution is given as the tree of resolvants.

Insert Figure 19 about here

68---

Axioms

1. ¬E(Y,Z)
2. B(X,Y,Z)
3. B(Y,Z,W)
4. B(W,Z,Y)
5. ¬B(X,Y,Z) B(Z,Y,X)
6. E(Z,Y) ¬B(W,Z,Y) ¬B(Z,Y,X)

Resolution Tree

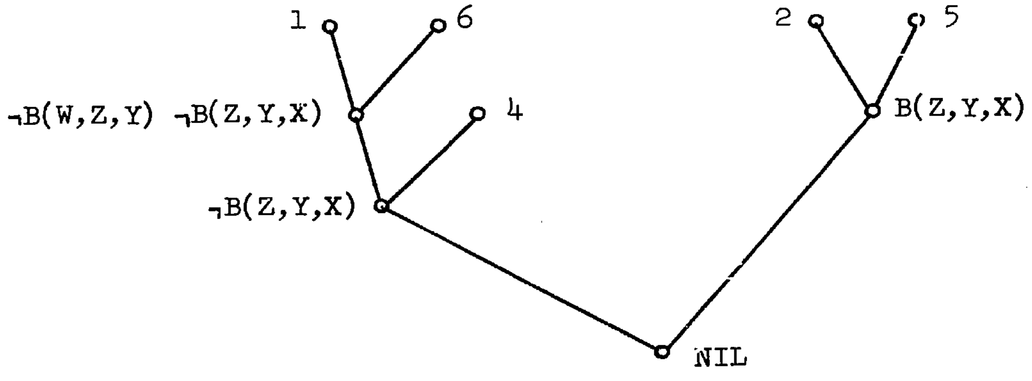


Fig. 19. Proof based on the Resolution Principle.

Although theoretically one might expect this approach to the SHOW command to work, numerous difficulties were encountered in experimental endeavors. Some difficulties are apparently due to the interactive nature of the theorem-prover used and to the strategies required 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 clauses (lemmas) or to change the set of strategies. This need for intervention, while often suitable for research, is not acceptable for teaching purposes. The 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. There is no obvious way, either from the structure of the expression, or from some interpretation of it, to determine which strategies to use. Since the student should not be required to interact with the theorem-prover, one set of strategies anticipating the kinds of problems the students will think of must be used for a given group of problems. The problem given in Figure 18 was not solvable by the theorem-prover. Apparently the program got confused because the search-space consisted entirely of ground clauses. Trouble also arose with the equality strategy (paramodulation) which would not substitute a constant for a constant.* As a result, many simple problems in group theory were not solvable. It would seem that, to implement a satisfactory SHOW command, the development of new strategies for resolution, or of theorem-provers with heuristic devices more closely linked to a particular range of problems, would fare better.

HELP. As mentioned in the introduction to this report, one of the important analysis problems faced in an instructional system for teaching the notion of mathematical proof is to help the student when he encounters difficulty in completing a proof or derivation. If the computer is to simulate the human tutor's ability to show the student how to proceed, the computer must be able to analyze and respond to the details of the student's work. It should not merely produce prestored answers to anticipated questions, or hints to anticipated difficulties. If the computer is to be able to adapt to the immediate needs of each student, it must be able to extract information from each student's responses, especially when they are only partial or erroneous, so as to initiate a dialogue relevant to what the student has been doing.

*This is an error in the program made available to us, not with the paramodulation strategy itself.

In order to sustain an informative dialogue, and thereby to realize a mechanized tutor, a heuristic theorem-prover was written. Ideally, a theorem-prover knows how to do the proofs the students are required to do. The theorem-prover can find solutions to problems that have premises. If one assumes that the lines the student has already requested are premise lines, then the theorem-prover can take them into account when it tries to discover a derivation. In this manner, the computer tutor is able to deduce what the student has already done in terms of what lines of his work can actually enter into a complete derivation. Using the information obtained by the theorem-prover, the computer can initiate a dialogue that will direct the student towards a successful solution of the problem.

To test this idea, a heuristically-based theorem-prover was written that solves problems of an Abelian group. The theorem-prover was interfaced to the instructional system via the command HELP. Any time the student feels the need for advice on how to continue, he types this command. The theorem-prover then attempts to find one or more solutions that take into account lines already generated in the derivation. The information so gained is given to a dialogue routine that proceeds to tutor the student. The precise details of how the theorem-prover operates, especially of how it is able to select several possible solution paths, as well as a description of the dialogue itself, will be given in a subsequent report [Goldberg].

PART IV. DEFINING PROBLEMS

The basic intent of this instructional program is to provide an interpretive system under which a user can explore the notion of mathematical proof. The emphasis is on self-exploration, not on the working through of a predetermined, linearly organized set of problems. Thus, two forms of curriculum specification are described: the syntax for problems that are stored on some peripheral device (such as the disk), and the commands the student has available for requesting problems at runtime. Recall that the command INIT (page 59) gives the student the option to interrupt the prestored curriculum on which he may be working and to switch to the second form of problem specification.

1. Prespecified Curriculum-TEACHER Mode

The syntax for curriculum stored on a peripheral file is given below. There are three types of problems:

1. A question that requires as an answer, a string of characters representing a letter, word, or number;
2. A derive problem, with a sequence of premises, for which the user constructs a derivation; and
3. A prove problem, i.e., a derive problem with no premises, for which the user is required to construct a proof. The expression so proved will have a name associated with it. The name labels the expression as either a theorem or a lemma. All theorem names have the form TH<positive integer>; names for lemmas are strings of one or more alphabetic characters.

The syntax presented is not true in the sense that all possible strings that can be generated from the grammar do not have meaning in the interpretive system. The acceptable groupings of the commands, including those which are optional, follow. Notice that it is necessary to enclose each problem within matched pairs of parentheses for the convenience of the LISP input function READ. In general, the entire problem, as well as any arguments to a command element are delimited by paired parentheses. The exception is COMMENT, which requires quotation marks to enclose the actual text to be printed. An illustration of problems for elementary algebra is offered in Appendix III. Appendix IV is the interaction of a student presented with these problems.

Syntax for Problems on the Curriculum File.

<problem>	::= [<problem number>(<problem type> <problem statement> <restrictions> <answer>]
<problem number>	::= a decimal number used for sequencing the problems
<problem type>	::= DERIVE | PROVE | QUESTION | Q
<problem statement>	::= (<well-formed expression>] PREMISE (<well-formed formula>) | P (<wff>) COMMENT "<text of the comment or question>" | COM NAME (<positive integer>) | NAME (<string of alphabetic characters>) |

```
<restrictions>            ::= RESTRICT (<possible restrictions>)

<possible restrictions>   ::= (YES<list of code names>)<possible restrictions> |
                              (NO<list of code names>)<possible restrictions> |
                              (ADD<axiom or theorem name>)<possible restrictions> |
                              (BLOCK)(possible restrictions) |
                              ∅

<answer>                  ::= ANS (<string for the exact match>) | ANSWER (<string>)
                              RANGE (<lower bound> <upper bound>) |
                              ALIST (<list of possible answers>) |
                              PROOF     Note:  If an answer is more than one
                                               item long, it is enclosed in
                                               parentheses in the ALIST.

<lower bound>             ::= integer number | NIL

<upper bound>             ::= integer number | NIL
```

<u>Acceptable Groupings of Problem Commands</u>.

