INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR

A COMPUTER MODEL FOR AXIOMATIC SYSTEMS

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF
THE UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN ELECTRICAL ENGINEERING

AUGUST 1977

By

Rosalind L. Ibrahim

Dissertation Committee:

W. W. Peterson, Chairman
N. T. Gaarder
A. Y. Lew
S. Lin
A. G. Mader

ABSTRACT

This dissertation describes the development of a computer system that can be used for informal axiomatic investigation of mathematics. A formal mathematics-like language is defined which enables a mathematician to write informal mathematical statements and proofs. Each step of a proof is checked for logical validity, and a user may develop and retain a system of axioms, theorems, dependencies and proofs written in this language. The intended user initially is a college mathematics student, who would use this system to develop proof-writing skills.

The language is an LALR(1) language parsed by a simulated push-down transducer. Although no rules of inference are ever mentioned explicitly in the input language, the rules of natural deduction are accommodated through the block structure and other features of the language. A proof step is represented as a mathematical statement and a (possibly null) list of reasons which are references to other proof steps or to axioms or previous theorems. Proof steps are translated to predicate calculus and then to clause form as the conjunction of the reasons and the negation of the statement. A resolution-based theorem prover, working in first-order logic without equality, seeks to verify each step.

The system is basically sound and has been used on a limited experimental basis for proofs in group theory, number theory and set theory. The language appears to be versatile and easy to use. The major problems have arisen regarding axiom selection and dealing with

equality. Future plans include building in equality and developing lesson plans.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 OBJECTIVES

This dissertation describes the development of a computer system that can be used for informal axiomatic investigation of mathematics. We consider this a 'model' for axiomatic systems in the sense that a model abstractly represents an entity and may be used to investigate that entity.

In axiomatic investigation, an important question that is asked is, "Is this sequence of statements a proof?". Great disparity arises when defining a proof. Definitions vary from the strict formal proofs displayed in logic books to the intuitive informality often found in a mathematics text. Formal proofs may be quite sound, but are cumbersome and long and are most appropriate for logic courses. Informal demonstrations, on the other hand, may be easier to read and write, but may also be lacking in sufficient logical rigor. They may be accepted by some audience, only to be disproved later.

We are dealing with informal proofs. In informal mathematical practice, the question of "what is a proof" is usually answered by presenting the proof to some audience of mathematicians. If the argument is convincing to them, the proof is accepted. With our model, the computer becomes the "audience" and determines whether the proof is valid.

Our model provides a language for representing mathematical statements and informal proofs, and a procedure for verifying informal

proofs expressed in this language. In other words, we have devised a method whereby proofs may be written quite informally, yet still possess a formal rational essence.

More specifically, the work described in this paper entails the development of the following:

1. An example of a formalized language for writing informal mathematical statements and proofs.
2. A proof-checking system for instructional use in proof-writing or general proof verification, of potential value to research mathematicians.
3. A facility for developing and retaining a system of axioms, theorems, dependencies and proofs written in this language.

The system provides rigorous logical testing of reasoning, yet requires informal proof procedures by the user, a step towards the formalization of mathematics.

The concept is also appropriate for modeling axiomatic systems in areas quite different from mathematics. The language is to a large extent user-defined and may be used for expressing logical statements in various contexts.

We hope that our model will contribute to the growing collection of knowledge regarding nonnumeric computer applications. Our efforts are concentrated along three avenues: enhancing the value of mechanical theorem proving procedures; providing computer applications appropriate for use in the educational process; investigating steps towards formalizing and validating mathematical argument and other complex logical reasoning.

The usefulness of mechanical theorem proving is enhanced in two ways: firstly, by providing easy access via a fairly natural input language; secondly, by using theorem proving procedures not to prove theorems in the usual sense, but to check the steps of an informal proof provided by a human. The user controls the theorem proving process and the computer aids in the proof by verifying individual steps. Our model leaves to the human those things people are best at, i.e., deciding what statements to use and writing proofs. Then we use the computer to do what it can do well: formal (tedious) logical manipulations to determine whether the proof is valid.

Our general educational objective is to teach logical thinking and we hope that our system will provide a tool capable of developing and improving the logical capacity of the user. The system does require a student to use language precisely and to think exactly. We provide a means of meeting the following learning objectives: construction of proofs, understanding the connections between theorems, viewing the collection of statements as a system that serves to study certain entities. Besides extending reasoning capability, the development of a system of statements stresses the importance of interrelationships. This is the basis of the systems approach which permeates many aspects of society today.

Mathematical arguments are formalized within the language we have designed, and subsequently verified. Thus our entire process is a step towards the formalization of mathematics.

1.2 SCOPE OF THE INVESTIGATION

In order to meet our objectives of modeling mathematical activities our investigation includes the following tasks:

1. the development of a general mathematics-like language with the following functional characteristics: appropriate for nonprogrammers, easy to read, easy to use, not specific to a particular mathematics discipline, allowing a variety of proof-writing techniques.
2. the inclusion in this language of the following technical characteristics or capabilities: to define the components of mathematical statements, to specify axioms, to introduce propositions, to identify and express the steps that make up a proof, and to seek verification of that proof.
3. the formal specification of this language in terms of its syntax, semantics, pragmatics; its linguistic classification; its recognizer; its translator; its target language.
4. the implementation of the model including the design and development of the language processor and its integration with an existing resolution theorem proving program.* The heart of the system, and our major concern, is the verification process of analyzing statements and proof steps, translating them to predicate calculus, converting predicate calculus to clause form, and submitting clauses to a theorem prover for verification by resolution techniques.

* This program was developed by Prof. W. W. Peterson, Department of Information and Computer Sciences, University of Hawaii, 1975.

The following general features are included with the language processor:

1. the retention of user defined definitions and statements, as well as system provided dependency information, in a file which we refer to as a user base.
2. the retention of partial or completed proofs in the user base.
3. editing of the user base (addition, alteration, and deletion of items) and various listing capabilities for all or part of the user base.

Our language and the scope of our investigation is within the general framework of the existing theorem proving program; that is we work within the first-order logic* without equality (axioms for equality must be specifically introduced). The theorem prover is an integral part of the model and the logical basis of resolution theorem proving and its input format will be discussed in this paper. Also, the particular search strategies used will be briefly described.

Evaluation of the current system has led to the consideration of possible extensions to the language and the theorem prover. These are discussed in Chapter 4.

* The resolution principle is inadequate to perform higher order inference. Although research efforts are being directed toward the mechanization of higher order logic (e.g., [38], [10]), the results have not been very promising ([12], [6]).

1.3 RELATED WORK

Several programs and systems have been written or described which are related to some aspects of our proof-checking system. These works fall in the areas of

1. Man-Machine Theorem Proving
2. Theorem Proving and Education
3. Proof-checking and Formalization

1.3.1 Man-Machine Theorem Proving

These systems are designed with the point of view that mechanical theorem proving will be more helpful to mathematicians, sooner, if efforts are directed away from unaided search for proofs of theorems towards a situation where human intervention is a key factor in exploring mathematical theorem proving. The degree and the intent of intervention varies from system to system.

Our system is similar in concept since we enable the user to provide possible proof steps leading to the establishment of a theorem. Thus in our system, the human interface is in guiding the theorem proving procedure by supplying proof steps and a number of justifications (reasons) to be used to verify the statement just asserted. However, we are interested in validating human reasoning rather than in generating new theorems.

Efforts made in man-machine theorem proving include the following.

1.3.1.1 "Semi-Automated Mathematics"

J. R. Guard, F. C. Oglesby, J. H. Bennett and L. G. Settle [17]

A series of large complex programs developed over a six year period at Applied Logic Corporation, SAM incorporates human intervention into a mechanical theorem proving procedure as follows: by selecting initial axioms, by halting the proof process and inserting additional axioms, by deleting formulas. SAM has been used with applications in group theory, linear algebra, and lattice theory and, with human intervention, succeeded in solving an open problem in lattice theory. Also, after a year's work at finding an appropriate axiom set, SAM was able to derive "all the essential results" (p. 61) of first-order generalized linear algebra.

The intent of SAM is to generate proofs for theorems or to generate interesting consequences of statements. Our system, however, is concerned with verification of proofs already conceived by our user.

1.3.1.2 "A Man-Machine Theorem-Proving System"

W. W. Bledsoe and Peter Bruell [4]

A system used at the University of Texas for investigating general topology, the system tries to prove a theorem and if it fails, the user can intervene to aid in the search for a proof. User commands include:

PUT - to instantiate or replace an existentially quantified variable appearing in a theorem by an expression in order to aid the theorem prover find that particular

expression (and thus prove the theorem).

LEMMA - to first prove the statement in the command and then add it to the hypothesis of the theorem.

USE - to add a stored theorem to the hypothesis of the current theorem.

ASSUME - to assume a current subgoal has been proved, and then continue searching for a proof.

The LEMMA command is similar to a proof step in our system except no specific reasons are attached to LEMMA statements. The USE statement provides a 'reason' in our sense, but applies to the entire proof-seeking procedure rather than to any particular statement entered by the user. Our user develops (in the proof) the expression which instantiates an existentially quantified variable in a theorem, so the PUT facility is not needed in our system. Our system as well allows the use of statements assumed true. These statements may be in the form of unverified theorems, or previous steps in this proof.

Bledsoe and Bruell's program, as reported, is still in the state of development and has proved no new theorems in topology. The authors say their next step will be to have a practicing topologist try to use the system in order to determine if the system might be practical.

The human is aiding the computer which controls the theorem proving process in the Bledsoe-Bruell system. Our philosophy is quite different however, for our user controls the process and provides proof steps. In our system, the computer aids the human by verifying the human thought process.

1.3.1.3 "An Interactive Theorem-Proving Program"

John Allen and David Luckham [3]

In this system, developed at the Stanford AI project, the user is allowed to direct and monitor the search for a proof at run time by altering the choice of theorem proving strategy to be employed. The user can access clause lists or resolvents of particular clauses and can set certain length (of clause) and depth (of nesting) bounds. The user must be familiar with mechanical theorem proving strategies and clause form of statements to use the interactive facility effectively. This type of interaction is very different from that provided in our system, since our user does not need to know anything about mechanical theorem proving. However, a similarity we have is that Allen and Luckham are experimenting to construct a system that they hope could be useful for checking the steps of informal mathematical proofs.

Allen and Luckham used their system to prove some results concerning questions of dependence in axiomatization of ternary Boolean Algebra. The strategies they implemented included set-of-support and paramodulation (built-in equality). Thus a problem could be formulated in either pure first-order logic or in first-order logic with identity. They found that paramodulation is sometimes but not always easier for their system to deal with. (Proofs in general were shorter, but took longer to generate.) Allen and Luckham also reported that unit preference seemed to lead to too many irrelevant clauses being generated at early levels for the more difficult problems they attempted.

Our theorem proving program also implements several strategies

including set-of-support and unit preference and we are experimenting to determine what methods might be most useful for our particular objectives. (See Chapter 4.)

1.3.2 Theorem Proving and Education

Some systems have been described which use the computer to assist in various applications of theorem proving in education.

1.3.2.1 "PROOF"

H. M. Gelder and J. M. Kraatz [14], [15]

A set of proof-writing lessons has been developed to operate under the PLATO system at the University of Illinois. The lessons allow students to try to derive particular exercises in elementary algebra. Students are given a set of axioms and use substitution rules to write a proof which is checked by the system. The programs apply only to elementary algebra and allow three particular modes of proof: transform equations to identity; derive an exercise from an axiom; assume antecedent of exercise and derive the consequent.

In some lessons, for each step, the student writes an instance of an axiom and PLATO writes a line of the proof which is a consequence of applying that axiom. In later lessons, the student writes the proof, chaining by substitution from statement to statement and PLATO checks whether a line is the consequence of the axiom chosen by the student. No conclusions have been drawn as to the effectiveness of this approach in teaching proof-writing.

Our system is more general than the PROOF system of Gelder and

Kraatz since it is not restricted to any particular mathematical discipline; the user employs a language which is self-defined to a large extent; more proof-writing methods are allowed, and of course it does not need to be run under the PLATO system. However, our educational objectives are similar, i.e., to help a mathematics student acquire proof-writing skills.

1.3.2.2 "Geometric Proofs"

Thomas J. Kelanic [21]

In this system, students enter statements ('givens') which describe a geometric figure and then attempt to prove something about that figure by providing statements and reasons. Geometric definitions are built into the system and the system will generate a reason for a statement the student enters if no reason is given. Also, various alternatives are presented to the student to guide the proof-writing endeavor. These alternatives are in the form of valid conclusions derivable from a step just entered.

This special purpose system is limited to geometric proofs and is based on pattern matching techniques. In these respects it is much less powerful than ours. However, the Kelanic system provides feedback to the student on the validity of the proof he or she is constructing, which is one of our objectives also.

1.3.2.3 "Computerized Help in Finding Logic Proofs"

Arthur E. Falk and Richard Houchard [13]

This system intends to assist students in developing the ability to detect inconsistency in logic statements. Students enter statements

in predicate calculus, make instantiations, and seek contradiction, being encouraged to follow a procedure similar to mechanical theorem proving. A proof to be emulated by the student is presented at the conclusion, for comparison.

The system is oriented only to logical symbol manipulation within the framework of an undergraduate logic course.

1.3.2.4 "Computer Understanding of Mathematical Proofs" Vesko Marinov [26]

After the development of our system, this paper came to our attention. The author briefly describes a computerized proof-checking system and states that it has been used for teaching axiomatic set theory in the philosophy department at Stanford University. However we are not aware of any published literature available on this system.

Two aspects appear to be similar to ours. First, the general objective is to use a computer to verify informal proofs as written by a college student. Second, Marinov's system uses resolution theorem proving for some of its verification procedures.

However significant differences exist concerning the notion of informal proof, the level of understanding of formal logic required by the user, the generality of the system, and the educational objectives.

In our system, a user never names a rule of inference and need not be concerned with the internal verification procedure at all. Rather, a user enters a proof step as a mathematical statement wtih reasons and lets the program use its own rules of inference. However, in Marinov's system, the user is in some steps required to refer explicitly

to a rule of inference, albeit a rule of natural deduction, such as universal generalization, conditional proof, or existential specification. Further, a user may need to isolate a part of a mathematical statement, such as "line 1, conjunct 2." The internal verification procedure also must be differentiated in the input to the system. Some steps are checked by truth table methods and others are checked by resolution. The student must make the choice.

Although both systems are designed for educational objectives, the goals and the intended student user are not the same. We wish to assist a mathematics student in developing and improving his or her general proof-writing skills. Marinov's system is designed to follow a particular text ([43]), and a particular curriculum, with the objective of teaching axiomatic set theory.

1.3.3 Proof-Checking and Formalization

The articles described here relate to our objective of formalizing mathematical argument. The ideas generate a sense of underlying importance for the area we are pursuing.

1.3.3.1 "Computer Programs for Checking Mathematical Proofs" John McCarthy [27]

Some early literature in artificial intelligence is concerned with proof-checking systems and the potential value of processing outlines of proofs. McCarthy suggests that (p. 219)

> Checking mathematical proofs is potentially one of the most interesting and useful applications of automatic computers. ... Proofs to be checked by

> computer may be briefer and easier to write than the informal proofs acceptable to mathematicians. ... (proof-checking) will permit mathematicians to try out ideas for proofs that are still quite vague and may speed up mathematical research.

He proceeded to outline a proof-checking system and suggested methods of defining rules of inference in LISP.

1.3.3.2 "Machine Verification of Mathematical Proof"

Paul Abrahams [1]

Inspired by McCarthy's ideas, Paul Abrahams proceeded to investigate a proof-checking system as his doctoral research project at MIT. Abrahams' system was designed to check textbook proofs. It accepted a proof as a list of LISP S-expressions representing proof lines and reasons. The reasons were essentially inference rules and their parameters (called macro-steps). From the input proof a formal proof was generated which was checked for validity using a logical system equivalent to the rules of inference of the natural deduction system of Suppes [42] but extended to allow derivation based on calculations. The system was applied with limited success to some proofs in propositional calculus from Russell and Whitehead's Principia Mathematica [40]. No attempts at more complicated theories were successful.

Abrahams' intent was to investigate the feasibility of using a computer to check mathematical proofs. He found that it was possible, but impractical to do so. Other early (pre-resolution) attempts at theorem proving met similar disappointing results which may be attributed in part to the logical formalism used.

1.3.3.3 "Toward Mechanical Mathematics"

Hao Wang [44]

In another early paper, Wang is concerned with formalization of mathematical argument and he writes that "machines may become of practical use more quickly for mathematical research, not by proving new theorems, but by formalizing and checking outlines of proofs." (p. 17). He proposed a new area of algorithmic endeavor into inferential analysis, or mechanization of mathematical logic. His aim was to use the computer to formalize or fill in the gaps in proofs in mathematical textbooks. Towards this goal Wang developed a set of programs which were used to generate formal mechanical proofs for over 300 theorems from Principia Mathematica [40].

In a subsequent article, "Mechanical Mathematics and Inferential Analysis" [45], Wang further expounds his quest for formalization through analysis of mathematical arguments. He stresses the need for formalization which are exact yet close to good common mathematical exposition. These ideas are also an important consideration in our system for one of our objectives is the development of a mathematics-like programming language for logical analysis.

1.3.3.4 "The Translation of Formal Proofs into English"

Daniel Chester [7]

This last article is of interest to us because it deals with understanding the structure of informal proofs. Chester examines the relationship between formal proofs and written proofs, but he pursues the topic in just the opposite manner. His system translates a formal

proof, within the system of natural deduction, into an English statement of a theorem and its proof. In the translation, he removes some statements and all references to rules of inference, but generates a meaningful narrative which explains how lines of the argument relate to each other. Our input is not as informal as his output for we are dealing with mathematical informality and he is concerned with English. However, his structural analysis of informal proofs in English is similar to our pragmatic analysis of informal proofs in mathematics.

1.4 ORGANIZATION OF THE PAPER

The material to be presented is organized as follows. Chapter 2 is devoted to the specification of the model. The model is defined in terms of its pragmatics, semantics and syntax. This chapter also includes a general description or overview of how the system appears to a user. In Chapter 3, the implementation features of the model are described. This presentation includes theoretical background of the procedures employed. Chapter 4 concludes our study with an evaluation of the system, a summary of our results, and suggestions for future endeavor.

CHAPTER 2

SPECIFICATION OF THE MODEL

2.1 METHOD OF SPECIFICATION

2.1.1 Formal Models

A formal model is usually constructed in order to obtain an abstract representation of an informal intuitive notion. The model may then be studied mathematically and, if the model is appropriate, this will help us understand the informal notion. Developing the formal model entails two things: abstracting the components of the informal system and abstracting the interrelationships among these components. Investigating the model requires using a well-defined procedure to manipulate the abstraction.

In this chapter we describe how the components and the interrelationships among components of axiomatic systems are represented in our model. In Chapter 3 we explain how the model is implemented.

2.1.2 Defining the Model

Our model* consists of a language for representing mathematical statements and informal proofs, and a procedure, invoked from within the language, for evaluating proofs. The procedure works in the

* Another way of viewing the model is as a nonprocedural language in the sense of Sammet [41] p. 20; i.e., processing the language invokes a set of procedures not directly specified in the language. The processing of the language includes the execution of a procedure. The procedure is therefore part of the language definition.

domain of the predicate calculus. The language accommodates informal axiomatics. Our purpose in this chapter is to specify the model by addressing three questions:

1. What is the form of the language?
2. How does the language relate to the procedure?
3. How does the language relate to informal axiomatics?

The first question is answered by giving the formal syntax of the language. The second question regards the semantics of the language and is resolved by explaining how the language is made equivalent to the predicate calculus. The third question is dealt with by considering the pragmatic definition of the language.

Thus, in order to formalize our model we shall investigate its syntax, semantics and pragmatics. Syntax is a rigorous statement of what strings of characters are correct or legitimate for use within the system. Semantics has to do with the meaning or interpretation of the strings within the system. Pragmatics is the relation of these strings and their meaning to the user with the idea that the users' intended use of a string will agree with its semantic interpretation within the system ([41] p. 52).

In order for our model to be useful, it must relate in a meaningful way to the informal activities of a mathematician. Pragmatics is what the language achieves, or according to Zamenek ([52] p. 489), "Pragmatics is the begin and the end of language investigation, and the syntax and semantics are only middle sections of it." Since we desire a useful model, we consider the practitioner and his or her use of statements and proofs. Furthermore, a user

must understand the semantics in order to use the model appropriately, and the syntax in order to use it correctly. Also, the system must recognize syntactically correct inputs and then translate and process them according to the semantic definition. Accordingly, a complete specification of our model addresses the three areas of syntax, semantics, and pragmatics.

2.2 OVERVIEW OF OUR MODEL

In this section we present an overview of selected features of the language. This section should acquaint the reader with the general capabilities of the system. Some sample runs are presented followed by a brief discussion of the pragmatics, semantics and syntax of our language. Then subsequent sections will be devoted to more formal treatment of the syntax, semantics, and pragmatics of the full language.

We say that our system facilitates informal axiomatic investigation. To clarify this statement let us first consider the general topic of axiomatic investigation as carried out by an informal mathematician. Investigation usually proceeds as follows:

1. denote predicates and functions to be used in statements.
2. decide upon axioms and definitions.
3. present a proposition and an informal proof that the proposition follows from the axioms, definitions and theorems.
4. verify the proof by submitting it to an audience of mathematicians. If the argument is convincing, the proof is valid and the proposition becomes a theorem. Otherwise, write another proof and try again or dismiss the proposition.
5. proceed in this manner to develop the theory.

With our system, a user proceeds in very much the same way with two exceptions: our language is used for representing statements and proofs, and our procedure is used to verify proofs, replacing the audience of mathematicians referred to in step 4. Thus with our system, the user gets an objective, impartial evaluation of a proof,

a quick response, and the opportunity to try and try again without boring or tiring the audience. In addition, our system conveniently retains statements, proofs, and dependency information and facilitates editing and listing of the information retained.

More specifically, a user of our system proceeds in this way:

1. define predicates and functions using PREDICATE and FUNCTION statements of our language.
2. identify axioms and definitions using AXIOM statements and write the axioms or definitions using the proof-writing language of our model.
3. identify propositions using the THEOREM statement and write the propositions in the proof-writing language.
4. identify a proof using the PROOF statement and write the proof in the proof-writing language.
5. seek verification of the proof by issuing the VERIFY command. If the procedure determines that the proof is valid, the proposition becomes a theorem. Otherwise the proof may be reentered or altered to try again.

Our system retains the following: predicate and function denotations, axioms, definitions, propositions, theorems, proofs, forward and backward dependency information for theorems, forward dependency information for axioms. Convenient editing and listing facilities for these items are provided.

The following sample illustrates how our system operates. Let us consider this run in three parts. The user enters all the information in part one (Figure 1A). This information defines predicates and

functions to be used in subsequent mathematical expressions, presents axioms and a statement of a theorem, and then a proof of the theorem. The last statement entered is a request to verify the proof. Part two (Figure 1B) is generated by the system as the proof is being verified. Part three (Figure 1C) illustrates some of the listing features provided by the system in response to user requests.

Now let us look more closely at part one. In the first three lines, the user has selected the symbol ID to represent a particular element (a constant) of the domain, the symbol . to represent a function requiring two arguments, and the symbol = to denote a binary predicate. Next, several mathematical statements have been entered. Some are to be considered axioms, and one has been identified as a theorem. These mathematical statements are comprised of a meaningful arrangement of the functions and predicates defined by the user, plus the quantifiers and logical operators provided in our language. The statements are checked for syntax errors as they enter the system. A reference name, e.g., A1 or GROUP1 , has been designated by the user for each statement. The system records a status with each statement which indicates whether that statement is an axiom, an unverified theorem (i.e., a proposition), or a verified theorem.

