Computer-Assisted Instruction in Logic: EMIL

JAMES W. GARSON and PAUL MELLEMA

University of Notre Dame

In most logic courses, the student is expected to acquire skills at finding proofs in some formal system. This learning is important not only in that it provides a foundation for mastery of the basic concepts of logic, but also because it gives the student the opportunity to learn some of the practical problems and strategies involved in creative thinking, particularly the creative thinking characteristic of mathematics and the sciences.

In the standard sort of course, students' abilities at finding proofs vary widely, so that those who do not have an initial knack are severely penalized. Even when strategies for proof-finding are carefully discussed in class, some students invariably complain that they cannot do a new problem on their own in spite of "understanding" the lectures. Part of the problem for these students is that they cannot convert a verbal explanation of techniques into a flexible tool for dealing with a new situation. Another problem is their inability to tell when they have applied the rules of a formal system correctly. They may proceed for quite some time with incorrect variants of the rules, completely unaware that anything is wrong.

With a bit of tutoring, most students with these difficulties improve rapidly. If students "think out loud" while attempting a proof, a gentle nudge here and there often leads to success. If they don't understand the rules, or simply haven't bothered to learn them, guiding them through a proof or two tends to straighten things out fairly quickly and improves their confidence and motivation. Just as in teaching most skills, effective strategies involve letting the student perform the task under guidance; lecturing on the proper procedure and telling students to "go home and do likewise" is relatively ineffective.

Of course there are good reasons why tutoring is not widely used in an introductory course in logic. These classes are usually quite large, so tutoring simply takes too much of the teacher's time. Because grading exercises in proof-finding is tedious, teachers tend to give students relatively few exercises that actually require them to create a proof. Even if students could learn on their own, they simply do not get enough practice to develop any skill, unless they "catch on" right away. Very often, the teacher relies on exercises that require a single answer, such as those that ask the student to fill in the justifications for the lines of a proof that is already completed. This does acquaint students with the rules, but it gives them no practice in the art of finding a proof.

Computer Applications

Computers make it possible to simulate the tutoring situation. Students can enter their proofs at the terminal and the computer can be programmed to see whether each line of the student's solution follows from previous lines, and to describe the difficulty if anything goes wrong. When trouble occurs, the computer can display the programmer's suggestions about how to proceed, and can show students what the rules "look like" in case they have forgotten them. There are a number of programs for teaching logic that incorporate one or more of these features. (For descriptions of the better known programs of this kind, see [1], [3], [6].)

This use of computers in education is particularly interesting because it has begun to depart radically from the multiple-choice format which became almost paradigmatic of computerized teaching materials. A proof-checking program does not require the student to come to any pre-selected answer, but to find a solution by any of a potentially infinite number of lines of attack. In a sense, the program does not demand an answer, but simply provides an ongoing check of the student's progress in achieving a result. It does not demand a set response so much as provide a tool which students can use in their own way to acquire a skill.

Compared to multiple-choice programs, proof-checking programs make heavy demands on the computer, the instructor, and the student. The computer must be provided with routines that interpret the student's "moves" at the terminal, determine whether they are correct, and respond in an intelligent way. The student and instructor must familiarize themselves with the procedures for operating the computer program, and must put up with the inconveniences caused by having to use a computer which is generally overburdened already, and which occasionally malfunctions. They must also put up with the inevitable mistakes a programmer makes in designing the logic teaching system. And yet, if we are to develop computer teaching systems that provide students with tools for learning, rather than merely with ongoing multiple-choice examinations, we must overcome these difficulties. Working our effective strategies for proof-checking programs can pave the way for developing less authoritarian styles of computerized education in other areas.

Proof-checkers have been around now for over a decade. Evaluations of their effectiveness have been consistently positive, yet very few such systems are actually used by instrucors who did not have a hand in their development. The purpose of this paper is to describe some of the obstacles to widespread use, and to suggest ways of overcoming these obstacles. The difficulties we shall discuss are these:

1. Proof-checking programs tend to restrict the instructor's choice of rules, and hence of textbooks.
2. The logic programs now available force students to construct their proofs from top to bottom, whereas the most natural way to construct a proof is to begin by deciding how to justify the *last* line.

3. If hints are available at all, they tend to ignore any progress the student may already have made. Because they force the student to reproduce the proof devised by the programmer, alternative approaches receive no encouragement.
4. Logic programs tend to be intolerant of typographical errors and of minor deviations from a rigidly defined input format; this tends to frustrate students who are not typists.
5. The programming languages available at most colleges and universities were not designed for writing logic programs.
6. Teachers can rarely count on finding the kind of help they need to get a program running at their school.

The first four of these obstacles can be surmounted by appropriate programming, and we shall sketch the approaches that look most promising to us. The last two require institutional change.

Restricted Choice of Rules

Proof-checkers generally come with a particular, and sometimes idiosyncratic, formulation of logic. Any teacher who uses a given program must learn, and construct the course around, the programmer's choice of rules. But there is a wide variety of formulations—one for each of a plethora of logic textbooks on the market. Given the persistent variations in notation, rules, and teaching strategies used by logic teachers, it is not reasonable to expect them to adopt a computer program that forces them to master an entirely new approach, or to limit candidate texts to those for which proof-checking programs have been written.

James Garson, working with Paul Mellema and Kook Huber, has written a proof-checker, named EMIL,[1] which can handle virtually any known formulation of the rules of first-order logic. The program does not need to be redesigned to accomodate new rules. The rules for a system are sent as data to the program. Changing the rules can be done in a few minutes on a computer that has file capabilities; on systems without such storage facilities, rule changes involve only a simple editing job. New rules can be typed into the program using a format which is perspicuous and widely used; no knowledge of programming, or of the internal workings of EMIL, is required. EMIL is presently being used in logic courses at Notre Dame and Rutgers. The program is capable of providing simultaneous service to students in several different logic courses, each with its own distinctive system of rules.

We wish to stress that this generality has simplified the programming task, rather than complicating it. No matter which rule a student applies at a given line of the proof, the same subroutine is used to check whether the rule was applied correctly. This subroutine is not particularly long (about 60 lines of the language PL/1); it would probably take as much effort to program two or three relatively complex rules of a specific system of logic. The advantage is that in-

dividual rules no longer need to be programmed in piecemeal fashion. Instead, a more general routine compares the attempted steps with argument forms stored as data.

Probably the best way to understand EMIL is to run through an example. Suppose for simplicity that we have just three rules:

HYPOTHESIS	MODUS PONENS	ADDITION
	A	
———	(A – > B) ———	A ———
A	B	(A v B)

These rules would be entered to EMIL by the instructor, in the following form:

```
'HYP'  –1  '/A'
'MP'    2  'A,(A – > B)/B'
'ADD'   1  'A/(A v B)'
```

In the first column we have the name the instructor has assigned to the rule (abbreviated so as to save the student extra typing at the terminal), followed by a code number which indicates how many premises the rule has and whether the rule introduces or discharges hypotheses. The third column contains the pattern for the rule. If several patterns go with a single rule, as in De Morgan's Law, the entry indicates multiple forms for that rule:

```
'DM'  1  '(– A v – B)/ – (A&B)'
'DM'  1  '(– A& – B)/ – (A v B)'
```

Now suppose a student has gotten this far in a proof: (Here and following **boldface** indicates the material that EMIL would print in computer-student exchanges.)

