TR. 269

A GENERALIZED SYSTEM FOR UNIVERSITY MATHEMATICS INSTRUCTION

by

Robert L. Smith and Lee H. Blaine

Institute for Mathematical Studies in the Social Sciences

Stanford University

1 Introduction

EXCHECK is a system for developing mathematically-based CAI courses. It is currently being used at Stanford University to teach a college-credit course in axiomatic set theory (Philosophy 161). New courses now under development include proof theory and the foundations of probability. EXCHECK includes facilities for defining the language of the theory as well as the curriculum material. Students interact with the EXCHECK proof checker, which accepts proofs presented in a style comparable to standard mathematical practice. Previous proof checkers have required that the student's proof be presented as a derivation in a primitive formal system. This paper describes the system's instructional uses. (1)

The design of this system had several goals. First, we wanted an instructional system that would provide a semantic base for our work on processing natural language and computer-generated audio. Axiomatic mathematics fits this description in that the underlying semantics is relatively well understood, but many of the interesting problems of natural language are also involved in the informal language of mathematics and the informal expression of mathematical proofs.

Second, we recognized that traditional proof checkers were inadequate for teaching mathematics, or for that matter, introductory logic. People understand mathematical concepts and arguments at a level much higher than the traditional formal systems of mathematical logic.

Finally, we designed the EXCHECK system to be an extensible system in which other curricula could be implemented with incrementally less work. This design is currently being tested and we will subsequently report on whether or not we have been successful.

2 Authoring a Mathematics Curriculum for EXCHECK

Designing an author language that can be used by subject-matter specialists to produce courseware is a classical problem of computer-assisted instruction. Many authoring systems "solve" this problem by offering the user computationally inferior programming languages in which it is difficult if not impossible to program something as complex as a proof checker. EXCHECK, which is written in two high-level programming languages originally intended for artificial-intelligence research (SAIL (2) and LISP (3)) and runs on the IMSSS PDP10/TENEX timesharing system, does not have such limitations. Several specialized processors are associated with the proof checker, and these processors are used by the curriculum authors for specifying the mathematical theories and curriculums in a machine-independent manner. A curriculum for EXCHECK is prepared and maintained by a curriculum author. Our authors have been professors and graduate students in philosophy rather than professional programmers.

2.1 The Languages for Mathematics Used by EXCHECK

The author must first characterize the language of the mathematical theory in question. Currently, the author writes a context-free grammar for the language, including macro templates that show how to build the internal representation from the parse of a given sentence of the theory. This system is described in N. Smith, 1974. Table 1 gives a selection of the formal and informal sentences of set theory that are recognized by our grammar for set theory, which has about 500 context-free rules. To facilitate use in other theories, such as probability theory, the set theory grammar is modularized.

Using a context-free language, it is easy to define fragments of mathematical languages in which the scopes of operators are well defined.

(1) We would like to thank Professor Patrick Suppes for his general direction of this project. Also we thank M. Hinckley, J. McDonald, V. Marinov, H. Graves, D. Ferris, P. Oppenheimer, and J. Prebus for their assistance. The research reported here was supported by the National Science Foundation under NSF Grant EPP 74-15016, A01.

(2) See Van Lehn, 1973 and R. Smith, 1975.

(3) See Quam, 1967 and Wolpert, 1975.

For example, the sentence

```
For all x,y
        if x is a set and y is a set then
            For all z,
                z is in x if and only if z is in y
```

is quite easily parsed by a context-free language. The more comprehensible paraphrase

```
Two sets are equal just in case
        they have the same elements
```

presents, for the current systems of natural language understanding, certain difficulties that form one of our research problems. (4)

The program associated with the definition of this language, CONSTRUCT, can then be used to test and examine the grammar interactively at a display terminal, and produce a compiled version of the grammar for inclusion in other programs of the system.