This information may be interpreted as follows: ID is the identity element of a group, . is the group operator, = has its usual meaning. Axiom A1 states that ID is a left identity, A2 states that a left inverse exists, and A3 is the associative law. The other axioms state properties of equality. The unverified theorem

```
FUN ID CONSTANT
FUN • BINARY
PRED = BINARY
AXIOM A1 FOR ALL X ID•X=X
AXIOM A2 FOR ALL X THERE EXISTS Y SUCH THAT Y•X=ID
AXIOM A3 FOR ALL X,Y,Z (X•Y)•Z=X•(Y•Z)
AXIOM EQREF FOR ALL X X=X
AXIOM EQSYM FOR ALL X,Y (X=Y -> Y=X)
AXIOM EQTRANS FOR ALL X,Y,Z (X=Y & Y=Z -> X=Z)
AXIOM EQSUBST2 FOR ALL X,Y,Z ((X=Y -> X•Z=Y•Z) & (X=Y -> Z•X=Z•Y))
THEOREM GROUP1 FOR ALL X THERE EXISTS Y SUCH THAT X•Y=ID
PROOF GROUP1  1• ASSUME B ARBITRARY
2• LET A SATISFY A•B=ID                          BY A2
3• LET C SATISFY C•A=ID                          BY A2
4• THEN B•A=ID•(B•A)                             BY A1,EQSYM
5•        =(C•A)•(B•A)                           BY #3,EQSUBST2,EQSYM
6•        =C•(A•(B•A))                           BY A3
7•        =C•((A•B)•A)                           BY A3,EQSUBST2,EQSYM
8•        =C•(ID•A)                              BY #2,EQSUBST2
9•        =C•A                                   BY A1,EQSUBST2
10•       =ID                                    BY #3
11•    B•A=ID                                    BY #4-#10,EQTRANS
12• CONCLUSION  THEOREM FOLLOWS                  BY #11
13• QED
VERIFY GROUP1
```

FIGURE 1. SAMPLE RUN

A. USER INPUT

```
*** VERIFYING STEP   1.   : ASSUME B ARBITRARY

*** VERIFYING STEP   2.   : LET A SATISFY A.B=ID                    BY A2
 REASONING IN STEP   2.   IS VALID

*** VERIFYING STEP   3.   : LET C SATISFY C.A=ID                    BY A2
 REASONING IN STEP   3.   IS VALID

*** VERIFYING STEP   4.   : THEN B.A=ID.(B.A)                       BY A1.EQSYM
 REASONING IN STEP   4.   IS VALID

*** VERIFYING STEP   5.   : =(C.A).(B.A)                  BY #3.EQSUBST2.EQSYM
 REASONING IN STEP   5.   IS VALID

*** VERIFYING STEP   6.   : =C.(A.(B.A))                  BY A3
 REASONING IN STEP   6.   IS VALID

*** VERIFYING STEP   7.   : =C.((A.B).A)                  BY A3.EQSUBST2.EQSYM
 REASONING IN STEP   7.   IS VALID

*** VERIFYING STEP   8.   : =C.(ID.A)                     BY #2.EQSUBST2
 REASONING IN STEP   8.   IS VALID

*** VERIFYING STEP   9.   : =C.A                          BY A1.EQSUBST2
 REASONING IN STEP   9.   IS VALID

*** VERIFYING STEP  10.   : =ID                           BY #3
 REASONING IN STEP  10.   IS VALID

*** VERIFYING STEP  11.   : B.A=ID                           BY #4-#10.EQTRANS
 REASONING IN STEP  11.   IS VALID

*** VERIFYING STEP  12.   : CONCLUSION  THEOREM FOLLOWS         BY #11
 REASONING IN STEP  12.   IS VALID

*** VERIFYING STEP  13.   : QED
***PROOF STATISTICS FOR THEOREM GROUP1        ***
    0 STEPS HAVE SYNTAX ERRORS
   11 STEPS ARE VALID
    0 STEPS ARE INVALID
    0 STEPS CAN NOT BE VALIDATED
PROOF DEPENDS ON FOLLOWING AXIOMS/THEOREMS
A2            WHICH IS VALID BY ASSUMPTION
A1            WHICH IS VALID BY ASSUMPTION
EQSYM         WHICH IS VALID BY ASSUMPTION
EQSUBST2      WHICH IS VALID BY ASSUMPTION
A3            WHICH IS VALID BY ASSUMPTION
EQTRANS       WHICH IS VALID BY ASSUMPTION

 STATUS OF THEOREM:  VERIFIED
```

```
 LIST DEF
  ********** LIST OF DEFINITIONS **********
SYMBOL              TYPE            # ARGUMENTS
------              ----            -----------

ID                FUNC-CONSTANT         0
=                 PRED-BINARY 1         2
•                 FUNC-BINARY 1         2
LIST STA
********** LIST OF STATEMENTS **********

     NAME           STATUS                    STATEMENT
     ----           ------                    ---------

  A1                AXIOM                     FOR ALL X ID•X=X
  A2                AXIOM                     FOR ALL X THERE EXISTS Y SUCH THAT Y•X=ID
  A3                AXIOM                     FOR ALL X,Y,Z (X•Y)•Z=X•(Y•Z)
  EQREF             AXIOM                     FOR ALL X X=X
  EQSUBST2          AXIOM                     FOR ALL X,Y,Z ((X=Y -> X•Z=Y•Z) & (X=Y -> Z•X=Z•Y))
  EQSYM             AXIOM                     FOR ALL X,Y (X=Y -> Y=X)
  EQTRANS           AXIOM                     FOR ALL X,Y,Z (X=Y & Y=Z -> X=Z)
  GROUP1            THEOREM VERIFIED          FOR ALL X THERE EXISTS Y SUCH THAT X•Y=ID
LIST GROUP1 DEP  IMM
********** LIST OF STATEMENTS **********
     NAME           STATUS                    STATEMENT
     ----           ------                    ---------
  GROUP1            THEOREM VERIFIED          FOR ALL X THERE EXISTS Y SUCH THAT X•Y=ID
----------------------
BACKWARD DEPENDENCIES
LEVEL-- 0
     A2             AXIOM                     FOR ALL X THERE EXISTS Y SUCH THAT Y•X=ID
     A1             AXIOM                     FOR ALL X ID•X=X
     EQSYM          AXIOM                     FOR ALL X,Y (X=Y -> Y=X)
     EQSUBST2       AXIOM                     FOR ALL X,Y,Z ((X=Y -> X•Z=Y•Z) & (X=Y -> Z•X=Z•Y))
     A3             AXIOM                     FOR ALL X,Y,Z (X•Y)•Z=X•(Y•Z)
     EQTRANS        AXIOM                     FOR ALL X,Y,Z (X=Y & Y=Z -> X=Z)
        PROOF

     1.      ASSUME B ARBITRARY                          BY A2
     2.      LET A SATISFY A•B=ID                        BY A2
     3.      LET C SATISFY C•A=ID                        BY A1,EQSYM
     4.      THEN B•A=ID•(B•A)                       BY #3,EQSUBST2,EQSYM
     5.      =(C•A)•(B•A)                            BY A3
     6.      =C•(A•(B•A))                            BY A3,EQSUBST2,EQSYM
     7.      =C•((A•B)•A)                            BY #2,EQSUBST2
     8.      =C•(ID•A)                               BY A1,EQSUBST2
     9.      =C•A                                    BY #3
     10.     =ID                                        BY #4-#10,EQTRANS
     11.     B•A=ID                                        BY #11
     12.     CONCLUSION  THEOREM FOLLOWS
     13.     QED
```

C. LISTING

states that every element has a right inverse.

A proof for theorem GROUP1 has been given next. The thirteen proof steps are checked for syntax errors as they are entered into the system. Then the VERIFY command causes the proof associated with theorem GROUP1 to be verified (as long as no syntax errors were diagnosed in the proof or the statement of the theorem).

Verification proceeds in a step by step manner generating the output in Figure 1B. In proof step 1, the symbol B was selected as an arbitrary element. Assumptions are not verified since a user is free to introduce any premise at will. In step 2 the symbol A has been chosen to represent the particular element which is the left inverse of B . The reason included in the proof step (Axiom A2) asserts that such an element does indeed exist and the system verifies that the instantiation is a valid one. The proof proceeds through step 11 to construct a right inverse (A) for the arbitrary element B . Reasons are given as preceding steps in the proof or as appropriate axioms. The system has found in each case that the reasoning is valid.

Then in step 12 a CONCLUSION statement is processed. At this time, any symbol declared arbitrary in the preceding ASSUME statement will be universally generalized if it appears in a proof step given as a reason for this conclusion. Thus B is universally generalized in step 11 and the system verifies that the theorem does follow. QED concludes the proof, and summary statistics are printed.

Figure 1C illustrates the output generated as a result of selected LIST statements of our system. These three statements request, respectively: all definitions provided by the user (in

alphabetical order); all statements entered by the user (in alphabetical order); and theorem GROUP1 , its immediate dependencies (backward only in this case), and its proof (if one exists).

Another sample is given in Figure 2, illustrating some additional features of the language. Of particular note are proof steps 1, 2, 6, and 7. Steps 1 and 2 introduce assumptions which declare that Q, R , and S are arbitrary (hence generalizable) variables and that in addition Q+R = Q+S and ¬R = S (¬ & and | can be used to symbolize the logical operators negation, conjunction, and disjunction respectively). CONTRADICTION in step 6 is simply treated by the system as the logical constant false. When a conclusion is verified (step 7) based on assumptions of the form described above, the following actions take place. First, any proof step given as a reason in the conclusion is treated as an implication where the assumptions are the hypothesis and the proof step is the conclusion. Second, any variables declared in the assumptions are generalized. (This example will be treated more fully later in this chapter.)

These two samples illustrate how the system appears to a user. Some of the flexibility of our system is exemplified as these proofs are taken from different mathematical disciplines and they represent different methods of proof. Our language provides many other features such as nested assumption-conclusion blocks, a variety of predicate and function forms, and editing of definition, statements, and proofs. These will be discussed as this chapter progresses.

Now let us briefly consider the pragmatics, semantics, and syntax of our language.

```
**********  LIST OF STATEMENTS  **********

    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  EQUALS          THEOREM VERIFIED          FOR ALL X,Y,Z (X+Y=X+Z -> Y=Z)

                PROOF
                -----

     1.      ASSUME Q,R,S SATISFY Q+R=Q+S
     2.      ALSO ¬R=S
     3.      ¬Q+R<Q+S & ¬Q+R>Q+S          BY #1, ORDER1
     4.      R>S | R<S                   BY #2, ORDER1
     5.      Q+R>Q+S | Q+R<Q+S           BY #4, ORDER2, ORDER3
     6.      CONTRADICTION               BY #3, #5
     7.      CONCLUSION THEOREM FOLLOWS BY #6
     8.      QED

*********  LIST OF DEFINITIONS  **********

SYMBOL            TYPE          # ARGUMENTS
------            ----          -----------

=              PRED-BINARY 1        2
+              FUNC-BINARY 1        2
>              PRED-BINARY 1        2
<              PRED-BINARY 1        2

**********  LIST OF STATEMENTS  **********

    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  EQUALS          THEOREM VERIFIED          FOR ALL X,Y,Z (X+Y=X+Z -> Y=Z)
  ORDER1          AXIOM                     FOR ALL X,Y((X=Y|X<Y|X>Y)&(¬X=Y|¬X<Y)&(¬X=Y|¬X>Y)&(¬X<Y|¬X>Y))
  ORDER2          AXIOM                     FOR ALL X,Y,Z (Y<Z -> X+Y<X+Z)
  ORDER3          AXIOM                     FOR ALL X,Y,Z (Y>Z -> X+Y>X+Z)
```

FIGURE 2. INDIRECT PROOF

Pragmatically, the examples have shown how one is able to achieve the essential ingredients of axiomatic investigation by using the appropriate elements of our language. Our language can be used to identify and name axioms, propositions, and theorems. Constants, functions, and predicates can be defined by our user as well. Finally, our user can write an informal proof and seek its verification. Thus we accommodate the basic components of axiomatics. Our subsequent discussion of pragmatics (in the next section) will be devoted for the most part to an examination of the logical and structural aspects of informal proofs.

Semantically, mathematical statements and proofs written in our language must be made equivalent to formulas in the predicate calculus in order to be processed internally. These formulas are translated to a specific internal form (clause form) and submitted to the theorem proving procedure when a proof is to be verified. After all steps have been processed, the proof is considered valid as long as no steps had logic or syntax errors, all reasons were axioms or proven statements, and the proof did in fact prove the theorem. In part 2.4 of this chapter we present the semantic definition of our language and explain how the elements of our language are interpreted internally and made equivalent to the predicate calculus.

The syntax of our language specifies the exact rules for grammatical formulation of acceptable strings of the language. The last part of this chapter will present the formal syntax of our language.

2.3 PRAGMATICS

This section is concerned with the practitioner, and the nature of mathematical statements, logic, and proofs from the pragmatic point of view. The intent is to identify and categorize common mathematical activities and then explain how these are accommodated in the model. The model is intended to be fairly natural for a mathematician. We are striving to exemplify Zemanek's thought that "very generally, formal expression can frequently be achieved by only little correction or manipulation of the informal pattern" ([52] p. 477). Here we try to identify this pattern. General references include [9], [22], [25], and [42].

2.3.1 The Practitioner

The intended user of our system initially is a college mathematics student. More generally, however, the practitioner we are concerned with may be the mathematician developing a theory, the author preparing a mathematics text, the instructor demonstrating the rationality of mathematics, or the student experimenting with proof-writing. Our practitioner is using mathematical statements to represent assumptions and assertions being made, logic to relate statements to each other, and proofs to determine whether statements qualify for admission to the theory being investigated. The remainder of this section will be devoted to examining what these items mean to the 'informal mathematician' and how these relate to the facilities provided for 'the user' of the computer system.

2.3.2 Mathematical Statements

2.3.2.1 Components of Statements

An examination of informal mathematical statements reveals that they are comprised of a meaningful arrangement of: constants, functions, and predicates, which are peculiar to the theory being investigated and are defined by the mathematician; logical operators and quantifiers, which are generally understood and appear in statements pertaining to most mathematical systems; English words and phrases, which of course vary considerably with the mathematician.

Our system provides for these components of statements as follows. First, we enable the user to name and define particular constants, functions, and predicates using the FUNCTION and PREDICATE statements of the language. In this respect our language is to a large extent user-defined. Symbols may also be selected by the user to represent arbitrary or specific elements (see next section). Second, the common use of logical operators and quantifiers is automatically provided as part of the proof-writing language. Lastly, a reasonable subset of English words and phrases is accommodated in the system to afford readability, some words being simply bypassed as noise words (such as "then" or "such that") and others being an integral part of the language (such as "assume," "let ... satisfy," and "conclusion").

2.3.2.2 Duration of Symbols

In mathematical statements, symbols representing elements, functions, or predicates may be classified as to the duration of their usage or the scope of their meaning. Symbols representing the

statements themselves may also be classified this way.

Firstly, some symbols are durable or permanent in the sense that they represent specific entities that have significance throughout the theory. In our system, symbols and definitions of this type are those entered with PREDICATE, FUNCTION, AXIOM, and THEOREM statements. They are retained in the user base and are accessible as objects of discourse in any subsequent statement or proof.

Secondly, some symbols are used temporarily to introduce a name for an entity in order to facilitate repeated reference to it within a proof or part of a proof. For example, an arbitrary element or an element known to exist may be named, and referred to by name in subsequent statements until the end of the proof or until the name is assigned to some other element later in the proof. Similarly, a statement in a proof may be numbered and referred to by step number in subsequent steps of the proof.

The block structure feature of our proof-writing language deals with this problem. The entire proof is considered a block, and within it blocks are initiated by ASSUME statements and terminated by CONCLUSION statements. Temporary symbols have meaning only in the context of the appropriate block of the proof. Thus, for example, a user denotes an arbitrary element (e.g., ASSUME B ARBITRARY in the proof in Figure 1A) or names an element known to exist (e.g., LET A SATISFY A.B=ID BY A2 in the same proof). This nomenclature endures for statements in the proof until the block is terminated (by the CONCLUSION statement in step 12 of that proof). Meanwhile, the symbols are reserved, and in our system they may not be redefined.

Simple elements (constants, but not functions or predicates) are allowed in these statements in our system. In an analogous manner, symbols (step numbers in this case) representing statements of the proof only have meaning and may be referenced only within the appropriate block of the proof. (Block structure is more precisely defined and exemplified in sections 2.4 and 2.5 of this chapter.)

Lastly, symbols may be used to introduce a name for an element to facilitate reference within that statement only. For example, consider the symbols X and Y in the following statement: FOR ALL X THERE EXISTS Y SUCH THAT Y.X = ID . These symbols are quantified variables and they have no meaning outside the statement. In our system, quantification is part of the language capability. Note however that we are working in the first order logic and thus we restrict symbols of this type also to represent simple elements.

2.3.3 Mathematics and Logic

The practitioner we are considering is the mathematician who carries out his activities in an informal way. However, the formal logical aspects of mathematics are extremely relevant to our model, and so we consider the mathematics-logic issue.*

How strictly mathematics and logic are or should be related is a topic of discussion among many mathematicians, logicians, and educators.

* Actually this issue is only a part of the more general philosophic argument regarding formalism. With the increasing use of the computer and its implied formalism, this topic is receiving much consideration as many parts of our society are becoming formalized. See Zamenek [52] pp. 494 ff.

Some feel that strict adherence to formal logical rules reduces mathematics to routine symbol manipulation and that genuine understanding comes from informal argument and not patterns of deduction. The argument continues that mathematics is a creative discipline not bound by formalism and that if a type of reasoning is appropriate in mathematics but not in logic, then the mathematical statement is not invalid, and logic must accommodate mathematics.

Others are convinced that formalization of mathematics in textbooks and in mathematical activities is desirable and necessary. Complete rational justification should ease the mind and strengthen the science. 'Proof' is not persuasive argument but logical demonstration since the results are to be considered conclusive and not probable. Whereas intuition and creative discovery are the beginning, formal validation should be the end. Therefore every formula should be stated explicitly in a well-defined language and justified by reference to a rule of inference.

In practice then, depending on his or her views, the mathematician may never refer to logical rules at all, may use traditional logical rules explicitly, or may develop some intermediary logical scheme of his or her own (e.g., [30]). However most mathematicians (including mathematical logicians) agree that carrying out mathematical activities completely in a formal way is too tedious.

Our system uses the resolution principle as its rule of inference. Therefore an explicit reference to any inference rule need never be made, for the logical rules are predefined in the language usage and resolution strategies. Accordingly our system offers a compromise

regarding the mathematics-logic issue discussed above. Although a computer model implies logical formalism and precision, since this is the inherent nature of computer usage, the system removes formal tedium for the user. Therefore the user need not be concerned with how a statement follows (i.e., by what rule of inference), but with whether or not it does follow (using the rules of logic embedded in the model). Our model aims in the direction of Wang's "logical mathematics," which he feels should be developed because

> what one needs is not just formalization in principle of mathematical textbooks but rather formalization in practice of mathematical activities. Our goal is to enrich logic (or mathematics) so that computers can aid pure mathematicians at least as much as they assist the applied scientists at present.
>
> [47] p. 155

Strict logical manipulation is the tool used in our system, but the user works in a mathematical domain quite free from strict logical formalism. In the translation is the key to removing the objections of the purists yet providing the rational essence demanded by the mathematical logicians. Strictly, proofs are formal. Pragmatically, they may range from inspired guess to informal proof.

2.3.4 Informal Proofs

In formal axiomatic theory, a proof of a theorem is a finite sequence of formulas such that each formula is either an axiom or is inferred from one or more previous formulas by a rule of inference. The last formula must be the theorem being proved. Every formula and

every rule of inference must be stated explicitly.

By contrast, there is no way to define such an imprecise notion as an informal proof. Informal proofs are mere outlines giving only mathematical highlights with no explicit reference to logical detail. We may consider an informal proof as an argument consisting of a sequence of mathematical statements leading naturally from some premises to a conclusion which states that the theorem follows. The statements may be justified by references to axioms or theorems of the mathematical discipline, or they may refer to preceding steps of the proof. For some statements, no justifications are provided. Logical rules are never explicitly mentioned and the only criterion regarding the clarity of the logic is that it should be capable of being followed by someone reasonably conversant in the field. Therefore steps may be large or small depending on the interpretation of "reasonably conversant," and hence on the intended audience.

The next two parts of this section will consider more specific aspects of informal proofs. Part 2.3.4.1 examines the implicit logic of informal proofs, and part 2.3.4.2 considers common types of informal proofs.

2.3.4.1 Informal Inference Rules and Natural Deduction

In this section we will discuss the decision procedures we assume the practitioner knows and the rules of inference for informal mathematicians that we assume underlie an assertion that a statement "follows by" some other statement. Many of these procedures fall within the general framework of what has been called natural deduction

([9], [42]). This discussion will also briefly relate these rules of inference to the procedures of our deductive system in order to support the contention that our system is rigorous. However we will more formally define a valid argument in our system when we discuss the logical basis of mechanical theorem proving in Chapter 3. We also refer the reader to section 2.4 of this chapter (semantics) for an indepth treatment of how these informal rules of logic are treated in our system.

Typically, an informal proof starts with some premise or hypothesis. Subsequent assertions may include justifications which are usually either explicit references to axioms or previous theorems, or explicit or implicit references to preceding steps of the argument. Conclusions are arrived at based on premises introduced, and the other preceding steps.

Now let us consider the premises of the argument. In informal proofs, the premise may be a known fact (axiom or theorem), or an assumption. The assumption may be the hypothesis of the theorem or any other condition which will be useful in developing the argument. The informal mathematician is free to introduce premises at any point in a proof. This capability is denoted the <u>premise</u> rule of natural deduction.

After premises have been introduced, the informal proof proceeds by arguing to some conclusion and then asserting that the premise implies the conclusion. That is, a typical argument runs "Assume P ... therefore Q. Hence we have shown that P implies Q." This procedure may be called <u>conditionalization</u> ([7]) or the conditional

proof rule of natural deduction ([9], [42]). It implies an informal proof structure regarding assumptions which have limited scopes (up to the conditional conclusion).

In our system, these two logical rules, premise and conditionalization, are accommodated in the block structure of our language. For example, in the uniqueness proof in Figure 3, a premise has been introduced in step 4. Then step 5 is shown to be valid and the assertion in step 6 is that the premise implies step 5 (implicitly by conditionalization). Note that the symbols Z1 and Z2 have been generalized in this conclusion as well. This will be discussed subsequently in this section.

Other rules may be denoted to indicate why a step is assumed to follow from the justifications given. A step of an argument may be assumed to follow because of standard inference. Some of the most common implicitly appealed to logical argument forms include the following:

1. modus ponens (P implies Q is true, P is true therefore Q is true).
2. transitivity (P implies Q is true, Q implies S is true, therefore P implies S is true).
3. proof by cases (P or Q is true, P implies S is true, Q implies S is true, therefore S is true).
4. contrapositive (not Q implies not P is true, therefore P implies Q is true).

These logical arguments are representative of those that we may group together because they are of the form "infer Y from X_1, X_2,

..., X_n because $(X_1 \wedge X_2 \wedge \ldots \wedge X_n) \rightarrow Y$ is a tautology." Let us call this general type of inference tautological implication.

Tautological implication is handled directly by the resolution principle implemented in the theorem proving program. Some examples of proof steps using these argument forms or variations of them are Figure 6, step 4 (modus ponens); Figure 5, steps 3, 7, 15, and 16 (proof by cases); Figure 2, steps 1, 2, 6, 7 (contrapositive). (Note that in all these examples, more than one rule of inference is implicitly required for each step.)

Mathematicians also make full (implicit) use of substitution and replacement operations. For example, a step may be justified by referring to an axiom (or previous step) and the step is assumed to follow because it is a substitution instance of that axiom (or previous step). (See also universal instantiation later in this section.) Or, proof steps may be introduced which employ logical equivalency rules such as DeMorgan's Law, Double Negation, conversion of quantifiers, etc. In the first case we substitute a term for a variable; in the second, we replace a statement by an equivalent statement in a different form. Let us call the former logical substitution and the latter tautological equivalence.

Of course an informal proof may contain a statement that is a strict tautology, such as "P or not P." Let us refer to this procedure simply as tautology.

The resolution principle accommodates logical substitution, tautological equivalence, and tautology. Figure 4, step 2 is an example of tautological equivalence and Figure 3, step 2 is an example

of logical substitution (for universally quantified variables).

The remaining rules are the rules of natural deduction which pertain to procedures for removing and inserting quantifiers. These four rules may be briefly described as follows.*

1. Universal instantiation. If a statement is true for all elements, it is true for a particular chosen (and named) element.
2. Existential instantiation. If there exists an element for which a statement is true, we may choose (and name) a particular element for which the statement is true. (Suppes [42] calls this an ambiguous name, p. 81).
3. Universal generalization. If a statement is true for an arbitrary element, then it is true for all elements.
4. Existential generalization. If a statement is true for some element, then there exists an element for which the statement is true.

Universal instantiation is logically justified by the resolution principle and the naming of the chosen element is made possible via the ASSUME statement of our language. See, for instance, Figure 6, steps 1 and 2 (the particular constant 0 is also logically substituted for the universally quantified variable Y in this case).

Existential instantiation is possible in our language by using the LET statement to denote the particular element. For example, in

* Note that in some of these rules there are restrictions regarding the previous use of the symbol chosen to represent the element, but this will not be discussed here. (See e.g., [25] ch. 1.) Our computer model however does check for proper symbol selection.

Figure 1A, steps 2 and 3, the symbols A and C are the ambiguous names.

Universal generalization is applied in our system when an arbitrary element has been introduced in an ASSUME statement and the CONCLUSION which terminates that block is being verified. For example, in Figure 3, step 6, the arbitrary elements Z1 and Z2 are generalized when the reason (step 5) is processed, and in the same proof, step 7, X and Y are generalized when the reasons (steps 3 and 6) are processed. (This subject is discussed further in the succeeding sections.)

Existential generalization requires no special language structures but is handled directly by the resolution principle. Examples of the implicit application of this rule are Figure 3, step 3 and Figure 4, step 5.

The following chart further exemplifies the logical interpretation of reasons for a variety of proof steps.

STATEMENT	REASONS	LOGICAL INTERPRETATION
P(X) OR ¬P(X)	none	statement should be tautologically true
P((F(A)))	FOR ALL X P(X)	logical substitution universal instantiation
Q(A)	P(A) P(A) → Q(A)	tautological implication
THERE EXISTS X SUCH THAT P(X)	P(A)	existential generalization
¬P(A) AND ¬Q(A)	NOT (P(A) OR Q(A))	tautological equivalence

In conclusion, we have shown that our system can accommodate the implicit logic that we assume an informal mathematician uses, and we have, for each rule, indicated whether it is a feature of the language or is directly processed by the resolution based theorem proving procedure. In the section on semantics we show more precisely how the language is processed in order to accommodate those inference rules not directly handled by resolution.

2.3.4.2 Common Types of Informal Proofs

Several classifications of informal proofs may be identified. There are direct or indirect proofs; proofs of statements of existence, or proofs of statements of generality; deductive or constructive proofs; proof by induction. The choice depends on the theorem and the mathematician. Since our computer system is intended for initial use by students to help them develop proof-writing skills, it is important that a variety of options be available. In meeting this requirement, several interesting issues emerged such as "how can we use an indirect validation procedure to validate an indirect proof" and "can an inductive proof be formulated and validated using a first-order deductive method." Proofs by contradiction work out particularly elegantly as described in section 2.4 on semantics.

The induction principle may be represented by the following axiom in second-order logic: "For all statements S, if S is true for 0 and, for all x, S is true for x implies S is true for x', then S is true for all x." Although our system only accommodates formulations in first-order logic, an inductive proof may be run using

our system by introducing the first-order axiom identifying the particular statement being investigated. (That is, delete "For all statements S " in the axiom above and replace S by the actual statement.) Alternately, this first-order axiom may be considered in two parts as is done in Figure 4. Here the general axiom IND_SET may be interpreted as the inductive axiom for an undefined statement S . Then the particular formula is introduced in axiom IND_FORM.

Examples of proof by induction and other varieties of proof styles have been given in the samples presented in this chapter. Figure 1 is a constructive proof; Figure 2, indirect; Figure 3, uniqueness; Figure 4, inductive; Figure 5, proof by cases. Also several different first-order theories were used as the subject matter for these proofs (elementary group theory, elementary number theory, set theory, linearly ordered sets).