1: P&Q HYP
2: (P&Q) – > R HYP

To derive R, by applying MP to lines 1 and 2, when EMIL prompts for line 3, by printing **"3:"**, the student replies:

R 1,2 MP

The first thing EMIL does is to scan the instructor's entries to find the name of a rule on file. He encounters MP, and so expects two line numbers. He locates these numbers and looks up the formulas entered on these lines (namely 'P&Q' and '(P&Q) – > R'), and constructs, in the computer memory, a sequence of formulas called SEQ:

(P&Q),((P&Q) – > R)/R (SEQ)

The formula to the right of the slash in SEQ is the formula just entered into the proof, as line 3, namely 'R'; to the left of the slash are the formulas cited in the justification for this latest line in the proof. (Outer parentheses added by EMIL

to make pattern matching routine simpler.) EMIL now checks to see whether the rule MP has been applied properly at line 3. To do this, he checks to see whether SEQ "matches" PAT, the pattern of the rule used at this line:

A,(A $->$ B)/B (PAT)

The flow chart for this matching subroutine is shown in Figure 1, and Figure 2 traces our present example through the flow chart.[2]

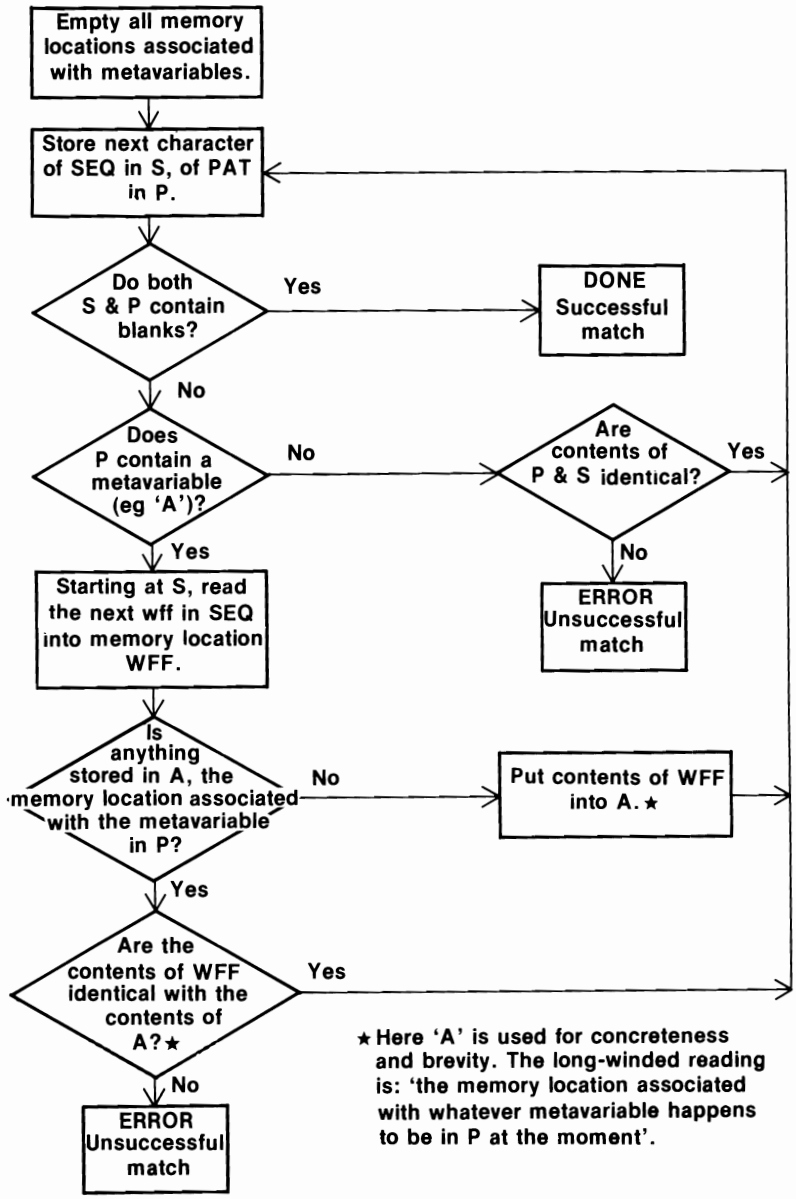

Figure 1: The Matching Subroutine

Figure 2: The Matching Subroutine (an example)

The numbers under each formula below indicate which symbols EMIL is looking at in PAT and SEQ at each stage of the matching process.

PAT: A,(A −> B)/B
1 2 3 4 5 6 7 8 9 10 11

SEQ: (P&Q),((P&Q) −> R)/R
1 2 3 4 5 6 7 8 9 10 11

At each stage EMIL is doing the following:

1. I see 'A' in PAT. Memory location A is empty so I load the wff starting at 1 in SEQ into location A. (A now contains '(P&Q)'.)
2. I see ',' in both SEQ and PAT. These match so on to the next ones.
3. These also match.
4. I see 'A' in PAT. Memory location A is full so I check to see whether the wff in the SEQ starting at 4 is identical to what is in A. It is, so I continue.
5. These match.
6. Match again.
7. I see 'B' in PAT. Memory location B is empty, so I put the next wff of SEQ (starting at 7) into B, and continue. (B now contains 'R'.)
8. A match.
9. Another match.
10. I see 'B' in PAT. Memory location B contains 'R'. I check to see whether the next wff in SEQ is the same as this. It is, so I continue.
11. No more symbols in PAT or SEQ. I'm done! SEQ is a correct instance of PAT, and the rule has been applied correctly.

Simple additions to the program allow EMIL to handle quantifier rules and rules of substitution of provable equivalents. For example, the rule of Universal Specification (or Instantiation, or Elimination) is presented to EMIL as follows:

'US' 1 '(X)A/S'

Both the 'X' and the 'S' in this schema are metavariables on a par with the prop-

ositional metavariables 'A', 'B', 'C', and 'D'. When the task of matching the student's formulas against this pattern is completed, EMIL checks to see whether anything is in S. If it is (and it will be if the above pattern has been successfully matched), EMIL checks to see whether it is the result of substituting some term for what was in X in the formula in A. In short, the 'S' here is shorthand for the result of substituting some term for all (free) X in A. EMIL keeps a record of terms used in a proof in order to limit rules like Existential Specification in the appropriate way.

This method for dealing with the quantifier rules can be generlized to other forms of substitution. For example, by associating the appropriate code numbers with "rules of substitution," the appropriate subroutine can be called to check whether A and S bear any of a number of relations of substitution. So rules for substitution of provable equivalents and identical terms can be accommodated using the same technique.

Options for Proof Notation

Even when the problem of handling the various formulations of the rules of logic has been solved, we must still deal with the variations in proof notation. Fitch systems use vertical lines to keep track of subproof structure (See [2]), whereas systems modelled after those of Suppes [8] and Lemmon [5] use dependency lists for that purpose. Some systems for quantification use lines, others flagged variables, and others still a special rank of terms. These are mainly variations in what a student expects to see at the terminal, and they are independent of how the rules are formulated. Designing a system that is modular across these differences is somewhat more difficult than designing one that can apply anybody's set of rules. Nevertheless, this problem should not be too difficult to solve. There are only a few major approaches to proof notation, while minor variations within these approaches should be easy to tolerate. The first versions of EMIL were written using the dependency list format, but James Garson rewrote a version using the Fitch style of notation without much difficulty. We are optimistic, then, that by designing three or four separate systems around the main notational approaches, we can provide any teacher with a version of EMIL which comes very close to the proof notation of his or her favorite textbook.

Unnatural Proof Construction

A second major difficulty with available proof checkers is that they force students to work from the top to the bottom of their proof. The best proof-finding strategies involve working at the bottom, then the top, then the bottom, and so on, until the proof "meets" in the middle. Terminals which print on paper do not allow the student to use a bottom-top-bottom procedure because it is impossible to type the top and the bottom of a proof on a page, and then go back to fill in lines in between. Printing terminals thus develop exactly the habit

of proof construction that one wants to break, and prevent the student from practicing the most effective proof-finding strategies. It would be better to ask that students construct their proofs with pencil and paper, and then enter the completed proof at the terminal. But then the computer does little more than grade exercises; the student does not get immediate feedback as each line is constructed, and the computer can give no guidance during the process of proof construction. Since the main problem for most students is finding proofs, the tutorial function of the computer, which was supposed to be its most valuable feature, cannot be used.

The best solution to this problem would be to use large graphics terminals, so that students could write into the top and bottom of their proofs just as they would with pencil and paper. The trouble with this solution is that graphics terminals, especially those with screens large enough to accommodate fair-sized proofs, are very expensive and unavailable to students at most colleges. Another difficulty is that standards for graphics programming languages are not well enough developed to allow programs written for one computer system to be transferred to another.