Question problems

1. [<number> (QUESTION COMMENT "the text of the question"
   ANSWER (an atomic answer]
2. [<number> (Q COM "the question text"
   RANGE (<lower bound> <upper bound>]
3. [<number> (Q COM "text" ALIST (list of possible answers]

Note: Q, COM, and ANS are abbreviations for QUESTION, COMMENT, and ANSWER, respectively. In 2., the answer is a number N such that <lower bound> $\leq$ N $\leq$ <upper bound>. In 3., the list of answers is (a1 a2 (a2 a4)), i.e., a list of atoms and sublists.

Derive problems

```
[<number>
(DERIVE (<well-formed formula>)
COMMENT "text of something the teacher may want to say about the problem"
PREMISE (<well-formed formula>)
PREMISE (<well-formed formula>)
RESTRICT ((YES list of rules)(NO list of rules)(ADD name of an axiom,
        rule, or theorem)(BLOCK))
PROOF]
```

Note: COMMENT, RESTRICT, and PREMISES are optional. Any number of premises, including none, may be included.

Prove problems

```
[<number>
(PROVE (<well-formed formula>)
NAME (<positive integer or an alphabetic string of characters>)
COM "text" RESTRICT ((YES list of rules)(NO list of rules)(BLOCK))
PROOF]
```

Note: If the name is a positive integer, then the name becomes TH<positive integer>. After the problem is solved, the name becomes part of the command language. NAME, COM, and RESTRICT are optional.

The RESTRICT option gives the curriculum writer special control over the command language. There are four possible control devices.

1. In order to add new commands (axiom names, theorems, and rules) to a student's command language, the optional list:

   (ADD <list of one or more names of axioms, rules, or theorems>)

   is included as one of the arguments following the word RESTRICT.
2. The writer may not want the student to interrupt the particular problem to be presented, i.e., use the INIT command. (Perhaps the curriculum is organized so that problems reference one another and the writer prefers to have a group of problems worked on in succession.) The interruption can be blocked with the argument (BLOCK). This option must appear in the specification of each derive or prove problem which the student may not interrupt.
3. The curriculum writer can also take special control over the actual solution to the derive or prove problem. He can insist that the student use one or more rules. These rules are included as an argument list beginning with the word PYES.
4. Or, he can insist that the student find a solution without using certain commands. This is the list beginning with the word PNO. The student is allowed to complete a solution before the interpreter checks to see if that solution meets the restrictions. If a command is incorrectly used, the student will be asked to work the problem again.

"Using a command" should be carefully defined since a student can request a command and then, without deleting it, not really have that command enter into the generation of the problem formula. The exact protocol of commands the student typed does not indicate if the student solved the problem within the restrictions set by the curriculum writer. Instead, a computed list of "relevant" commands, i.e., commands which materially entered into the generation of the problem formula must be examined. The algorithm for computing the list of relevant commands is given in Figure 20.

Insert Figure 20 about here

----

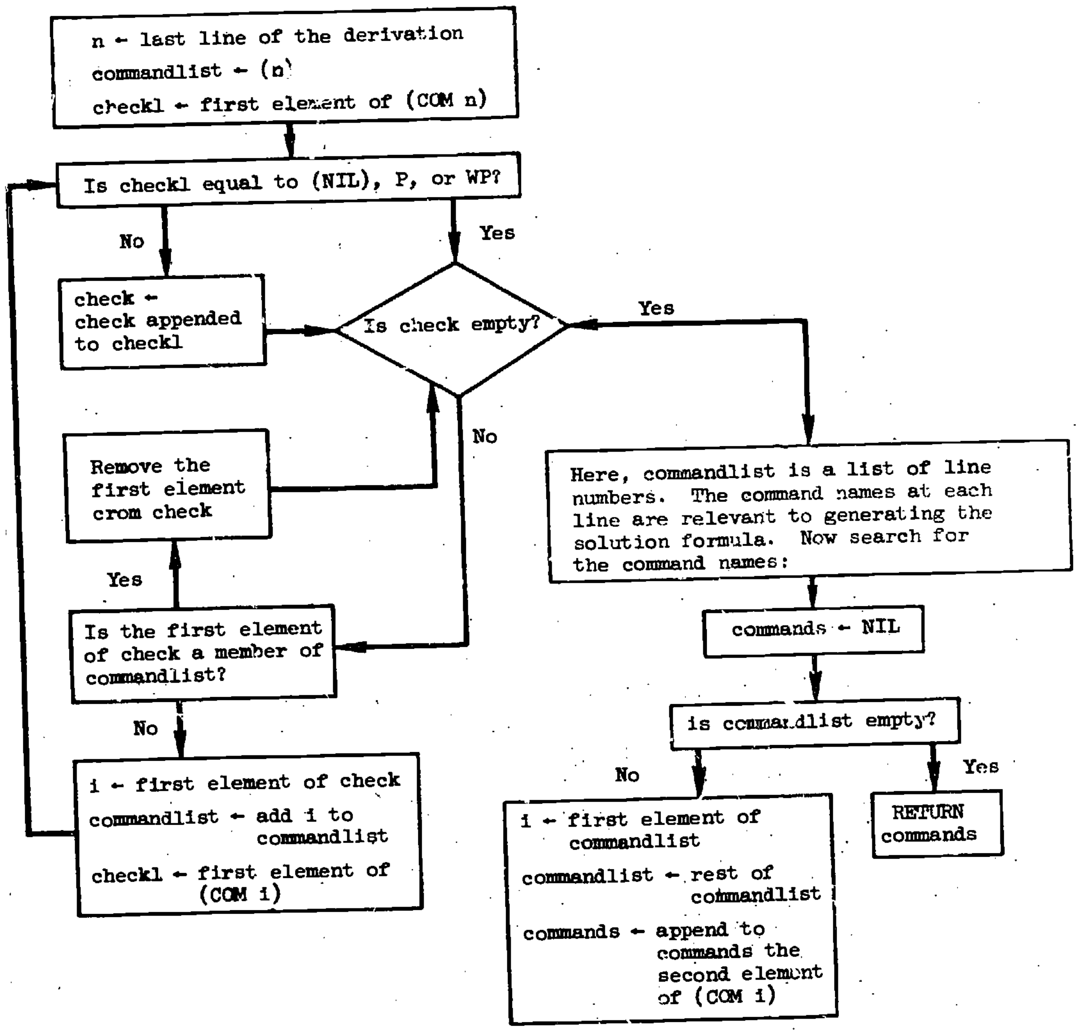
2. Commands for Requesting Problems in the STUDENT Mode

The student requests only the derive and prove problems, not the questions. This give him a chance to rework his proof techniques, repeat problems he may have had trouble solving, try some easier problems if he feels he is not ready for the ones he is being given, or try more challenging ones. Any lemmas he proves become part of his command language and are thus available to him for later derivations. The syntax for requesting problems is compatible with the command language.