Table 1

The Language for Set Theory
as Recognized by the EXCHECK Grammar

In the following we give several formulas of set theory, labelled Formal, and equivalent English-like expressions, labelled Informal, that are recognized by the EXCHECK grammar. The expressions labelled Paraphrase are not currently recognized or generated by EXCHECK, but represent more readable versions of these sentences.

```
Formal
        (A x)(E y)(x sub y)
Informal
        For every x there is a y such that
                x is a subset of y
Paraphrase
        Everything is a subset of something
```

```
Formal
        Function(F) & F:A -> B
Informal
        F is a function and F maps A into B
Paraphrase
        Function F maps A into B
```

```
Formal
        {x : x neq x} = 0
Informal
        The set of all x such that
                x is not equal to x is empty
Paraphrase
        The set of those things not equal to
                themselves is empty
```

```
Formal
        (A A)[Dinfinite(A) IFF
            (E C)(C psub A & C := A)]
Informal
        For all A, A is Dedekind-infinite
            just in case
                there is a C such that
                    C is a proper subset of A
                    and C is equipollent to A
Paraphrase
        A set is Dedekind-infinite just in case
                it has a proper subset
                equipollent to itself.
```

(4) See Smith and Rawson, A multi-processing approach to natural language, forthcoming, for a description of our current work in handling the complex scopes of natural language operators.

2.2 Defining the Theory

After giving the language of the theory, the curriculum author can state the axioms, definitions and theorems of the mathematical theory. For example, the definition of the notion of subset:

```
For all x,y
        [x is a subset of y if and only if
            For all z, if z is in x then z is in y]
```

Another processor, the theory executive (THRYEX), then interactively checks the definitions that the author has given, compiling a binary version that can be read with random-access when the student uses the system. The current set theory system contains over 500 theorems. This is actually too many as evidenced by the fact that only about 100 theorems were used by the students who took the set theory course in the spring quarter of 1975.

2.3 Curriculum Text

The curriculum itself is prepared using the mathematics executive (MATHEX) program. This is an interactive program that accepts student exercises and text material, checking for errors and allowing on-line correction. MATHEX produces a list of exercises for the program as well as a curriculum text. The curriculum text is processed automatically by a text justification system (5) to produce a hard-copy version for the students.

The student uses the computer terminal mostly for the construction of proofs. Very little presentation of text material is done at the terminal, and only a few "multiple choice"-type questions are given. The computer is being used primarily as a proof-checking laboratory, not for presenting simple facts. The hard-copy text itself is over 200 pages which, while coordinated to the computer exercises, is written in a traditional style similar to Suppes, 1972, upon which it is based.

3 The Runtime System

The program the student sees, EXCHECK, contains the bulk of our research effort. This system is actually two separate programs, one written in SAIL and the other in LISP, communicating by means of the multiple-process ("fork") facilities of the TENEX timesharing system. Together these programs require about 250,000 words of paged PDP10 memory. Figure 2

(5) PUB Tesler, 1972.

shows a conceptual chart of the components of EXCHECK.

3.1 Dialog with the Student

The input system contains a context-free parser, initialized to the language defined by the curriculum author as described above. There are also command editing facilities, command name recognition, and automatic command defaulting. For example, suppose the student wishes to specify for the variables in a quantified expression, using the SPECIFICATION command. The following dialogue shows how this might be done.

```
(1)  For all a,b there is c such that
       for all b1 if b1 < a & b1 > b then
         c < a & c > b V c = b1

*1specIFICATION

    [Here the student typed enough of the
    command name to cause the entire command
    to be recognized.]

Do you want to specify for (A a,b) *$
Variables *$a,b

    [The command processor accepts the ENTER
    key as requesting that the program find
    the obvious thing to do at that point.
    This particularly saves typing in
    substitution sequences where there is an
    obvious "guess". We use $ to represent a
    typing of the ENTER key by the student.
    In the immediately preceding interaction,
    the program asked the student which
    variables he wanted specified. The
    student typed the ENTER key without
    naming any variables, thus signaling the
    program that he wished the default list
    of variables. The program computes the
    default list of variables and prints it
    out so that the student will know what
    value is being used; in this case the
    variables are a and b.]

Substitute for a *$a

Substitute for b *$b

Do you want to specify for (E c) *$
Ambiguous name for c *$c

Do you want to specify for (A b1) *$
Substitute for b1 *$b1

1 SPECIFICATION  Substitute b for b, a for a;
                   c for c; b1 for b1
      (2)  If b1 < a & b1 > b then
             c < a & c > b V c = b1

    [The program prints out the result.]
```