At Notre Dame, we have programmed a feature for EMIL, called GOAL, which solves part of the problem using standard equipment. If the student wishes to enter a line at the bottom of his or her proof, he or she indicates this wish by typing an appropriately large line number (e.g., 99), followed by a colon. The terminal responds by spacing over to the right-hand side of the paper, where it repeats the student's line number, and then waits for the student to enter the formula to be derived at that point in the proof. In this way, the "top" of a developing proof appears on the left of the page, in its natural order, and the "bottom" of the proof appears on the right of the page, in reverse order. Here is an example:

1:	(P&Q)&R HYP		
2:	99:	**99:**	P&(Q&R)
2:	P&Q 1SIMP		
3:	98:	**98:**	Q&R
3:	75:	**75:**	P

When the student first enters a GOAL formula to the system, it may not be clear how that formula is to be justified; but as soon as the justification becomes clear, it may be entered by typing the appropriate line number, again followed by a colon. This induces the terminal to space over to the right side of the page, repeat the line number and the formula already entered, and then to wait for the student to enter the justification. In the example above, the student will surely know, having entered lines 98 and 75, how line 99 is to be justified. His or her next move, then, may well be this:

3:	99:	**99:**	**P&(Q&R)** 75, 98 CONJ

Students should probably be encouraged, whenever they enter a new goal formula into the proof, to indicate how that formula will figure in the justification

of some subsequent line in the proof; a formula is unworthy of the status of a goal unless it can make some clearly explicable contribution to completion of the proof.

A student working on a complex proof with the help of GOAL may become confused, after a time, about the structure of his or her proof, especially since the goal formulas will appear on the page in (roughly) reverse order. To remedy this confusion, the student can ask the computer to list his or her proof. The terminal will respond by printing out all the steps of the proof on the left side of the page, in their natural order, with ellipsis points separating the "top" of the proof from the "bottom." Here is how our example would look:

```
1:  (P&Q)&R   HYP
2:  P&Q       1SIMP
        .
        .
        .
75: P
        .
        .
        .
98: Q&R
99: P&(Q&R)   75, 98   CONJ
```

Of course this is not an ideal way to work; the common terminals type rather slowly, thus discouraging the student from listing the proof often. Still, it beats a system where the student is forced into a linear, top-to-bottom pattern of proof construction, and this is much more economical than use of display terminals.

Inflexibility of Hints

The more valuable aspects of tutoring cannot be duplicated on the computer unless it is capable of assessing the student's progress, and offering advice about how to proceed, if and when the student gets stuck and requests help. There are two main strategies for doing this. The first, used by BERTIE [6], is simply to store a completed version of the proof that the student is working on, and to store comments on the various lines which are intended to help a student who asks for help after completing that particular line. If the comment on a particular line proves unhelpful, the student can ask to see the next line of the stored proof, or indeed any number of lines, up to and including the entire proof.

This hint strategy requires that a completed proof be stored in the computer, along with appropriate comments, for every problem students will work on. It also presupposes that there is likely to be only one reasonable sequence of steps that leads to the conclusion. If students approach a problem in an unusual way, there may not be enough similarity between their proof and the stored proof for the computer to be of any help. Finally, it presupposes a top-to-bottom pattern of proof construction. But very often the very next steps in a

proof will not reveal a strategy that leads to success; such strategies must rather be "explained" with reference to what happens much later on. By restricting attention to the steps of the proof in sequence, this sort of hint routine fails to help students to appreciate the "global" strategies which require knowledge, not just of where the proof has been, but also of where it is going. These, we maintain, are generally the most useful strategies.

Another technique is to write a program that allows the computer to generate a solution to the student's problem, and to recognize standard situations during the course of that solution. QUINC is an example of a system of this kind (See [1]). This strategy eliminates the need for storing a proof, with commentary, for each problem to be attempted, since the computer generates its own solutions. But this strategy runs the risk of generating strange proofs, which students are unlikely to recapitulate. Also, each formulation of the rules of logic will require its own custom-tailored program for generating proofs. The program to generate comments must be very carefully written to avoid giving misleading advice. Worst of all, this strategy still does not help students to see global strategies; like the stored-proof strategy, this strategy uses a top-to-bottom approach to proof construction, and confines itself to giving advice about the very next step in the proof.

We have programmed EMIL to solve these problems for the propositional calculus (see Figure 3). Our hint algorithm presupposes a proof-checker that permits the student to build a "stack" of goal formulas, as illustrated in the preceding section.

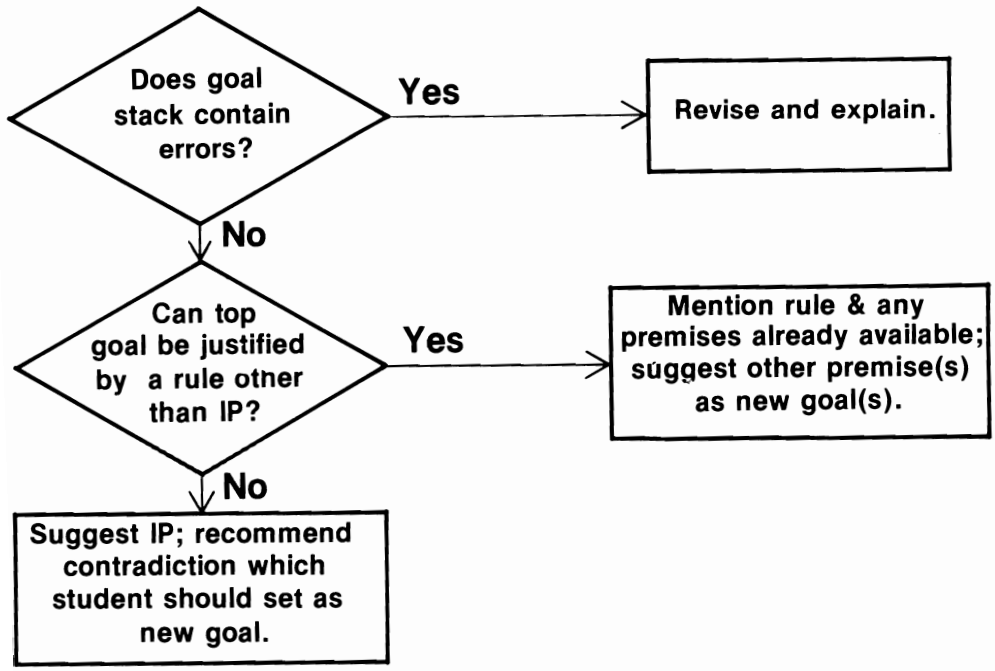

Figure 3: The Hint Algorithm

When the student asks for a hint, our algorithm begins by reviewing the student's goal stack for serious errors; if a student is headed in the wrong direction, the best advice consists in pointing this out, so that the student can try a different strategy. If, on the other hand, nothing is amiss in the student's work thus far, our hint algorithm next looks for some rule, other than indirect proof, which could be used to justify the top goal on the student's stack. The algorithm will mention the first such rule it finds. In addition, it will call the student's attention to any lines in the top of the proof[3] which can be used, along with the suggested rule, in justifying the top goal. Finally, if any formula(s) must be proven before the suggested rule can be applied, the algorithm will list the missing formula(s) and suggest that the student add it (them) to the top of the goal stack. All this assumes, of course, that the top goal can be justified without recourse to indirect proof. If the search for a direct-proof rule should fail, the algorithm will recommend the appropriate hypothesis for *reductio*, and make a suggestion about what contradiction the student should try to prove. A detailed account of the structure of our hint routine is given in the Appendix.

We want to emphasize what we take to be the principle virtue of our approach to giving advice. Our advice giver directs the student's attention *not* to the very next line to be entered at the top of the proof, but instead to the bottom of the proof, which contains a list of goals which the student is trying to achieve. This reinforces the strategy of working backwards, which is highly effective not only in logic, but in many other problem solving contexts.