1. To request a derive problem, the student types:

```
DERIVE: <well-formed formula>
P         (1)  <well-formed formula>
P         (2)  <well-formed formula>
.
.
.
```

The P command stands for "premise." It is similar to entering a formula for a working premise, but this line cannot be deleted by the DLL rule. Once the student starts typing commands other than P, he can no longer type the P command.



n ← last line of the derivation
commandlist ← (n)
checkl ← first element of (COM n)
Is checkl equal to (NIL), P, or WP?
No
Yes
check ← check appended to checkl
Is check empty?
Yes
No
Remove the first element crom check
Yes
Is the first element of check a member of commandlist?
No
i ← first element of check
commandlist ← add i to commandlist
checkl ← first element of (COM i)
Here, commandlist is a list of line numbers. The command names at each line are relevant to generating the solution formula. Now search for the command names:
commands ← NIL
is commandlist empty?
No
Yes
i ← first element of commandlist
commandlist ← rest of commandlist
commands ← append to commands the second element of (COM i)
RETURN commands

Fig. 20. Algorithm for computing the solution string--the list of commands used in generating the problem formula.

Notationally, the list of commands is saved in an array named (COM i), where i is the line number. (COM i) has the form:

(<list of line numbers> (command name) <list of occurrence references> (<information following the colon>))

2. To request a prove problem, the student types:

   PROVE: <well-formed formula>

   After the student completes the proof, the program will ask him for a

   NAME: <string of alphabetic characters>

   The student's response, an alphabetic string, names the formula as a lemma which now becomes part of his command language.

3. The student may also derive a new rule of inference if he feels it will help him in constructing new derivations. The format is:

   RULE: <name for the new rule--an alphabetic string>

   FROM: <name of an axiom or theorem in the student's command language>

   The same information has been requested here as was requested in the TEACHER mode version of deriving a rule of inference (page 54) and the identical processing algorithm is carried out.

PART V. SUMMARY OF THE COMMAND LANGUAGE

The convention is to underline items typed by the user; all other information is typed by the program. Each user command is terminated by an ALTMODE (Enter Key) which appears as a dollar sign ($). In most cases, the ALTMODE is not shown. Note that the user can type ALTMODE in order to escape from any command sequence.

In the following, α is an individual variable, β 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) of the closed formula associated with 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 ALTMODE key without one of the two possible substitution pairs. The substitution procedures are carried out iteratively.

Proof Procedures

1.	WP	Working Premise	
	WP	(i)	<WFF>
2.	CP	Conditional Proof	
	WP	(i)	φ
		(j)	ψ
	i.jCP	(n)	$\varphi \rightarrow \psi$
3.	IP	Indirect Proof (reductio ad absurdum)	
	WP	(i)	$\underline{\varphi}$
		(j)	ψ
		(k)	NOT ψ
	i.j.kIP	(n)	NOT φ

4. Introduce a variable of universal generalization

GEN: <variable>

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) $\varphi(\alpha)$	(i) $\varphi(\alpha)$
iUG: α	(n) $\forall\alpha\varphi(\alpha)$	iUG: α (n) $\forall\alpha\varphi(\alpha)$

This version of UG need only check the last GEN introduced.

This version of UG requires checking for α free in any antecedent lines.

Primitive Rules of Inference

1. AA Affirm the Antecedent (*modus ponens*)

(i) $\varphi \rightarrow \psi$

(j) φ

i.jAA (n) ψ

2. Quantification Rules (and UG above)

ES Existential Specification

(i) $\exists\alpha\varphi(\alpha)$

iES

α :: β (n) $\varphi(\beta)$ where β must be new to the derivation.

EG Existential Generalization

(i) $\varphi(\beta)$

iEG <sequence of occurrence numbers>

α : β (n) $\exists\alpha\varphi(\alpha)$ where substitute α for each occurrence of β referenced.

US Universal Specification

(i) $\forall\alpha\varphi(\alpha)$

iUS

α :: β (n) $\varphi(\beta)$

3. Logic of Identity IDS and IDC

(i) $\varphi(\beta)$

iIDC <sequence of occurrence numbers>

$\alpha : \beta$	(m)	$\forall\alpha(\alpha=\beta \rightarrow \varphi(\alpha))$	where substitute α for each
m IDS	(n)	$\varphi(\beta)$	occurrence of β referenced.

4. Interchange Rules RE and RQ

(i) $\varphi(\alpha)$

(j) $\alpha = \beta$

i.jRE <occurrence number>

(n) $\varphi(\beta)$ where replace β for the free occurrence

(o) $\varphi(\psi)$ of α referenced.

(p) $\psi \leftrightarrow \psi'$

i.jRQ <occurrence number>

(q) $\varphi(\psi')$ 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>

Miscellaneous Commands

1. Delete the last line DLL

<line number> DLL

All lines other than premises, beginning with <line number> 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. Enter a line ENT

ENT (i) <WFF>

This command is useful for testing expressions for well-formedness or for debugging the curriculum without having to do the proofs.

3. Use of Mechanical Theorem-Provers SHOW and HELP

   SHOW is described in Part II, Section 5.

   HELP is explained in Parts III and IV.

4. Obtain the initiative to request problems at runtime.

   INIT

   a. Derive problems

      DERIVE: <WFF>

      P (1) <WFF>

      P (2) <WFF>

      .
      .
      .

      The student can enter any number of premise lines, continuing until he enters a command different from P or DLL.

   b. Prove Problems

      PROVE: <WFF>

      NAME:: <alphabetic string>

      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 results from the algorithm presented in Part III, Section 5.

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 patterns (under the name PATTERN) on the property list of each rule listed in Appendix II. Note that the logical connectives are written as variable names (e.g., "ARROW", "ORSGN"). After the teacher has specified the vocabulary, the variables in the rules are replaced by the corresponding logical constants. The RESTRICT options are written as LISP S-expressions. The procedure that processes derived rules evaluates these S-expressions in order to check for restrictions on the values of the dummy variables or the requested expressions (REQ), or to (re)compute substitution pairs for the substitution list.

References

Allen, J., & Luckham, D. An Interactive Theorem-Proving Program. In B. Meltzer & D. Michie (Eds.), Machine Intelligence 5. New York: American Elsevier, 1970. Pp. 321-336.

Goldberg, A. Ph.D. dissertation (in preparation).

Kalish, D., & Montague, R. Lo[illegible]c: 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 Demonstrations. Springer-Verlag, 1970.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., & Levin, M. I. LISP 1.5 Programmer's Manual. Cambridge, Massachusetts: The MIT Press, 1962.

Mendelson, E. Introduction to Mathematical Logic. Princeton, New Jersey: D. Van Nostrand, 1964.

Robinson, J. A. A Machine Oriented Logic Based on The revolution Principle. Journal of the ACM, 1965, 12, 23-41.

Suppes, P. Computer-Assisted Instruction at Stanford. Technical Report No. 174, Institute for Mathematical Studies in the Social Sciences, May 19, 1971.

Suppes, P., & Binford, F. Experimental Teaching of Mathematical Logic in the Elementary School. The Arithmetic Teacher, March, 1965, 187-195.

Suppes, P., Jerman, M., & Brian, D. Computer-Assisted Instruction: Stanford's 1965-66 Arithmetic Program. New York: Academic Press, 1968.