We should mention here that our computer system is an application in nonnumerical processing. As such it does not perform any arithmetic during the evaluation of a proof. Therefore, any numerical calculations required in order for a proof step to follow must be indicated in an appropriate axiom.

```
**********  LIST OF STATEMENTS  **********

    NAME          STATUS                  STATEMENT
    ----          ------                  ---------

  EQREF           AXIOM                   FOR ALL X X=X
  EQSYM           AXIOM                   FOR ALL X,Y (X=Y <-> Y=X)
  EQTRANS         AXIOM                   FOR ALL X,Y,Z (X=Y & Y=Z -> X=Z)
  EQUALITY        AXIOM                   FOR ALL X,Y,Z ((X=X) & (X=Y <->Y=X) & (X=Y & Y=Z -> X=Z) )
  SUM_UNIQUE      THEOREM VERIFIED        FOR ALL X,Y THERE EXISTS Z SUCH THAT FOR ALL U,V
                                          (X+Y=Z & (X+Y=U & X+Y=V -> U=V)  )

*********  LIST OF DEFINITIONS  **********

SYMBOL            TYPE          # ARGUMENTS
------            ----          -----------

=             PRED-BINARY 1         2
+             FUNC-BINARY 1         2


**********  LIST OF STATEMENTS  **********

    NAME          STATUS                  STATEMENT
    ----          ------                  ---------

  SUM_UNIQUE      THEOREM VERIFIED        FOR ALL X,Y THERE EXISTS Z SUCH THAT FOR ALL U,V
                                          (X+Y=Z & (X+Y=U & X+Y=V -> U=V)  )

              PROOF
              -----

    1.    ASSUME X,Y ARBITRARY
    2.    X+Y = X+Y  BY EQREF
    3.    THERE EXISTS Z SUCH THAT X+Y=Z  BY #2
    4.    ASSUME Z1,Z2 SATISFY X+Y=Z1 AND X+Y=Z2
    5.    Z1 = Z2  BY #4,EQUALITY
    6.    CONCLUSION FOR ALL U,V(X+Y=U AND X+Y=V -> U=V) BY #5
    7.    CONCLUSION THEOREM FOLLOWS BY #6,#3
    8.    QED
```

FIGURE 3. UNIQUENESS PROOF WITH NESTED BLOCK STRUCTURE

```
**********  LIST OF STATEMENTS  **********

    NAME            STATUS                  STATEMENT
    ----            ------                  ---------

  EQREF             AXIOM                   FOR ALL X X = X
  IND_FORM          AXIOM                   FOR ALL X (S TRUE_FOR X <-> X=0 OR THERE EXISTS Y SUCH THAT X=Y')
  IND_SET           AXIOM                   FOR ALL X (S TRUE_FOR 0 AND (S TRUE_FOR X -> S TRUE_FOR X' ))
                                            IMPLIES FOR ALL Y S TRUE_FOR Y
  NUM1              THEOREM VERIFIED        FOR ALL X (X=0 OR THERE EXISTS Y SUCH THAT X = Y')

*********  LIST OF DEFINITIONS  **********

SYMBOL              TYPE          # ARGUMENTS
------              ----          -----------

S               FUNC-CONSTANT         0
TRUE_FOR        PRED-BINARY 1         2
0               FUNC-CONSTANT         0
=               PRED-BINARY 1         2
'               FUNC-POSTFIX          1

**********  LIST OF STATEMENTS  **********

    NAME            STATUS                  STATEMENT
    ----            ------                  ---------

  NUM1              THEOREM VERIFIED        FOR ALL X (X=0 OR THERE EXISTS Y SUCH THAT X = Y')

                PROOF
                -----

    1.      0 = 0 BY EQREF
    2.      0=0 OR THERE EXISTS Y SUCH THAT 0=Y'  BY #1
    3.      ASSUME X ARBITRARY
    4.      X' = X'  BY EQREF
    5.      THERE EXISTS Y  X' = Y'  BY #4
    6.      X = 0 OR EXISTS Y X = Y' -> X' = 0 OR EXISTS Y X' = Y'  BY #5
    7.      CONCLUSION FOR ALL Z (Z=0 OR EXISTS Y Z=Y' -> Z' = 0 OR EXISTS Y Z' = Y')    BY #6
    8.      THEOREM FOLLOWS  BY #2,#7,IND_SET,IND_FORM
    9.      QED
```

FIGURE 4. PROOF BY INDUCTION

```
**********  LIST OF STATEMENTS  **********

    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  THM2            THEOREM VERIFIED          FOR ALL X,Y (X<Y+ 1 -> X < Y OR X=Y)
                  PROOF
                  -----
    1.      ASSUME A,B SATISFY A<B+1
    2.      LET X SATISFY A+X=B+1  BY DEF_LT,#1
    3.      X=1 OR 1 < X     BY THM1
    4.      ASSUME ANY SATISFIES X=1
    5.      THEN A+1 = B+1  BY #2,#4,EQPLUS,EQUALITY
    6.      A=B  BY #5,EQPLUS
    7.      CONCLUSION  X=1 -> A=B     BY #6
    8.      ASSUME ANY SATISFIES 1 <X
    9.      LET Y SATISFY 1+Y=X  BY DEF_LT,#8
   10.      THEN A + (1+Y) = B+1  BY #2,#9,EQUALITY,EQPLUS
   11.      A+(Y+1) = B+1              BY #10,COMMUTE,EQUALITY
   12.      (A+Y)+1 = B+1            BY #11,ASSOC,EQUALITY
   13.      A+Y=B  BY #12,EQPLUS,EQUALITY
   14.      A < B   BY #13,DEF_LT
   15.      CONCLUSION 1 < X -> A < B  BY #14
   16.      A=B OR A<B  BY #3,#7,#15
   17.      CONCLUSION THEOREM FOLLOWS  BY #16
   18.      QED
**********  LIST OF STATEMENTS  **********
    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  ASSOC           AXIOM                     FOR ALL X,Y,Z (X+(Y+Z)) = ((X+Y)+Z)
  COMMUTE         AXIOM                     FOR ALL X,Y,Z (X+(Y+Z)) = (X+(Z+Y))
  DEF_LT          AXIOM                     FOR ALL X,Y (X < Y IFF EXISTS Z SUCH THAT X + Z = Y )
  EQPLUS          AXIOM                     FOR ALL X,Y,Z ((X=Y <-> X+Z=Y+Z) & (X=Y <-> Z+X=Z+Y ))
  EQUALITY        AXIOM                     FOR ALL X,Y,Z ((X=X) & (X=Y -> Y=X) & (X=Y & Y=Z -> X=Z))
  THM1            AXIOM                     FOR ALL X (X= 1 OR 1 < X)
  THM2            THEOREM VERIFIED          FOR ALL X,Y (X<Y+ 1 -> X < Y OR X=Y)
```

FIGURE 5. PROOF BY CASES

```
********** LIST OF STATEMENTS **********

    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  SUCCESSOR       THEOREM VERIFIED          FOR ALL X (X + 0' = X')

                PROOF
                -----

    1.      ASSUME X ARBITRARY
    2.      X + ^ ' = (X+ 0)'   BY N2
    3.      X+0 = X    BY N1
    4.      X + 0' = X'   BY #2,#3,EQSUC,EQTRANS
    5.      CONCLUSION THEOREM FOLLOWS BY #4
    6.      QED

********** LIST OF STATEMENTS **********

    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  EQSUC           AXIOM                     FOR ALL X,Y (X=Y ->X' = Y' )
  EQTRANS         AXIOM                     FOR ALL X,Y,Z (X=Y & Y=Z -> X=Z)
  N1              AXIOM                     FOR ALL X (X+0 = X)
  N2              AXIOM                     FOR ALL X,Y (X+Y' = (X+Y) ' )
  SUCCESSOR       THEOREM VERIFIED          FOR ALL X (X + 0' = X')
```

FIGURE 6. NUMBER THEORY

```
********** LIST OF STATEMENTS **********

    NAME          STATUS                  STATEMENT
    ----          ------                  ---------

  DEMORGAN        THEOREM VERIFIED        FOR ALL X,Y(N X V N Y =N(X A Y))

                PROOF
                -----

    1.      ASSUME S,T,U SATISFY S E N(T A U)
    2.      ¬S E (T A U) BY #1,AX1
    3.      ¬S E T | ¬S E U BY #2,AX3
    4.      S E N T | S E N U  BY #3,AX1
    5.      CONCLUSION FOR ALL X,Y,Z(X E N(Y A Z)-> X E N Y V N Z) BY #4,AX2
    6.      ASSUME S,T,U SATISFY S E (N T V N U)
    7.      S E N T | S E N U  BY #6,AX2
    8.      ¬S E T | ¬ S E U  BY #7,AX1
    9.      ¬ S E (T A U)  BY #8,AX3
   10.      CONCLUSION FOR ALL X,Y,Z(X E (N Y V N Z) -> X E N(Y A Z)) BY #9,AX1
   11.      THEOREM FOLLOWS BY #5,#10,AX4
   12.      QED

********** LIST OF STATEMENTS **********

    NAME          STATUS                  STATEMENT
    ----          ------                  ---------

  AX1             AXIOM                   FOR ALL X,Y(X E N Y <-> ¬(X E Y))
  AX2             AXIOM                   FOR ALL X,Y,Z(X E Y|X E Z<->X E Y V Z)
  AX3             AXIOM                   FOR ALL X,Y,Z(X E Y & X E Z <-> X E Y A Z)
  AX4             AXIOM                   FOR ALL X,Y,Z((X E Y <->X E Z) <-> Y=Z)
  DEMORGAN        THEOREM VERIFIED        FOR ALL X,Y(N X V N Y =N(X A Y))
```

FIGURE 7. SET THEORY

```
**********  LIST OF STATEMENTS  **********

    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  DEMORGAN        THEOREM VERIFIED          FOR ALL X,Y(N X V N Y =N(X A Y))

                PROOF
                -----

    1.      ASSUME S,T,U ARBITRARY
    2.      S E N(T A U)-> ¬(S E (T A U))  BY AX1
    3.      S E N(T A U) -> ¬(S E T)|¬(S E U) BY #2,AX3
    4.      S E N(T A U) -> S E N T|S E N U  BY #3,AX1
    5.      S E N(T A U) -> S E (N T V N U)  BY #4,AX2
    6.      S E (N T V N U) ->S E N T |S E N U  BY AX2
    7.      S E (N T V N U) -> ¬(S E T)|¬(S E U)  BY #6,AX1
    8.      S E (N T V N U)-> ¬(S E (T A U))  BY #7,AX3
    9.      S E (N T V N U)-> S E N(T A U)  BY #8,AX1
   10.      CONCLUSION  THEOREM FOLLOWS  BY #5,#9,AX4
   11.      QED


**********  LIST OF STATEMENTS  **********

    NAME          STATUS                    STATEMENT
    ----          ------                    ---------

  AX1             AXIOM                     FOR ALL X,Y(X E N Y <-> ¬(X E Y))
  AX2             AXIOM                     FOR ALL X,Y,Z(X E Y|X E Z<->X E Y V Z)
  AX3             AXIOM                     FOR ALL X,Y,Z(X E Y & X E Z <-> X E Y A Z)
  AX4             AXIOM                     FOR ALL X,Y,Z((X E Y <->X E Z) <-> Y=Z)
  DEMORGAN        THEOREM VERIFIED          FOR ALL X,Y(N X V N Y =N(X A Y))
```

FIGURE 8. SET THEORY--VARIATION OF PROOF IN FIGURE 7

2.4 SEMANTICS

Wegner defines semantics as

> ... the study of the relation between objects and their representations. Usually we are given a set of representations of objects, specified in a 'language' which allows relations between objects to be expressed and wish to say something about the 'meaning' or 'denotation' of the representation.
>
> [48] p. 151

Our language is used for expressing informal mathematical statements and proofs. Since the model we have developed is based on the predicate calculus, our semantic definition is the relation between the predicate calculus and the mathematical statements and proofs expressed in our language.

We will first discuss mathematical semantics, or the meaning of mathematical statements. Next we discuss the semantics of proofs, which deals with: the interpretation of special statements that may appear in proofs; proof steps (statements and reasons); and the block structure of proofs in our language.

2.4.1 Mathematical Semantics

We have chosen the general model of the first order predicate calculus to serve as the vehicle for denoting mathematical statements. Therefore the object we wish to represent is a well-formed formula of the predicate calculus.

2.4.1.1 Well-Formed Formulas

The predicate calculus itself is a formal language and the symbols

used have meaning restricted to a specific universe of discourse. The construction of well-formed formulas in the predicate calculus involves symbols classified and interpreted in the following way.

- logical operators--quantifiers and propositional connectives, which have a fixed interpretation defined by logical axioms.
- n-place functions with $n \geq 0$

  When $n = 0$ these functions denote specific elements (or constants) of a domain D in an interpreted system, or in an uninterpreted system these functions denote unspecified elements of an unspecified domain D. In general, these functions are interpreted as mapping D^n into D.
- n-place predicates with $n \geq 0$

  With $n = 0$ we have logical constants denoting truth values. When $n > 0$, predicates are interpreted as mapping D^n into true or false. In an interpreted system a truth value is assigned to each predicate.
- variables--interpreted as mathematical variables or arbitrary values over the domain of elements in D, or over some preassigned set of elements in D.
- punctuation marks ( ) ,

Now let us review a typical definition of well-formed formulas in the predicate calculus ([6])

- Terms

  1. a constant is a term

2. a variable is a term

3. if f is an n-place function symbol and t_i are terms then $f(t_1,\ldots,t_n)$ is a term.

- Atoms

  If P is an n-place predicate symbol and t_i are terms then $P(t_1,\ldots,t_n)$ is an atom.

- Well-formed formulas

  1. an atom is a well-formed formula

  2. if F, G are well-formed formulas then NOT F, F OR G, F AND G, F IMPLIES G, and F IFF G are well-formed formulas.

  3. if F is a well-formed formula and x is a free variable in F , then FOR ALL x F and THERE EXISTS x F are well-formed formulas.

Parentheses* may be introduced to cause evaluation of logical operators to be performed in the manner indicated. Otherwise the operators will be applied according to the following hierarchy*: quantification, negation, disjunction, conjunction, implication, equivalence.

2.4.1.2 Mathematical Statements

We have defined a well-formed formula as the object we are trying to represent by a mathematical statement expressed in our higher level

* Note that precedence rules are relevant for processing mathematical statements in our language but that the internal form of the predicate calculus used for representing mathematical statements is an equivalent parenthesis-free Polish notation.

language. Our semantic definition rests now in specifying how we may represent the symbols and well-formed formulas just described.

- The logical operators are: the quantifiers FOR ALL and THERE EXISTS and THERE EXIST ; and the propositional connectives are: NOT or ¬ for negation, OR or | for disjunction, AND or & for conjunction, IMPLIES or → for implication, and IFF or ↔ for equivalence.
- A constant is a symbol chosen by the user and entered into the user base as a constant. The FUNCTION statement of the operating language* is used to define a constant.
- A function is a symbol chosen by the user and entered into the user base as a function, along with its arguments placed appropriately as defined by the function type. The FUNCTION statement of the operating language is used to denote the symbol and to define the type. The function types are: infix binary, postfix unary, prefix unary, or n-place list, with $n > 0$. The arguments must be constants, variables, or functions.
- A predicate is a symbol chosen by the user and entered into the user base as a predicate, along with its arguments (if any) placed appropriately as defined by the predicate type. The PREDICATE statement of the operating language is used to denote the symbol and to define the type. The predicate types

* The FUNCTION and PREDICATE statements of the operating language are defined syntactically in section 2.5.2.

are the same as the function types described above. In addition zero-place predicates are also allowed. The arguments must be constants, variables or functions.

- A variable is a symbol chosen by the user which is none of the above symbols, nor is it a reserved word of the language.
- A predicate is a mathematical statement.
- IF S_1 and S_2 are mathematical statements then NOT S_1, $\neg S_1$, S_1 AND S_2, S_1 & S_2, S_1 OR S_2, $S_1 | S_2$, S_1 IMPLIES S_2, $S_1 \rightarrow S_2$, S_1 IFF S_2, and $S_1 \leftrightarrow S_2$ are mathematical statements.
- If S is a mathematical statement then FOR ALL varlist S, THERE EXIST varlist S, and THERE EXISTS varlist S are mathematical statements where varlist is a list of variables separated by commas and any variable in varlist should be free in S.
- The hierarchy of evaluation of operators is as described in the next section on syntax.
- Parentheses may be introduced to cause evaluation of logical operators to be performed in the specified manner, and to specify the scope of quantifiers.
- No variables may appear in mathematical statements unless they are quantified. Note that symbols which are quantified variables inherit the scope of the quantifier in the usual logical sense. That is, they only have meaning in that part of the statement quantified.

2.4.1.3 Interpretation

Our mathematical statements are always considered in the context of an interpreted system. This is because all symbols used must be defined either by the user or the language and the definition provides the interpretation.

User defined symbols appearing in mathematical statements are interpreted as a result of interpretation operators PREDICATE and FUNCTION . Statements themselves are interpreted by means of the AXIOM, THEOREM and VERIFY interpretation operators.

The fundamental elements under consideration in a mathematical system comprise what is typically called the 'domain' or 'basic set' of the system. Operators and relations are defined over elements of this set, constants are particular elements of this set, and variables refer to elements of this set. By basing our model on first-order theories, we have restricted our domain to be a domain of individual elements.

In certain first-order axiomatic systems, this domain may consist of elements of more than one type, and functions and predicates may be defined over certain types of elements in the domain. Such a system is called a 'many-sorted' theory and could be modelled by a 'many-sorted' logic ([8] p. 339). The current version of the proof-checking system however will require that many-sorted systems be formulated as one-sorted systems, i.e., with a domain consisting of only one type of element. If such a transformation is required, it could be accomplished as follows ([46] pp. 323-333). Suppose the original system consisted of n categories of fundamental elements. The corresponding one-sorted

system would require the introduction of n one-place predicates such that the predicate would be true iff the variable was of the appropriate type. Any mathematical statement referring to a variable of a specific type would require the corresponding property as a hypothesis. For example, points and lines might be fundamental objects in geometry with variable names starting with say p and ℓ, respectively. A statement might thus read

$$\forall\, p\, \forall\, \ell\ (\ \ldots\ p\ \ldots\ \ell\ \ldots\ )$$

If we introduce P and L as predicates indicating the variable is a point or a line, we may quantify over a general domain and translate such a statement to

$$\forall\, x\, \forall\, y\ ((P(x) \wedge L(y)) \rightarrow (\ \ldots\ x\ \ldots\ y\ \ldots\ ))$$

Since the proof-checking system will deal with a one-type domain, it need not be specified nor do variable, function, nor predicate definitions need to refer to any restricted range of elements.

In our system, functions and constants always denote some object in a single unspecified (but understood) domain D of interpretation. The FUNCTION operator names a function symbol and is understood to map that symbol to a function from D^n to D, where n is the implicitly or explicitly stated number of arguments required for use of the symbol.

Predicates denote a truth value. The PREDICATE operator provides an interpretation by mapping the symbol named to a function from D^n to true or false. Note that D does not include the truth values

as elements since predicates are not allowed to be the arguments of predicates or functions.

The logical operators in mathematical statements always denote the same entities (under all interpretations provided for other symbols used). They are defined as in the predicate calculus, i.e., by logical axioms or truth table analysis. The logical operators are mapped to a function from sets of truth values to a truth value. Quantification is interpreted as quantification over the single unspecified domain D. Therefore variables, like constants and functions, denote some element or all elements of D.

Since our mathematical statements are always under an interpretation, they denote a truth value. A truth value is assigned to a mathematical statement as a result of the interpretation operators AXIOM, THEOREM, and VERIFY. Although we are not actually concerned with the 'truth' of a mathematical statement, we prefer to use this terminology because it lends clarity to the discussion. Mathematical theories are more concerned with 'acceptance' of statements as a logical consequence of other accepted statements, rather than any iota of truth. But a statement Q is a logical consequence of another statement P if $P \rightarrow Q$ is a tautology (or if $P \rightarrow Q$ is logically true). In this case we may 'accept' Q, if P was already 'accepted.' To simplify terminology, any statement we accept we will call a 'true' statement. Further, we assume the interpretation we are under at any time is a <u>model</u> for the theory, meaning an interpretation for which the axioms are true statements. Accordingly we define the interpretation of mathematical statements as follows.

An axiom is a mathematical statement interpreted as true. This means the interpretation operator AXIOM maps the named statement to true.

Now let us define a proposition, similarly to Robinson [37] as that which asserts that a mathematical statement Q follows from a set P of mathematical statements. In our system, a proposition is a statement which is an unverified theorem. That is, the interpretation operator THEOREM maps the named statement to a proposition where P is the set of true statements in the system and Q is the named statement. A proposition is not mapped to a truth value. However, the interpretation operator VERIFY may further interpret a proposition, for if the THEOREM is verified then Q is mapped to true and the subset of P actually required for verification is retained in dependency lists.

2.4.2 The Semantics of Proofs

In this section we will investigate the relation of informal proofs and the predicate calculus. The discussion will include an overview of mechanical theorem proving and its required input form, since this is the object we need to be able to represent in our informal proof-writing language. Next, the basic types of statements used in the proof-writing language will be described in terms of semantic definition. These discussions will show how the proof-writing language incorporates the rules of natural deduction while translating proof steps into the form of the predicate calculus required for resolution.

2.4.2.1 Mechanical Proof Procedures and the Predicate Calculus

A mechanical proof procedure is a general method to verify the validity of a statement. A statement is valid if every interpretation of the statement is true. A proof procedure can determine that a statement T is a logical consequence of statements $A_1 \ldots A_n$ if it can be found that the statement $(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \rightarrow T$ is valid. Equivalently, a statement is valid if and only if its negation is inconsistent (false under all interpretations). Therefore T logically follows from $A_1, A_2, \ldots A_n$ if the statement $(A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge \neg T)$ is inconsistent.

The resolution principle, introduced by J. A. Robinson in 1965 [36], is an inference rule that is particularly suited to computer implementation and it is used to demonstrate that a statement is inconsistent. Resolution requires that the statement originally be expressible in first-order predicate calculus. Then the statement must be translated into a special form called clause form which is input to the resolution-based theorem proving program. If the statement is indeed inconsistent, this can be determined in a finite number of processing steps.

Let us consider a proof step in the proof-writing language as a pair (S,R) where S is a statement and R is a possibly empty set of reasons, R_i . Both S and R_i represent statements expressible in the first-order predicate calculus. In order to determine whether the statement is a logical consequence of the reasons, another statement of the form $(R_1 \wedge R_2 \wedge \ldots \wedge R_n \wedge \neg S)$ can be generated, translated to clause form and submitted to a resolution-based theorem prover for

verification.

The theorem proving program may derive a contradiction, which would indicate that the statement in the proof step did follow from the reasons given. Alternately, the program may determine that no contradiction can be derived and in this case the proof step included faulty reasoning. A third case may arise where the theorem prover has not found a contradiction within time and search bounds imposed, but the search was not conclusive. In this situation, the proof step may or may not follow from the reasons given.

The first-order predicate calculus is a language suitable for representing most mathematical statements as well as assertions that are not mathematical at all. The flexibility offered in the language of our model leaves the definition of most symbols up to the user and thus a statement processed by this system need not necessarily have a mathematical interpretation. However, regardless of the original intention of statements entered, all must be translated into the first-order predicate calculus for processing by the system. Also, reasons specified explicitly or determined implicitly must be translated to predicate calculus and an appropriate conjunction of the reasons and the negation of the statement must be provided prior to the translation to clause form for the thoerem prover.

Clause form is an equivalent representation of formulas of the predicate calculus. It is described and exemplified in Chapter 3 when we discuss the implementation of the translator. Translation of the user language actually results in an intermediate form of the predicate calculus used prior to and during conversion to clause form. The

intermediate form used is standard postfix Polish, which is both useful and convenient in view of the LR(k) parsing strategy employed. This translation phase will be described and exemplified in Chapter 3 as well. Also the logical deductions performed in resolution theorem proving will be discussed in that chapter.

The relation of the proof-writing language of our model to the predicate calculus will be described next.

2.4.2.2 Statements in the Proof-Writing Language

In this section we show how the steps of a proof are made equivalent to formulas of the predicate calculus. We discuss the block structure of our language and the internal treatment of arbitrary and particular symbols which enables the rules of natural deduction to be processed by our system.

Statements in the proof-writing language may be grouped as follows.

- Mathematical statements--statements that represent well-formed formulas of the predicate calculus. Mathematical statements are used for statements of axioms and theorems in the user base, as described in the preceding section on mathematical semantics. They are also used in proof steps.
- Assumptions--statements that denote special symbol usage, and conditions assumed to be true during the course of a proof. Assumptions are used to declare symbols and conditions and to initiate a block within which the assumptions will be valid.
- Conclusions--statements that represent logical conclusions derived on the basis of previous assumptions made. Conclusion

statements terminate the last unterminated assumption block initiated.

- Instantiations--statements that facilitate existential instantiation or the naming of particular elements that exist.
- Other--Three system defined statements are available for use in proof steps. These are CONTRADICTION, THEOREM FOLLOWS , and QED .

All statements in the language are labelled. Statements in the user base (theorems or axioms) have a name. Statements appearing in a proof have a proof step number.

The basic statement types will now be described with regard to semantic definition.

2.4.2.2.1 Mathematical statements in proofs

The mathematical semantics described in the preceding section pertains to the use of named statements, that is, statements which represent axioms or theorems. Axioms or theorems used as reasons in proof steps, or the statement of the theorem being proved used as a statement in a proof step (theorem follows) must conform to the semantic description provided above.

However, with mathematical statements in proof steps (i.e., numbered statements) the following additional semantic information applies:

- a predicate may be a binary (infix) predicate with a vacuous left side in a proof step following a step which consisted of

a binary (infix) predicate (with or without a left side). In this case, the right side of the previous predicate is interpreted as the missing left side.

- a constant may be a symbol which has appeared in a list of symbols following ASSUME in a proof step, and whose scope includes the step in which the symbol appears. We also call this type of constant an arbitrary constant.
- a constant may be a symbol which has appeared in a list of symbols following LET in a proof step, and whose scope includes the step in which the symbol appears. We also refer to this type of constant as a particular constant.
- a variable may be a symbol known as an arbitrary constant but considered as a variable during conclusion processing. We also call this type of variable an implicitly quantified variable.

The above semantic definitions refer to "scope" and "conclusion processing." These terms are defined in the following discussion of block structure.

2.4.2.2.2 Block structure

- Scope

Proofs are structured in what we call proof blocks. Proof blocks are initiated by ASSUME statements and terminated by CONCLUSION statements. In addition, the entire proof is considered a block beginning with the first statement and concluding with QED . The

set of steps comprising the proof block includes the delimiting steps as well as any intervening steps (including nested blocks) that are active or unerased as described below.

A symbol introduced in a proof step only has meaning within that step and the succeeding set of steps which are in the block to which the proof step belongs. This set of steps is the scope of the symbol.

Two types of symbols may be introduced in proof steps: symbols representing elements and symbols representing statements. Symbols representing elements are introduced (or declared) in ASSUME or LET statements. These symbols are local to this proof block, and have no meaning outside of it. Neither do they have any meaning in axioms or theorems given as reasons or included implicitly (i.e., THEOREM FOLLOWS) in this proof block. We do not allow symbols to be redeclared within a proof block. However, a symbol may freely be reused in subsequent proof blocks, since the nomenclature expires when the proof block is terminated.

Symbols representing statements are step numbers assigned to proof steps. Any proof steps appearing in a proof block may be referenced as reasons only within that proof block, except conclusion proof steps may be referenced at the next higher block level.

Conclusions denote the end of a proof block. After conclusions are processed, the proof block, excluding the conclusion itself, is in effect erased. Thus, steps in the terminated proof block may not be referenced and symbols declared no longer have any special significance. Although the conclusion itself may be referenced in subsequent steps, any symbols appearing within the statement must be meaningful in the

context of the referencing step. Also, note that conclusions of nested proof blocks are no longer accessible when the nesting proof block is terminated.

Of course, after a QED is processed, any symbol conventions introduced in the entire proof are released.

• Symbol Usage and Accommodation of Informal Logic

The block structure of our language facilitates the special symbol handling that is necessary in order to let our resolution procedure process the logical rules of universal instantiation and universal generalization. Our discussion here deals with how certain symbols are sometimes treated as constants and sometimes as variables in the predicate calculus in order to accomplish this. The block structure also accommodates premise and conditionalization (as described in section 2.3, pragmatics, of this chapter). We describe how this is accomplished now as well.

In addition to delimiting a proof block as described above, an ASSUME statement serves two purposes.

1. It may identify symbols which are to be considered arbitrary elements of the proof domain. The statement

   ASSUME A, B ARBITRARY

   for example, would indicate that the symbols A and B refer to arbitrary elements. Internally, any explicit occurrence of these symbols in statements within the scope of this declaration will result in these symbols being treated as arbitrary constants or as implicitly quantified variables (the latter

during conclusion processing only).

2. Alternately, an ASSUME statement may specify that a list of symbols are to be identified as arbitrary elements but also that one or more conditions are to be temporarily assumed true (within this proof block). For example, statements of the form

```
ASSUME Q,R,S,T SATISFY  Q+R = Q+S
ALSO  ¬R = S
```

indicate that T represents an arbitrary element of the domain and Q, R , and S are elements that may be considered "arbitrary" except that the conditions Q+R = Q+S and ¬R = S are true.