Our hint routine is designed primarily for natural deduction systems of propositional logic, though at present it works well with Pospesel's version of predicate logic [7] as well. It would not be particularly appropriate for systems where subsequent lines of a proof are determined, or very nearly determined, by the previous lines; the tree systems of Jeffery's *Formal Logic* [4] are examples of this kind of system. In such cases, the hint algorithm could be simplified immensely, and could concern itself almost entirely with matters such as the choice of terms in instantiating the quantifiers. Systems such as this give students more practice in carrying out an algorithm, but much less in creative problem-solving. Given that such systems are much easier to deal with on the computer, there is a danger that they wil be more widely adopted as computers are more widely used to teach logic. We say "danger" because it is more important, in a logic course, to learn creative problem-solving strategies, than to learn to follow a predetermined set of rules.[4] We hope that the description of this hint routine will encourage others to teach systems that give students practice in this creative process, even if they use a computer.

Troubles for Students

If a program is difficult for the teacher and the student to use, then it will tend *not* to be used. Because of poor programming languages (discussed in next section), it is annoyingly difficult to program the computer to be flexible about what the student enters at the terminal. Most students are poor typists, and

almost everyone has difficulty adjusting to the rigid format which most programs require. Designing a system that is gentle with the user, that is, which accepts and corrects common mistakes, and allows a variety of formats, is difficult, partly because it is not often clear what the common typing mistakes or omissions at the terminal are liable to be.

In designing EMIL, we have tried to insure flexibility in a few important areas. The first is in EMIL's routine for determining whether a student's input formula is well-formed. People are generally poor at writing well-formed formulas. We generally drop the outermost parentheses, put in spaces or use an implicit hierarchy of connectives to determine grouping, etc. When parentheses are used, we inevitably forget to match them properly. EMIL's wff-checker is designed to tolerate all these errors and more. The program simply fixes the errors found, and prints back a corrected version if there is a possibility that the student may not know what EMIL takes the formula to be.

For example, if the student types P $->$ Q) $->$ R, EMIL assumes the formula in question is ((P $->$ Q) $->$ R); if P $->$ Q $->$ R is typed EMIL associates to the left: (P $->$ (Q $->$ R)); however, if P&Q $->$ R is typed, EMIL resolves the parentheses according to the convention that & is grouped before $->$, and so treats this input as ((P&Q) $->$ R). If attempts at repair of the input fail, EMIL politely indicates the point where the difficulty arose. The following interaction illustrates this.

2: P – Q HYP
YOUR FORMULA IS NOT WELL FORMED
I HAD TROUBLE NEAR THE *: P* – Q

The wff-checking routine does a very accurate job of making sense of formulas which are technically ill-formed. We have tested this part of the program with several people familiar with logic, and EMIL has consistently repaired ill-formed formulas according to their expectations on what was intended. It is interesting to note that the wff-checker does not involve a lot of elaborate programming. It uses a simple stack parser, plus information on the standard hierarchy used with the connectives.

We also thought it important to build flexibility into the format used to justify lines in a proof. The version of EMIL which uses dependency lists, allows the student to enter a new line of a proof in the following sort of format:

3: P $->$ Q 1, 2 MP /1 2

This asks EMIL to try to derive (P $->$ Q) from lines 1 and 2 by *modus ponens*, where the dependency list contains assumption lines 1 and 2. But this is only one way the line might have been typed. The formula may be omitted, in which case EMIL figures it out for himself (if he can): the slash and the dependency list may also be omitted, in which case EMIL supplies it. Furthermore, the numbers 1, 2 preceding 'MP' may be separated by spaces or a dash (the same is true of numbers in the dependency list), or they may follow the rule name instead of preceding it. Finally, spaces between formula, rule name, numbers, and the

slash are optional, and are only needed to distinguish, for example, the number twelve ('12') from the pair of numbers one and two ('1 2'). Consequently, all the following inputs would be accepted by EMIL, and treated as equivalent to our first example:

3: MP1 2
3: P – > Q1-2MP/1,2
3: (P – > Q) MP,1,2,/1 2

When formula and/or dependency list is missing, and the application of a rule is successful, EMIL retypes the line and includes the missing information for the student's later reference.

EMIL also allows the student to use several names for the same rule. For example, the student may want to type MODUS PONENS, or – > OUT, instead of MP. This can be done simply by adding new lines to the data file containing the list of rules:

'MOD' 2 'A,(A – > B)/B'
'– > 0' 2 'A,(A – > B)/B'

Because the rule names are abbreviated in the data file, all the student will actually need to type correctly is 'MOD' or '– > O'. Extra characters are ignored; they may without penalty be typed incorrectly, or even omitted altogether.

Some inflexibility remains. The formula (if present) must appear first, and the dependency list must begin with a slash, and appear last. These limitations could be removed, but our present programming language (PL/1) makes this task annoyingly complex. With more sophisticated text-processing languages, more could be done.

We do not claim to have solved the problem of making a natural activity out of work at the terminal, but programming the computer to be intelligent about the student's input can go a long way toward making computerized logic programs easier to work with. This is important in promoting their widespread acceptance.

Programming Languages

Hardly any of the present logic systems will run on many computers. The Stanford program LOGIC is written in SAIL, a language peculiar to that campus. (EMIL is not in much better shape; he is written in PL/1, which runs only on certain IBM products.) The authors of BERTIE went to heroic lengths to insure that their program would work on just about any computer that "understands" BASIC. Nevertheless, BERTIE will not run on our IBM 370, because the version of BASIC available to us (VSBASIC) is simply not designed to handle such a large program as BERTIE. The reason BERTIE is so long is that the authors had to restrict themselves to the commands common to virtually every version of BASIC. The resulting language is so impoverished that elaborate programming is required for the simplest of tasks. A number of attempts have been made to try to stan-

dardize programming languages so that programs can be easily "transported," but the tendency is to standardize around bare-bones languages, rather than to provide the standard language with a full complement of programming tools. The result is that it is extremely difficult to write programs in the standard language, and even if this is done, there is no guarantee that the resulting program will be small enough to run at most campuses.

If this weren't enough, there is another problem. BASIC and FORTRAN are presently the most widely used languages, and they are thus prime candidates for solving the transportation problem. But FORTRAN was designed primarily for mathematical computation in a research setting, and BASIC was developed as a simplified extension of FORTRAN. The result is that both languages are poor at handling text, while good at handling numbers. The problem is that unless interaction with students is to be in multiple-choice format, the student's response must be treated as text. Since it is so difficult to program the computer to respond appropriately to text, there is a strong tendency to want to adopt the multiple-choice format when programming in these languages. Of course the multiple-choice format cannot be used with a proof-checker. Furthermore, the formulas of logic are text, not numbers, so all the major programming operations in writing a proof-checker must be carried out using the most primitive tools of BASIC and FORTRAN. Languages that are good at text processing, such as LISP, SNOBOL, TRAC, and even APL, are simply not used enough to be widely available in educational institutions.

What is the solution? For the moment, we think it is to bite the bullet and write programs in FORTRAN using the ANS standard, thus abandoning virtually all text manipulation capabilities. This requires skillful programming, and a taste for tedium. The ultimate solution must be for programmers of educational materials to demand and get a full variety of text-manipulation commands in the standards for transportable languages.

Troubles for Teachers

Adopting a proof-checker is a bother. It requires finding a program, arranging for terminal time for the students, teaching them to use it, and handling the inevitable problems that crop up with the system. For a teacher who has had no experience working with computers, the whole prospect can be a bit frightening, particularly since there is often no one to help out if anything goes wrong with the program, or if programming changes must be made. Ironically, students are better off in this respect. Quite a few can program in some language or other, and many more have had experience working at a terminal, if only to play computer games like FOOTBALL and STARTREK.

Our present system for disseminating computerized materials encourages teachers very little. Most teachers have little idea what it is like to work with a computer program, so they are not likely to commit themselves to use of a particular program unless they have seen it demonstrated. But demonstrations are not easy to arrange at a teacher's campus, since at least minor modifications of

the program will usually be necessary to get it to run there. It would probably be better to arrange for demonstrations during annual conventions, since a single set-up could have a large audience. The main problem, however, is that given present institutional arrangements, it is not in anyone's interest to give demonstrations. Developers of a program are funded, if at all, for *development*. The success of their work is not measured in terms of how widely their program is adopted. If success is measured at all, it is done by controlled psychological experiments that show the educational advantage of the system in the environment where it runs. Once the system has been constructed, and shown to be educationally effective, its developer turns to something else.