The SPECIFICATION command, though reasonably simple, illustrates the combination of heuristics and logic that is typical of EXCHECK. The heuristics help to determine what is a plausible next step in the current command, and the logical routines certify that the step is correct logically. Also, since it is important not to generate frustrating syntactic errors when there is a charitable interpretation of the student's command, the command evaluation functions try to make plausible matches for the student's input. For example, we avoid generating syntax errors when the order of arguments is different from that given in the manual, in those cases where changing the order gives a correct application.

3.2 Output System

The output system includes functions for printing the steps of the student's proof in readable form, plus reviewing various parts of the proof. The students work on high-speed displays; EXCHECK uses a split-screen so that the student can place a portion of his proof in semi-permanent display at the top of the screen, while dialog with EXCHECK scrolls at the bottom.

The history component maintains a profile of the student concerning his location in the exercises, and saves partially completed proofs for the student, so that the construction of a proof can extend over several sessions. Completed proofs are listed on a high-speed printer and delivered to the students, since we want them to have copies of the work they do on the display terminals. Complete data collection for analysis is also done. We have found that it is best to save data in several different layers of detail so that the simplest kinds of data analysis will be easily performed, but it will still be possible to perform more complete analysis.

The curriculum driver component supervises the presentation of exercises to the student. Although the driver contains routines for text presentation and answer judging, this is not an important component in our application.

3.3 Proof Checker

The EXCHECK proof checker (see Figure 1) is a general purpose proof checker for many-sorted theories. In direct opposition to earlier proof checkers EXCHECK does not require proofs to be expressed as derivations in a standard formal system for first-order logic. The design philosophy of EXCHECK is to accept a sequence of statements as a proof of the last statement if each statement in the sequence either can be seen to be true in the theory under consideration or can be seen to be a consequence -- in the theory under consideration -- of previous statements in the sequence. Note that there is no *a priori* restriction of methods -- either for determining that a statement is true in the theory under consideration or for determining that a statement is a consequence of prior statements. Also, there are no "stored proofs" associated with the curriculum. Hence, the student can present whatever proof he can find and understand.

Earlier checkers (for both instructional and general applications) required that proofs be expressed as derivations in a standard elementary formal system for first-order logic. This involves two rather severe restrictions neither of which is plausible when considered in the light of

mathematical practice. The first major restriction is to pure first order logic. This in turn involves two restrictions:

> 1) The only way to show that a statement is true in the theory under consideration is to give a first order derivation of it from the axioms and previous theorems (instead of using decision procedures and calculations where appropriate);
>
> 2) Consequence in these earlier checkers is restricted to first order logical consequence rather than consequence in the theory under consideration.

The second restriction is to elementary formal systems. In these systems the logical inferences allowed by the rules are far too small to be used for checking naturally occurring proofs.

One of our major research goals is the development of methods which allow the user to express his proofs in standard mathematical style. Part of this is the development of natural inference procedures -- natural in the sense that they justify the inferential steps actually made in standard mathematics.

In standard mathematical practice the details are not explicitly handled -- and for a good reason: detail simply gets in the way of understanding. Usually left implicit are:

> 1) References to logically necessary previous results -- axioms, definitions, and theorems, or procedures.
>
> 2) References to inference rules or procedures -- in particular logical detail is almost never done explicitly.

As an example of the latter consider this example from Suppes, 1972 p. 35:

> [Suppose there were a y such that]
>
> y is in {x: x is not equal to x}
>
> Then by Theorem 47
>
> y is not equal to y

Theorem 47, mentioned above, is the theorem of concretion, which states for any formula FM,

> for each z,
> if z is in {x: FM(x)}, then FM(z)

Such inferences are common in mathematics -- giving as a justification for an inference that it follows "by" applying or using a previous result. Part of our problem is to analyze such inferences and to develop procedures adequate for their expression. This divides roughly (certainly not cleanly) into a logical (proof-theoretic) component and a linguistic component. The proof-theoretic component provides the actual inference procedures and the linguistic component interprets from the actual expression of the inference and its context in the proof into explicit procedure calls to be evaluated. For example, for the above inference the most likely explicit procedure call in EXCHECK is to a procedure called IMPLIES, which applies a previous result to a line or lines of the proof. Thus, one would have the following command sequence (with student input underlined):

WP (1) y is in {x: x is not equal to x}
1th$EOREM *47
(2) y is not equal to y

Doing this inference completely (in the sense of doing the details that are left implicit) involves a considerable amount of proof checking machinery.