Note that this use of the ASSUME statement implicitly incorporates the premise rule of natural deduction, followed by universal instantiation with a named but arbitrary element. That is, the above ASSUME statement represents the following analogous argument in natural deduction.

```
For all  X, Y, Z  (X+Y = X+Z  and  ¬Y = Z)  (premise)
Let  Q, R, S  be arbitrary elements
Q+R = Q+S  and  ¬R = S  (universal instantiation)
Let  T  be an arbitrary element.
```

ASSUME statements may be processed in three different ways during the verification of a proof.

1. <u>As the proof step currently under consideration</u>

In this case, no translation to Polish predicate calculus is involved since no verification of the step is made.

(Premises may be introduced at will.) Symbols listed enter the symbol table as arbitrary constants, and a new proof block is initiated.

2. As a reason stated explicitly in a proof step

In this case the formula following SATISFY or ALSO will be converted to Polish predicate calculus and included in the string of statements which will be converted to clause form. Explicit references to an ASSUME ... ARBITRARY statement as a reason would not be meaningful in a proof and would be ignored.

3. Implicit referencing of assumptions in conclusion processing

When a conclusion statement is reached, reasons that are proof steps within the proof block being terminated are processed in a special way. First, a reason will be translated into an implication statement where the conjunction of the assumptions is the hypothesis and the step statement is the conclusion. Note that this process implements the condition-alization rule of natural deduction. That is, we may conclude an argument by implicitly inferring that the premises imply the conclusions.

Secondly, any symbols declared in an ASSUME statement will now be treated internally as universally quantified variables. (Arbitrary constants become implicitly quantified variables.) This accomplishes universal generalization of arbitrary symbols. For example, consider the following indirect proof, reproduced from Figure 2, of the theorem

```
FOR ALL X,Y,Z  (X+Y = X+Z  IMPLIES  Y = Z)
1.  ASSUME Q,R,S SATISFY Q+R = Q+S
2.  ALSO  ¬R = S
3.  ¬Q+R < Q+S AND ¬Q+R > Q+S        BY #1, ORDER1
4.  R > S OR R < S                   BY #2, ORDER1
5.  Q+R > Q+S OR Q+R < Q+S           BY #4, ORDER2, ORDER3
6.  CONTRADICTION                    BY #3, #5
7.  CONCLUSION THEOREM FOLLOWS       BY #6
8.  QED
```

The reason #6 used in step 7 would be processed as

(Q+R = Q+S AND ¬R = S) IMPLIES CONTRADICTION

or equivalently

¬(Q+R = Q+S AND ¬R = S) OR CONTRADICTION

but CONTRADICTION means 'false' so the reason for step 7 would be the negation of the assumptions. In addition, Q, R , and S would be considered universally quantified variables in this internal representation of the reason.

Only the symbols declared arbitrary in the nearest preceding unterminated ASSUME statement are generalized at this time. Thus for example in the proof in Figure 3, Z1 and Z2 are generalized when the reason (step 5) for the conclusion statement in step 6 is processed, and X and Y are generalized in the reasons given for the conclusion statement in step 7.

Notice that, based on the above discussion, the semantic

difference between constants and variables appearing in proof blocks actually depends on the context of the symbol.

Assumptions or conclusions may be continued by preceding the continuation statement with ALSO (as for example in step 2 of the preceding proof). This is equivalent to using a logical AND to conjoin the statements of the steps, but each statement receives a step number. The user may wish to distinguish parts of assumptions or conclusions this way to allow for more specific references in succeeding proof steps (as is done in step 4 of the above proof).

2.4.2.2.3 Instantiations

When a LET statement is the step currently being verified it will be interpreted in the following way.

LET sym1, sym2 SATISFY formula BY reason(s)

is equivalent to

There exist sym1, sym2 such that (formula)

and the reason(s) given will be used to verify that such elements do indeed exist. Thus the symbols in the list are treated as quantified variables when the step is being verified. Subsequent reference to this step as a reason will result in translating the formula to Polish predicate calculus and any occurrence of those symbols in the list will result in their use as constants. The symbols will be considered particular constants throughout their scope.

Pragmatically, LET statements effect existential instantiation.

2.4.2.2.4 Other statements

Three statements are provided by the system. These are CONTRADICTION, THEOREM FOLLOWS , and QED.

• CONTRADICTION

CONTRADICTION actually stands for the logical constant false. Therefore when CONTRADICTION is processed as the statement of a proof step, or when it is processed as a reason in a CONCLUSION statement it is simply ignored, as explained below.

Consider again a proof step of the form (S,R) , S being a statement and R being a reason. Processing a proof step results in generation of the statement R AND ¬S. If S is false, ¬S is true, and (R AND true) ↔ R . Therefore if the statement was CONTRADICTION, it is ignored.

Now consider a conclusion proof step (S,R) and let A represent the assumptions. Processing of this step generates a statement of the form (A IMPLIES R) AND ¬S . But (A IMPLIES R) ↔ (¬A OR R) and if R is CONTRADICTION (false) then (¬A OR false) ↔ ¬A . Therefore if the reason was contradiction, it is ignored.

• THEOREM FOLLOWS

This statement merely causes the theorem being proved to be extracted from the user base and inserted as the statement in the proof step. Note that THEOREM FOLLOWS must occur as a step in every proof.

- QED

This statement must be the last statement of a proof. It is required so that the system can check for proper proof structure when the proof has been completed.

2.5 SYNTAX

The complete language of our system includes the following capabilities:

- A method of representing mathematical statements
- A method of representing proofs
- A method of naming components of mathematical statements and interpreting them as elements, functions, or predicates
- A method of naming mathematical statements and interpreting them as axioms, theorems, or propositions
- A method of seeking verification of proofs.

We will consider the complete language as two sublanguages. The methods of representing mathematical statements and proofs are provided in the "proof-writing language," and the interpretation and execution features are available through the "operating language." This section will formally define the syntax of both languages.

2.5.1 The Proof-Writing Language

The syntax of the proof-writing language will be formally defined by means of the metalinguistic formulas of BNF where

- a sequence of characters enclosed in metabrackets < > represents a metalinguistic variable
- :: = is a metalinguistic connective
- a sequence of characters not a connective and not in brackets represents itself or a class of symbols similar to it.

In general, we will choose mnemonic names for the character sequences and a reference in the text to that name without metabrackets will refer to that entity.

The grammar below is used to define mathematical proofs. It is also used to define isolated steps within a proof as well as mathematical statements which do not appear in proofs. We will first describe the overall grammar, and then identify the particular subsets of it which define lesser structures.

The grammar as processed by the system is presented. Note that in some cases, formulas appear which seem redundant to the actual syntactic definition. These have been introduced into the grammar specifically to aid in the translation process, because semantic processing may occur only when a reduction is made. (In Chapter 3 we will describe semantic processing procedures.) Also, the symbol @ has been introduced to aid in internal processing. It never appears in the user's actual input, but the translator interprets this symbol to mean 'end of line.'

A formula of the form NEW-GOAL :: = <proof> actually appears in the grammar that is processed by the system. It was added by the syntax analyzer used in this system, and it transforms the input grammar to an augmented grammar (see Parsing in Chapter 3). This formula will not be included in the discussion here.

Step numbers have been excluded from the grammar. However, all proof steps and any statement beginning with ASSUME, ALSO, CONCLUSION , or QED must be entered with a step number. Step numbers are described under section 2.5.2, the operating language.

The overall grammar may be viewed most conveniently in parts or subgrammars. These subgrammars define successively what is meant by a proof, a proof block, a proof step, a statement, a well-formed formula, a predicate, and a function.

2.5.1.1 Definition of a Proof

The following formulas define a proof:

<PROOF> :: = <PROOFBODY> <QEDSTMT> @

<PROOFBODY> :: = <BLOCK>

<PROOFBODY> :: = <PROOFBODY> <BLOCK>

<BLOCK> :: = <PROOFBLOCK>

<BLOCK> :: = <PROOFSTEP>

A proof is structured as an ordered sequence of one or more blocks followed by a QED statement, and a block is either a proof block or a proof step.

If no premises or assumptions need to be introduced, the proof may be simply a series of proof steps. Proof blocks may be intermixed or used exclusively however.

2.5.1.2 Definition of a Proof Block

The following formulas define a proof block:

<PROOFBLOCK> :: = <ASSERTION> <CONCLUSION>

<ASSERTION> :: = <ASSUMPTION>

<ASSERTION> :: = <ASSERTION> <BLOCK>

<ASSUMPTION> :: = <ARBASSUME>

<ASSUMPTION> :: = <SATASSUME>

<ARBASSUME> :: = ASSUME <ARBSTMT>

<SATASSUME> :: = ASSUME <SATSTMT> <WFF> @

<SATASSUME> :: = <SATASSUME> ALSO <WFF> @

<ARBSTMT> :: = <VAR> ARBITRARY

<ARBSTMT> :: = <VAR> , <ARBSTMT>

<SATSTMT> :: = <VAR> SATISFIES

<SATSTMT> :: = <VAR> , <SATSTMT>

<CONCLUSION> :: = CONCLUSION <STMT> <REASONS> @

<CONCLUSION> :: = <CONCLUSION> ALSO <STMT> <REASONS> @

A proof block is structured as an assertion followed by a conclusion, and an assertion is an assumption followed by any number of proof steps or proof blocks. That is, proof blocks may be nested. Up to 20 levels of nesting are allowed.

2.5.1.3 Definition of a Proof Step

The following formulas define a proof step:

<PROOFSTEP> :: = <STMT> <REASONS> @

<REASONS> :: =

<REASONS> :: = <REASONLIST>

<REASONLIST> :: = BY <REASON>

<REASONLIST> :: = <REASONLIST> , <REASON>

<REASON> :: = <STEPNUM>

<REASON> :: = <IDENTIFIER>

<REASON> :: = <STEPNUM> - <STEPNUM>

<STMT> :: = <WFF>

<STMT> :: = <LETSTMT>

<STMT> :: = CONTRADICTION

<STMT> :: = THEOREMFOLLOWS

<LETSTMT> :: = <LETHEAD> SATISFIES <WFF>

<LETHEAD> :: = LET <VAR>

<LETHEAD> :: = <LETHEAD> , <VAR>

A proof step is a statement followed by a possibly empty list of reasons. Two special statements, CONTRADICTION and THEOREM FOLLOWS , are indicated here and represent themselves.

If a step number is given as a reason, its scope must include the referencing step. A sequence of step numbers is implied by a reason of the form stepnum-stepnum . The latter should follow the former. An identifier given as a reason is the name of an axiom or theorem.

2.5.1.4 Definition of a Well-Formed Formula

A general mathematical statement is a logical expression or a well-formed formula. The following subgrammar defines a well-formed formula:

<WFF> :: = <ANDPRIM>

<WFF> :: = <ANDPRIM> <EQUIVOP> <ANDPRIM>

<WFF> :: = <ANDPRIM> <IMPOP> <ANDPRIM>

<ANDPRIM> :: = <ORPRIM>

<ANDPRIM> :: = <ANDPRIM> <ANDOP> <ORPRIM>

<ORPRIM> :: = <NOTPRIM>

<ORPRIM> :: = <ORPRIM> <OROP> <NOTPRIM>

<NOTPRIM> :: = <QPRIM>

<NOTPRIM> :: = <NOTOP> <QPRIM>

<QPRIM> :: = <PRIM>

<QPRIM> :: = <QUANT> <QPRIM>

<QUANT> :: = <UNIV> <VAR>

<QUANT> :: = <EXIST> <VAR>

<QUANT> :: = <QUANT> , <VAR>

<PRIM> :: = (<WFF>)

<PRIM> :: = <PREDICATE>

A well-formed formula is a rule for determining a logical value. The logical connectives are equivop (equivalence) , impop (implication) , andop (conjunction) , orop (disjunction) , and notop (negation) and they have the conventional logical meaning.

In general, logical evaluation of repeated operators is carried out from left to right except as noted below. According to the syntax, the following precedence rules hold:

1. quantification
2. negation
3. disjunction
4. conjunction
5. equivalence and implication

(Thus A AND B OR C AND D means A AND (B OR C) AND D).

In addition, any well-formed formula within parentheses will be evaluated by itself and this value is used in subsequent evaluations.

Thus, parentheses may be used to effect a desired order of evaluation.

This precedence is set up in compliance with the usual logical conventions employed when defining well-formed formulas of the predicate calculus, ([6] p. 30 etc.) with one exception. For clarity, we have required that any arguments of the equivalence or implication operators must be at a lower level. Therefore formulas of the following form are not acceptable:

P IMPLIES Q IMPLIES S

P IMPLIES Q IFF S

Explicit ordering must be specified by using parentheses in these cases. For example,

(P IMPLIES Q) IMPLIES S

P IMPLIES (Q IMPLIES S)

(P IMPLIES Q) IFF S

or P IMPLIES (Q IFF S)

For the other logical operators, evaluation of operators at the same level is carried out from left to right. That is, P OR Q OR S is accepted and evaluated as (P OR Q) OR S .

Note that quantification may apply to a predicate alone or to a formula in parentheses. That is, FOR ALL X NOT P(X) is not accepted, but FOR ALL X (NOT P(X)) is. Also FOR ALL X P(X) AND Q(X) is interpreted as (FOR ALL X P(X)) AND Q(X) (that is, X in Q(X) is not quantified).

2.5.1.5 Definition of Predicates and Functions

The last subgrammar defines predicates and functions.

<PREDICATE> :: = <LOGCON>

<PREDICATE> :: = <PREDSYMBOL> (<PRIMLIST>)

<PREDICATE> :: = <FN> <BINRELIN> <FN>

<PREDICATE> :: = <BINRELFIRST> <FN>

<PREDICATE> :: = <FN> <POSTUNREL>

<PREDICATE> :: = <PREUNREL> <FN>

<BINRELIN> :: = <BINREL>

<BINRELFIRST> :: = <BINREL>

<FN> :: = <FNSYMBOL> (<PRIMLIST>)

<FN> :: = <FIFTH>

<FN> :: = <FN> <BINOP4> <FIFTH>

<FIFTH> :: = <FOURTH>

<FIFTH> :: = <FIFTH> <BINOP3> <FOURTH>

<FOURTH> :: = <THIRD>

<FOURTH> :: = <FOURTH> <BINOP2> <THIRD>

<THIRD> :: = <SECOND>

<THIRD> :: = <THIRD> <BINOP1> <SECOND>

<SECOND> :: = <PRIMARY>

<SECOND> :: = <PREUNOP> <PRIMARY>

<SECOND> :: = <PRIMARY> <POSTUNOP>

<PRIMARY> :: = <CONST>

<PRIMARY> :: = <VAR>

<PRIMARY> :: = (<FN>)

<PRIMLIST> :: = <PRIMARY>

<PRIMLIST> :: = <PRIMLIST> , <PRIMARY>

All predicates and functions used in well-formed formulas are defined by the user. There are six types of predicate symbols: logical constants, binary (infix) relations, postfix unary relations, prefix unary relations, binary relations with vacuous lefthand side, and list-type predicate symbols. Each may only be used with the user-specified or inherently required number of arguments.

Predicates or relational operators have only functions (not other predicates) as arguments. That is, only first-order relations are allowed. Therefore no hierarchy of evaluation applies.

Functions, on the other hand, may have other functions as their arguments. The syntax defines the following precedence rules for evaluation of functions:

1. unary operators
2. binary operators, level 1, 2, 3, 4 respectively.

Parentheses may be introduced, as in the case of wffs , to cause evaluation of the enclosed function by itself before subsequent use as an argument of a function or predicate.

Both list-type functions and list-type predicates require that each argument by a primary, that is, either a constant, a variable, or a function of any other type in parentheses. Also, a list-type function must appear within parentheses if it is to be used as an argument of any type of function.

e.g.: A + F(X) is not acceptable as an argument

A + (F(X)) is acceptable

P((F(X)), (A+B)) is acceptable and all parentheses are required.

To avoid the necessity of parentheses around list-type functions the grammar could be changed as follows: in order to evaluate list-type functions first, delete

<FN> :: = <FNSYMBOL> (<PRIMLIST>)

and insert

<PRIMARY> :: = <FNSYMBOL> (<PRIMLIST>)

This would allow list-type functions to appear as arguments of list-type functions without parentheses and would allow their use similarly as arguments of other functions. However, unary and binary functions used as arguments of list-type functions or predicates would still need to be enclosed in parentheses.

Changing the definition of PRIMLIST as follows would avoid using parentheses for LIST or other function types as arguments of a list-type function or predicates, but parentheses would be required to use list-type functions as arguments of other non-list-type functions.

<PRIMLIST> :: = <FN>

<PRIMLIST> :: = <PRIMLIST> , <FN>

2.5.1.6 Other Symbols

The other symbols appearing in the grammar are listed below. They are classified as to whether they represent themselves, a class of similar symbols, or an easily identified group of symbols.

1. Symbols Representing Themselves (system-defined)

ASSUME	,
ALSO	-
ARBITRARY	(
CONCLUSION	)
BY	
CONTRADICTION	
LET	
THEOREMFOLLOWS	(actually represents THEOREM FOLLOWS or FOLLOWS only (THEOREM is a noise word))

2. Symbols Representing a Class of Similar Objects

a. System Defined

SATISFIES	represents SATISFY or SATISFIES	
<EQUIVOP>	logical equivalence operators	IFF or ↔
<IMPOP>	logical implication operators	IMPLIES or →
<ANDOP>	logical conjunction operators	AND or &
<OROP>	logical disjunction operators	OR or |
<NOTOP>	logical negation operators	NOT or ¬
<UNIV>	universal logical quantifier	ALL
<EXIST>	existential logical quantifier	EXIST or EXISTS

b. User Defined

<STEPNUM> a step number of the form xx or xx.xx where x is a digit and leading or trailing zeroes need not be expressed

<VAR> a symbol selected by the user which is not already defined or declared by the user not reserved by the system. A symbol is defined only for the duration of its scope. (See semantics.)

<CONST> a symbol defined by the user as indicated in the chart below, or declared as a constant in an ASSUME or LET statement. Again the definition or declaration only endures for its scope. (See semantics.)

The other user defined symbols are explained in the following chart.

Symbol in the Grammar	Meaning	How Entered (in operating language*)
<LOGCON>	a logical constant (propositional symbol)	PREDICATE logcon CONSTANT
<PREDSYMBOL>	a predicate (list-type) symbol	PREDICATE predsymbol (# arguments)
<POSTUNREL>	a postfix unary relation	PREDICATE postunrel POSTFIX
<PREUNREL>	a prefix unary relation	PREDICATE preunrel PREFIX
<BINREL>	a binary (infix) relation	PREDICATE binrel BINARY
<FNSYMBOL>	a function (list-type) symbol	FUNCTION fnsymbol (# arguments)
<BINOP4>	a binary (infix) function precedence level 4	FUNCTION binop4 BINARY (4)
<BINOP3>	a binary (infix) function precedence level 3	FUNCTION binop3 BINARY (3)
<BINOP2>	a binary (infix) function precedence level 2	FUNCTION binop2 BINARY (2)
<BINOP1>	a binary (infix) function precedence level 1	FUNCTION binop1 BINARY (1)
<PREUNOP>	a prefix unary function	FUNCTION preunop PREFIX
<POSTUNOP>	a postfix unary function	FUNCTION postunop POSTFIX
<IDENTIFIER>	an axiom or a theorem	AXIOM identifier statement or THEOREM identifier statement
<CONST>	a constant or zero-place function	FUNCTION const CONSTANT

* The operating language is described in the next section.

3. Others

<QEDSTMT>	proof step consisting of QED
@	internal use - end of line of a proof step

2.5.1.7 Other Uses of the Grammar

The grammar described above defines a proof, and is used in its entirety when a proof is translated and verified. However, our system also uses this grammar for syntax checking when axioms, theorems and proofs are entered into the user base. In this case, no translation occurs and no structural scheme is imposed. Each statement is treated as a single entity.

In order to use the grammar this way, a theorem or axiom must be a wff . For a proof step we accept QED , or we bypass CONCLUSION or ALSO , if appropriate, and look for a proof step.

The grammar is also used for translating reasons. For step numbers, the step is extracted and we seek a stmt , bypassing ASSUME, LET, SATISFY, ARBITRARY, CONCLUSION or ALSO as appropriate and stopping at BY or @ . When an identifier appears as a reason, the theorem or axiom referenced is retrieved and interpreted as a wff .

2.5.1.8 Syntax Notes

• Keywords

The following words are provided in the proof-writing language and are reserved (always have the system-defined meaning) when they are used in statements. This means that they may not be chosen to define predicates or functions. They may however be chosen for names of

theorems or axioms since these names would never appear in a statement although they may appear in a list of reasons. The keywords are:

QED	CONTRADICTION	OR
ASSUME	FOLLOWS	NOT
SATISFY	ARBITRARY	IMPLIES
SATISFIES	LET	IFF
ALSO	ALL	EXIST
CONCLUSION	AND	EXISTS
BY		

• Noisewords

The following words are provided for use at will to improve readability or they may be omitted to abbreviate commonly used expressions. They will always be ignored and are therefore reserved words and cannot be used as names for axioms, theorems, predicates or functions.

noiseword	intent
THEN	use anytime to improve readability
SUCH THAT	"
FOR	may abbreviate: ALL for FOR ALL
THERE	may abbreviate: EXISTS for THERE EXISTS or EXIST for THERE EXIST
THEOREM	may abbreviate: FOLLOWS for THEOREM FOLLOWS

• Spacing

1. Any number of blanks may separate words in the proof-writing language.
2. No blanks are required around the delimiters ( ) & , | ¬ or

- in a reason.

3. No blanks are required following → or ↔ , but a special character may not precede these strings if they are to be interpreted as the logical operators.
4. No blanks are permitted after # for a step number given as a reason.
5. In general, no blanks need to separate words (noisewords, keywords, user defined words) which can be identified by the class of characters they are composed of. The classes of characters are

Alphanumeric	A to Z 0 to 9 and underscore
Special characters	= + - * . % ; : < > ? $ # @ '

Any character not belonging to the class will terminate the word.

- Miscellaneous

1. Use of - and #

In reasons (after BY) these symbols are reserved.

In statements, they must be user defined.

2.5.2 The Operating Language

The operating language provides a method of naming components of mathematical statements and interpreting them as elements, functions or predicates. It also provides for the naming of mathematical statements and their interpretation as axioms or propositions. The statements and components named are permanent in the sense that they

have significance throughout the theory (see section 2.3). The operating language also includes the command which causes evaluation of proofs, leading possibly to the interpretation of propositions as theorems. A listing command is provided as well.

We will present the syntax of the operating language by using the COBOL metalanguage. Lower-case letters denote metalinguistic variables, braces { } denote choice, brackets [ ] denote optional, upper-case letters denote optional fixed words in the language, upper-case underlined words denote required fixed words in the language, ellipsis ... denotes an arbitrary number of repetitions of the previous unit. We also underline part of a fixed word to denote that only those letters need be specified. (Only the first three characters of keywords are actually processed, so that keywords may be abbreviated to three letters.)

The general formal for statements in the operating language is

keyword [name] [details]

where keyword identifies the statement type, name is an identifier provided by the user, and details provide further information.

Seven statements of the language will be described here. The first five involve defining and describing the mathematical system under investigation. These are the PREDICATE, FUNCTION, AXIOM, THEOREM , and PROOF statements. The sixth statement is the VERIFY statement which is a request for proof verification. The last statement we will describe is the LIST statement.

2.5.2.1 Defining Predicates and Functions

The statements used to define the symbols which are to be interpreted as predicates and functions are as follows:

{ PREDICATE / FUNCTION } symbol [type]

where: symbol is the 1 to 12 character symbol to be used to denote this item. The characters are composed of either A to Z, 0 to 9 and underscore (alphanumeric) or a combination of the special characters = + - * . % ; : ? $ # ' only.

type is one of the following:

- (numargs) where numargs is a digit representing the number of arguments which must appear with this symbol, enclosed in parentheses and separated by commas, when used in a mathematical statement.
- BINARY [(preclevel)] indicating a binary (infix) relation or operator with optional precedence level specified for binary operators. (Otherwise 1 is assumed.) Preclevel is a number from 1 to 4. Functions with level 1 are evaluated first, and equal levels are evaluated from left to right when the mathematical statement is processed.
- PREFIX for unary operators or relations preceding the argument.
- POSTFIX for unary operators or relations following the argument.

- CONSTANT for identifying particular elements of the domain by name, or for naming logical constants.

If type is not specified, the definition will be deleted from the user base.

The following examples illustrate the definition and subsequent usage of predicates and functions, assuming X, Y represent elements of the domain.

```
operating language:        PREDICATE LESS_THAN BINARY
proof-writing language:    X LESS_THAN Y

operating language:        PREDICATE P(3)
proof-writing language:    P(X,X,Y)

operating language:        PRED = BIN
proof-writing language:    X = Y

operating language:        PRED IS_A_PRIME POST
proof-writing language:    X IS_A_PRIME

operating language:        PRED TRUE CONSTANT
proof-writing language:    TRUE

operating language:        FUNCTION PRODUCT(2)
proof-writing language:    PRODUCT(X,Y)

operating language:        FUN * BIN(1)
proof-writing language:    X * Y

operating language:        FUN ' POSTFIX
proof-writing language:    X'

operating language:        FUNC INVERSE_OF PREFIX
proof-writing language:    INVERSE_OF X

operating language:        FUN ID CONSTANT
proof-writing language:    ID
```

2.5.2.2 Defining Mathematical Statements

The statements used to define mathematical statements are:

$$\left\{\begin{array}{l}\underline{\text{AXIOM}}\\ \underline{\text{THEOREM}}\end{array}\right\} \quad \text{name} \quad [\text{statement}]$$

where: name is the 1 to 12 character name assigned to this statement, and is an alphanumeric symbol string, and statement is the actual representation of the statement in the proof-writing language.

If no statement is specified, the statement will be deleted from the user base and any theorems, as well as their forward dependencies, dependent on the deleted statement will be set to 'unverified.'

The status of a theorem will initially be set at 'unverified' indicating that this statement is a proposition.

Examples:

```
AXIOM  ASSOC  FOR ALL X,Y,Z (X.Y).Z = X.(Y.Z)
AXIOM  LEFTINV  FOR ALL X THERE EXISTS Y SUCH THAT Y.X = ID
THEOREM  RIGHTINV  FOR ALL X THERE EXISTS Y SUCH THAT X.Y = ID
THEOREM  THM3  FOR ALL X,Y (X<Y+1 → X<Y | X=Y)
```

2.5.2.3 Proofs

The PROOF statement enables the creation of a proof. The format is:

$$\underline{\text{PROOF}} \quad \text{name} \quad \left\{\begin{array}{l}\text{step number} \quad \text{proof step}\\ \left\{\begin{array}{c}\text{step number\_m}\\ 0\end{array}\right\} - \left\{\begin{array}{c}\text{step number\_n}\\ \text{QED}\end{array}\right\}\end{array}\right\} \ldots$$

where name is the name assigned to a statement which must have been

defined via a THEOREM statement, and a proof step is a step of the proof as defined by the proof-writing language. The step number is of the form xx or xx.xx where x is a digit and leading or trailing zeroes need not be expressed.

Some additional facilities are provided regarding step numbers and null statements as described below since step numbers are also used to add, alter, or delete parts of proofs.

1. an entry of the form

   step number proof step

   will be processed in the following way:

   a. If there is no previously existing proof, or if there is no matching step number in the current proof, this statement causes the step to be added to the proof in numerically ascending order by step number.

   b. If there is a matching step number in the current proof, the step entered will replace the existing one, or delete it if a null step (i.e., no step) is entered.