Even when the teacher has been convinced of the desirability of using a proof-checker, there are more problems to face. To obtain a program, the teacher will generally make use of CONDUIT, an organization which gathers and distributes educational programs written in BASIC and FORTRAN, generally at a cost of about $50. Programs from CONDUIT are available on tape or cards, and a complete listing of the program is supplied, along with fairly good documentation. The programs supplied are written in "transportable" versions of BASIC or FORTRAN, or at least their authors have been strongly encouraged to write programs according to certain standards.

So CONDUIT performs an extemely valuable service; and yet the teacher needs more than CONDUIT provides. When the materials arrive, there is still the need to load the program and fix whatever problems still remain. The teacher without experience is in no position to change the program, and must seek the help of someone on campus willing and able to get the program to run, and to diagnose and fix any problems involved in "fine-tuning" the program to the needs of the teacher. There are bound to be mistakes, or oversights, or simply decisions about how the program works that don't fit well with the teacher's methods or philosophy. On most campuses no one is employed to do this, and so the teacher has to rely on finding a person who is familiar with computers and willing to donate time.

Few institutions supply teachers with the backup they need to use a program in their class. Ideally, the people who supply this support should be the writers of the program being used, and not someone at the teacher's campus. Even with excellent documentation for a program, a person who does not understand logic, or the strategies used in the program's construction, is not likely to be able to do a good job of modifying a program for the teacher.

This suggests a new conception of the role and duties of authors of computerized materials. Part of their job, for which they should be paid at their usual rate, should consist of consulting with users of their programs. This has two advantages. First, it provides the support teachers need, and hence improves their motivation to try out the new materials. Second, it provides feedback to the designers of the program, feedback that will be of value in improving the product for everyone who uses it. Too often, programmers of educational materials presume that they can anticipate the needs and reactions of the students and teachers. The assumption is that if a program runs, and runs as the

designers intended it to, there is nothing more to be done. But if the program is to be used widely, the question of how it should operate has not been answered, and won't be answered, until there is adequate information on how well students and teachers handle it, and how changes might improve it. Valuable suggestions for redesign of the program are bound to result from carefully monitoring the actual use of the program in a real setting. This monitoring, evaluation, and revision of the program allows it to grow in a way that responds to people's reactions to it. Not only is it sure to improve the quality of the product, but it also involves the teacher and the students in the ongoing process of improving the system. As a result, they develop a sense of "ownership" in the materials they are using, instead of feeling that the program has descended upon them in an unchallengeable form.

Computerized materials have an advantage over textbooks in this respect. Editing a program, and putting out a new "edition," is much less difficult than re-publishing a book. In fact, we may expect that textbooks will eventually be stored in digital form, so as to take advantage of the flexibilities that already exist for computer programs. For the present, however, the danger is that commercial interests will enter the field of computer-assisted instruction, and begin producing educational programs that will play somewhat the same role as *Cliff's Notes* play now. Such an enterprise may soon be commercially feasible; already it is possible to buy display terminals with built-in computers for under $1,000, and there is every indication that costs for this sort of equipment will continue to drop, perhaps to as little as $100 per unit, paralleling the drop in prices for pocket calculators.[5] It will be unfortunate if commercial vendors come to dominate the field of computerized educational materials, since vendors have little incentive to adjust their programs to the needs of educators, once they have marketed a half-way workable product. The prospect of commercial intervention lends urgency to the suggestion of the preceding paragraph, that academic program authors should receive support for the ongoing revision and maintenance of their products.

Conclusion

We have tried to explain some of the major obstacles to the use of proof-checkers in teaching logic. The feasibility and value of these systems has already been demonstrated. The main problem now is to design systems and institutions that solve enough of the practical problems so that they will be widely adopted. We need to find ways of communicating the advantages of proof-checkers to teachers, and to provide the support they need in order to be confident that their initial investment of time and effort will yield a return. One way to help is to design systems that are easy to use, and which fit well with what goes on in the classroom. We hope the discussion of the first four obstacles will help show how this can be done.

In the last two sections, we have raised issues that apply to the use of computers in education in general: the need to promote wider distribution of

languages that are suited to the needs of instructional programmers, and the need to develop institutions that allow communication between program designers, teachers, and students. At present there is a large body of computerized course material, very little of which is widely used. Part of the explanation is that it is far easier to get funds to develop such material than it is to get money for developing the skills and institutions we need to make effective use of these systems. The latter activity does not fall neatly into the category of research, but rather lies at the interface between teaching and research. Until colleges and universities feel the need for better programming languages, and for better communication between teachers and program authors, the development of instructional programs will continue to be an "academic" exercise.

Appendix: Details on the Hint Giver

The structure of our Hint Giver falls into three main parts. (See Figure 3). The first is to evaluate the goals the student has entered at the bottom of the proof for errors. Our goal-stack evaluation routine (see Figure 4) will in fact check for four kinds of problems. First, it is a serious error for a student to enter a formula as a goal, without being able to say exactly how derivation of that formula will help in completing the proof. We must insist that the student supply justifications for all goals except the top one or two[6] on the stack. (As they are being supplied, these justifications will be checked by the matching subroutine—Figure 1.)

Once the student has supplied valid justifications for all non-top goals, it will be clear which goal steps do, and which do not, discharge hypotheses. This puts us in a position to look for a second kind of serious goal-stack error: planning the end of a proof in such a way that the final conclusion rests on one or more hypotheses not given in the statement of the problem. Next comes a check for an oversight, rather than an error: perhaps there is some easy way to derive one of the lower goals on the stack, without having to derive the top (few) goal(s) at all. Finally, it may be that some goal simply does not follow from assumptions available at its particular spot in the subproof structure, and this is of course a serious error.[7]

If there are no errors in the goal stack, then the hint algorithm begins the second process: the search for a rule, other than indirect proof, which can be used to derive the goal at the top of the student's stack (see Figure 5). This search forms the heart of the hint algorithm, and will make extensive use of a modified version of the matching subroutine (Figure 1). In searching for hints, neither the name of the rule, nor the numbers of the lines being operated on, will be given; rather, these are what we must find, if we can. What we *are* given is the conclusion to be derived, and its logical form provides the starting point for our search. Taking each rule in turn, we check to see whether the conclusion schema in this particular rule matches the logical form of the top goal on the stack. If not, then of course this rule cannot be used to justify the top goal, and we go on to the next rule. (More will have to be said later about what we mean by "next" here.)

Once we find a rule whose conclusion schema is of the appropriate form, the premise schema(ta) can be matched against lines already derived by the student. With many rules, this matching process will be necessary for the generation of useful advice. Specifically, a match on the conclusion schema will not be