We should add that in the current version of the program we are not attempting to deal with elliptic references to previous results.

There is a clear adequacy condition for such natural inference procedures, namely, are they adequate to express natural (preexisting) inferences, i.e., are they strong enough to justify the inferences actually made? To test adequacy one takes, as we have done above, the statements of the preexisting (textbook) proof and cites the natural inference procedures as justification. In the above, we used our IMPLIES procedure to justify the inference. If the procedures justify the inferences taken, then they are adequate for natural inference in the corpus under study. There is as yet no mathematical characterization of natural inference, and hence testing of procedures for natural inference will have to be case studies as above. Also, procedures adequate for one mathematical theory will not in general be adequate for another.

3.4 Further Extensions of the Proof Checker

Currently the proof checker is limited to employing inference procedures only after the premises of the inference have been established -- e.g., induction theorems are applied only after the basis case and the induction cases have been established. This means that the user must attend to detail -- such as writing out the induction hypothesis -- which the program could easily handle if the user could specify beforehand which inference procedure he intended to use.

To continue with the induction example, if the user specified the form of induction to be used, and if necessary what he is doing induction on, the program could easily take care of all of the details, setting up the induction hypothesis and the induction cases, with the user filling in only the "essential details". We have used induction as an example because of its familiarity, but the phenomenon is a general one. Almost every proof procedure can be applied this way, with a substantial savings of effort on the part of the user. Work is currently underway to extend EXCHECK to work in this fashion and more generally as an interactive theorem prover.

Hence, we intend to have available a number of

proof procedures, heuristic theorem proving algorithms, and special purpose algorithms (such as BOOLE, the decision method we designed and implemented for class algebra). Again this is in keeping with standard mathematical practice: people learn a variety of these proof procedures, which then form a part of the common background and are implicitly referenced in the proofs. Proofs can then be presented by giving a sketch of the proof (or even: the "idea behind the proof" or "essential steps") and the hearer can then use the common base of procedures to construct the full proof.

4 Student Usage of EXCHECK

The EXCHECK system was used during the 1974-75 school year to teach axiomatic set theory to Stanford students. In the spring quarter of 1975, eight students completed the course. The analysis of the data collected from those students, which we summarize here, will be detailed in a forthcoming technical report.

The students were given grades on the basis of preassigned standards of completed material -- theorems that they had to prove. Five students received a grade of A, one B, and two PASS grades. The amount of time spent at the terminal varied from a high of 84 hours to a low of 32, with a mean of 50.8 hours and a standard deviation of 21.1. Individual variations in work styles accounted for much of the variability. For example, S5 (student number 5), who worked 84 hours at the terminal, did most of his studying for the course interactively, reading his text and trying things out with the proof checker. Other students would enter the computer classroom with a completed proof written out.

The average number of proofs done was 52.6, with student S7 doing the minimum of 41 and student S5 (who did most of his studying at the terminal) doing the high of 65.

4.1 Student Use of Inference Rules

With only eight students in the course last spring, the data is of limited significance. However, we had the following main hypothesis about the use of the rules of inference: students who did better in the course would tend to use high-level rules of inference, and students who did rather poorly would tend break up these inferences into smaller steps, using rules elementary rules. These rules are available in the checker, but not emphasized in the instruction. This hypothesis has not been disconfirmed in the data.

For example, student S4, who received a grade of PASS, tended to avoid using the rules that we intuitively thought of as "high-level", and instead used the "low-level" rules most frequently. On the other hand, student S6 who received a grade of A+, generally had the opposite pattern. A few of these data are shown in Table 2 below.