Suppes, P., & Ihrke, C. Accelerated Program in Elementary-School Mathematics--The Fourth Year. Psychology in the Schools, 1970, 7, 111-126.

Appendix I

Forming Clauses for the Formal Theorem-Prover

1. SKOLEM TRANSFORMATION routine

Below is a sketch of the seven functions used to form the Skolem transformation of a formula φ.

input string $\eta \leftarrow \varphi$

CLOSURE $\eta \rightarrow \eta$ — Ensure that the formula η is a closed formula.

ELIMARROW $\eta \rightarrow \eta$ — Eliminate the implications signs. So all occurrences of the pattern $(\rightarrow A\ B)$ become $(OR(\neg A)B)$. Any more occurrences of $(\rightarrow A\ B)$?

NO YES

NEGSCOPE $\eta \rightarrow \eta$ — Reduce the scope of the negations signs. Each negation sign $(\neg)$ should apply to at most one predicate letter. So all the

occurrences of:	become:
$(\neg(\& A\ B))$	$(OR(\neg A)(\neg B))$
$(\neg(OR\ A\ B))$	$(\&(\neg A)(\neg B))$
$(\neg(\neg A))$	A
$(\neg(\exists\ x\ A))$	$(\forall\ x(\neg A))$
$(\neg(\forall\ x\ A))$	$(\exists\ x(\neg A))$

NO YES

DISTR $\eta \rightarrow \eta$ — Distribution laws for the quantifiers $\exists$ and $\forall$ are applied. All the occurrences

of:	become:
$(OR(\exists\ x\ A)(\exists\ x\ B))$	$(\exists\ x(OR\ A\ B))$
$(\&(\forall\ x\ A)(\forall\ x\ B))$	$(\forall\ x(\&\ A\ B))$

NO YES

α

1

PRENEX $\eta \to \eta$

η is now in the form $\delta\varphi$ where δ is a string of quantifiers and φ is the matrix.

Standardize the variables--rename variables to ensure that each quantifier has a unique dummy variable. This is done while moving all the quantifiers to the front of η, preserving order. If α is a variable of generalization, renamed as α', the φ is within the scope of the quantifier, then any α contained in φ is renamed as α'. In order to ensure that the actual binding operator still binds the original variables within its scope, variables of generalization which occur in a φ are also renamed as above. When a variable of generalization is found which has already been renamed, it is renamed again. Thus, order and the scope of binding operators are preserved.

$\downarrow$

CNF $\eta \to \eta$

η is now in the form $\delta\varphi'$ where φ' is φ in conjunctive normal form.

Conjunctive Normal form:
The matrix φ resulting from PRENEX η is put into an equivalent form which is the conjunction of a finite set of disjunctions.
(OR A (&B C)) becomes (&(OR A B)(OR A C))

$\downarrow$

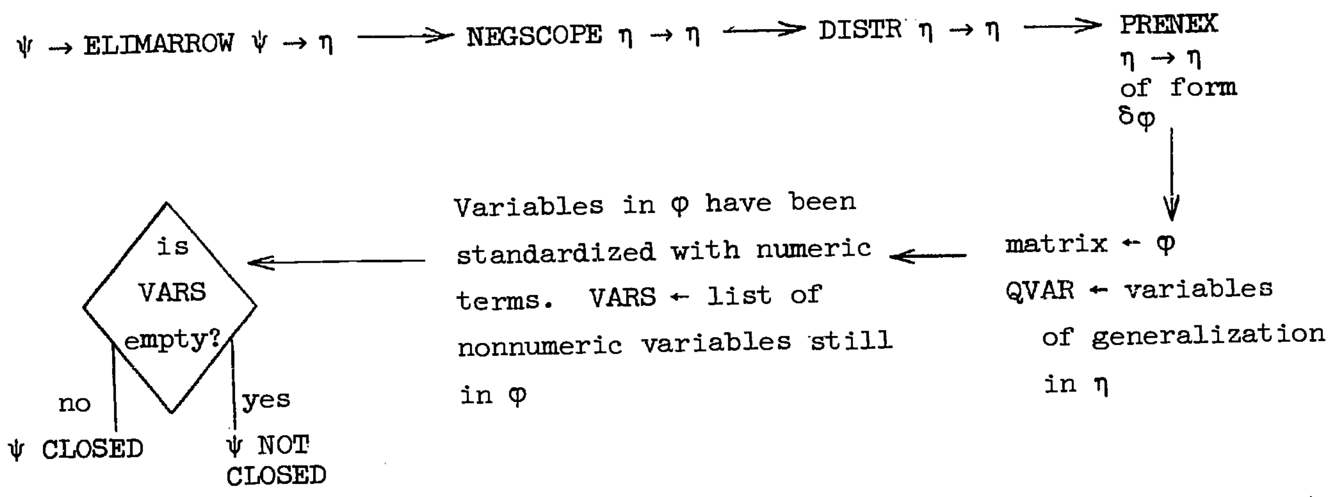
SKOLEM $\eta \to \eta$

Eliminate the existential quantifiers. The usual skolemization is carried out with the skolem functions represented as f<i>, i=1,2,.... The universal quantifiers are dropped.

$\downarrow$

η

2. Is a formula ψ CLOSED?



ψ → ELIMARROW ψ → η
NEGSCOPE η → η
DISTR η → η
PRENEX η → η of form δφ
matrix ← φ
QVAR ← variables of generalization in η
Variables in φ have been standardized with numeric terms. VARS ← list of nonnumeric variables still in φ
is VARS empty?
no
yes
ψ CLOSED
ψ NOT CLOSED

3. FORM a CLAUSE from φ

A clause for the theorem-prover is a disjunction of literals where a literal is either an atomic formula or its negation.

φ → SKOLEM TRANSFORMATION φ → η

CONJLIST η → η
L ← eliminate the conjunction sign.
L is a list of all the conjuncts; essentially all occurrences of (& A B) becomes the list (A B).

DISTLIST η → η
All terms which are constants must be nested, in list form, one list deeper. All disjunctions are eliminated so that a clause becomes a list of the conjuncts which in turn are a list of the disjuncts.