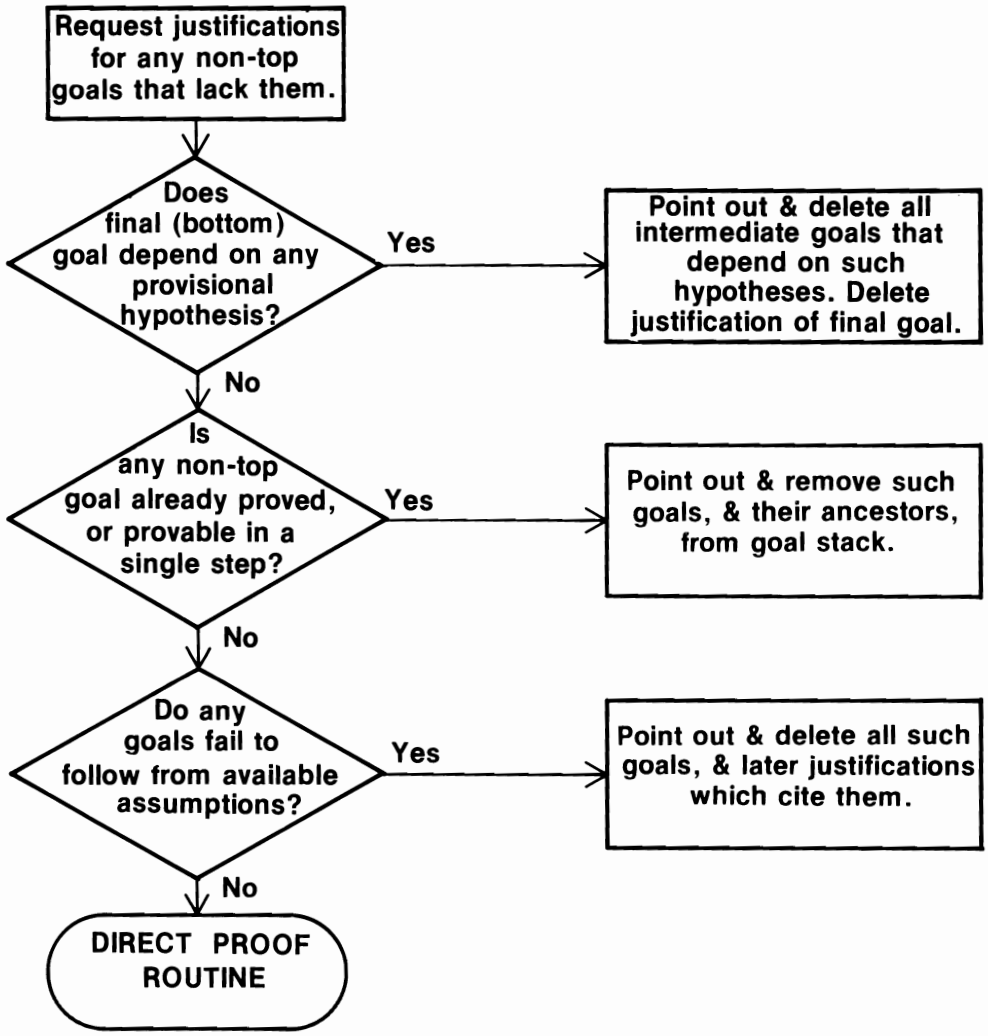

Figure 4: Goal-Stack Evaluation Routine

enough in the case of those rules whose premise schemata include metavariables which do not appear in the conclusion schema. Consider, for example, Constructive Dilemma: (A v B), (A $->$ C), (B $->$ C)/C. Any goal formula whatever will match the conclusion schema, C, but until we can determine which formulas (if any) play the roles of A and B in this proof, we cannot give any specific advice about which formula(s) the student should try to prove next.

By way of contrast, consider DeMorgan's Law: (– A& – B)/ – (A v B). A conclusion match alone is all we need to generate good advice here. If the student's top goal has the form – (A v B), then we know *exactly* which formula the student would have to use in justifying the top goal using DM.

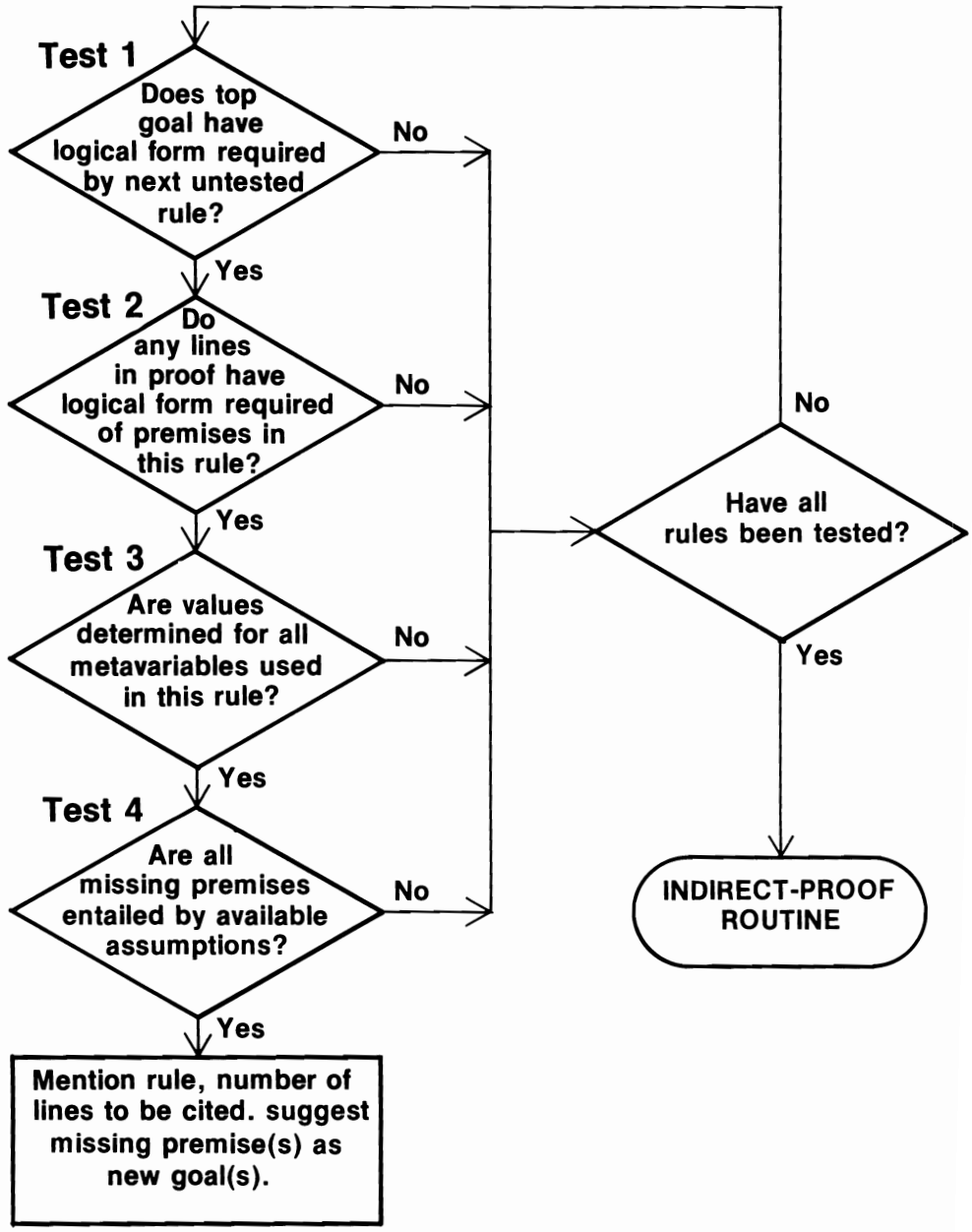

Figure 5: Direct-Proof Routine

An important function of the premise search, then, is to fix the values of all the metavariables used in stating the rule being tested. If this cannot be done, we cannot give *clear* advice; but success in evaluating the metavariables cannot by itself guarantee *good* advice. Suppose, for instance, that a student requests advice in the following situation:

1: P –> S HYP
2: S –> Q HYP
3: – R v Q HYP
4: P v R HYP
5: P –> Q HS 1,2
6: 99:

Suppose the direct proof routine is testing MP: (A –> B),A/B. Clearly the top goal, Q, matches the conclusion schema, B. The premise search would also show that line 5 matches the premise schema (A –> B), and this would suffice to determine the value of the hitherto indeterminate metavariable, A. The missing premise for MP, then, is P, and one is tempted to recommend that the student add P to the top of the goal stack. But while this advice would be clear enough, it would not be good advice, since P does not follow from the given premises. Thus the final test in the direct-proof routine must be a test of whether the advice we are about to give is workable. That is, do the available assumptions actually entail all the missing premises for the rule being tested?

We can now explain what "next rule" means in Test 1 of Figure 5. It is necessary to distinguish at least three classes of direct-proof rules, corresponding to three distinct phases in the operation of the direct-proof routine. DeMorgan's Law, in the variant mentioned above, is a representative of one important class of rules. We have already noted one interesting property of DM: if the conclusion match (Test 1) is successful, then we know immediately, without a premise match (Test 2), exactly which formula the student would have to cite in justifying the top goal via DM. But this particular variant of DM (i.e., (– A& – B)/ – (A v B)) has two other properties of even greater interest.