TABLE 2

Usage of Rules by Student S4 and S6

"High-level" Rules

Rule	Use by S4	Use by S6	Average over 8 students
IMPLIES and RIMPLIES together	26	204	150.25
Application of a theorem in VERIFY rule	52	74	100.88
Application of a definition in VERIFY rule	17	92	62.88
Two rules for equality replacement (RE and RER)	4	30	43.51
TAUTOLOGY	5	18	14.38
BOOLE	5	12	8.63
RF (Replace Formula)	5	19	9.75

"Low-Level" Rules

AA (a logical rule for elimination of the logical implication symbol)	67	21	30.63
DIFF (another logical rule)	48	5	8.25

The data thus far do not disconfirm our a priori beliefs about the importance of the computationally complex rules. It is interesting to note that some computationally simple rules, especially those for manipulating conjunctions, were clustered with the higher-level rules in terms of usage. In examining the actual student proofs from the spring quarter (1975), we discovered that the conjunction rules were being used as auxiliaries to the IMPLIES rule. We have since modified the IMPLIES rule to incorporate these uses of the conjunction rules.

5 Future Extensions

Our main research interest is to study the role of natural language and computer-generated audio in the context of the EXCHECK system. Informal mathematical reasoning is an example, in a relatively structured domain, of complex dialogues about semantically rich objects. Informal proofs, as given in classrooms and textbooks, are highly linguistic in character in that the fragment of English used exploits the ability of natural language to provide stress, ellipsis, pronominal reference, and so on, to produce readable and convincing proofs.

Now that EXCHECK is operational for Stanford students and several additional curricula are being implemented, we are concentrating our efforts on developing computer models of the expression of mathematical arguments.