2. an entry of the form

   step number_m - step number_n

   a. Deletes all those steps.

   b. Step number_m may be 0 meaning starting with the first step, and step number_n may be QED meaning up to and including the last step of the proof.

2.5.2.4 Verifying Proofs

The VERIFY statement effects proof verification. The format of

this statement is

VERIFY name

where name is the name of a proof (and a theorem) which is to be checked for logical validity. All steps of the proof will be checked and if no faulty logic is detected, the status of the named statement will be set to 'verified' indicating that the statement is indeed a theorem.

2.5.2.5 Listing the Contents of the User Base

The LIST statement enables listing of statements, definitions, and proofs on the user base. The format is:

LIST { STATEMENTS / DEFINITIONS / name } [ALPHA letter [TO letter]] [DEPEND {IMMED / ALL}]

If the STATEMENTS option is chosen, axioms and theorems will be listed in alphabetical order by name. Optionally, an initial letter or range of initial letters may be specified for listing only statements with corresponding names. If the ALPHA clause is not included, all statements are listed. Dependencies will be listed for each statement if the DEPEND option is used. Either immediate or full dependencies may be requested, the choice referring to both backwards and forwards dependencies. If the DEPEND clause is not present, no dependency information will be listed.

The DEFINITIONS option will cause listing of function and predicate definitions in alphabetical order by name. Again the ALPHA

clause may restrict listing to the indicated alphabetic subset. The DEPEND clause has no meaning when DEFINITIONS has been specified.

'Name' refers to the unique 1 to 12 character name assigned to theorems and axioms. By specifying 'name' the user indicates he or she desires that statement to be printed, along with its proof if available. Dependency information may be requested as well.

2.5.2.6 Syntax Notes

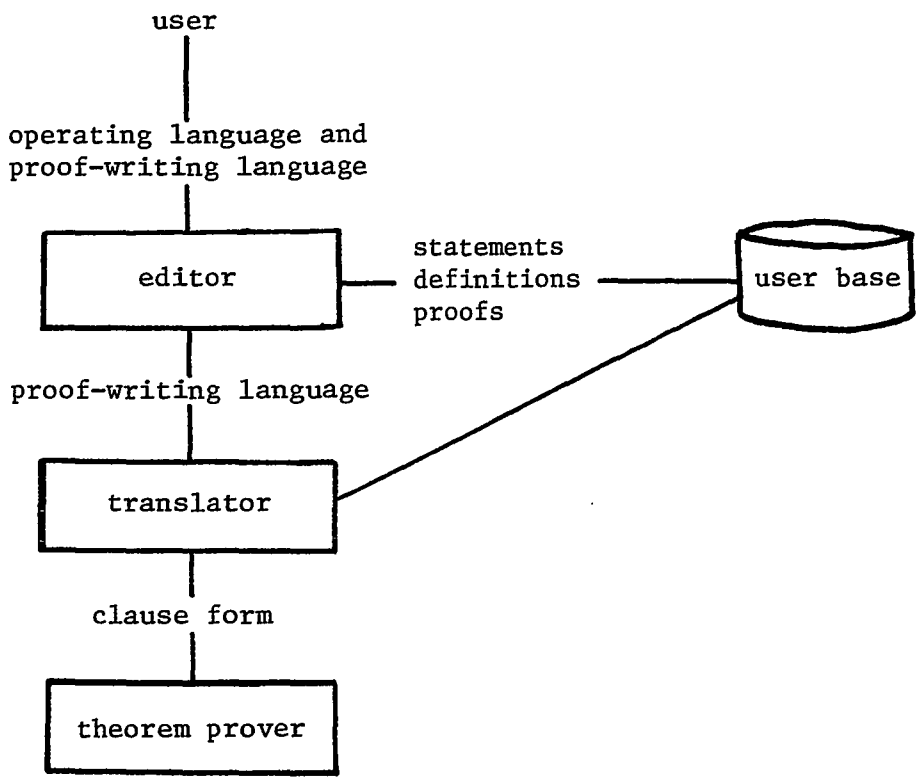
All keywords start in position 1 of the input record and a continuation statement has position 1 blank. At least one blank must separate words in the operating language. For proof steps, one step may appear following 'name' on the PROOF statement, and subsequent step numbers must start in position 1. Proof steps may also be continued by starting the continuation line past position one.

CHAPTER 3

IMPLEMENTING THE MODEL

3.1 IMPLEMENTATION OVERVIEW

The system is coded in PL/1 and is currently being run in batch mode on the system/370. There are three logical phases or subsystems: the editor, the translator, and the theorem prover. The editor directs overall processing by recognizing keywords in the operating language. It also maintains the user base. The translator is called by the editor to perform syntax checking or to translate the proof-writing language into predicate calculus and then clause form so that proof verification may take place. The theorem prover is called by the translator when a proof step is to be tested for validity.



More specifically, the following major tasks are carried out by the three subsystems:

The editor

1. recognizes keywords of the operating language and directs control accordingly
2. maintains user tables of predicates, functions, statements, and proof steps
3. provides a variety of capabilities for listing and editing information on the user base
4. calls the translator for
   - syntax checking of statements
   - syntax checking of proof steps
   - verification of a proof

The translator

1. performs syntax checking of statements and proof steps
2. translates proof steps into predicate calculus, and then into clause form
3. calls the theorem prover to verify steps of a proof
4. records information in user tables on status of proof steps and theorems, as well as dependency information

The theorem prover

1. seeks a contradiction from clauses received
2. returns one of three conditions: step valid, step not valid, validity of step cannot be determined

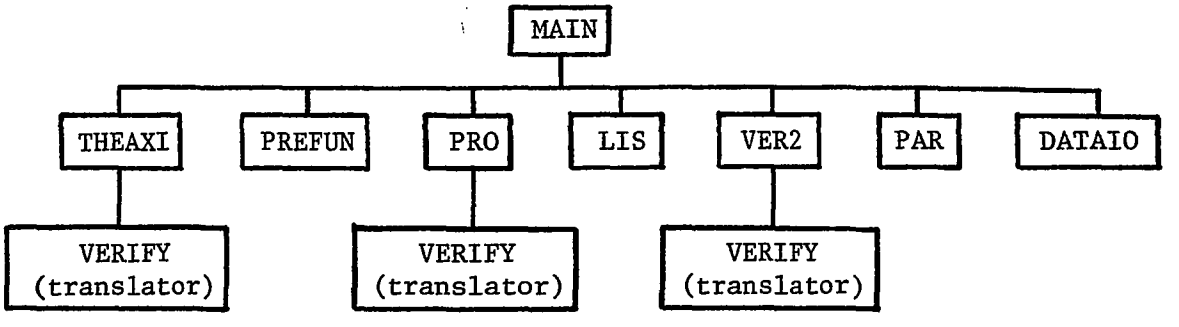
The next three sections will describe the editor, the translator, and the theorem prover, respectively.

3.2 THE EDITOR

3.2.1 Overview

The editor is the 'front-end' of the system, recognizing keywords in the operating language and directing control accordingly. The major task performed by the editor is the maintenance of the domain or user base of predicates, functions, statements and proof steps. The editor calls the translator (VERIFY) for syntax checking of statements and proof steps prior to their being saved in the user base. It also calls the translator for verification of a proof.

The editor consists of a main routine and seven subroutines. The main routine processes input records and upon recognition of the keywords of the operating language, it calls the appropriate processing module.



The THEAXI module is called to process THEOREM and AXIOM statements, PREFUN is called to process PREDICATE and FUNCTION statements, PRO is called for PROOF statements, VER2 and LIS for VERIFY and LIST statements respectively. Also the PAR module is called to set up parameter information provided in the input stream, and the DATAIO module is called for saving and restoring the user

base. The next section will briefly describe these modules.

3.2.2 The Modules

3.2.2.1 MAIN

The main routine receives processing parameters and calls DATAIO to restore the user base if it was previously saved for this user. Then the input stream is read in and for each statement, MAIN calls the appropriate module (THEAXI, PREFUN, LIS, PRO, PAR, VER2) for processing. DATAIO is called to save the user base at the end of the job.

3.2.2.2 THEAXI

THEAXI is called by MAIN for either of the following statements:

$$\left\{\begin{array}{l}\underline{\text{THEOREM}} \\ \underline{\text{AXIOM}}\end{array}\right\} \quad \text{name} \quad [\text{statement}]$$

It calls VERIFY (in the translator) for preliminary syntax checking of the mathematical statement entered. Statements with syntax errors are entered into the statement table of the user base, but they are flagged. If no axiom or theorem is already in the user base with the name indicated, the statement is entered. If a statement with the same name already is present in the user base, the new statement replaces it. If no statement was entered, the entry with that name is deleted.

Statements are accessed and stored by means of hash codes and alphabetic lists. Statement names are hashed to hash buckets which point to forward linked lists by hash code. Also alphabetic buckets

(one for each allowable character) point to forward linked lists (in alphabetical order within the list) by the first character of the name.

When statements are entered, forward pointers are set up and adjusted appropriately. When statements are deleted, the statement table is compressed and all statement and dependency pointers are adjusted accordingly. In addition, if a statement is either replaced or deleted, all forward dependencies (i.e., all statements whose validity depended on this replaced or deleted statement) are set to an 'unverified' status.

'Theorem' statements are entered into the user base initially with the 'unverified' status. (They are actually propositions as described in Semantics in Chapter 2.)

3.2.2.3 PREFUN

PREFUN is called by MAIN to process the following statements:

$$\left\{\begin{array}{l}\underline{\text{PRE}}\text{DICATE} \\ \underline{\text{FUN}}\text{CTION}\end{array}\right\} \quad \text{symbol} \quad [\text{type}]$$

The symbol representing the function or predicate is entered into the definition table of the user base. These symbols are accessed and stored in a manner similar to entries in the statement table, that is each entry is in two forward linked lists; one by hash code and one alphabetically. When symbols are added, pointers are adjusted accordingly. If the symbol is already in the table, the new definition replaces the old one. When symbols are deleted, the table is compressed.

3.2.2.4 PRO

PRO is called by MAIN to process a PROOF statement and to process any proof steps following it.

PROOF name { step number proof step { step number_m / 0 } - { step number_n / QED } } ...

First the name in the PROOF statement is checked to determine that a theorem with that name has been previously entered in the user base. If it has not, no proof steps will be processed. Proof steps to be entered into the proof table are first checked for syntax errors by the translator (VERIFY) . Steps with errors are entered, but flagged. Steps are entered in ascending order by step number, regardless of their physical sequence in the input stream. Proof steps are replaced if a matching step number is found for the step in the proof table. When one or more proof steps are to be deleted, the table is compressed.

3.2.2.5 VER2

VER2 is called by MAIN when a VERIFY statement is encountered:

VERIFY name

If no theorem with that name has been entered, or if the theorem or any of the proof steps has syntax errors, then no verification will be attempted. Otherwise, VER2 calls VERIFY , the controlling routine of the translator, to attempt verification of the proof for the named statement.

3.2.2.6 LIS

The LIS module is called by MAIN when the LIST keyword is recognized. This module implements the many listing options available to the user for examining the user base. The options include listing statements, definitions, proofs, and dependency information. Statements or definitions may be listed selectively in alphabetical order for all or part of the alphabet and full or immediate dependency information may be requested.

3.3 THE TRANSLATOR

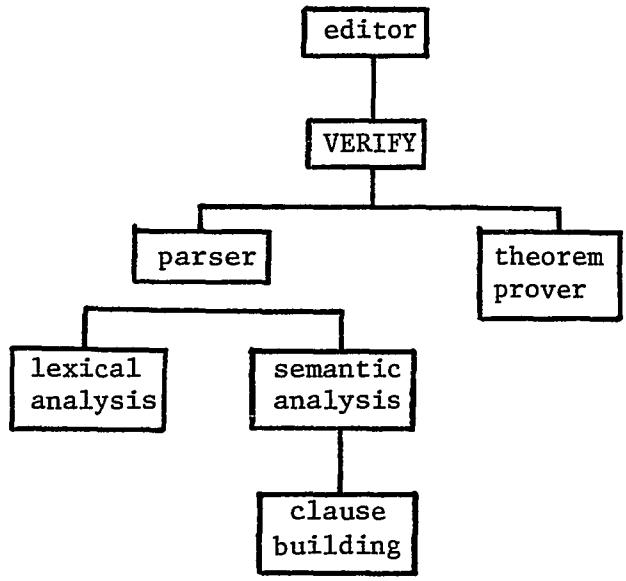
3.3.1 Overview

The major function of the translator is to translate the proof-writing language into clause form. This process may be viewed in four phases: lexical analysis, syntactic analysis, semantic analysis, and clause building.

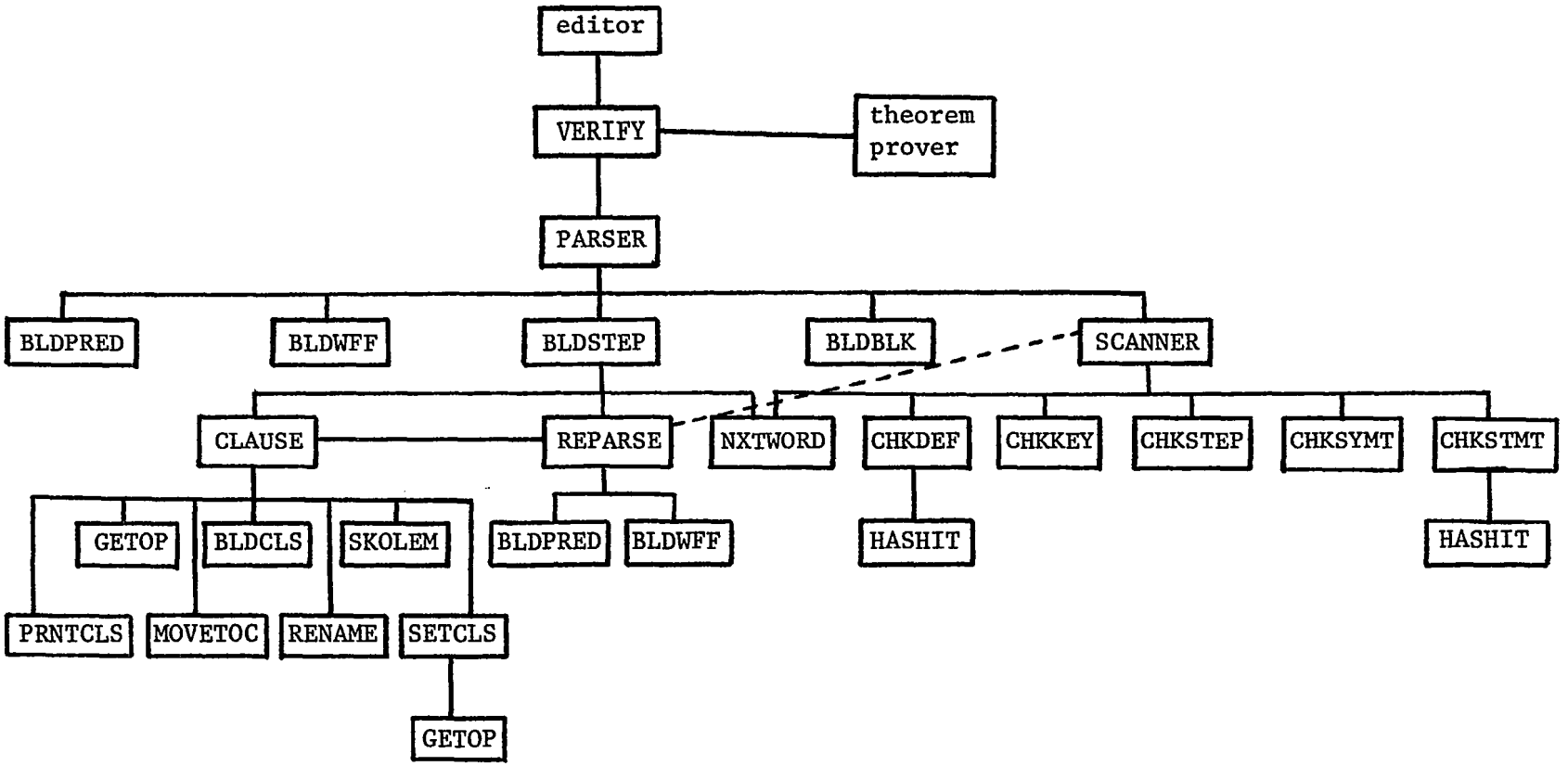
The lexical analysis routines identify words or tokens in the source language and pass them to the syntactic analysis procedures (parsers) which recognize syntactically correct strings. The parsers call the semantic routines, when appropriate, to generate the internal form of the language, which is further translated to clause form in the last phase.

For each proof step that requires verification, the translator calls the theorem prover which seeks a contradiction from the clauses passed.

The controlling routine of the translator, VERIFY , is called by the editor to perform syntax checking or to verify a proof. Schematically the translator may be viewed as follows:

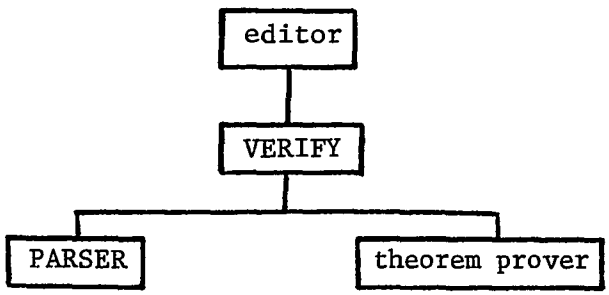


More specifically, twenty-three modules comprise the translator: the controlling routine (VERIFY) ; the parsers (PARSER, REPARSE); the lexical analysis routines (SCANNER, NXTWORD, CHKDEF, CHKKEY, CHKSYMT, CHKSTMT, CHKSTEP, HASHIT) ; the semantic routines (BLDPRED, BLDWFF, BLDSTEP, BLDBLK) ; the clause building routines (CLAUSE, GETOP, BLDCLS, RENAME, SKOLEM, PRNTCLS, MOVETOC, SETCLS) . They are related as follows.


editor
VERIFY
theorem prover
PARSER
BLDPRED
BLDWFF
BLDSTEP
BLDBLK
SCANNER
CLAUSE
REPARSE
NXTWORD
CHKDEF
CHKKEY
CHKSTEP
CHKSYMT
CHKSTMT
GETOP
BLDCLS
SKOLEM
BLDPRED
BLDWFF
HASHIT
HASHIT
PRNTCLS
MOVETOC
RENAME
SETCLS
GETOP

3.3.2 The Controlling Routine

VERIFY is the controlling routine of the translation phase. It is called by the editor for syntax checking or proof verification and it calls PARSER for translation activities and the theorem prover for verification.



VERIFY is called by THEAXI or PRO modules of the editor for syntax checking, or by VER2 in the editor for verification of a proof.

For syntax checking of a proof step or a statement, VERIFY sets a 'scan only' flag on and calls PARSER . PARSER returns an indication of whether or not a syntax error occurred and VERIFY returns this result to the editor.

For verification of a proof, VERIFY performs the following for each step of the proof.

1. Call PARSER , which returns with an indication of whether or not a syntax error occurred (syntax errors involving more than one proof step, i.e., structural errors, are not detected when proof steps are initially scanned, but are only diagnosed during verification). An indication of whether or not this step is to be verified is also returned (assumptions for

example are not).

2. If the step is to be verified, the theorem prover is called, and upon return VERIFY prints a message as to the status of that step (valid, invalid, or validity not determined).

After all steps have been processed, VERIFY prints summary information (number of syntax errors, number of valid steps, etc.) and dependency information (statements that the proof referenced). Then additional checks are made for required use of THEOREM FOLLOWS and QED , and the status of the theorem is determined and recorded. The status will be set to 'verified' only if no errors occurred, all steps were valid, and all dependencies were either axioms or verified theorems. If the theorem is 'verified,' VERIFY stores forward and backward dependency information.

3.3.3 Parsing*

We have presented the syntactic definition of our language in Chapter 2 by means of the BNF metalanguage. BNF grammars define context-free languages. A wide class of context-free languages are those specified by LR(k) grammars. Our proof-writing language is specified by an LR(1) grammar, or more specifically, by an LALR(1) grammar, and thus the language is a deterministic context-free language.

The deterministic context-free languages can be recognized and processed by a deterministic one-way pushdown transducer. The

* For references on linguistic classifications, concepts from automata theory, and parsing see [2], [16], [19], [28]. CFSM generation is discussed in [32], [33], [34] and implementation details are given in [20].

implementation of our parser is a simulation of a deterministic one-way pushdown transducer which can also look ahead one symbol. The transducer is represented as a set of tables which are the output of a collection of programs conceived by Professor David Pager of the University of Hawaii and implemented by this author. These programs accept as input a grammar in BNF form, which is transformed to an augmented grammar. The transducer is developed in an incremental manner incorporating a look ahead feature to resolve inadequacies (conflicting actions at states). It accepts LALR(1) grammars. The simulated transducer is called a characteristic finite state machine (CFSM) ([11]) and the CFSM tables are used in the parsers of our translation program.

The CFSM is represented as a set of lists corresponding to processing alternatives at each state of the parse. At any time, what action is to be carried out is determined by the current state of the machine and the current scanned symbol. This state symbol pair determines one of the following actions.

1. Stack the scanned symbol
2. Reduce the stack
3. Record a syntax error

After an action has been carried out, we must ascertain the next state/symbol pair in order to continue.

1. After a stacking action, we proceed to the next (possibly the same) state as indicated in the action list. The next scanned symbol will be the next symbol in the input string.
2. After a reduction has been made, we proceed to the state we

were in when we decided to stack the left-most symbol of the right part of this production. The left part (a nonterminal symbol) of this production becomes the next scanned symbol.

3. After a syntax error has been detected, we return to the state we were in before we started processing the current proof step. No more processing of this step will be made, and the next scanned symbol will be the first symbol of the next proof step, when it is processed.

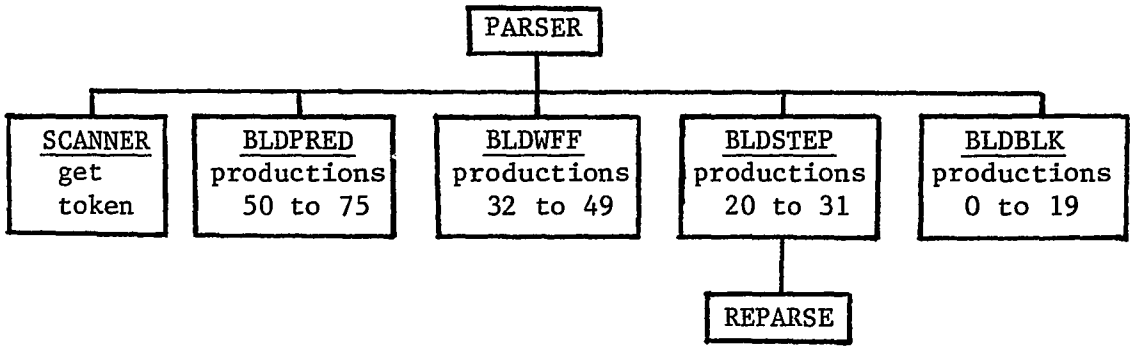
Another concern is the contents of the stack after an action has been carried out.

1. After a stacking action, the stack remains unchanged except for the addition of the scanned symbol. The length of the stack is therefore augmented by one.
2. After a reducing action, those rightmost symbols in the stack that represented the right part of the production are popped (removed) and the rest of the stack remains intact. The length of the stack is therefore decreased by the number of symbols in the right part of the production.
3. After a syntax error has occurred, any symbols placed on the stack during the processing of the current proof step are popped. The rest of the stack however remains unchanged.

Initially the stack is empty and we start in state 0 of the CFSM . The parser directs the movement through the CFSM , calling SCANNER when a new symbol is needed from the input string, stacking the scanned symbol when appropriate, calling REDUCE (in PARSER) when a reduction is to be made, and calling BACKUP (also in PARSER)

to recover from a syntax error. The successful completion of a proof will result in the CFSM being in an accept state, the stack being emply, and the scanned symbol being the goal symbol. (Note that the stack is actually parallel stacks retaining four items as described in the next section on lexical analysis.)

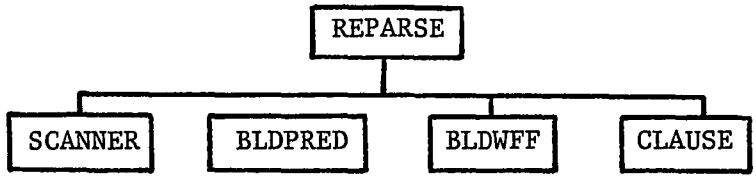
The translator actually contains two parsing modules, PARSER and REPARSE . PARSER is the main parsing procedure, controlling and directing the CFSM through the parse of a complete proof. REPARSE is used to direct the parsing and processing of reasons. Both modules are described next.



PARSER controls syntactic and semantic processing routines of the system. The input string is processed in a single left to right scan. When another word (denoted a "token") is needed from the input string, SCANNER is called to extract and recognize it. Certain symbol usage errors (e.g., undefined words) will be detected at this time, as well as improper references given as reasons. Erroneous usage of the language will be flagged as syntax errors by the parser.

As the input string is parsed, the semantic routines BLDWFF, BLDPRED, BLDSTEP, and BLDBLK will be invoked via REDUCE in PARSER

to apply actions upon recognition of appropriate language elements (predicates, wffs, steps, or blocks, respectively). BLDPRED and BLDWFF generate the intermediate form of the predicate calculus (postfix-Polish), BLDSTEP invokes the final translation phase from postfix-Polish to clause form, and BLDBLK handles block structure and scope processing. Note that no semantic routines will be called however if we are only scanning a statement or a proof step for syntax errors.



The secondary parser, REPARSE , is used to parse reasons. It operates very similarly to PARSER and uses the same stacks (but in a different unused area, carefully maintaining the original stack information on the entire proof). REPARSE is called by the semantic routine BLDSTEP when a reason needs to be processed.

REPARSE starts in state 0 and calls SCANNER for tokens via RESCAN , an internal procedure which preprocesses the tokens returned by SCANNER . The semantic routines for processing reasons are BLDPRED and BLDWFF only. When a reason has been converted to Polish intermediate form, REPARSE calls CLAUSE to carry the translation through to clause form. Then, after restoring the original parse stacks, REPARSE returns to BLDSTEP .

3.3.4 Lexical Analysis (Token Recognition)

The input to the verification procedure is a string of characters. Certain groups of characters function as a unit according to the specification of the language. For example, a string of characters may form a keyword, or a variable name. Lexical analysis procedures identify such a string of characters as a single entity commonly called a token. Aho and Ullman [2] define a token as a symbol string associated with a pair (token type, data) where token type is a category such as "variable" or "reason" and data represents the terminal symbols or a pointer to information about the symbols.

In our compiler the token type is indicated in TOKEN and the symbol string in WORD . Further identifying information is kept in TOKENID . The SCANNER directs lexical analysis, translating input symbols to (TOKEN, WORD, TOKENID) triples.