First, this rule, like other forms of DM, is *semantically reversible*; this tells us that if we already know that we can derive a formula of the form – (A v B) from a given set of assumptions, then the same assumptions can be used to derive (– A& – B), the premise in this rule. Thus suppose that a student requests advice on how to prove – (S v T), based on some assumptions or other. In evaluating the goal stack, the hint algorithm will already have verified that those assumptions really are sufficient to entail – (S v T). The direct-proof routine will discover that the conclusion schema for DM matches the student's top goal. As we saw before, no premise search is needed to determine what the student needs to prove in order to derive – (S v T) by DM, so Tests 2 and 3 can be bypassed. But in addition, by virtue of DM's semantic reversibility, we can also bypass Test 4: if – (S v T) follows from available premises, then so does the premise, – S& – T.

Second, this form of DM is what we shall call a *complexity-increasing* rule.

By this we mean that students will generally have more difficulty dealing with its conclusion than with its most complex premise. On its face, this rule *reduces* syntactical complexity, but this overlooks the fact that conjunctions are generally easier to derive, and to work with once derived, than are formulas involving disjunction. Students are likely to find – S& – T easier to prove than – (S v T). Hence it's good advice, in this context, to suggest – S& – T as a new goal.

In our view, a conditional is less complex (i.e., easier to get or work with) than a disjunction, but more complex than a conjunction. A biconditional is clearly more complex than a conditional, perhaps about on a par with disjunctions. In EMIL's notation, the biconditional operator has three characters (<– >), the conditional two (– >), and the conjunction one (&). Thus the complexity of a formula, as we see it, can be measured by counting its characters—with the proviso that the disjunction operator must be counted as three characters.

The general point about semantically reversible, complexity-increasing rules is this: Whenever the conclusion schema of such a rule matches the student's top goal, we can confidently advise use of that rule, without having to check whether the missing premise follows, and without having to search the top of the proof. Such rules permit us to bypass Tests 2, 3, and 4 in Figure 5.[8]

But while this is attractive from the standpoint of computing efficiency, it is not necessarily going to lead to the *best* advice in all situations. It will cost computing time, of course, to look at the top of the student's proof, but if the student has been making progress, then advice based on information about the bottom *and* the top of the proof is likely to be better than advice based only on information about the bottom of the proof. Also, the student's confidence may be subtly bolstered by hints that build on his or her work, and subtly undermined by hints that ignore his or her efforts. For these reasons, we believe that complexity-increasing (hereafter, CI), semantically reversible (SR) rules should be the last group to be processed by the direct-proof routine.

Before considering CI, SR rules, then, the direct-proof routine should look at non-SR rules of two kinds: complexity-reducing (CR) rules with more than one premise, and CI rules with any number of premises. With most rules in this class, we will need to examine the top of the proof in order to fix the values of one or more metavariables. (Addition, A/(A v B), is an exception.) Test 4 will also be necessary; only SR rules can bypass that phase of the direct-proof routine. Still, certain processing economies are possible in connection with Test 2, and these economies turn out actually to suppress a certain kind of bad advice which would otherwise be given.

Suppose, for example, that we are testing the rule of *modus ponens*: A, (A – > B)/B. No matter what the top goal is, it will have the logical form required by the conclusion schema of this rule. Moreover, the first premise we test from the top of the proof will match the premise schema A. We are in danger, then, of suggesting that the student try to prove (A – > B); advice which is almost sure to lengthen the proof unnecessarily.

We can suppress this bad advice, and at the same time eliminate a fair bit of processing, by rejecting multi-premise rules of this class whenever no line from the top of the proof matches the *longest* premise schema.[9] This amounts, in the case of MP, to a refusal to recommend use of this rule except where the conditional premise is already available. In rules where no longest premise exists, we shall have to match *each* premise schema against formulas at the top of the proof in order to assure good advice. Consider, for example, Constructive Dilemma: (A v B), A $->$ C), (B $->$ C)/C. If the top of the student's proof already contains a formula of the form (A v B), we may be justified in suggesting (A $->$ C) and (B $->$ C) as new goals (assuming, of course, that these formulas pass Test 4). But even if no disjunctive premise is available, CD may still be a good rule to use in justifying the top goal: if (A $->$ C) and (B $->$ C) are both available, the student should perhaps be sent in pursuit of (A v B). Thus, there is no *one* premise schema in CD that *must* be matched in order for CD to be a good bet, and an unabridged premise search is required.

So far, we have mentioned CI, SR rules, and non-SR rules which either increase complexity, or else have more than one premise, and reduce complexity. We turn now to the left-overs: rules whose conclusion schemata are neither more nor less complex than their most complex premise schema; one-premise rules that reduce complexity; and, if there are any, SR, CR rules with more than one premise. These three subclasses have something interesting in common: they involve premises at least as complex as the conclusions they yield. In general, then, even if the top goal happens to match the conclusion schema for one of these rules, we should not suggest that the student add the missing premise(s) to the top of the goal stack, since these missing premises will be just as hard to derive, generally speaking, as the formula *now* on top of the stack. We recognize that there will be special circumstances in which this will not be true, and in which the best advice will be to use a "left-over" rule, after deriving a missing premise. But we believe that such circumstances will rarely arise, and that they will be costly to detect, in terms of processing time.

Accordingly, we propose to recommend that the top goal be derived by a "left-over" rule when, but only when, the student has already derived all necessary premises. Thus Tests 3 and 4 can be passed over when we are processing "left-over" rules, and Test 2 must be tightened up to require that all needed premises be found. This class of rules must be processed first, since any advice involving these rules will be of excellent quality: such advice will tell the student how to derive the top goal in a single step, without setting any new goals. Figure 6 summarizes this discussion of the phrase "next rule" in Test 1, Figure 5.

Before we turn to the indirect-proof routine, we wish to point out an additional processing economy that could be achieved in the direct-proof routine—at a price. Test 2 promises to be very time-consuming, and some students may prefer to take over some of the work of premise-searching, in return for shorter waits for hints. For these students, we plan to build into our hint algorithm a keyword that will "switch off" Test 2 in certain situations. First, the switch would eliminate Test 2 from the goal-stack evaluation routine

(Figure 4). Second, the switch would altogether eliminate consideration of Class A ("left-over") rules from the direct-proof routine. Finally, the switch would cancel the premise search as soon as all metavariables had been evaluated, so that a formula already derived might be listed as a "missing" premise. We don't know how much damage this switch will do to the quality of advice given by the hint algorithm, but we suspect that premise-searching is one of the things students do best.

	Class definition	Processing economies (see Figure 5)
A	CR rules of one premise, rules that are neither CR nor CI, and any multi-premise SR, CR rules there may happen to be.	Tests 3 and 4 bypassed; Test 2 must be tightened up to require finding of all needed premises
B	Non-SR rules that are either CR and multi-premise, or else CI.	For multi-premise rules, Test 2 can be abbreviated if one premise is longer than any other.
C	SR, CI rules.	Tests 2, 3, and 4 bypassed.

Abbreviations
CR: complexity-reducing
CI: complexity-increasing
SR: semantically-reversible

Figure 6: Classes of Direct-Proof Rules, in Order of Processing.

We come now, finally, to the "last resort" of proof construction: indirect proof (Figure 7). Once an assumption for indirect proof has been made, the entailment test (Test 4, Figure 5) becomes useless as a means of discerning and rejecting bad advice. Before, we could eliminate some possible "new-goal" suggestions on grounds that the putative new goal would not be derivable from available premises. But in indirect-proof situations, the available premises entail a contradiction, and consequently they entail *any* formula we might consider putting forward as a new goal. We know, of course, that we're looking for a contradiction, but that still leaves us with infinitely many possible goals, and clearly some contradictions will be easier to prove than others. How can we steer the student onto the contradiction which is easiest to prove?

It is unlikely that our indirect-proof routine represents the best answer to this question; indeed, we hope it will be possible to improve on this routine. Nonetheless, this routine gives pretty good advice in many indirect-proof situations, and it may at any rate provide a starting point for an ongoing discussion of the problem of goal selection in indirect-proof contexts.

After informing the student that indirect proof is called for, the indirect-proof routine begins working at the proof on its own, using a "scratchpad" that

is invisible to the student. First, the routine applies all the CR rules, in all possible ways, to lines which the student has already derived; the resulting formulas, if any, are placed on the scratchpad.