Appendix II

The Rules of Logic

```
AA   *AFFIRM THE ANTECEDENT

PREMISE ((ARROW %P% %Q%) %P%)
CONCL   %Q%
NOP     2

CC   *COMMUTE CONJUNCTION

PREMISE ((ANDSGN %S% %P%))
CONCL   (ANDSGN %P% %S%)
NOP     1
OCCUR   1

CD   *COMMUTE DISJUNCTION

PREMISE ((ORSGN %S% %P%))
CONCL   (ORSGN %P% %S%)
NOP     1
OCCUR   1

CE   *COMMUTE EQUALS

PREMISE ((EQSIGN %A% %B%))
CONCL   (EQSIGN %B% %A%)
NOP     1
OCCUR   1

DC   *DENY THE CONSEQUENT

PREMISE ((ARROW %Q% %R%) (NEGSGN %R%))
CONCL   (NEGSGN %Q%)
NOP     2

DD   *DENY THE DISJUNCT

PREMISE ((ORSGN %S% %P%) %R%)
CONCL   %Q%
NOP     2
(DEFPROP RESTRICT
(COND
 ((DENIAL (CADR (ASSOC (QUOTE %S%) MAINSUB))
          (CADR (ASSOC (QUOTE %R%) MAINSUB)))
  (SETQ MAINSUB (SUBST (QUOTE %Q%) (QUOTE %S2%) MAINSUB)))
 ((DENIAL (CADR (ASSOC (QUOTE %P%) MAINSUB))
          (CADR (ASSOC (QUOTE %R%) MAINSUB)))
  (SETQ MAINSUB (SUBST (QUOTE %Q%) (QUOTE %S%) MAINSUB)))
 (T (ERRMSG 56)))
EXPR)

DM   *DEMORGAN'S LAWS

PREMISE (OR
           ((ANDSGN %Q% %R%))
           ((ORSGN %Q% %R%))
           ((NEGSGN (ANDSGN %Q% %R%)))
           ((NEGSGN (ORSGN %Q% %R%))))
CONCL   (OR
           (NEGSGN (ORSGN %P% %S%))
           (NEGSGN (ANDSGN %P% %S%))
           (ORSGN %P% %S%)
           (ANDSGN %P% %S%))
NOP     1
(DEFPROP RESTRICT
(AND (SETQ MAINSUB
           (APPEND (LIST
                    (LIST (QUOTE %P%)
                          (COND
                           ((AND
                             (NOT
                              (ATOM
                               (CADR (ASSOC (QUOTE %Q%) MAINSUB))))
                             (EQ
                              (CAADR (ASSOC (QUOTE %Q%) MAINSUB))
                              NEGSGN))
                            (CADADR (ASSOC (QUOTE %Q%) MAINSUB)))
                           (T
                            (LIST NEGSGN
                                  (CADR
                                   (ASSOC (QUOTE %Q%) MAINSUB)))))))
                   MAINSUB))
```

```
    (SETQ MAINSUB
          (APPEND (LIST
                   (LIST (QUOTE %S%)
                         (COND
                          ((AND
                            (NOT
                             (ATOM
                              (CADR (ASSOC (QUOTE %R%) MAINSUB))))
                            (EQ
                             (CAADR (ASSOC (QUOTE %R%) MAINSUB))
                             NEGSGN))
                           (CADADR (ASSOC (QUOTE %R%) MAINSUB)))
                          (T
                           (LIST NEGSGN
                                 (CADR
                                  (ASSOC (QUOTE %R%) MAINSUB)))))))
                  MAINSUB)))
EXPR)

DN    *DOUBLE NEGATION

PREMISE (OR
             ((NEGSGN (NEGSGN %S%)))
             ((%S%)))
CONCL   (OR
             %S%
             (NEGSGN (NEGSGN %S%)))
NOP      1

DS    *DISJUNCTIVE SYLLOGISM

PREMISE ((ORSGN %P% %Q%) (ARROW %P% %S%) (ARROW %Q% %R%))
CONCL   (ORSGN %S% %R%)
NOP      3

FC    *FORM A CONJUNCTION

PREMISE (%S% %Q%)
CONCL   (ANDSGN %S% %Q%)
NOP      2

FD    *FORM A DISJUNCTION

PREMISE (%S%)
CONCL   (ORSGN %S% REQ)
NOP      1
REQ      S

HS    *HYPOTHETICAL SYLLOGISM

PREMISE ((ARROW %P% %Q%) (ARROW %Q% %R%))
CONCL   (ARROW %P% %R%)
NOP      2

LB    *LAW OF THE BICONDITIONAL

PREMISE (OR
             ((BICOND %P% %Q%))
             ((ANDSGN (ARROW %P% %Q%) (ARROW %Q% %P%))))
CONCL   (OR
             (ANDSGN (ARROW %P% %Q%) (ARROW %Q% %P%))
             (BICOND %P% %Q%))
NOP      1

LC    *LEFT CONJUNCT

PREMISE ((ANDSGN %S% %R%))
CONCL   %S%
NOP      1

LT    *LOGICAL TRUTH

CONCL   (= REQ REQ)
NOP      0
REQ      T
TYPE     T

RC    *RIGHT CONJUNCT

PREMISE ((ANDSGN %S% %R%))
CONCL   %R%
NOP      1
T
*
```

88

Appendix III

Sample Curriculum File

```
[1 (QUESTION
 COMMENT " 'CA' MEANS 'COMMUTE ADDITION'.  CA ALLOWS
YOU TO SWITCH THE TERMS AROUND THE...

A)  '=' SIGN
B)  '+' SIGN
C)  '<' SIGN. "
  ALIST (B /+]

[2 (Q
  COM "COMMUTE ADDTION IS AN AXIOM WHEREAS
COMMUTE EQUALS IS A...

A) PREMISE
B) DEFINITION
C) RULE OF INFERENCE."
ANSWER (C]

[3 (Q
  COM "HERE IS THE CA AXIOM:  A+B=B+A.
THE CA AXIOM IS A TRUE EQUATION NO
MATTER WHAT NUMBERS 'A' AND 'B' ARE.
WHICH OF THESE IS AN EXAMPLE OF CA...

A)  5+4=5+4
B)  5+4=4+5
C)  4+5=9. "
  ANS (B]

[4 (DERIVE (5 /+ 4 = 4 /+ 5)
  COM  "HERE IS AN EXAMPLE OF HOW TO USE THE
CA AXIOM:

DERIVE:  5+4=4+5

CA$     A+B=B+A
A::5$
B::4$     (1)  5 + 4 = 4 + 5

TRY THE PROOF. "
RESTRICT ((ADD CA))
PROOF ]

[5 (DERIVE (13 /+ 2 = 2 /+ 13)
 PROOF]

[6 (DERIVE (A /+ B = 3 /+ 6)
  COM "TRY USING THE SHORTFORM OF THE CA AXIOM."
  PREMISE (A /+ B= 6 /+ 3)
  RESTRICT  ((NO RE))
  PROOF]

[7 (Q
  COM "WHAT NUMBER CAN 'A' BE IF 'A < 6' IS
A TRUE STATEMENT?"
  RANGE (NIL 5]

[8 (DERIVE (A < 6)
 P  ((A /+ 3 = 5) THEN (A < 6))
 P  (B=3)
 P  (A /+ B = 5)
 PROOF]

[9 (DERIVE (A=B)
COM "'Z' STANDS FOR THE ZERO AXIOM: A+O=A"
P (B /+ 0 = A /+ 0)
RESTRICT ((NO SE)(ADD Z))
PROOF]

[10 (PROVE (0 /+ A = A)
NAME (1)
COM "YOU NOW KNOW FIVE AXIOMS TO USE IN PROOFS."
RESTRICT ((BLOCK)(YES CA Z))
PROOF]

[11 (Q
COM "WE WILL CALL EQUATION O+A=A, THEOREM 1"
COM "SINCE YOU HAVE PROVED THEOREM 1, YOU CAN
USE IT IN OTHER PROOFS.  STUDY THIS...

DERIVE  O+6=6
TH1   O+A=A
A::6   (1)  O+6=6

'TH1' STANDS FOR...

A) THEOREM 1
B) AXIOM 1
C) RULE 1."
ALIST (A (THEOREM 1]

[12 (DERIVE (O /+ 7 = 7)
COM "USE THEOREM 1 IN THIS PROOF. "
RESTRICT ((NO CA Z))
PROOF]
```