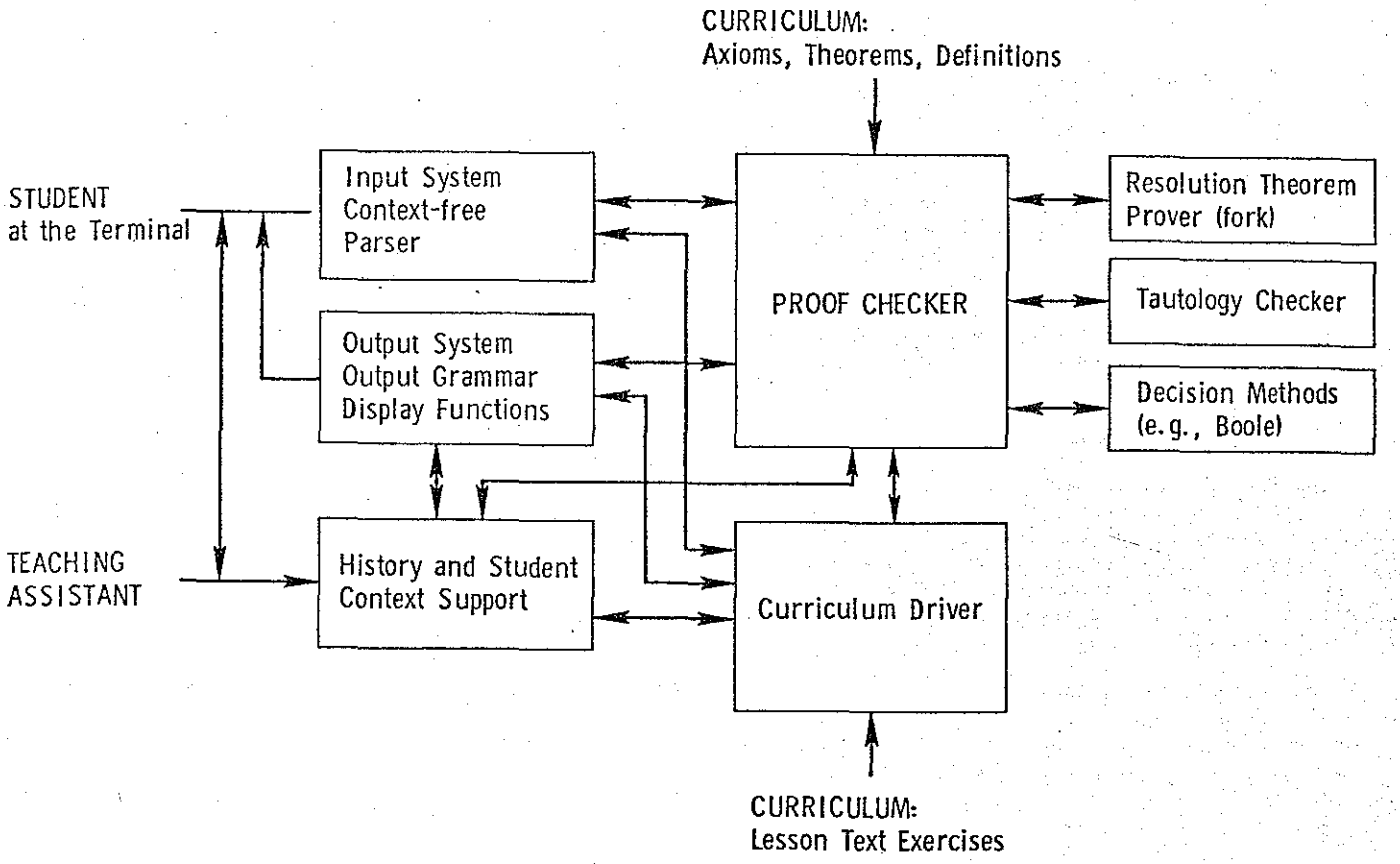

Figure 1. EXCHECK instructional system.