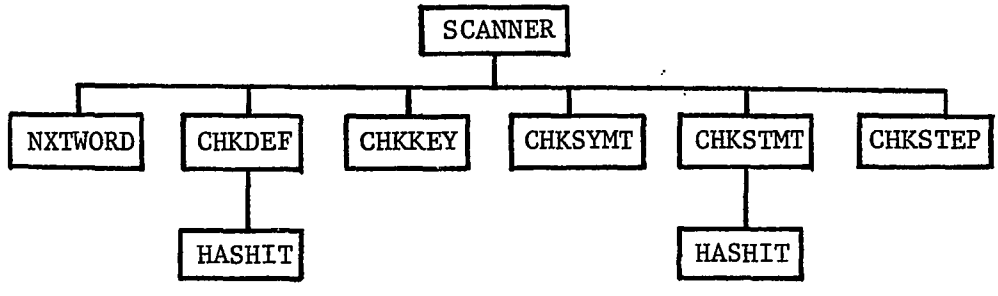
In general, TOKEN is used in the parsing procedure, and WORD and TOKENID are used by the semantic routines. The TOKENID may contain the step number of the current step, or the step number given as a reason, or a pointer to the user statement table for a theorem or an axiom given as a reason.

Aho and Ullman classify lexical analyzers as indirect or direct. ([2] p. 61.) An indirect analyzer looks only for a particular token when scanning an input string. Our lexical analyzer operates directly since, given an input string and a pointer into the string, the token immediately to the right of the pointer is determined, and then the pointer is moved past that part of the string forming the token just determined. Actually, in our case, the part of the string recognized

as a token is removed from the string and thus the next token is taken from the head of the string, proceeding until the string is empty.

In choosing tokens, one considers a tradeoff between the complexity of the scanner and the ease of parsing. Our particular choice of tokens was made to favor a more complicated scanner, enabling the lexical analysis routines to process many symbols up to the point where table entries were made or code generated. Our philosophy was

1. to keep the CFSM small, thus reducing translation time.
2. to keep the syntax of the grammar simpler.
3. to provide flexibility for changing tokens without necessitating a change in the grammar.



When a parser needs another token it calls the scanner which isolates and identifies the next word in a left to right scan of the input string. Words are recognized in the following way.

The scanner calls NXTWORD to extract and delimit a word from the string. NXTWORD fully identifies certain words and others are returned to the scanner for further analysis. NXTWORD classifies words as follows.

1. Noise words: (THEN, SUCH THAT, FOR, THERE, THEOREM).
2. Identifiers: words other than noise words beginning with and

including only the alphanumeric characters A to Z, 0 to 9 and _ . Any other symbol including a blank will terminate an identifier.

3. Single character delimiters: reserved characters, extracted as single characters, regardless of their immediate context. These delimiters are & (logical and), | (logical or), ¬ (logical not), ((left parenthesis),) (right parenthesis), and , (comma). A hyphen appearing in a reason is also processed as a single character delimiter. Note that its use in a statement is not reserved however, and the user may define a hyphen for use as a predicate or function.
4. The symbolic implication and equivalence operators: → and ↔ . These are extracted if they are the next character in the string regardless of whatever character follows the > . Note that this does not prohibit the use of these symbol combinations in user definitions if they follow another special character in a special character string. (See 6 below).
5. Step numbers as reasons: the numeric (0 to 9) characters with an optional decimal point following a # in a reason.
6. Special character strings: any string consisting only of one or more of the characters = + - * . % ; : < > ? $ # @ ' . Any other character including a blank will terminate the string. Note that these strings never appear as reasons, but are used only for predicate or function definitions. The use of - or # in reasons is treated differently (see 3 and 5 above).

7. NXTWORD will detect any other input symbol and print it with an ILLEGAL CHARACTER message.

The scanner, upon receiving the next word from NXTWORD , further identifies the words and assigns appropriate tokens to return to the parser. The scanner proceeds as follows.

1. Noise words are ignored.
2. If the next word is FOLLOWS the statement of the theorem is extracted from the user base and inserted in the input string replacing FOLLOWS and scanning proceeds on the theorem.
3. Identifiers are analyzed in the following manner:
   - If this is not a reason (i.e., if it is not past BY in the input string)
     a. The identifier is checked against a list of (reserved) key words, and if it is on this list it is returned as such. Note that keywords therefore may not be defined as predicates or functions for identifiers will always be checked as keywords first. Keywords may be used for statement names however since identifiers used as reasons are not checked against a keyword list. As such they are not strictly reserved words. The CHKKEY module identifies keywords from the following list: QED, ASSUME, SATISFY, SATISFIES, ALSO, BY, CONCLUSION, CONTRADICTION, FOLLOWS, ARBITRARY, LET, ALL, AND, OR, NOT, IMPLIES, IFF, EXIST, EXISTS.
     b. If an identifier is not a keyword it is next checked against user defined predicate and function names. The

CHKDEF module performs this search.

c. If the string being scanned is not a theorem or axiom from the user base the identifier is next checked against the symbol table of constant symbols declared in the proof. (These declarations only apply to statements given in proof steps.) The CHKSYMT module scans the symbol table which includes only symbols declared whose scope includes the statement being scanned.

d. Lastly, if the identifier is none of the above it is assumed to be a variable. Note that the grammar specifies that only variables may be used in ASSUME and LET statement symbol lists, and thus this processing precludes redeclaration of symbols within a block.

- If this is a reason (after BY in the input string) then CHKSTMT is called to check for the existence of the statement in the user base. If the statement is not found, the message UNDEFINED STATEMENT _____ GIVEN AS REASON will be issued.

4. Single character delimiters, implication and equivalence operators are returned to the parser with no further checking or identification.

5. Special character strings are checked by the CHKDEF module which searches user definitions for these strings. The appropriate token is returned for a matching predicate or function, or the message UNDEFINED WORD _____ will be

issued.

6. For a step number given as a reason, CHKSTEP is called to determine whether this is a legal reference. A reference to a step with a syntax error, or a step whose scope does not include the current step, or a step that cannot be found will result in an appropriate diagnostic.

3.3.5 Intermediate Form

The intermediate form used to represent strings prior to their conversion to clause form is postfix Polish form of the predicate calculus. This form is both useful and efficient in terms of the parsing strategy employed. The form we use is defined below.

- constants--in order of appearance during a left to right scan
- variables as arguments--in order of appearance
- functions--operator follows arguments

a + b	becomes	a b +
f(x,y)	becomes	x y f
x '	becomes	x '
inverse x	becomes	x inverse

- predicates--operator follows arguments

x = y	becomes	x y =
p(x,y)	becomes	x y p
x is_even	becomes	x is_even
complement a	becomes	a complement

- logical operators follow arguments

a = b or c = d	becomes	a b = c d = or
not (p(x) implies q(y))	becomes	x p y q implies not

- quantifiers follow quantified variables which follow their arguments

for all x,y p(x,y)	becomes	x y p y all x all
there exists x such that for all y,z p(x,y,z)	becomes	x y z p z all y all x exists

3.3.6 Semantic Routines

3.3.6.1 Predicate Recognition

BLDPRED is called by the parser (or by REPARSE) to process productions related to predicate formation. This processing yields a predicate in intermediate postfix Polish form. A predicate in this form consists of an entry in a predicate stack comprised of three things: a pointer to the user base for this predicate definition (PREDDEF); a pointer to the beginning of a predicate string (PREDPTR); the length of the predicate string (PREDLN) . The predicate string itself consists of the predicate and the arguments of the predicate in postfix Polish form. There are three types of entries in the predicate string:

1. proof defined constants (i.e., symbols declared arbitrary or listed in a LET statement) in which case the entry is a pointer to the symbol table.
2. variables, in which case the entry is a pointer to the

variable table.

3. user defined functions (including user defined constants) or the predicate entry being processed, in which case the entry is a pointer to the user table.

BLDPRED forms these entries when applying the approrpiate reduction based on the state of the parse. The following productions are processed: (50 to 75)

PRODUCTION 50 <PRIM> :: = <PREDICATE>

PRODUCTION 51 <PREDICATE> :: = <LOGCON>

PRODUCTION 52 <PREDICATE> :: = <PREDSYMBOL> (<PRIMLIST>)

PRODUCTION 53 <PREDICATE> :: = <FN> <BINRELIN> <FN>

PRODUCTION 54 <PREDICATE> :: = <BINRELFIRST> <FN>

PRODUCTION 55 <PREDICATE> :: = <FN> <POSTUNREL>

PRODUCTION 56 <PREDICATE> :: = <PREUNREL> <FN>

PRODUCTION 57 <BINRELIN> :: = <BINREL>

PRODUCTION 58 <BINRELFIRST> :: = <BINREL>

PRODUCTION 59 <FN> :: = <FNSYMBOL> (<PRIMLIST>)

PRODUCTION 60 <FN> :: = <FIFTH>

PRODUCTION 61 <FN> :: = <FN> <BINOP4> <FIFTH>

PRODUCTION 62 <FIFTH> :: = <FOURTH>

PRODUCTION 63 <FIFTH> :: = <FIFTH> <BINOP3> <FOURTH>

PRODUCTION 64 <FOURTH> :: = <THIRD>

PRODUCTION 65 <FOURTH> :: = <FOURTH> <BINOP2> <THIRD>

PRODUCTION 66 <THIRD> :: = <SECOND>

PRODUCTION 67 <THIRD> :: = <THIRD> <BINOP1> <SECOND>

PRODUCTION 68 <SECOND> :: = <PRIMARY>

PRODUCTION 69 <SECOND> :: = <PREUNOP> <PRIMARY>

PRODUCTION 70 <SECOND> :: = <PRIMARY> <POSTUNOP>

PRODUCTION 71 <PRIMARY> :: = <CONST>

PRODUCTION 72 <PRIMARY> :: = <VAR>

PRODUCTION 73 <PRIMARY> :: = (<FN>)

PRODUCTION 74 <PRIMLIST> :: = <PRIMARY>

PRODUCTION 75 <PRIMLIST> :: = <PRIMLIST> , <PRIMARY>

Variables are entered into the predicate string and into the variables table as they are encountered, and the same variable may appear several times in the variable table for a given statement. (This is because some, but not all, instances of a variable may need to be replaced by skolem functions during generation of clause form (see sections 3.3.7 and 3.3.8 on clause form)).

Constants have at this time already been entered into the symbol table so BLDPRED only places them in the predicate string when they are encountered.

If a list of arguments is required for a predicate or function, BLDPRED checks that the number of arguments is correct and if not, issues a diagnostic:

WRONG NUMBER OF ARGUMENTS FOR _____

DEFINED TO HAVE ____ ARGUMENTS BUT USED WITH ____

Up to ten levels of nesting are allowed.

Since the reduction of variables and constants is always

encountered prior to any reductions involving operators by the design of the grammar, BLDPRED simply moves functions to the predicate string when a reduction including an operator is encountered, and the result is a string of variables, constants, and operators (functions) in standard postfix Polish form (the operators following their arguments).

For example,

		postfix Polish
A + B	becomes	A B +
F(X,Y)	becomes	X Y F
G((F(X,Y)),Z)	becomes	X Y F Z G
A + B + C	becomes	A B + C +

Once a string has been built up in this manner, the processing of a predicate symbol results in entering the predicate symbol in the string, and setting up the predicate stack entry which points to the string just built. Also an entry in the wffstring (see next section) is set up to point to this predicate.

Since the grammar allows the use of a binary predicate with a vacuous left hand side, special attention must be paid when processing such strings. Two occurrences of such predicates must be considered: when directly processed as a statement; when accessed as a reason.

When a binary predicate with a vacuous left hand side is processed as the statement of a proof step, the right hand side of the preceding binary predicate must be retrieved. It is available in the last predicate string that was built. This string is recognizable because BLDPRED always marks the beginning and end of the polish

string for the right hand side of any binary predicate that is processed as the statement of a proof step. Then this string (which was built for the preceding statement) is copied into the predicate string being set up for the current statement.

Further processing must be done at this point since the current step might later be referenced as a reason. To retain the appropriate left hand side for this case, the string just retrieved is retained along with the step number for later access. Then, when a step containing a binary predicate with a vacuous left hand side is encountered as the reason of a proof step, that predicate can be processed by retrieving the appropriate left hand side string which was saved when the statement was processed the first time (i.e., as the statement part of a proof step).

3.3.6.2 Formula Recognition

The semantic routines that process well-formed formulas are found in BLDWFF . The parser calls BLDWFF to handle reductions dealing with logical connectives and quantifiers. The LET statement is also processed by BLDWFF . The productions processed are as follows: (32 to 49)

PRODUCTION 32 <LETSTMT> :: = <LETHEAD> SATISFIES <WFF>

PRODUCTION 33 <LETHEAD> :: = LET <VAR>

PRODUCTION 34 <LETHEAD> :: = <LETHEAD> , <VAR>

PRODUCTION 35 <WFF> :: = <ANDPRIM>

PRODUCTION 36 <WFF> :: = <ANDPRIM> <EQUIVOP> <ANDPRIM>

PRODUCTION 37 <WFF> :: = <ANDPRIM> <IMPOP> <ANDPRIM>

PRODUCTION 38 <ANDPRIM> :: = <ORPRIM>

PRODUCTION 39 <ANDPRIM> :: = <ANDPRIM> <ANDOP> <ORPRIM>

PRODUCTION 40 <ORPRIM> :: = <NOTPRIM>

PRODUCTION 41 <ORPRIM> :: = <ORPRIM> <OROP> <NOTPRIM>

PRODUCTION 42 <NOTPRIM> :: = <QPRIM>

PRODUCTION 43 <NOTPRIM> :: = <NOTOP> <QPRIM>

PRODUCTION 44 <QPRIM> :: = <PRIM>

PRODUCTION 45 <QPRIM> :: = <QUANT> <QPRIM>

PRODUCTION 46 <QUANT> :: = <UNIV> <VAR>

PRODUCTION 47 <QUANT> :: = <EXIST> <VAR>

PRODUCTION 48 <QUANT> :: = <QUANT> , <VAR>

PRODUCTION 49 <PRIM> :: = (<WFF>)

When BLDWFF is called, predicates have been transformed to Polish strings (see previous section). Now a wffstring (a string corresponding to a well-formed formula) in Polish form also, is built consisting of quantified variables, quantifiers, predicates, and logical operators.

The predicate entries in the wffstring consist of pointers to the predicate stack set up by BLDPRED . Quantified variables are entered into the variable table by BLDWFF as they are encountered. However, variables and their quantifiers are entered into the wffstring in right to left order (opposite to the order of encounter) with quantifiers following variables, which would be the appropriate polish representation since quantifiers are treated in effect as prefix unary logical operators.

For example, an input string in the following form, where P is any predicate:

FOR ALL X,Y THERE EXISTS W,Z SUCH THAT P

would appear in a polish wffstring symbolically as

P Z ∃ W ∃ Y ∀ X ∀

Internally, P would be represented by a pointer to the predicate stack entry for predicate P ; Z, W, Y , and X would be represented by respective pointers to the variable table; and ∃ and ∀ would be represented by their token values.

As reductions which include the standard logical operators are processed, their token values are included in the wffstring . Since, according to the design of the grammar, all arguments will have already been entered on the string, this procedure generates polish form. For example, P ∨ (Q ∧ S) becomes P Q S ∧ ∨ .

When a LET statement is checked, it is translated to an existentially quantified statement with special handling of the variables listed. The variables are entered into the variable table (as needed for translation to clause form) and into the wffstring followed by the existential quantifier, but in addition, named variables are entered into the symbol table as constants because any later reference to those symbols (within the current block, if any) denotes a reference to a particular declared constant. Thus a statement of the form LET X,Y SATISFY P is translated to P Y ∃ X ∃ and X and Y are thereafter declared constants.

3.3.6.3 Step Recognition

The BLDSTEP procedure is called by the parser when the input string is ready to be reduced to a proof step. At this point the proof statement has been translated to intermediate polish form. BLDSTEP invokes the CLAUSE routine which translates Polish to clause form. BLDSTEP also extracts reasons denoted in the proof step and calls REPARSE , the parser and controlling program for processing reasons. Upon completion of processing the production

<PROOFSTEP> :: = <STMT> <REASONS> @

all conversion to clause form will have been accomplished for the entire proof step and the linkage to the theorem proven will have been established. The following productions are processed: (20 to 31)

PRODUCTION 20 <PROOFSTEP> :: = <STMT> <REASONS> @

PRODUCTION 21 <REASONS> :: = <REASONLIST>

PRODUCTION 22 <REASONS> :: =

PRODUCTION 23 <REASONLIST> :: = BY <REASON>

PRODUCTION 24 <REASONLIST> :: = <REASONLIST> , <REASON>

PRODUCTION 25 <REASON> :: = <STEPNUM>

PRODUCTION 26 <REASON> :: = <IDENTIFIER>

PRODUCTION 27 <REASON> :: = <STEPNUM> - <STEPNUM>

PRODUCTION 28 <STMT> :: = <WFF>

PRODUCTION 29 <STMT> :: = <LETSTMT>

PRODUCTION 30 <STMT> :: = CONTRADICTION

PRODUCTION 31 <STMT> :: = THEOREMFOLLOWS

BLDSTEP performs as follows:

1. When the statement of a proof step is recognized, it is negated (by placing a logical not at the end of the wffstring) and BLDSTEP calls CLAUSE to translate the wffstring to clause form. A statement reading CONTRADICTION results in no processing at all since contradiction means 'false' and not false (negation of the statement) means 'true' and $P \wedge$ true where P is any formula, is logically equivalent to P .
2. Processing of reasons is more complex. If the reason is an axiom of theorem, that statement is extracted from the user base and REPARSE is called to process the statement. The statement is also added to a list of dependencies for the theorem being proved.

If the reason is given as a step number within the current block, that step in the current proof is extracted. If the current step being processed is a CONCLUSION , the reason must be treated as an implication, with the current assumptions as the hypothesis and the step extracted as the conclusion. The GETASSUME procedure within BLDSTEP performs this function, generating the disjunction of the reason with the negation of the conjunction of the assumptions in the current block. If the step given as a reason was a CONTRADICTION then the reason consists of the negation of the assumptions only, since $P \rightarrow$ CONTRADICTION is equivalent to $\neg P \vee$ CONTRADICTION is equivalent to $\neg P$.

Once the assumptions have been prefixed to the step, REPARSE is called to process the reason string. For example,

1. Assume ... Satisfy P
2. Also Q
   .
   .
   .
6. R
7. Conclusion S by #6
8. Also T by #6

The reason for both steps 7 and 8 would be processed as

$$\neg((P) \wedge (Q)) \vee (R)$$

If no reasons are given for a proof step, the statement alone will be subject to logical verification in the theorem prover. Note that GETASSUME invokes NXTWORD in order to locate the formula given in an ASSUME ... SATISFY ... or ALSO statement.

3.3.6.4 Block Recognition

Block structures are identified and processed by BLDBLK . The major considerations for block handling are to enable access only to steps and declarations whose scope includes a step being processed. The scope is determined as a particular set of symbol table entries and a particular set of proof steps. BLDBLK provides up to 20 levels of nesting, and symbols may be redeclared in different independent blocks, but retain one usage within a block (including nesting). A block table entry for each block is set up. It consists of a pointer to the symbol table to the first entry this block, a pointer to the block table entry at the next highest level, a pointer to step table first entry this block, and the number of steps in this block.

When an ASSUME statement is processed, a new block table entry is set up with a pointer to the next free entry in the symbol table, and a pointer to the current setp; the number of steps in this block is initialized to one; and a backward pointer to the last block table entry made. The symbols in the ASSUME statement are entered into they symbol table as arbitrary constants.

As subsequent proof steps are completely processed, the number of steps associated with the current block, and also any nesting blocks, is incremented by one. When a conclusion is reached, all steps in the current block are released (by setting a status entry) making them inaccessible, and any symbol table entries made in this block are deleted. The next highest block then becomes the current block.

PRODUCTION 0 NEW GOAL :: = <PROOF>

PRODUCTION 1 <PROOF> :: = <PROOFBODY> <QEDSTMT> @

PRODUCTION 2 <PROOFBODY> :: = <BLOCK>

PRODUCTION 3 <PROOFBODY> :: = <PROOFBODY> <BLOCK>

PRODUCTION 4 <BLOCK> :: = <PROOFBLOCK>

PRODUCTION 5 <BLOCK> :: = <PROOFSTEP>

PRODUCTION 6 <PROOFBLOCK> :: = <ASSERTION> <CONCLUSION>

PRODUCTION 7 <ASSERTION> :: = <ASSUMEBLOCK>

PRODUCTION 8 <ASSERTION> :: = <ASSERTION> <BLOCK>

PRODUCTION 9 <ASSUMEBLOCK> :: = <ARBASSUME> @

PRODUCTION 10 <ASSUMEBLOCK> :: = <SATASSUME>

PRODUCTION 11 <ARBASSUME> :: = ASSUME <ARBSTMT>

PRODUCTION 12 <SATASSUME> :: = ASSUME <SATSTMT> <WFF> @

PRODUCTION 13 <SATASSUME> :: = <SATASSUME> ALSO <WFF> @

PRODUCTION 14 <ARBSTMT> :: = <VAR> ARBITRARY

PRODUCTION 15 <ARBSTMT> :: = <VAR> , <ARBSTMT>

PRODUCTION 16 <SATSTMT> :: = <VAR> SATISFIES

PRODUCTION 17 <SATSTMT> :: = <VAR> , <SATSTMT>

PRODUCTION 18 <CONCLUSION> :: = CONCLUSION <STMT> <REASONS> @

PRODUCTION 19 <CONCLUSION> :: = <CONCLUSION> ALSO <STMT>
<REASONS> @

3.3.7 Clause Form

A particular version of predicate calculus is used in resolution deductive programs. This language, called clause form, is a language of predicates. The only explicit logical operator in the language is negation. The sentences of the language are sets of assertions of predicates with or without negation. For example, assuming P, Q are predicate symbols, and x, y are variables, the following is a sentence:

$$\{\neg P(x),\ Q(x,y)\}$$

The set is called a clause and the members are called literals. A clause is interpreted as the logical disjunction of its literals: that is, the above clause is interpreted as meaning

$$\neg P(x) \quad \text{or} \quad Q(x,y)$$

Any variables in the clause are assumed to be universally quantified. Thus the above clause is equivalent to

$$\forall x \forall y (\neg P(x) \text{ or } Q(x,y))$$

Existential quantification is also expressible in this language. Consider a formula of the form

$$\exists x P(x,y)$$

The variable x represents some unnamed element. In clause form, the element is given a name, say s, called a skolem constant. Thus the formula becomes $P(s,y)$.

If the unnamed element depends on a universal variable, for example

$$\forall y \exists x P(x,y)$$

the variable x is replaced by a skolem function of that variable. Thus the above formula may be represented as $P(s(y),y)$. In a similar manner, a skolem function of any degree may be set up, as appropriate, to replace the occurrence of an existentially quantified variable.

A set of clauses is interpreted as the conjunction of its members.

Any well-formed formula in the predicate calculus can be translated to clause form by following a series of transformations. Each transformed formula is tautologically equivalent to its predecessors. These transformations are carried out in the folloiwng steps (assuming P, Q, and R are formulas).

1. Eliminate the equivalence operator replacing

   $$P \leftrightarrow Q \text{ by } (P \rightarrow Q) \wedge (Q \rightarrow P)$$

2. Eliminate the implication operator replacing

   $$P \rightarrow Q \text{ by } \neg P \vee Q$$

3. Reduce the scope of negation to a single predicate by replacing

$$\neg\forall\ x\ P \quad \text{by} \quad \exists\ x\ \neg P$$

$$\neg\exists\ x\ P \quad \text{by} \quad \forall\ x\ \neg P$$

$$\neg\neg P \quad \text{by} \quad P$$

$$\neg(P \wedge Q) \quad \text{by} \quad \neg P \vee \neg Q$$

$$\neg(P \vee Q) \quad \text{by} \quad \neg P \wedge \neg Q$$

4. Rename all quantified variables so that each is unique
5. Replace existentially quantified variables by skolem functions,

   e.g., replace $\exists\ y\ P(x)$ by $P(s)$

   replace $\forall\ x\ \exists\ y\ P(x,y)$ by $\forall\ x\ P(x,s(x))$

   and so forth
6. Delete all quantifiers
7. Distribute disjunction over conjunction, replacing

$$P \vee (Q \wedge R) \quad \text{by} \quad (P \vee Q) \wedge (P \vee R)$$

$$(P \wedge Q) \vee R \quad \text{by} \quad (P \vee R) \wedge (Q \vee R)$$

We now have an equivalent representation of the original formula which is the conjunction of a finite set of disjunctions of predicates and/or negations of predicates containing only universally quantified variables. The conjunction operators conjoin clauses and we may remove them with this understanding. The disjunction operators disjoin literals and we may remove them as well.

The following formula (which is the induction axiom used in Figure 4) will be transformed for illustration purposes.

$$\forall\ x\ (S(0) \wedge (S(x) \rightarrow S(x')) \rightarrow \forall\ x\ S(x)$$

1. Eliminate equivalence operator (not applicable)
2. Eliminate implication operator

a. $\forall x\ (S(0) \wedge (\neg S(x) \vee S(x'))) \rightarrow \forall x\ S(x)$

b. $\neg\forall x\ (S(0) \wedge (\neg S(x) \vee S(x'))) \vee \forall x\ S(x)$

3. Reduce scope of negation

a. $\exists x\ \neg(S(0) \wedge (\neg S(x) \vee S(x'))) \vee \forall x\ S(x)$

b. $\exists x\ (\neg S(0) \vee \neg(\neg S(x) \vee S(x'))) \vee \forall x\ S(x)$

c. $\exists x\ (\neg S(0) \vee (\neg\neg S(x) \wedge \neg S(x'))) \vee \forall x\ S(x)$

d. $\exists x\ (\neg S(0) \vee S(x) \wedge \neg S(x'))) \vee \forall x\ S(x)$

4. Rename quantified variables

$$\exists x\ (\neg S(0) \vee (S(x) \wedge \neg S(x'))) \vee \forall y\ S(y)$$

5. Skolemize (using sk for skolem constant)

$$(\neg S(0) \vee (S(sk) \wedge \neg S(sk'))) \vee \forall y\ S(y)$$

6. Delete quantifiers (all variables are now universally quantified)

$$(\neg S(0) \vee (S(sk) \wedge \neg S(sk'))) \vee S(y)$$

7. Distribute disjunction over conjunction

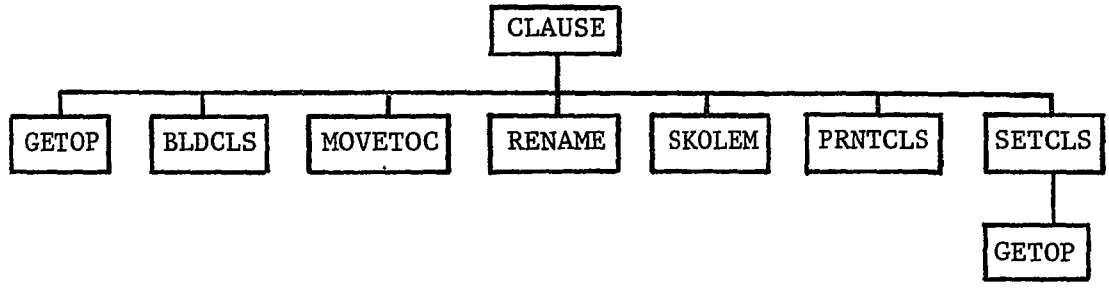
a. $(\neg S(0) \vee S(sk)) \wedge (\neg S(0) \vee \neg S(sk')) \vee S(y)$

b. $(\neg S(0) \vee S(sk) \vee S(y)) \wedge (\neg S(0) \vee \neg S(sk') \vee S(y))$

Thus the original formula is equivalent to the following set of clauses:

$$\{\{\neg S(0),\ S(sk),\ S(y)\},\ \{\neg S(0),\ \neg S(sk'),\ S(y)\}\}$$

3.3.8 Transformation to Clause Form



The CLAUSE module directs the translation process from Polish intermediate form to clause form. It is invoked by BLDSTEP directly, for processing statements of proof steps, or indirectly from BLDSTEP via REPARSE for processing reasons of proof steps. In either case the wffstring (see preceding section on formula recognition) holds the Polish representation of the statement or reason. This string, consisting of quantified variables, quantifiers, predicates, and logical operators, is first moved to a clause area (denoted c-area) and is scanned several times during the transformation.