Appendix IV

Using the Curriculum of Appendix III

```
*(START)
WHO ARE YOU (TYPE S OR T)?
*S$
PLEASE TYPE YOUR NUMBER.
*6$
ADELE GOLDBERG
OK?
*Y$

1
 'CA' MEANS 'COMMUTE ADDITION'.  CA ALLOWS
YOU TO SWITCH THE TERMS AROUND THE...

A)  '=' SIGN
B)  '+' SIGN
C)  '<' SIGN.

*B$
CORRECT...

2
COMMUTE ADDTION IS AN AXIOM WHEREAS
COMMUTE EQUALS IS A...

A) PREMISE
B) DEFINITION
C) RULE OF INFERENCE.

*B$
     TRY AGAIN
*C$
CORRECT...

3
HERE IS THE CA AXIOM:  A+B=B+A.
THE CA AXIOM IS A TRUE EQUATION NO
MATTER WHAT NUMBERS 'A' AND 'B' ARE.
WHICH OF THESE IS AN EXAMPLE OF CA...

A)   5+4=5+4
B)   5+4=4+5
C)   4+5=9.

*B$
CORRECT...
```

```
4
HERE IS AN EXAMPLE OF HOW TO USE THE
CA AXIOM:

DERIVE:  5+4=4+5

CA$    A+B=B+A
A::5$
B::4$     (1)  5 + 4 = 4 + 5

TRY THE PROOF.

DERIVE      (5 + 4)=(4 + 5)

:*CA$    A + B = B + A
A::*5$
B::*4$      (1)  (5 + 4)=(4 + 5)

CORRECT...

5
DERIVE      (13 + 2)=(2 + 13)

:*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


:*DERIVE: 2+3=2+3$
DERIVE      (2 + 3)=(2 + 3)


:*CA$    A + B = B + A
A::*2$
B::*3$      (1)  (2 + 3)=(3 + 2)

:*1CE1$     (2)  (3 + 2)=(2 + 3)

:*1.2RE1$   (3)  (2 + 3)=(2 + 3)

CORRECT...

TYPE A DERIVE, PROVE OR RULE COMMAND


:*FIN$

NOW REDO THE PROBLEM YOU INTERRUPTED

5
DERIVE      (13 + 2)=(2 + 13)

:*CA$    A + B = B + A
A::*13$
B::*2$      (1)  (13 + 2)=(2 + 13)

CORRECT...
```

```
6
TRY USING THE SHORTFORM OF THE CA AXIOM.

DERIVE      (A + B)=(3 + 6)
P           (1)  (A + B)=(6 + 3)

:*CA$   A + B = B + A
A::*6$
B::*3$      (2)  (6 + 3)=(3 + 6)

:*1.2RE1$  (3)  (A + B)=(3 + 6)

CORRECT...
     YOU MAY NOT USE RULE RE IN THIS PROBLEM
     TRY AGAIN
DERIVE      (A + B)=(3 + 6)
P           (1)  (A + B)=(6 + 3)

:*1CA2$     (2)  (A + B)=(3 + 6)

CORRECT...

7
WHAT NUMBER CAN 'A' BE IF 'A < 6' IS
A TRUE STATEMENT?

*4$
CORRECT...

8
DERIVE      A < 6
P           (1)  ((A + 3) = 5)->(A < 6)
P           (2)  B = 3
P           (3)  (A + B)= 5

:*3.2RE1$  (4)  (A + 3)= 5

:*1.4AA$$   (5)  A < 6

CORRECT...

9
'Z' STANDS FOR THE ZERO AXIOM: A+0=A

DERIVE      A = B
P           (1)  (B + 0)=(A + 0)

:*
     NO COMMAND REQUESTED?

:*Z$   A + 0 = A
A::*B$      (2)  (B + 0)= B

:*Z$   A + 0 = A
A::*A$      (3)  (A + 0)= A

:*1.2RE1$  (4)  B =(A + 0)

:*4.3RE1$  (5)  B = A

:*CE1$      (6)  A = B

CORRECT...
```

```
10
YOU NOW KNOW FIVE AXIOMS TO USE IN PROOFS.

PROVE       (0 + A)= A

:*INIT$
     YOU MAY NOT REQUEST YOUR OWN PROBLEMS NOW!

:*Z$   A + 0 = A
A::*A$      (1)  (A + 0)= A

:*1CAA<A>1$     (2)  (0 + A)= A

CORRECT...

11
WE WILL CALL EQUATION 0+A=A, THEOREM 1

SINCE YOU HAVE PROVED THEOREM 1, YOU CAN
USE IT IN OTHER PROOFS.  STUDY THIS...

DERIVE  0+6=6
TH1   0+A=A
A::6   (1)  0+6=6

'TH1' STANDS FOR...

A) THEOREM 1
B) AXIOM 1
C) RULE 1.

*THEOREM 1$
CORRECT...

12
USE THEOREM 1 IN THIS PROOF.

DERIVE      (0 + 7)= 7

:*TH1$   0 + A = A
A::*7   $      (1)  (0 + 7)= 7

CORRECT...
LESSON OVER...
GOODBYE...ADELE
I
*
```

Appendix V

Some Sample Proofs

The first proof given is of a theorem of logic which is motivated by Russells' paradox. Simply let 'F' be interpreted as the membership relation of set theory; then, the sentence to be proved asserts that there is no set which consists of exactly those sets which are not members of themselves. Special attention should be paid to the fact that some of the lines of the proof are justified on the basis of either universal specification or existential specification. Both of these rules involve proper substitution of a term A for a variable X.

The second theorem of first-order logic for which a proof is provided is a variant of Russell's paradox (again, for 'F' read 'ϵ'). The antecedent of the theorem is an instance of the well-known Aussonderung axiom due to Zermelo (thus, sometimes referred to as 'Zermelo's Axiom'); it asserts the existence of those sets which are a subset of some given set as defined by some well-formed formula of the language. It is expected that from this axiom one can prove the non-existence of the troublesome universal class and this is what the consequent of the theorem asserts. Note that instances of previously proved theorems are needed for the proof. Of special significance is the fact that in obtaining these instances substitution for predicates is needed; for example, the formula 'F(v,v)' is substituted for the 0-place predicate 'Q'.