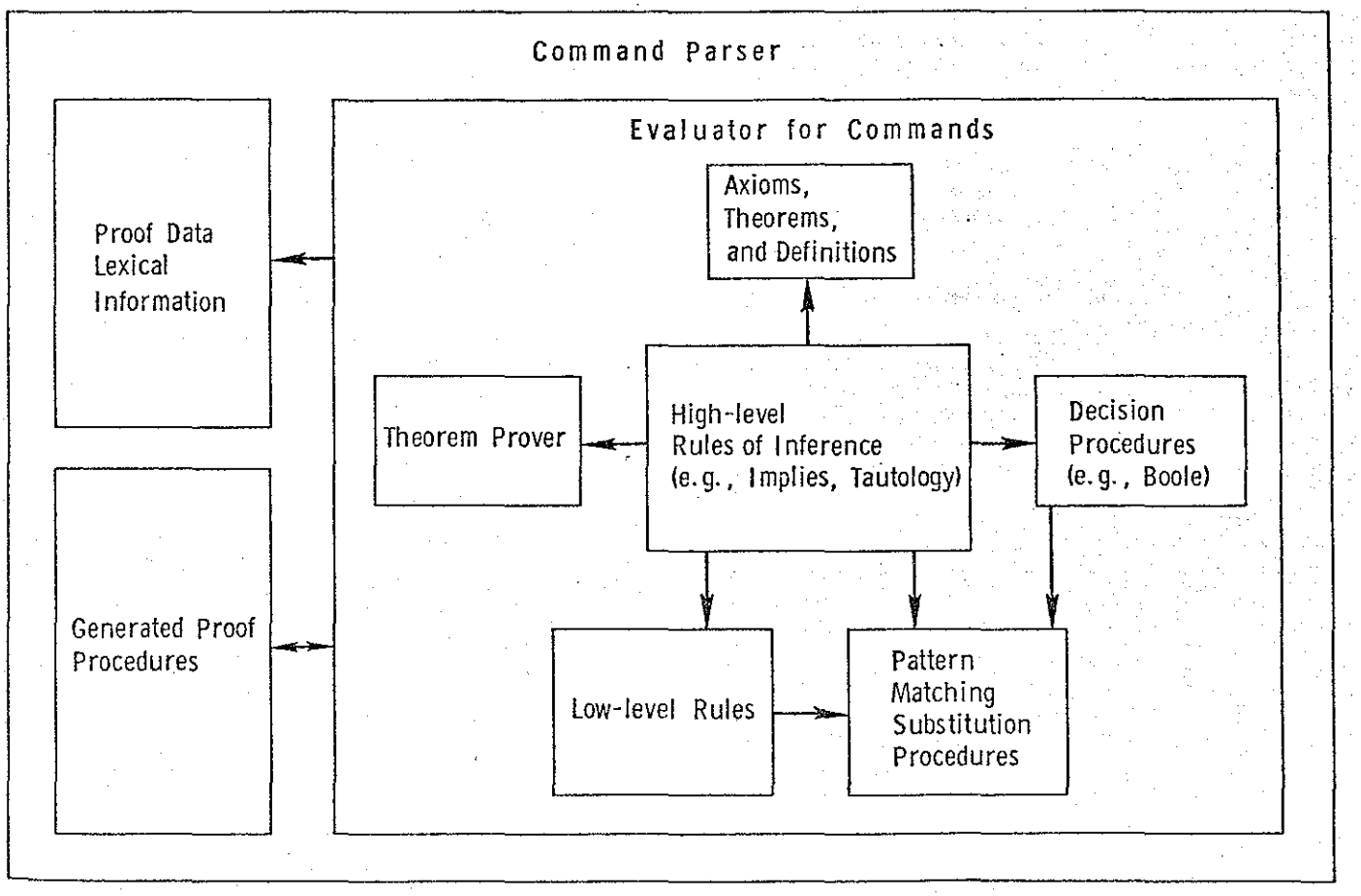

Figure 2. EXCHECK proof checker.

Figure 3 shows a proof from an advanced part of set theory, the theory of ordinal numbers. It is a proof by induction over the ordinals. The commands of the current checker are sufficiently powerful to make the proof of manageable size, but the structure of the proof is not perspicuous.

Figure 4 shows an analysis of the structure of the proof in Figure 3, generated by a program we have developed. Notice that the structure has been made clear by finding the important parts of the proof, and printing out the relationship of those parts. We are incorporating this proof analysis program into EXCHECK, as a part of a routine to summarize and explain the parts of a student's developing proof. The explanation will combine graphic display with computer-generated audio.

Underlying this effort to explain and summarize the proof is a search for a better language for expressing the proof. This summarizing should involve a theory of the significant structure of the proof -- what has to be said and what can be left unsaid. We believe that this theory will be formulated in terms of a basis of strategies for doing proofs. For the proof in Figures 3 and 4, this would amount to saying that the proof is an induction on M(b). This is the usual information for doing this proof, and it would be expected that the program and/or student would have the capacity to set up a proof by induction.

6 Instructional Uses of EXCHECK

We envision a system for mathematics instruction that can help the student understand and construct a complex and informal mathematical proof. With EXCHECK, it is possible for the student to present proofs informally and naturally, but have the program check for correctness as the proof progresses.

There are, of course, totally different approaches to teaching mathematics with a computer, and most of these are not axiomatic at all. A common approach is to provide the student with a simple but powerful programming language, such as LOGO (Feurzig et al., 1969), and encourage him to write interactive programs that manipulate displays, TURTLE-type devices, and so on. We see this as a valid approach, but not as a substitute for teaching inferential reasoning and proof construction in the manner of EXCHECK.

There are also implications for the instructional staff. Teaching assistants have remarked to us that they have more opportunity to interact directly with students in this context since the computer is handling the rote grading of the "homework".