Three utility modules, GETOP, BLDCLS , and MOVETOC , are invoked repeatedly during the translation process. GETOP is called to locate the operands of specific logical operators; BLDCLS to build a replacement string in a temporary area (denoted t-area); MOVETOC , to insert the t-area back into the original c-area .

Assuming A, B are well-formed formulas and x, y are variables, the algorithm proceeds as follows:

1. Eliminate equivalence operator. Strings of the form A B $\leftrightarrow$ are replaced by A B $\rightarrow$ B A $\rightarrow$ $\wedge$. GETOP, BLDCLS , and MOVETOC are invoked. Note that BLDCLS sets up completely new predicate entries for repeated predicates since the variables may later be treated differently in each predicate. For example, the infix statement

   $\exists$ x $\forall$ y P(x,y) $\leftrightarrow$ Q becomes

   $\exists$ x $\forall$ y P(x,y) $\rightarrow$ Q $\wedge$ Q $\rightarrow$ x y P(x,y)

   and after the implication operator is eliminated (in step 2) it becomes apparent that the variable y in the leftmost

predicate P(x,y) will be skolemized to a function of x , while this will not be true for the variable y in the rightmost copy of the predicate P(x,y) .

2. Eliminate implication operators. Strings of the form A B → become A ¬ B | . GETOP, BLDCLS , and MOVETOC are invoked.
3. Reduce the scope of a negation operator to a single predicate. This requires several passes of the clause string.
   a. The first pass moves a negation operator past a quantifier within its scope and transforms that quantifier from universal to existential or vice versa as appropriate. e.g., Strings of the form A × ∃ ¬ become A ¬ × ∀ and strings of the form A × ∀ ¬ become A ¬ × ∃ .
   b. The second pass removes double negation. Strings of the form A ¬ ¬ become A . BLDCLS and MOVETOC are invoked.
   c. The third pass applies De Morgan's law to negation operators operating on a disjunction or conjunction operator. Strings of the form A B ∧ ¬ become A ¬ B ¬ | and strings of the form A B | ¬ become A ¬ B ¬ ∧ . GETOP, BLDCLS , and MOVETOC are invoked.

   The three pass cycle is performed until no string transformations occur.
4. Rename all quantified variables. When a quantifier is encountered, GETOP is invoked to determine the scope of the quantifier. Then RENAME is called to assign a number to each occurrence of the variable quantified within the scope

of its quantifier. This number will be unique within a wffstring (statement or reason of a proof step).

5. Skolemize to replace existentially quantified variables. Existentially quantified variables are replaced by skolem functions of degree k , where k is the number of universally quantified variables on which the existential quantifier is dependent. GETOP determines the scope of the quantifier and SKOLEM is invoked to establish the function and replace the variable. Skolem functions are unique within each proof step (statement and reasons).
6. Delete all quantifiers. BLDCLS and MOVETOC are called to delete all quantifier-variable pairs from the c-area .
7. Distribute disjunction over conjunction to form a conjunction of disjunctions (conjunctive normal form). Strings of the form ABC ∧ | become AB | AC | ∧ and strings of the form AB ∧ C | become CA | CB | ∧ . Any operand of the disjunction which itself is not a conjunction will be distributed. If both operands are conjunctions, the rightmost will be distributed. GETOP, BLDCLS , and MOVETOC are invoked. The c-area is rescanned until no transformations occur.
8. Lastly, SETCLS is called to set up the linkage for the theorem prover. The clauses are attached to the linkage area as they are converted to clause form.

3.3.8.1 Clause Building Utility Modules

- GETOP

GETOP is called by CLAUSE or SETCLS to locate the operand or scope of an operator recognized by CLAUSE . Three arguments (K, P, and L) are involved. K points to the rightmost entry of the operand in the c-area . P and L are returned as pointer to the c-area , and length of the operand, respectively.

An operand is either a predicate (possibly negated or quantified) or a binary logical operator and its operands (possibly negated or quantified). GETOP scans the c-area from right to left starting at position K . If a predicate is found first, GETOP returns with P and L set appropriately. Else if a binary operator (equivalence, implication, and, or) is found, the scan will continue until both operands of this operator, which may also include other binary operators, have been located. For example, consider the string

A	B	∧	C	∧	D	∧
1	2	3	4	5	6	7

To locate the operand whose rightmost position is 5 (K = 5) GETOP would look for 2 operands for the operator in 5. They are C and operator ∧ , but operator ∧ needs 2 operands. They are A and B , GETOP returns with P = 1 , L = 5 .

When GETOP is called to determine the scope of a quantifier, K points to the quantifier and the processing is as above.

• BLDCLS

BLDCLS is called to set up an entry in a temporary string area (the t-area). It operates in three modes depending on its arguments (I, PTR, LN) which are as follows:

I = -1 if moving part of the c-area directly to the t-area

= TOKEN (always > 0) if that token (representing a logical operator) is to be entered in the t-area

= -2 if moving from the c-area as in I = -1 but also new predicate entries must be set up for all predicates moved. (This is used when duplicating entries in equivalence removal.)

PTR, LN used only if I is negative, PTR points to the first entry of the c-area to be moved and LN is the number of entries to move

• MOVETOC

MOVETOC moves strings built in the t-area back to the c-area , replacing or inserting strings as required. The two arguments P1 and P2 represent the part of the c-area (from entry P1 up to and including entry P2) to be replaced. The t-area is also set empty by MOVETOC .

• PRNTCLS

This module is used only for debugging purposes to print the

clause area, predicate stack, and variable table.

3.3.8.2 The Other Modules

- RENAME

RENAME is called by CLAUSE when a quantifier is recognized in the c-area . It will assign a unique number to the quantified variable when it appears in the scope of the quantifier. RENAME receives four arguments: I , the position of the quantifier; OP1P , the position of the first entry in the operand (scope) of the quantifier; OP1L , the length of the operand up to and including the quantifier itself; NEW , which is true if this is the first call to RENAME for this wffstring . The first time, the number is set to 0 . The variable to be renamed (in entry I-1) is renamed by assigning a fixed (unique within this step) number to it in the variable table. Note that only the specific variable table entries pointed to by predicates in the scope will be renamed. That same variable appearing in other places in the variable table will not be renamed.

- SKOLEM

SKOLEM is called by CLAUSE to replace existentially quantified variables by skolem functions. These variables are pointed to in predicate entries, and the pointers are changed to point to an entry in the skolem table. SKOLEM determines the degree of the function from the number of universally quantified variables the existential quantifier depends on and enters a skolem function of this degree and with those variables as arguments.

For example a wff of the form $\forall x \exists y P(x,y)$ would be represented in Polish intermediate form internally as

wffstring	predstk	predstr	
P y ∃ x ∀	predptr = predstr entry predln = 2 ⋮	prtype = 2 prptr = vtab entry for x	prtype = 2 prptr = vtab entry for y

The algorithm determines that occurrences of y in predicate P should be replaced by a skolem function of x . After SKOLEM , the predstr entry for variable y would be changed to

	skolem table	skstring
prtype = 4 prptr = skolem table entry	skname = SK.nn (nn is current skolem number) sknarg = 1 skptr = skstring entry	vtab entry for x

- SETCLS

SETCLS sets up the linkage to the theorem prover. After conversion to clause form has been completed, the c-area contains one clause, or a conjunction of clauses, each clause consisting of a disjunction of literals. The c-area therefore contains only predicate pointers and negation, disjunction and conjunction operators. For example, suppose three clauses c_i , i = 1,2,3 were generated, where P_{ij} represent literals, as follows:

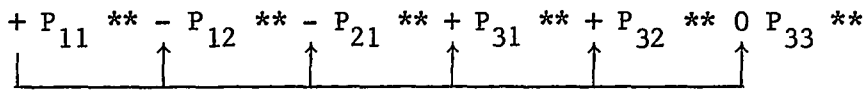
$$c_1 = (P_{11} \vee P_{12})$$
$$c_2 = (P_{21})$$
$$c_3 = (P_{31} \vee P_{32} \vee P_{33})$$

The c-area contains, $c_1 \wedge c_2 \wedge c_3$ in polish form as follows:

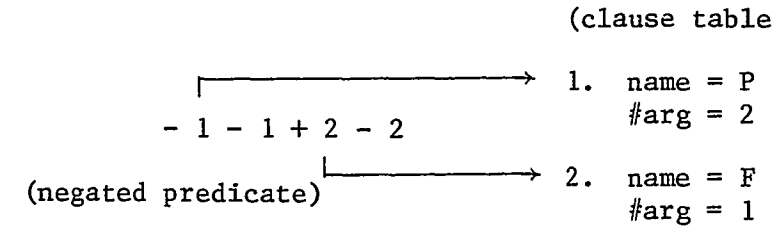
$$P_{11}\ P_{12}\ \vee\ P_{21}\ \wedge\ P_{31}\ P_{32}\ \vee\ P_{33}\ \vee\ \wedge$$

SETCLS reformats this string for linkage to the theorem prover. The linkage is set up as a string with entries representing predicates and their arguments (if any) or pointers used to distinguish predicates within clauses, and clauses within the string. This string will be processed from left to right by the theorem prover and predicates and their arguments are set up in prefix-polish form. Pointers precede and endmarkers follow predicate entries. The pointers point to succeeding predicate entries and they are positive if the next predicate is in the same clause, and negative if the next predicate is the first entry of the next clause. The last predicate of the string is preceded by a pointer set to 0 . Predicate and function entries consist of pointers to a clause table, set up by SETCLS , containing the symbolic name and the number of arguments. Negative pointers represent negated predicates and variables are represented by their renamed value (see RENAME) with a negative sign.

The preceding example would be set up as follows:



where ** represents a predicate endmarker. Expanding this, if P_{11} were a predicate of the form $\neg P(x,F(y))$ with x, y variables it would be represented by



assuming x was renamed 1 and y was renamed 2 .

As SETCLS sets up this clause string, GETOP is invoked to identify clauses which are operands of the conjunction operator if one appears in the c-area .

As predicates are moved to the clause string, constants and variables are checked as follows:

1. If a proof-defined constant is encountered and the CONCLUSION proof step is being processed which terminates the declaration, constants declared arbitrary are transformed to variables (thus implementing universal generalization).
2. If a variable appears with no renamed value (indicating it was not quantified) then the message "SYMBOL xxx WAS NEITHER QUANTIFIED NOR DECLARED" is issued resulting in a syntax error for this step.

SETCLS also sets up a pointer to the clause or set of clauses which represent the statement as opposed to the reason part of the proof step. This pointer (BEGSOS) therefore denotes those clauses to be placed in the set of support, if the set of support strategy is to be used in the theorem prover (see next section).

3.4 THE THEOREM PROVER

3.4.1 Overview

The theorem prover is called by the translator when a proof step has been translated to clause form and is ready to be checked for validity. This procedure forms the logical basis of the system and it implements the resolution principle.

3.4.1.1 Inference Procedures

Resolution was introduced in 1965 by J. A. Robinson [36] and it dramatically improved upon previous efforts in mechanical theorem proving. The resolution principle states that from any two clauses, one may infer their resolvent. Briefly*, a resolvent is formed by first replacing terms for variables in a pair of clauses in such a manner that a literal appears in one clause and its negation appears in the other. Then a new clause is formed which includes the instantiated literals from both parent clauses except for the complementary pair which is deleted (along with any copies of each literal of this pair which may have appeared in the parent clauses). This clause is called a resolvent of the pair of clauses.

The substitution applied is called the most general unifier of the literals which were omitted from the resolvent. The resolution $R(S)$ of any set of clauses S is the set of all clauses which are members of s or resolvents of members of S, and the nth resolution of S, denoted $R^n(S)$, is defined as $R^0(S) = S$ and for $n \geq 0$,

* For more information on resolution see [6], [31], [39].

$R^{n+1}(S) = R(R^n(S))$.

The Resolution Theorem states that if S is any finite set of clauses, then S is unsatisfiable if and only if $R^n(S)$ contains the empty clause for some $n \geq 0$.

Generating a resolvent can involve finding a substitution which unifies more than two literals, as explained above. In our system, this general procedure is carried out as two processes: binary resolution and factoring. Binary resolution unifies two literals (one of them negated) from the parent clauses and generates a (binary) resolvent. Factoring unifies matching literals within a single clause and generates a factor. These combined are equivalent to the more general procedure.

For example, consider the two clauses C and D where $C = \{P(x,a), P(x,y), Q(x)\}$ and $D = \{\neg P(w,a), \neg R(w)\}$. Assume that a is a constant and x, y , and w are variables. By substituting w for occurrences of x and a for occurrences of y , these clauses become $\{P(w,a), P(w,a), Q(w)\}$ and $\{\neg P(w,a), \neg R(w)\}$ respectively. $P(x,a)$, $P(x,y)$ and $\neg P(w,a)$ have been unified and the resolvent of C and D is the clause $\{Q(w), \neg R(w)\}$.

Alternately, clause C may be factored by unifying $P(x,a)$ and $P(x,y)$ with the substitution of a for y . The factor of C is the clause $\{P(x,a), Q(x)\}$ which may be resolved with D (by substituting w for x) to obtain the binary resolvent $\{Q(w), \neg R(w)\}$.

3.4.1.2 Search Strategies*

Given a finite set of clauses as input, a direct implementation of the Resolution Theorem would consist of computing the sequence S, $R(S)$, $R^2(S)$,... until encountering some $R^n(S)$ which contains the empty clause or which does not contain the empty clause but equals $R^{n+1}(S)$. In the first case the refutation has been obtained. In the second case, S is determined to be satisfiable. For some inputs however, by Church or Turing's** results, the procedure will not terminate.

The level or depth of the refutation is n . The direct implementation, by generating $R(S)$, $R^2(S)$, etc. corresponds to a breadth first search for a refutation, or a level saturation method. However, many clauses may be generated that are irrelevant or redundant. Also, it may be possible to find a refutation more quickly if strategies are adopted for the selection of which clauses to attempt to resolve.

In order for resolution theorem proving to be more practical and efficient, several refinements have been suggested which speed up the search for a refutation. These include simplification strategies and choice strategies, where the former attempt to delete as many resolvents as possible without excluding the chance of finding a proof, and the

* General references for this discussion include [6], [24], [29], [31].

** In 1936, Church and Turing independently proved that the predicate calculus is undecidable, i.e., that there is no general decision procedure to determine whether an arbitrary formula of the predicate calculus is valid. However, it can be said that the predicate calculus is semidecidable since procedures do exist (resolution for example) which can determine whether a valid formula is indeed valid.

latter help determine an ordering for selecting clauses to resolve or a criterion for not attempting to resolve certain clauses.

3.4.1.2.1 Simplification strategies

The following strategies are implemented in the theorem proving program:

- any clause containing a tautology is usually* deleted. (Tautologies are always true so deletion of a tautology from an unsatisfiable set of clauses does not affect the unsatisfiability of the remaining set of clauses and it also implicitly eliminates all the irrelevant clauses which could have been generated from the tautologies.)
- any identical clause or alphabetic variant of a clause already retained is deleted. (These are particular cases of elimination by subsumption.)**
- any literal which already appears in a clause is deleted.
- clauses whose preference priority heuristic exceeds a given value are deleted. The preference priority represents an estimate of the complexity of the clause and is a function of the number of literals, the length of the literals, and a preference constant. This strategy represents an estimate of what the probable simplest proof will look like, and it deletes

* Complementary pairs within a clause must be identical including the index numbers in order for the clause to be deleted.

** A clause C subsumes another clause D if an instantiation of C is contained in D . The subsumed clause may be eliminated.

clauses which look too complex.

3.4.1.2.2 Choice strategies

Choice strategies are employed to guide the selection of clauses to be resolved. Some strategies order the search and others determine a criterion for not attempting to resolve certain clauses.

- Ordering strategies--Ordering strategies offer an alternative to the level-saturation method. The intent is to try to 'look ahead' for a contradiction at the nth level before completing all the lower levels. The unit preference strategy [49] is based on the philosophy that, since the refutation procedure seeks an empty clause it seems reasonable to try to resolve unit clauses first. Therefore unit clauses are examined as far as a specified level. Resolvents are also checked against unit clauses. This ordering is extended to consider units versus doubletons next, etc.

  In our system, a modified unit preference strategy is used. It is based on the preference priority assigned to each clause.

- Refinement strategies--Refinement strategies deal with establishing certain criteria to be used when selecting clauses to be resolved. These criteria eliminate some clauses from consideration so that fewer resolutions need be performed at each level, and the overall search will typically result in narrower deeper searches.

  Two refinement strategies are implemented in our system.

These are the set-of-support strategy, where the clauses resulting from the negation of the statement in the proof step form the initial set-of-support; and lock resolution. (These strategies are not combined.) Each strategy is briefly defined below.

Semantic resolution techniques deal with dividing clauses into two groups and then not allowing clauses within the same group to be resolved with each other. A special case of semantic resolution is the set-of-support strategy proposed by Wos, Robinson, and Carson [50]. This strategy divides the set of clauses into two subsets based on satisfiability. For a set of clauses S, a subset M is called a set-of-support if $S - M$ is satisfiable. Then a set-of-support resolution of two clauses is one such that both clauses are not from $S - M$, and a set-of-support deduction is a deduction in which every resolution is a set-of-support resolution. Each clause produced is also in the set-of-support.

If we interpret S as the set of clauses representing $A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge \neg T$ where A_i are axioms and T is a theorem, then we should expect the axioms to be consistent and choose M to be the set of clauses originating from the negation of the theorem. Therefore, we are avoiding resolutions between clauses representing the axioms, for we should expect this to be pointless (i.e., not lead to a contradiction).

The set-of-support strategy is complete, meaning that if S is unsatisfiable and M is a subset such that $S - M$ is satisfiable,

then there is a set-of-support deduction of the empty clause from S with M as a set-of-support. (For a proof, see [6].)

Another refinement strategy, called lock resolution, was introduced by R. S. Boyer in 1971 [5]. Lock resolution uses an arbitrary assignment of indices to literals in clauses to order the literals. Then resolution is only permitted on literals of the lowest index in each clause. Literals in resolvents retain the lowest indices (if more than one is possible) of their parent clauses and if there is more than one occurrence of the same literal in a clause, then only the one with the lowest index is retained.

A lock resolvent of a pair of clauses is the clause obtained by resolving literals of the lowest index from each clause. A lock deduction from a set of clauses S is a deduction in which every clause in the deduction is either a clause in S or a lock resolvent. In many cases, a lock deduction of the empty clause is much shorter than a level-saturation deduction. Lock resolution is complete (see [6] for a proof) but lock resolution is not compatible (i.e., not complete when combined) with most other resolution strategies such as the set-of-support or some deletion strategies.

3.4.1.3 General Operating Procedures

The following parameters can be specified for use with the theorem prover:

- a preference constant--used in calculating the preference priority (default = 7)
- a resolution count--which limits the number of clauses

generated to this number (default = 200)

- a strategy choice--to select set-of-support or lock resolution (default = lock resolution)
- a print option--to control the amount of printed output for checking

The theorem prover returns one of the following indications to the translator:

- proof step is valid--meaning a contradiction was found
- proof step is invalid--meaning no contradiction was found and all resolutions were attempted
- validity cannot be determined--meaning the resolution count was exceeded or space restrictions were exceeded or generated clauses were thrown away and the search was otherwise completed before a contradiction was found

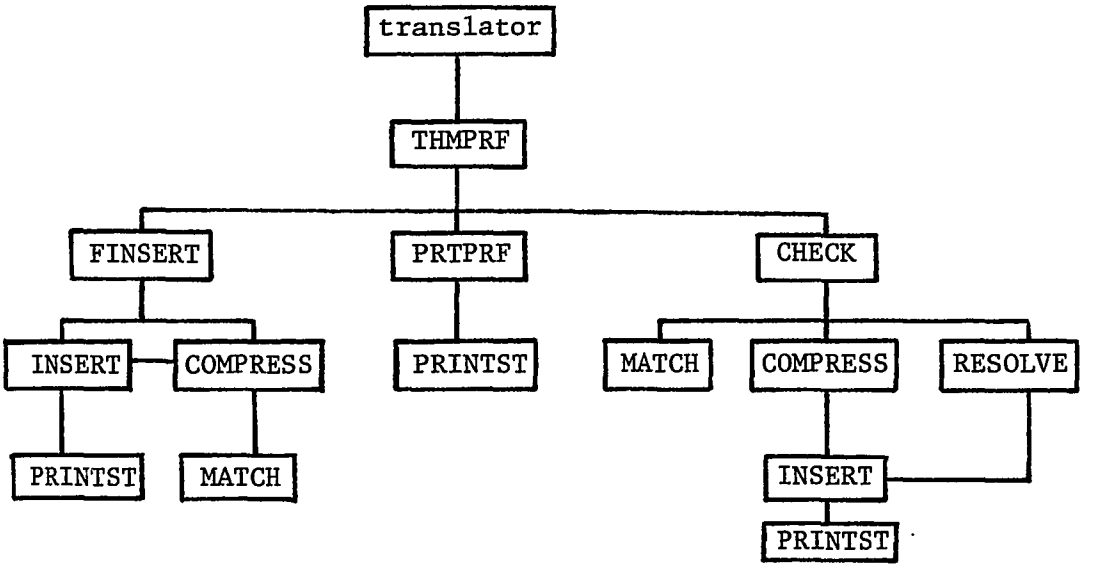
The major activities performed by the theorem prover and the modules responsible for these activities are:

1. Selection of clauses to attempt to resolve (THMPRF)
2. Selection of literals to attempt to unify (CHECK)
3. Unification and clause generation (MATCH, RESOLVE, COMPRESS)
4. Clause simplification and maintenance (FINSERT, INSERT)
5. Proof reconstruction and printing (PRTPRF, PRINTST)

THMPRF oversees the search for a proof and determines the next two clauses to attempt to resolve; CHECK looks for candidates for unification; MATCH carries out the unification algorithm; RESOLVE

generates binary resolvents; COMPRESS generates factors if matching literals in a clause can be unified; FINSERT preprocesses the initial set of clauses; INSERT simplifies and maintains clauses in a work area; PRTPRF and PRINTST generate printed output.

These modules are related as follows:



3.4.2 Selecting Clauses (THMPRF)

THMPRF is the controlling procedure of the theorem proving subsystem, guiding the overall search for a proof by selecting the clauses to attempt to resolve. THMPRF first calls FINSERT to insert the original set of clauses into a work area. The work area is used throughout the procedure to contain the original clauses and the generated clauses.

Clauses are selected for resolution based on the preference priority, with a pair of clauses having the smallest sum of preference priorities selected first. This implements a variant of the unit

preference scheme. In addition simple unit clauses are selected before more complex (longer) unit clauses. Also, after a successful resolution, the next two clauses are selected to minimize the sum of the preference priorities taking into account newly generated clauses as well. This means we may search very deeply down one part of the search tree before attempting to resolve longer clauses at a lower level. The philosophy is to try to 'look ahead' to find a proof at a deeper level before generating all clauses at a given level.

If the set-of-support strategy is being used, no pair of clauses which are not in the set-of-support will be selected, and the set-of-support includes clauses generated from clauses of which at least one is in the set-of-support. If the set-of-support is not being used then unit preference alone is employed.

THMPRF calls CHECK when it has selected two clauses for examination. When the entire procedure has concluded, THMPRF optionally calls PRTPRF to print information on the result of the processing including the deduction of the empty clause if one was found.

3.4.3 Selecting Literals (CHECK)

CHECK examines the predicates in the pair of clauses selected by THMPRF , seeking matching predicates of opposite sign. Since the predicates within clauses are sorted, the task is simplified. Two strategies are available. For the set-of-support strategy a merge-type search through the clauses is made. For lock resolution, the first two predicates of the two clauses are checked for a complementary match, and then (if these were found) subsequent predicates are checked for a

match, since all predicates of the same arbitrary index must be considered.

In either case, for each pair MATCH is called to attempt unification, and then RESOLVE if a most general unifier was found. If RESOLVE generates the empty clause, control returns to THMPRF . Otherwise COMPRESS is called to attempt factoring, and then the search continues for more matching predicates of opposite sign. CHECK returns the minimum number of predicates in a clause generated during execution, or an indication that no clauses were generated.

3.4.4 Unification and Clause Generation (MATCH, RESOLVE, COMPRESS)

MATCH seeks to unify the two literals passed to it (by CHECK or by COMPRESS). The procedure finds the disagreement set by examining parallel entries in the two literals. If the entries are different and neither is a variable, unification fails. If exactly one entry is a variable then, as long as the variable does not already appear in the expression being substituted or in substitutions set up so far, the substitution is acceptable. Otherwise unification fails. If both entries being examined are variables, one is selected for the substitution. A substitution table is set up during MATCH which holds the composition of the substitutions at any time. If unification is successful, it contains the most general unifier of the two literals. Unification fails however if the end of one literal is reached before the other.

RESOLVE actually applies the substitution that MATCH has set up in the substitution table and forms the resolvent of the two clauses.

As the new clause is being formed, RESOLVE orders the literals according to predicate indices (see FINSERT). RESOLVE generates the resolvent in a temporary area and then calls INSERT to enter the clause in the work area.

COMPRESS attempts to unify any matching literals found in the clause being considered and if unification is possible, the factor is entered in the work area. COMPRESS is called by CHECK and calls MATCH to determine the most general unifier of all matching literals. Then COMPRESS unifies the literals and calls INSERT to enter the factor in the work area. This factoring procedure, along with the generation of binary resolvents, is equivalent to more general resolution procedures.

3.4.5 Clause Simplification and Maintenance (FINSERT and INSERT)

FINSERT is called by THMPRF in order to prepare the initial set of clauses for insertion into the work area. FINSERT calls INSERT to insert each clause and COMPRESS to attempt to factor each clause.

Indices used for ordering predicates are computed by FINSERT . This is done as follows. Each predicate is initially (i.e., when passed from the translator) represented by a pointer to a table. The pointer is negative if the predicate is negated. The pointer is multiplied by 200 and augmented with a unique constant to form the index for lock resolution. The unique constant is a positive integer for non-negated predicates and a negative integer greater than -100 for negated predicates. The absolute value of this index is used for

sorting predicates within clauses, and subsequently to establish the literal which may be used in lock resolution.

FINSERT also marks those original clauses which form the set-of-support if this strategy is being used.

INSERT is called by FINSERT, COMPRESS , and RESOLVE to sort a clause, simplify it, and insert it into the work area if it should be kept.