The third example is an alternative derivation of the fact that the universal class is not a set. Premise (1) asserts that every set is such that the cardinal number of the class of all subsets of it is greater than the cardinal number of the set itself; this is, of course, Cantor's theorem. Premise (2) asserts that the class of all subsets of a given set is itself a set; this is the Power Set axiom of axiomatic set theory. Premise (3) asserts that for sets x,y such that y contains x, it is not the case that the cardinal number of x is greater than the cardinal number of y. The final premise asserts that the universal class contains every set.

```
DERIVE     NOT(E Y(A X(F(X,Y) IFF(NOT F(X,X)))))

:*WPS       (1)        *E Y(A X(F(X,Y) IFF (NOT(F(X,X))))))$

:*11<1>ES$
Y::*Z$      (2)        A X(F(X,Z) IFF(NOT F(X,X)))

:*2US$
X::*Z$      (3)        F(Z,Z)IFF(NOT F(Z,Z))

:*3LB$      (4)        (F(Z,Z) ->(NOT F(Z,Z)))&((NOT F(Z,Z)) -> F(Z,Z))

:*4LC$      (5)        F(Z,Z)->(NOT F(Z,Z))

:*4RC$      (6)        (NOT F(Z,Z))-> F(Z,Z)

:*WP$       (7)             *NOT F(Z,Z)$

:*6.7AA$    (8)             F(Z,Z)

:*7.7.8IP$ (9)         F(Z,Z)

:*WP$       (10)             *F(Z,Z)$

:*5.10AA$  (11)              NOT F(Z,Z)

:*10.10.11IP$
            (12)        NOT F(Z,Z)

:*1.9.12IP$(13)  NOT(E Y(A X(F(X,Y) IFF(NOT F(X,X)))))

CORRECT...

DERIVE      (A Z(E Y(A X(F(X,Y) IFF(F(X,Z) &(NOT F(X,X)))))))->(NOT(E
                 Z(A X F(X,Z))))

:*WP$       (1)        *A Z(E Y    (A X(F(X,Y) IFF (F(X,Z)&(NOT F(Y,X)))))
)$

:*WP$       (2)             *E Z(A X F(X,Z)))$

:*2ES$
Z::*W$      (3)             A X F(X,W)

:*1US$
Z::*W$      (4)             E Y(A X(F(X,Y) IFF(F(X,W) &(NOT F(X,X)))))

:*4ES$
Y::*V$      (5)             A X(F(X,V) IFF(F(X,W) &(NOT F(X,X))))

:*5US$
X::*V$      (6)             F(V,V)IFF(F(V,W) &(NOT F(V,V)))

:*PS:THA$      (Q  IFF(R  &(NOT Q)))->(R  IFF(Q  &(NOT C)))
::*Q:F(V,V)$
::*R:F(X,W)$
::*X:V$
::*$
           (7)              (F(V,V) IFF(F(V,W) &(NOT F(V,V))))->(F(V,W)
                              IFF(F(V,V) &(NOT F(V,V))))

:*7.6AA$    (8)             F(V,W)IFF(F(V,V) &(NOT F(V,V)))

:*PS:THB$       NOT(Q  &(NOT Q))
::*Q:F(V,V)$
::*$
```

93

```
          (9)             NOT(F(V,V) &(NOT F(V,V)))

:*8LB$      (10)            (F(V,W) ->(F(V,V) &(NOT F(V,V))))&((F(V,V)
                            &(NOT F(V,V))) -> F(V,W))

:*10LC$     (11)            F(V,W)->(F(V,V) &(NOT F(V,V)))

:*11.9DC$   (12)            NOT F(V,W)

:*3US$
X::*V$      (13)            F(V,W)

:*2.12.13IP$
            (14)          NOT(E Z(A X F(X,Z)))

:*1.14CP$   (15)  (A Z(E Y(A X(F(X,Y) IFF(F(X,Z) &(NOT F(X,X))))))->(
                 NOT(E Z(A X F(X,Z))))

CORRECT...

DERIVE     NOT S U

:*P$        (1)  *A X(S(X)-> G(A(B(X)),A(X)))$

:*P$        (2)  *A X(S(X)->S(B(X)))$

:*P$        (3)  *A X(A Y((S(X) & S(Y))->(C(Y,X)->
*                     (NOT G(A(X),A(Y))))))$

:*P$        (4)  *A X(S(X)->C(U,X))$

:*WP$       (5)        *S(U)$

:*2US$
X::*U$      (6)        S U-> S B U

:*6.5AA$    (7)        S B U

:*3US$
X::*B(U)$   (8)        A Y((S B U & S Y) ->(C(Y,B U) ->(NOT G(A B U,A Y
                       ))))

:*8US$
Y::*U$      (9)        (S B U & S U)->(C(U,B U) ->(NOT G(A B U,A U)))

:*7.5FC$    (10)        S B U& S U

:*9.10AA$   (11)        C(U,B U)->(NOT G(A B U,A U))

:*4US$
X::*B(U)$   (12)        S B U-> C(U,B U)

:*12.7AA$   (13)        C(U,B U)

:*11.13AA$ (14)         NOT G(A B U,A U)

:*1US$
X::*U$      (15)        S U-> G(A B U,A U)

:*15.5AA$   (16)        G(A B U,A U)

:*5.14.16IP$
            (17)  NOT S U

CORRECT...
```

(Continued from inside front cover

96 R. C. Atkinson, J. W. Breisford, and R. M. Shiffrin. Multi-process models
April 13, 1966. (J. math. Psychol., 1967, 4, 277-300).
97 P. Suppes and E. Crothers. Some remarks on stimulus-response theories of la
98 R. Bjork. All-or-none subprocesses in the learning of complex sequences. (J
99 E. Gammon. The statistical determination of linguistic units. July 1, 1966.
100 P. Suppes, L. Hyman, and M. Jerman. Linear structural models for response
Minnesota Symposia on Child Psychology. Minneapolis, Minn.: 1967. Pp
101 J. L. Young. Effects of intervals between reinforcements and test trials in pa
102 H. A. Wilson. An investigation of linguistic unit size in memory processes.
103 J. T. Townsend. Choice behavior in a cued-recognition task. August 8, 196
104 W. H. Batchelder. A mathematical analysis of multi-level verbal learning. A
105 H. A. Taylor. The observing response in a cued psychophysical task. Augus
106 R. A. Bjork. Learning and short-term retention of paired associates in relatio
August 11, 1966.
107 R. C. Atkinson and R. M. Shiffrin. Some Two-process models for memory.
108 P. Suppes and C. Ihrke. Accelerated program in elementary-school mathemati
109 P. Suppes and I. Rosenthal-Hill. Concept formation by kindergarten children
110 R. C. Atkinson and R. M. Shiffrin. Human memory: a proposed system and it
111 Theodore S. Rodgers. Linguistic considerations in the design of the Stanford
112 Jack M. Knutson. Spelling drills using a computer-assi ed instructional sys
113 R. C. Atkinson. Instruction in initial reading under computer control: the S
114 J. W. Breisford, Jr. and R. C. Atkinson. Recall of paired-associates as a
115 J. H. Stelzer. Some results concerning subjective probability structures wit
116 D. E. Rumelhart. The effects of interpresentation intervals on performance in
117 E. J. Fishman, L. Keller, and R. E. Atkinson. Massed vs. distributed prac
118 G. J. Groen. An investigation of some counting algorithms for simple additio
119 H. A. Wilson and R. C. Atkinson. Computer-based instruction in initial rea
120 F. S. Roberts and P. Suppes. Some problems in the geometry of visual perc
121 D. Jamison. Bayesian decisions under total and partial ignorance. D. Jamis
uncertainty. September 4, 1967.
122 R. C. Atkinson. Computerized instruction and the learning process. Septem
123 W. K. Estes. Outline of a theory of punishment. October 1, 1967.
124 T. S. Rodgers. Measuring vocabulary difficulty: An analysis of item varial
parts. December 18, 1967.
125 W. K. Estes. Reinforcement in human learning. December 20, 1967.
126 G. L. Wolford, D. L. Wessel, W. K. Estes. Further evidence concerning
models. January 31, 1968.
127 R. C. Atkinson and R. M. Shiffrin. Some speculations on storage and retri
128 John Holmgren. Visual detection with imperfect recognition. March 29, 19
129 Lucille B. Mlodnosky. The Frostig and the Bender Gestalt as predictors of
130 P. Suppes. Some theoretical models for mathematics learning. April 15,
1967, 1, 5-22)
131 G. M. Olson. Learning and retention in a continuous recognition task. M
132 Ruth Norene Hartley. An investigation of list types and cues to facilitate
133 P. Suppes. Stimulus-response theory of finite automata. June 19, 1968.
134 N. Moler and P. Suppes. Quantifier-free axioms for constructive plane g
F. Oort (Ed.), Compositio Mathematica. Vol. 20. Groningen, The Ne
135 W. K. Estes and D. P. Horst. Latency as a function of number of resp
136 M. Schlag-Rey and P. Suppes. High-order dimensions in concept identi
137 R. M. Shiffrin. Search and retrieval processes in long-term memory. A
138 R. D. Freund, G. R. Loftus, and R. C. Atkinson. Applications of multi
December 18, 1968.
139 R. C. Atkinson. Information delay in human learning. December 18, 1968.
140 R. C. Atkinson, J. E. Holmgren, and J. F. Juola. Processing time as infl
March 14, 1969.
141 P. Suppes, E. F. Loftus, and M. Jerman. Problem-solving on a compute
142 P. Suppes and Mona Morningstar. Evaluation of three computer-assisted in
143 P. Suppes. On the problems of using mathematics in the development of th
144 Z. Domotor. Probabilistic relational structures and their applications. Ma
145 R. C. Atkinson and T. D. Wickens. Human memory and the concept of reinf
146 R. J. Titiev. Some model-theoretic results in measurement theory. May 2
147 P. Suppes. Measurement: Problems of theory and application. June 12,
148 P. Suppes and C. Ihrke. Accelerated program in elementary-school math
149 D. Rundus and R. C. Atkinson. Rehearsal in free recall: A procedure f
150 P. Suppes and S. Feldman. Young children's comprehension of logical