We also have the goal of advising and helping the student with the computer, both in terms of particular problems (when he needs "help"), and in terms of general structural comments about how to express what the student wants to say. Before this will be possible in any serious way, it is necessary for the program to have a general model of mathematical reasoning in terms of the semantic and linguistic components involved, combined with the ability to accept heuristics and strategies for the theory at hand.

```
Derive:  Lim(a) & b > 0 -> Lim(a . b)

WP      (1)  Lim(a)
ABBREVIATION
        (2)  M(d) <-> d > 0 -> Lim(a . d)
VERIFY Using:  Df. Greater, Th. 5.1.18
        (3)  M(0)
WP      (4)  M(a1)
WP      (5)  a1´ > 0
VERIFY Using:  Df. Otimes
        (6)  a . a1´ = a . a1 + a
1 VERIFY Using:  Th. 5.4.10
        (7)  Lim(a . a1 + a)
7,6 RER 1
        (8)  Lim(a . a1´)
5,8 CP  (9)  M(a1´)
4,9 CP  (10)  M(a1) -> M(a1´)
10 UG   (11)  (A a1)(M(a1) -> M(a1´))
WP      (12)  Lim(c) & (A b)(b > c -> M(b))
WP      (13)  c > 0
1 VERIFY Using:  Df. Limit ordinal, Th. 5.1.13,
    Df. Greater, Df. Less
        (14)  a > 0
12,14 IMPLIES Using:  Th. 5.4.20
        (15)  Lim(a . c)
13,15 CP
        (16)  M(c)
12,16 CP
        (17)  Lim(c) & (A b)(b > c -> M(b)) -> M(c)
17 UG   (18)  (A c)(Lim(c) & (A b)(b > c -> M(b))
                ->
                  M(c))
17UG    (18)  For all c,
                  If Lim(c) & (A b)(b > c -> M(b))
                    then  M(c)
3,11,18 IMPLIES Using:  Th. T3-induction
        (19)  (A a2)M(a2)
19 US b for a2
        (20)  M(b)
1,20 CP
        (21)  Lim(a) -> M(b)
21 TAUTOLOGY
        (22)  Lim(a) & b > 0 -> Lim(a . b)
*** QED ***
```

Figure 3

Proof of a Theorem from Ordinal Number Theory

```
Prove:  Lim(a) & b > 0 -> Lim(a . b)
Abbreviation   M(d) <-> d > 0 -> Lim(a . d)

Show:  Lim(a) & b > 0 -> Lim(a . b)
  By tautology
  |--------------------
  |
  |Show:  Lim(a) -> M(b)
  |   By conditional proof
  |   |--------------------
  |   |
  |   |Show:  M(b)
  |   |    Using  Th. T3-induction
  |   |   |--------------------
  |   |   |
  |   |   |Show:  M(0)
  |   |   |   By logic
  |   |   |    Using  Th. 5.1.18, Df. Greater
  |   |   |
  |   |   |Show:  (A a1)(M(a1) -> M(a1´))
  |   |   |   By conditional proof
  |   |   |   |--------------------
  |   |   |   ||   Th. 5.4.10
  |   |   |   ||   Df. Otimes
  |   |   |   |--------------------
  |   |   |
  |   |   |Show:  For all c,
  |   |   |         If Lim(c) & (A b)(b > c -> M(b))
  |   |   |            then  M(c)
  |   |   |   By conditional proof
  |   |   |   |--------------------
  |   |   |   ||   Th. 5.4.20
  |   |   |   ||   Df. Limit ordinal
  |   |   |   ||   Th. 5.1.23
  |   |   |   ||   Df. Greater
  |   |   |   ||   Df. Less
  |   |   |   |--------------------
  |   |   |--------------------
  |   |--------------------
  |--------------------
```

Figure 4

Program Generated Graphic Analysis
of a Proof of the Limit Ordinal Theorem

References

1. Feurzig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. Programming languages as a conceptual framework for teaching mathematics (Report 1889). Boston: Bolt Beranek, & Newman, 1969.

2. Quam, L.H., & Diffie, W. Stanford LISP 1.6 Manual (Operating Note 28.6). Stanford, Calif.: Stanford Artificial Intelligence Laboratory, Stanford University, 1967.

3. Smith, N. W. A question-answering system for elementary mathematics (Tech. Rep. 227). Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1974.

4. Smith, R. L. TENEX SAIL (Tech. Rep. 248). Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1975.

5. Suppes, Patrick. Axiomatic Set Theory, Dover Publications, New York, 1972.

6. Tesler, L. PUB: The document compiler, Stanford Artificial Intelligence Laboratory, Operating Note 70, September, 1972.

7. Van Lehn, K. SAIL users manual (Artificial Intelligence Memo 204). Stanford, Calif.: Stanford Artificial Intelligence Laboratory, Stanford University, 1973.

8. Wolpert, T. M. TENEX UCI-LISP. Unpublished manuscript, Stanford University, 1975.