If the clause is empty, INSERT just records this fact and returns an appropriate indicator that the empty clause was successfully deduced. Otherwise, INSERT sorts the predicates within the clause using the insertion method. Predicates are sorted by the absolute value of the index, causing, for the same predicate, non-negated appearances to precede negated ones. If both indices are the same, the sort uses the number of the first item that differs within the predicates. If two entire predicates are identical, one of them is deleted (this is a special case of subsumption) and if a tautology is encountered (i.e., a complementary identical pair) the entire clause is eliminated.

The preference priority for the clause is then computed as the number of literals plus the total length (number of literals, functions, variables, and constants) divided by a preference constant set by the user or assumed at value 7. If the preference priority exceeds the complexity heuristic of 20, the clause is rejected. If an identical clause is already in the work area, the clause is rejected. Otherwise, the clause is inserted into the work area into a forward linked list by preference priority. The parent clause pointers are also kept, if applicable.

If the clause is successfully inserted, the resolution count is adjusted by one and if the count exceeds the resolution count parameter or if the work area is now full, then INSERT indicates this condition and the entire procedure will subsequently terminate without being able to determine the validity of the proof step.

3.4.6 Proof Reconstruction and Printing (PRTPRF and PRINTST)

PRTPRF is optionally called by THMPRF to print summary information on processing including the proof if indeed a deduction of the empty clause was found. PRTPRF traces the proof and calls PRINTST to print each clause.

PRINTST prints the clause and its origin (i.e., by hypothesis or by inference from another clause or by inference from a pair of clauses). It is called by PRTPRF and optionally by INSERT .

CHAPTER 4

CONCLUSIONS

In this chapter, we conclude our report by presenting an evaluation of the system, some suggestions for future enhancements, and a summary of our accomplishments.

4.1 EVALUATION

The system has been operational and in use for several months. It is being used by us and a graduate student* in mathematics. We will evaluate our work in two parts. First, we consider the use of the language itself, and then we discuss the use of the system including our experiences in axiom selection and proof verification.

4.1.1 The Language

4.1.1.1 Versatility

The language has been shown to be useful for several methods of proof-writing. The examples given in Chapter 2 (Figures 1 through 8) illustrate direct proofs, indirect proofs, existence proofs, uniqueness proofs, proof by induction, and proof by cases. Examples are included from elementary group theory, number theory, and set theory. The language does not cater to any particular discipline since the elements

* Bill Abbley, Department of Mathematics, University of Hawaii, has been using the system to investigate the axioms and the theorems of group theory leading to the establishment of LaGrange's Theorem. He has been following the development presented in Topics in Algebra, by I. N. Herstein [18].

of the language are treated abstractly.

The versatility of the language is further illustrated in Figure 9 which is a variation of a word problem found in Nilsson [31]. Here the individuals in the domain are people and the predicates are personal attributes. The constructive existence proof determines that there is a member of the club (specifically Mike) who is a climber, but not a skier.

Figure 10 illustrates another use of the language. Here we prove a theorem directly, rather than check the steps of a proof. The system may be used this way simply by providing a single proof step which states that the theorem follows from the axioms. This particular theorem is one frequently submitted to theorem proving programs that are reported in the literature (e.g., [24]). It states that in an associative system having left and right solutions, there is a right identity element. The predicate P(x,y,z) means the product of x and y equals z .

4.1.1.2 Ease of Use

Our graduate student reported that the language of our system was "easy to learn" and "easy to use" because of its similarity to the language used by mathematicians. In his proof-writing he tended to use parentheses instead of depending on the hierarchy of operators. Also he was inclined to use AND and OR for the logical connectives, rather than the symbolic alternatives of & and | , which he would have preferred to have available for function or predicate definition.

The user, we recall, actually defines much of the language by

```
**********  LIST OF STATEMENTS  **********

    NAME          STATUS                  STATEMENT
    ----          ------                  ---------

  PUZZLE          THEOREM VERIFIED        THERE EXISTS SOMEONE SUCH THAT (SOMEONE IS_A_MEMBER AND SOMEONE
                                          IS_A_CLIMBER AND NOT  SOMEONE IS_A_SKIER)

              PROOF
              -----

    1.      NOT MIKE LIKES_RAIN AND NOT MIKE LIKES_SNOW  BY TONY_VS_MIKE,TONY_LIKES
    2.      THEN MIKE IS_A_CLIMBER AND NOT MIKE IS_A_SKIER  BY #1,RAIN,SNOW,CLUB,   THE_ROSTER
    3.      THEOREM FOLLOWS    BY #2,THE_ROSTER
    4.      QED
**********  LIST OF STATEMENTS  **********

    NAME          STATUS                  STATEMENT
    ----          ------                  ---------

  CLUB            AXIOM                   FOR ALL X (X IS_A_MEMBER IMPLIES X IS_A_SKIER OR X IS_A_CLIMBER)
  PUZZLE          THEOREM VERIFIED        THERE EXISTS SOMEONE SUCH THAT (SOMEONE IS_A_MEMBER AND SOMEONE
                                          IS_A_CLIMBER AND NOT SOMEONE IS_A_SKIER)
  RAIN            AXIOM                   FOR ALL X (X IS_A_CLIMBER IMPLIES NOT X LIKES_RAIN)
  SNOW            AXIOM                   FOR ALL X (X IS_A_SKIER IMPLIES X LIKES_SNOW)
  THE_ROSTER      AXIOM                   JOHN IS_A_MEMBER AND TONY IS_A_MEMBER AND MIKE IS_A_MEMBER
  TONY_LIKES      AXIOM                   TONY LIKES_RAIN AND TONY LIKES_SNOW
  TONY_VS_MIKE    AXIOM                   (TONY LIKES_RAIN IFF NOT MIKE LIKES_RAIN) AND    (TONY LIKES_SNOW
                                          IFF NOT MIKE LIKES_SNOW)
*********  LIST OF DEFINITIONS  **********

SYMBOL          TYPE          # ARGUMENTS
------          ----          ----------

IS_A_CLIMBER  PRED-POSFIX          1
IS_A_MEMBER   PRED-POSFIX          1
IS_A_SKIER    PRED-POSFIX          1
JOHN          FUNC-CONSTANT        0
LIKES_RAIN    PRED-POSFIX          1
LIKES_SNOW    PRED-POSFIX          1
MIKE          FUNC-CONSTANT        0
TONY          FUNC-CONSTANT        0
```

FIGURE 9. WORD PROBLEM

```
LIST RIGHTID

**********  LIST OF STATEMENTS  **********

    NAME            STATUS                  STATEMENT
    ----            ------                  ---------

  RIGHTID           THEOREM VERIFIED        THERE EXISTS Y SUCH THAT FOR ALL X P(X,Y,X)


                PROOF
                -----

    1.      THEOREM FOLLOWS BY LEFTSOLT,RIGHTSOLT,ASSOC
    2.      QED

LIST STA


**********  LIST OF STATEMENTS  **********

    NAME            STATUS                  STATEMENT
    ----            ------                  ---------

  ASSOC             AXIOM                   FOR ALL X,Y,Z,U,V,W (  (P(X,Y,Z) & P(Y,U,V) & P(X,V,W) -> P(Z,U,W) )
                                            & ( P(X,Y,Z) & P( Y,U,V) &   P(Z,U,W) -> P(X,V,W))  )
  LEFTSOLT          AXIOM                   FOR ALL Y,Z EXISTS X P(X,Y,Z)
  RIGHTID           THEOREM VERIFIED        THERE EXISTS Y SUCH THAT FOR ALL X P(X,Y,X)
  RIGHTSOLT         AXIOM                   FOR ALL X,Z EXISTS Y P(X,Y,Z)

LIST DEF


*********  LIST OF DEFINITIONS  **********

SYMBOL              TYPE          # ARGUMENTS
------              ----          -----------

P  1            PRED-LIST             3
```

FIGURE 10. THEOREM PROVING

denoting the predicate and functions to be used and by choosing from a variety of methods for their use (i.e., postfix, prefix, infix, list-type, and four precedence levels may be denoted for infix functions). Therefore readability of the language is to a great extent determined by the user. However, it does require some analysis on the user's part to find an appropriate symbolic representation for some entities. Consider a sequence $a_1, a_2, \ldots, a_n$ for example. In our language it cannot be represented in the usual way. However, it could be represented as a single object such as SEQUENCE(A,N) or, the ith object in a sequence might be denoted NTH(S,I) where S is a sequence.

Also the user must transform informal statements to their logical equivalency by employing only the particular logical operators and quantifiers provided in the input language. Thus, for example, an informal mathematical statement such as "the sum of any two numbers is a number" must be represented in our language by some statement such as "for all x, y there exists z such that x + y = z ."

Two statement forms that our graduate student wanted to use were the following (reading E as "is an element of")

```
        ASSUME X E S ARBITRARY
and     LET X E S SATISFY ...
```

These statements are syntactically incorrect in our language and must be restated as

```
        ASSUME X SATISFIES X E S
```

and LET X SATISFY X E S AND ...

4.1.1.3 Error Checking, Diagnostics, Recovery

An abundance of error checking procedures are carried out by the language processor. Diagnostics include:

ILLEGAL CHARACTER _____

UNDEFINED WORD _____

WRONG NUMBER OF ARGUMENTS FOR _____ DEFINED TO HAVE _____
ARGUMENTS BUT USED WITH _____

SYMBOL _____ WAS NEITHER QUANTIFIED NOR DECLARED

INVALID USE OF _____

UNDEFINED STATEMENT _____ GIVEN AS REASON

PROOF DID NOT PROVE THEOREM

PROOF STRUCTURE INVALID--LAST STEP NOT QED

STEP _____ HAS SYNTAX ERRORS

STEP _____ IS NOT ACCESSIBLE

STEP _____ CANNOT BE FOUND

The last three refer to step numbers given as reasons.

Any proof step containing syntax errors detected during verification (structural errors) is effectively ignored by the parser (the stacks are backed up to the end of the previous proof step) to enable processing to continue through the proof. However, verification will not be attempted initially if syntax errors were found in the theorem or any step of the proof during the initial syntax scan.

These diagnostics were found to be useful and adequate in

determining the cause of syntax errors.

4.1.2 Use of the System

4.1.2.1 Selection of Axioms

We have found that one of the most difficult tasks in using this system effectively is making a judicious choice of axioms.

First, all necessary axioms must be included. The axioms for equality in particular must be present including substitution axioms for all predicates and functions being used. This situation could be cumbersome and annoying for a proof-writer since equality is one of the easiest (hence implicitly assumed) relations for humans to understand.

Another example where an easily overlooked axiom was excluded was the following. Our graduate student was trying to prove that if K is a subgroup, then the identity element is an element of K . To do this, he picked an arbitrary element C of K (i.e., ASSUME C SATISFIES C E K). Then he showed that $C.C^{-1}$ is equal to the identity, $C.C^{-1}$ is an element of K and therefore the identity element is in K . The only valid conclusion from this argument is "If C E K then ID E K ," but he wanted to conclude "ID E K ." This problem must be handled by introducing the necessary (but again easily assumed) axiom stating that a subgroup is nonempty. Then the element C could be chosen as a particular element of the subgroup by means of a LET statement (i.e., LET C SATISFY C E K BY NONEMPTY). The argument can then be validated.

Another consideration in axiom selection is the formulation of axioms in such a way that they are useful in several proofs. That is, an axiom may be stated so that it is in a convenient form for proving one theorem, but it should be general enough to use in subsequent proofs as well. Our graduate student found that this was a time consuming task.

Also, the axioms and theorems must be developed carefully and consistently when dealing with a many-type domain. In this case a predicate for each element mentioned should be included in order to specify its type. This ensures the validity of the assertions proved about an entity. Thus, for example, care should be taken to specify whether a variable represents a group, a subgroup, an element of a group, an element of a subgroup, etc. Note that if these procedures are not followed, an invalid theorem may become verified since in the mechanical procedure, variables are freely substitutable.

4.1.2.2 Proof Verification

In our early experiences, we found that several times a proof step certainly looked logical but that the system could not validate it. Upon further examination we found several reasons for this.

1. The step actually was invalid because
   a. the axioms or reasons were not stated properly
   b. a necessary axiom or reason had been excluded
2. The step actually was valid but it was "too large" for the theorem prover to handle (within search bounds imposed) and
   a. too many axioms were given (i.e., extraneous reasons were

included)

b. a different internal proof finding strategy could validate the step.

The first two cases were discussed above. The second and third cases both frequently involved equality, first by leaving out a required axiom of equality, and then overcompensating by including unneeded equality axioms. For example, consider the following:

10. A + (1 + Y) = B + 1 ...

11. A + (Y + 1) = B + 1 BY #10, COMMUTE, EQPLUS, EQUALITY

12. (A + Y) + 1 = B + 1 BY #11, ASSOC, EQPLUS, EQUALITY

The axioms referenced are:

ASSOC: FOR ALL X,Y,Z (X + (Y + Z)) = ((X + Y) + Z)

COMMUTE: FOR ALL X,Y,Z (X + (Y + Z)) = (X + (Z + Y))

EQPLUS: FOR ALL X,Y,Z ((X = Y ↔ X + Z = Y + Z) AND
(X = Y ↔ Z + X = Z + Y))

EQUALITY: FOR ALL X,Y,Z ((X = X) AND (X = Y → Y = X) AND
(X = Y AND Y = Z → X = Z))

The validity of step 12 could not be determined. An unnecessary axiom (EQPLUS) had been included and its removal enabled the step to be verified. Note that this axiom was also superfluous in step 11, but this did not affect verification of that step.

The following illustrates another type of problem which was solved by restructuring proof steps.

10. B * A = B * (ID * C) ...

11. = (B * ID) * C ...

12. = B * C ...

13. = ID ...

14. B * A = ID BY #10 - #13, EQUAL

The axiom referenced is

FOR ALL A,B,C (A = A AND ((A = B AND B = C) → A = C)
AND ((A = B) ↔ (B = A)))

The validity of step 14 could not be determined. When step 12.5 was included, giving shorter step spans, both steps 12.5 and 14, as illustrated below, were accepted.

12.5 B * A = B * C BY #10 - #12, EQUAL

13. = ID ...

14. B * A = ID BY #12.5, #13, EQUAL

Note however that in Figure 1 (Chapter 2), step 11 used seven steps as reasons and the proof was verified. However, only the transitivity property of equality was given in that proof step.

There were also times when a proof step could be validated using one of the internal theorem proving strategies available, but not with another. We are analyzing this situation to try to find the best combination of strategies and heuristics to use.

These experiences emphasize the fact that great care must be taken when setting up axioms initially and in choosing the axioms to

be used in proof steps. Also, the internal strategies used must be carefully evaluated. We expect continuing use of the system to help determine some useful guidelines.

4.2 FUTURE WORK

This system may be considered the first version of an ongoing effort and much work remains in order to develop a truly practical proof-checking system. Evaluation through continuing use should indicate directions of improvement. At this time, we envision several areas for future work. These include building more "knowledge" into the system, providing additional feedback to users, improving interactive facilities, extending the language, investigating strategies for the theorem prover, developing and carrying out utilization plans.

4.2.1 Building More Knowledge into the System

4.2.1.1 Equality

With the current system, all equality axioms must be provided and referenced explicitly by the user. This has been found to be bothersome to the proof-writer. A solution is to extend the system's capabilities regarding the treatment of equality. Extensions could be made to the language processor or to the theorem prover.

The language processor, for example, could provide automatic inclusion of the axioms for equality, including for every predicate and function introduced by the user, the generation of an appropriate substitution axiom. These axioms could be given predetermined names which the user would state as reasons in appropriate proof steps or alternately, the language processor could automatically include the appropriate axioms as determined during the scan of a proof step.

Another approach is to modify the theorem prover to handle

equality automatically by basing its logic on the first-order predicate calculus with equality. Several methods have been suggested for accomplishing this, most notably the paramodulation principle of Wos and Robinson [51].

Either method should improve the system from the user's point of view. Internally, paramodulation looks attractive since Wos and Robinson have shown that it can handle equality effectively. However, it still exhibits "much purposeless behavior" ([12] p. 128). Allen and Luckham [3] found that formulation of a problem with equality was sometimes, but not always, easier for their program to deal with than pure first-order formulation, and some of their initial results showed that equality formulation proofs were shorter, but took longer to generate.

4.2.1.2 Knowledge Relative to a Particular Discipline

As the system becomes used for selected theories, it might be appropriate to provide a way for the user to specify certain axioms to be used implicitly under certain circumstances, thus incorporating a certain core of knowledge into the axiom set. This would enable a user to list proof steps with no specific reasons or to specify the reason as OBVIOUS . Then the core axioms would automatically be provided with that proof step when verification is attempted.

4.2.1.3 Arithmetic

The current system does not 'know' that 1 + 1 = 2 (although this may be specified as an axiom). Some limited arithmetic capability would be useful.

4.2.2 Providing Additional Feedback to User

It would be desirable to give the user additional proof analysis. For example, extraneous reasons could be noted. Also some feedback on axioms could be provided, possibly testing for contradictions in axioms or performing independence tests.

4.2.3 Improving Interactive Facilities

The system will presently be converted from batch to interactive mode. Additional editing and running features will accompany this conversion, such as a feature for renumbering proof steps and a facility for checking isolated steps of a proof.

4.2.4 Extensions to the Language

One modification to the language which would be convenient would be to allow an assumption of a formula to be made without including a list of variables. IF ... THEN might be introduced as another alternative for IMPLIES and $\rightarrow$. The symbols & and | could be deleted as logical operators (representing conjunction and disjunction) and included as characters allowable for user defined symbols instead. A uniqueness quantifier might be introduced as well.

It would also be useful to provide a means of including comments in the language.

4.2.5 Investigate Strategies of Theorem Proving

It is difficult at best to determine which strategy should be used for a given mathematical discipline or any given problem. The current

system has already pointed this out. Some guidelines have been given in the literature ([3], [23], [26]). We will continue experimentation by implementing and analyzing some additional strategies in the theorem prover (besides possibly paramodulation as discussed above). Furthermore, our criterion for the "best" strategy is not the same as criteria for other applications because we want to be able to verify a step if it is reasonable that an audience of mathematicians would.

4.2.6 Utilization Plans

Lastly, much work needs to be done in order to develop the axioms and exercises for students who will use the system. Axiom choice and problem formulation can be crucial factors in the effectiveness of the system and a self-paced course in proof-writing must be planned with great care. Beyond that we must evaluate the effectiveness of this system and this approach in a learning environment.

4.3 SUMMARY

The purpose of the research reported here was to develop a method whereby informal proofs could be verified by a computer. This investigation led us from informal mathematics to formal logic via formal linguistics. Several languages were analyzed: the statements of informal mathematics, our input language, formulas of the predicate calculus, clause form. We tried to make a reasonable subset of the language of informal mathematics equivalent to clause form. Several deductive systems were considered as well: informal mathematical reasoning, natural deduction, first-order logic, the resolution principle. We attempted to use the resolution principle to simulate informal mathematical reasoning.

Evaluation based on current usage has shown that the system is basically sound. It accomplishes the primary tasks we set out to investigate. Thus, in conclusion, the efforts described here have resulted in a working system which appears to be versatile and flexible and usable within the limited areas of our experience. Our original goal of a system usable at least by mathematics students still appears feasible, with continued development.

REFERENCES

[1] Abrahams, Paul. Machine Verification of Mathematical Proof. Sc.D. Dissertation, Massachusetts Institute of Technology, 1963.

[2] Aho, A. V. and J. D. Ullman. The Theory of Parsing, Translating, and Compiling: Vol. I Parsing; Vol. II Compiling. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

[3] Allen, John and David Luckham. "An Interactive Theorem-Proving Program." Machine Intelligence, Vol. 5 (Ed. B. Meltzer and D. Michie, New York: American Elsevier, 1970, pp. 321-336.

[4] Bledsoe, W. W. and Peter Bruell. "A Man-Machine Theorem-Proving System." Artificial Intelligence 5, 1974, pp. 51-72.

[5] Boyer, R. S. Locking: A Restriction of Resolution. Ph.D. Thesis, University of Texas at Austin, 1971.

[6] Chang, Chin-Liang and Richard Char-Tung Lee. Symbolic Logic and Mechanical Theorem Proving. New York: Academic Press, 1973.

[7] Chester, Daniel. "The Translation of Formal Proofs into English." Artificial Intelligence 7, 1976, pp. 261-278.

[8] Church, Alonzo. Introduction to Mathematical Logic, Vol. 1. Princeton University Press, Princeton, 1956.

[9] Copi, Irving M. Symbolic Logic. 2nd ed., New York: Macmillan, 1965.

[10] Darlington, J. L. "A Partial Mechanization of Second-Order Logic." Machine Intelligence, Vol. 6 (Ed. B. Meltzer and D. Michie, New York: American Elsevier, 1971, pp. 91-100.

[11] DeRemer, F. L. "Simple LR(k) Grammars." Communications of the ACM, Vol. 14, No. 7, July 1971, pp. 453-460.

[12] Elspas, Bernard, Karl N. Levitt, Richard J. Waldinger and Abraham Waksman. "An Assessment of Techniques for Proving Program Correctness." Computing Surveys, Vol. 4, No. 2, June 1972, pp. 97-147.

[13] Falk, Arthur E. and Richard Houchard. "Computerized Help in Finding Logic Proofs." Proceedings of the 1972 Conference on Computers in Undergraduate Education.

[14] Gelder, H. M. and J. M. Kraatz. "PROOF." Computer-Based Education Research Laboratory, Urbana, Illinois, August 24, 1973.

[15] Gelder, H. M. and J. M. Kraatz. "PROOF Progress Report." Computer-Based Education Research Laboratory, Urbana, Illinois, September 16, 1974.

[16] Gries, David. Compiler Construction for Digital Computers. New York: John Wiley & Sons, Inc., 1971.

[17] Guard, J. R., F. C. Oglesby, J. H. Bennett and L. G. Settle, "Semi-Automated Mathematics." Journal of the ACM, Vol. 16, No. 1, January 1969, pp. 49-62.

[18] Herstein, I. N. Topics in Algebra. Blaisdell Publishing Co., 1964.

[19] Hopcroft, John E. and Jeffrey D. Ullman. Formal Languages and Their Relation to Automata. Reading, Mass.: Addison-Wesley Publishing Co., 1969.

[20] Ibrahim, Rosalind. LR(0) CFSM Generation and Resolution of Inadequancies via Lane Tracing. Program Documentation, University of Hawaii, February 1973.

[21] Kelanic, Thomas J. "Geometric Proofs." Creative Computing, March 1976, pp. 60-61.

[22] Kershner, R. B. and L. R. Wilcox. The Anatomy of Mathematics. New York: The Ronald Press Company, 1950.

[23] Lawrence, J. Dennis and J. Denbigh Starkey. "Experimental Tests of Resolution-Based Theorem-Proving Strategies." Information Sciences 10, 1976, pp. 131-154.

[24] Luckham, David. "Some Tree-Parsing Strategies for Theorem Proving." Machine Intelligence, Vol. 3 (Ed. D. Michie), New York: American Elsevier, pp. 95-112.

[25] Margaris, Angelo. First Order Mathematical Logic. Waltham, Mass.: Blaisdell Publishing Co., 1967.

[26] Marinov, Vesko. "Computer Understanding of Mathematical Proofs." Institute for Mathematical Studies in the Social Sciences, Stanford University (undated).

[27] McCarthy, John. "Computer Programs for Checking Mathematical Proofs." Proceedings AMS Symposium on Recursive Function Theory, New York, 1961, pp. 219-227.

[28] McKeeman, W. M., J. J. Horning and D. B. Wortman. A Compiler Generator. Englewood Cliffs, N.J.: Prentice-Hall, 1970.

[29] Meltzer, Bernard. "The Programming of Deduction and Induction." Artificial and Human Thinking (Ed. Alick Elithorn and David Jones), Amsterdam: Elsevier Scientific Publishing Co., 1973, pp. 19-33.

[30] Nidditch, P. H. Introductory Formal Logic of Mathematics. Illinois: The Free Press of Glencoe, 1957.

[31] Nilsson, Nils J. Problem Solving Methods in Artificial Intelligence. New York: McGraw Hill Book Co., 1971.

[32] Pager, D. "A Solution to an Open Problem by Knuth." Information and Control, Vol. 17, No. 5, December 1970, pp. 464-473.

[33] Pager, D. Efficient Programming Techniques for LR(k) Parsing. University of Hawaii, Information Sciences Program, Tech. Report No. PE236, January 1972.

[34] Pager, D. On the Incremental Approach to Left-to-Right Parsing. University of Hawaii, Information Sciences Program, Tech. Report No. PE238, January 1972.

[35] Peterson, W. W. A Resolution-Based Theorem Proving Program. 1975.

[36] Robinson, J. A. "A Machine-Oriented Logic Based on the Resolution Principle." Journal of the ACM, Vol. 12, No. 1, January 1965, pp. 23-41.

[37] Robinson, J. A. "The Generalized Resolution Principle." Machine Intelligence, Vol. 3 (Ed. D. Michie), New York: American Elsevier, pp. 77-93.

[38] Robinson, J. A. "Mechanizing Higher Order Logic." Machine Intelligence, Vol. 4, (Ed. B. Meltzer and D. Michie), New York: American Elsevier, 1969, pp. 151-170.

[39] Robinson, J. A. "An Overview of Mechanical Theorem Proving." Theoretical Approaches to Non-Numeric Problem Solving (Ed. R. B. Banerji and M. D. Mesarovic), New York: Springer-Verlag, 1970, pp. 2-20.

[40] Russell, B. and A. Whitehead. Principia Mathematica, Vol. I. 2nd ed., New York: Chelsea, 1938.

[41] Sammet, Jean E. Programming Languages: History and Fundamentals. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1969.

[42] Suppes, P. Introduction to Logic. Princeton: Van Nostrand, 1957.

[43] Suppes, P. Axiomatic Set Theory. New York: Dover, 1972.

[44] Wang, Hao. "Toward Mechanical Mathematics." IBM Journal of Research and Development 4, 1960, pp. 2-22.

[45] Wang, Hao. "Mechanical Mathematics and Inferential Analysis." Computer Programming and Formal Systems (Ed. P. Braffort and D. Hirschberg), Amsterdam: North Holland Publishing Co., 1963, pp. 152-160.

[46] Wang, Hao. Logic, Computers and Sets. New York: Chelsea Publishing Co., 1970.

[47] Wang, Hao. "Remarks on Mathematics and Computers." Theoretical Approaches to Non-Numeric Problem Solving (Ed. R. B. Banerji and M. D. Mesarovic), New York: Springer-Verlag, 1970, pp. 152-160.

[48] Wegner, Peter. "Programming Language Semantics." Formal Semantics of Programming Languages (Ed. Randall Rustin), Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1972, pp. 149-248.

[49] Wos, L. D. Carson and G. A. Robinson. "The Unit Preference Strategy in Theorem Proving." Proceedings AFIPS 1964 Fall Joint Computing Conference, pp. 616-621.

[50] Wos, L. G. A. Robinson and D. F. Carson. "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving." Journal of the ACM, Vol. 12, No. 4, 1965, pp. 536-541.

[51] Wos, L. and G. A. Robinson. "Paramodulation and Set of Support." Proc. Symposium on Automatic Demonstration, New York: Springer-Verlag, 1970, pp. 276-310.

[52] Zamenek, H. "Formalization, History, Present, and Future." Programming Methodology (Ed. Clemens E. Hacki), New York: Springer-Verlag (Berlin), 1975, pp. 477-501.