(Continued on back cover)

...ory?

...l. for memory with applications

of language learning. June 12, 1

J. math. Psychol., 1965, 2, 1

...s and latency performance in

Pp. 160-200).

...n paired-associate learning. Au

...n. August 3, 1966

1965.

August 9, 1966.

...gust 10, 1966.

...ation to a paired-associate learning

[illegible]

ons to a continuous presentation task.
2, 1966.
, 1, 182-195).
in arithmetic. (In J. P. Hill (ed.),
August 1, 1966.

of interpresentation intervals.
[illegible]
ry 30, 1967.
February 27, 1967.
rch 21, 1967.
um in initial reading. June 1, 1967.
1967.
[illegible] procedures. July 21, 1967.
1967.
[illegible] task. August 11, 1967.
[illegible] drills. August 18, 1967.
1967.
the Stanford Project. August 25, 1967.
7, (*Synthese*, 1967, *17*, 172-201)
subjective probabilities under total

English and Japanese-English vocabulary

sumptions of visual detection
rm memory. February 21, 1968.
ril 12, 1968.
earch and *Development in Education*

ry acquisition. May 29, 1968.
s. (In J. C. H. [illegible] and
[illegible] 1968. Pp. [illegible]-152.)
[illegible] learning. July 1, 1968.
(*Psychon. Sci.*, 1968, *11*, 141-142)
y to continuous recognition tasks.

[illegible] in the visual display.
5, 1969.
2, 1969.
2, 1969.
[illegible]

r. August 7, 1969.
August 12, 1969.
15, 1969.

(Continued from inside back cover)

151 Joaquim H. Laubsch. An adaptive teaching system for optimal item allocation. November 14
152 Roberta L. Klatzky and Richard C. Atkinson. Memory scans based on alternative test stimul
153 John E. Holmgren. Response latency as an indicant of information processing in visual searc
154 Patrick Suppes. Probabilistic grammars for natural languages. May 15, 1970.
155 E. Gammon. A syntactical analysis of some first-grade readers. June 22, 1970.
156 Kenneth N. Wexler. An automaton analysis of the learning of a nimiature system of Japanese
157 R. C. Atkinson and J. A. Paulson. An approach to the psychology of instruction. August 14
158 R. C. Atkinson, J. D. Fletcher, H. C. Chetin, and C. M. Stauffer. Instruction in initial readi
August 13, 1970.
159 Dewey J. Rundus. An analysis of rehearsal processes in free recall. August 21, 1970.
160 R. L. Klatzky, J. F. Juola, and R. C. Atkinson. Test stimulus representation and experimen
161 William A. Rottmayer. A formal theory of perception. November 13, 1970.
162 Elizabeth Jane Fishman Loftus. An analysis of the structural variables that determine proble
December 18, 1970.
163 Joseph A. Van Campen. Towards the automatic generation of programmed foreign-language in
164 Jamesine Friend and R. C. Atkinson. Computer-assisted instruction in programming: AID.
165 Lawrence James Hubert. A formal model for the perceptual processing of geometric configura
166 J. F. Juola, I. S. Fischler, C. T. Wood, and R. C. Atkinson. Recognition time for informati
167 R. L. Klatzky and R. C. Atkinson. Specialization of the cerebral hemispheres in scanning fo
168 J. D. Fletcher and R. C. Atkinson. An evaluation of the Stanford CAI program in initial readi
169 James F. Juola and R. C. Atkinson. Memory scanning for words versus categories.
170 Ira S. Fischler and James F. Juola. Effects of repeated tests on recognition time for inform
171 Patrick Suppes. Semantics of context-free fragments of natural languages. March 30, 197
172 Jamesine Friend. Instruct coders' manual. May 1, 1971.
173 R. C. Atkinson and R. M. Shiffrin. The control processes of short-term memory. April 19,
174 Patrick Suppes. Computer-assisted instruction at Stanford. May 19, 1971.
175 D. Jamison, J. D. Fletcher, P. Suppes and R. C. Atkinson. Cost and performance of compute
176 Joseph Offir. Some mathematical models of individual differences in learning and performanc
177 Richard C. Atkinson and James F. Juola. Factors influencing speed and accuracy of word r
178 P. Suppes, A. Goldberg, G. Kanz, B. Searle and C. Stauffer. Teacher's handbook for CAI
179 Adele Goldberg. A generalized instructional system for elementary mathematical logic. Octo

14, 1969.
mulus representations. November 25, 1969.
arch tasks. March 16, 1970.

se. July 24, 1970.
14, 1970.
ading under computer control: the Stanford project

ental context effects in memory scanning
blem-solving difficulty on a computer-based test
[illegible] instructional material. January 23, 1971.
January 25, 1971.
[illegible] February 10, 1971.
[illegible]
[illegible] information in long-term memory.
ading (Grades K through 3). March 12, 1971.
[illegible] information in long-term memory.
971.

[illegible] 1971.
[illegible] assisted instruction for compensatory education
[illegible] June 28, 1971.
d recognition. August 12, 1971.
CAI course. September 1, 1971.
October 11, 1971.