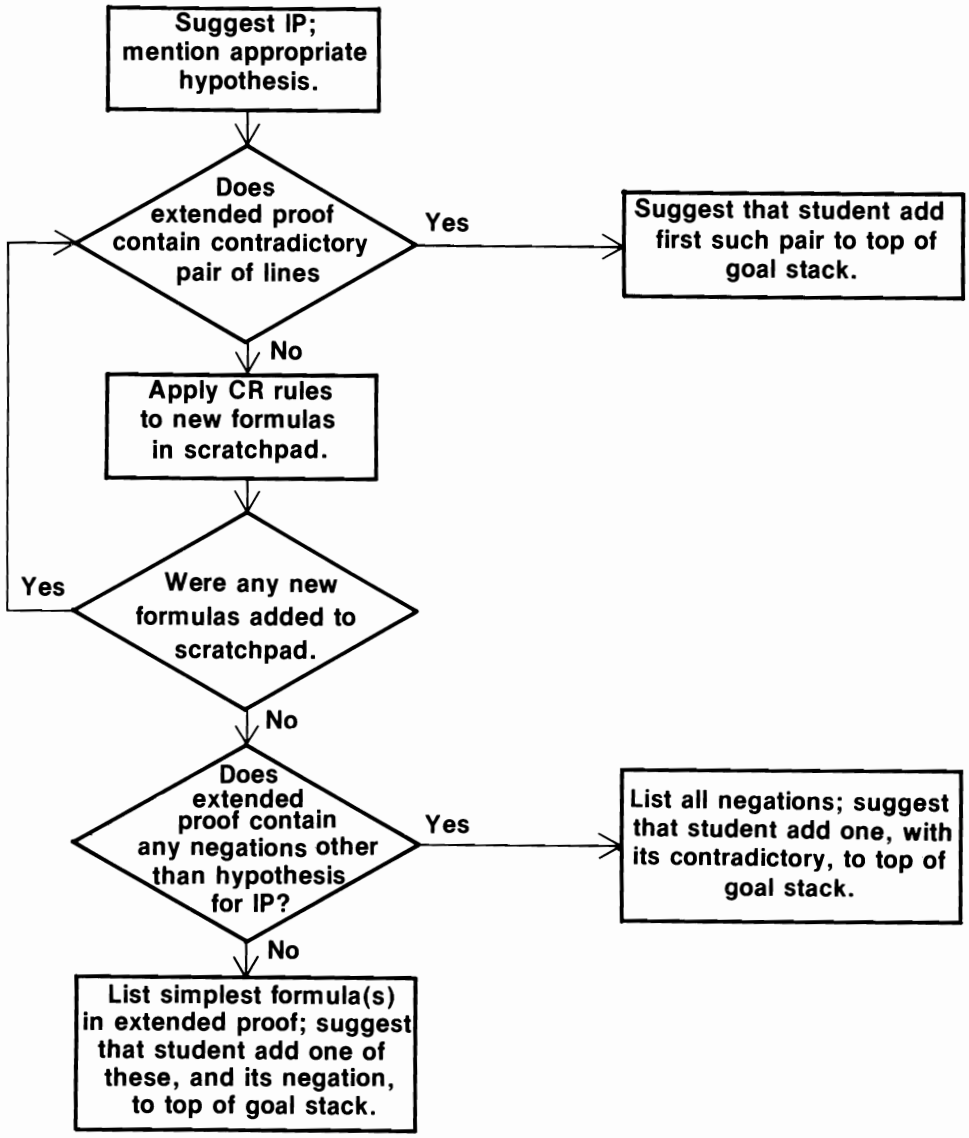

Figure 7: Indirect-Proof Routine

Looking now at the extended proof (the top of the student's proof, augmented by the scratchpad), the routine searches for a pair of lines, one of which is the negation of the other. If the routine finds a contradictory pair, it naturally suggests that the student try to prove that particular contradiction. If no contradictory pair is found, the routine again tries to add new formulas to

its scratchpad, using the CR rules. From now on, however, each rule application must involve at least one of the formulas that entered the scratchpad during the *previous* round of CR applications. (This restriction is intended to suppress repetitions in the scratchpad.) If any new formulas are added to the scratchpad in this way, we search the extended proof for their contradictories.

By keeping track of which rules have been applied during the creation of the scratchpad, we can insure that this process never enters an infinite loop. The restriction of complexity-*reducing* rules is intended to guarantee termination, sooner or later. Of course this limitation may be excessive, leading to premature termination; perhaps we could improve the quality of our advice by allowing the use of some non-CR rules, without jeopardizing the finitude of our scratchpad. One must steer a course between making the scratchpad too large (infinite, or finite but garbage-laden), on the one hand, and on the other, of making it too small, so that it fails to include the most easily provable contradiction of all.

If and when the iteration ceases, we may still not have found a contradictory pair. In that case, we must content ourselves with giving advice of more dubious quality. First, we look over the extended proof (barring the hypothesis for *reductio*) to see whether any of the formulas in it are negations. If we discover any negations, we list them all, and suggest that the student try to prove one of them, together with its contradictory. Should the extended proof be devoid of negations, we list the class of minimally complex formulas,[10] and suggest that the student prove one of these formulas, along with its negation.

One unfortunate feature of our indirect-proof routine is that is may require the student to make a choice among several alternative new goals. We could, of course, make the choice for the student, in some arbitrary way, but until we can discover some principled basis for making the choice, we believe the student deserves the opportunity to make his or her own choice—arbitrarily, if need be. At least our indirect-proof routine has the virtue of cutting an initially infinite set of alternatives down to finite, and hopefully manageable, size.

References

[1] Falk, A. and Houchard, R. "Computerized Help in Finding Proofs," (xerox) Western Michigan University, Kalamazoo, Michigan.

[2] Fitch, F. *Symbolic Logic, an Introduction*, Ronald Press, 1952.

[3] Goldberg, A. "Computer-Assisted Instruction: The Application of Theorem-Proving to Adaptive Response Analysis," Technical Report #203, May 25, 1973, Institute for Mathematical Studies in the Social Sciences, Stanford University, Stanford, California.

[4] Jeffrey, R. *Formal Logic: Its Scope and Limits*, McGraw Hill, 1967.

[5] Lemmon, E. J. *Beginning Logic*, Hacket Publishing Company, 1978.

[6] Moor, J. and Nelson, J. "Computer-Assisted Instruction in Logic: BERTIE," *Teaching Philosophy* 2:1 (Spring 1977), pp. 1-6.

[7] Pospesel, H. *Predicate Logic*, Prentice-Hall, 1976.

[8] Suppes, P. *Introduction to Logic*, van Nostrand, 1957.

Notes

Paul Mellema is primarily responsible for the Appendix of this paper, while James Garson is primarily responsible for the body of the paper. Thanks to George Thompson for helpful suggestions.

1. We are following the tradition (established by the authors of BERTIE) of honoring our heroes in naming EMIL for Emil Post. Post showed how to mimic the operation of any formal system within a universal system. In showing that the universal system exists, Post sketched a basic strategy to be used in designing a universal proof-checker. Although EMIL's design departs a good deal from Post's work, the spirit remains.

2. The matching subroutine may also be useful in checking whether students have correctly translated an English sentence into logical notation, since it recognizes the logical form of a wff, regardless of the sentence, or predicate, letters used. A good translation checker would also check for logical equivalence between the expected pattern and the student's translation.

3. By "top of the proof," we mean everything in the proof except goals which are still pending, and have not as yet been linked, by a justificatory chain, to the starting assumptions (if any) of the problem.

4. There is another reason for prefering to teach natural deduction systems, apart from the practice they give in creative thinking. These systems organize a proof in a way which mirrors the way in which people ordinarily communicate their arguments.

5. Cost could be reduced even further by using a standard TV receiver as the display device, as has been done by makers of programmed electronic games and household computers.

6. In the example given in the main body of the paper ("Unnatural Proof Constructions"), goal lines 75 and 98 both had to be entered without justifications.

7. Since first-order logic is undecidable, the test for this error obviously cannot be extended, with total reliability, to systems of quantificational inference. Still, it would probably be useful to verify validity of proposed deductions in universes of up to, say, four individuals.

8. Strictly speaking, there are semantically reversible complexity increasing rules for which we would need Test 3 to obtain good advice. However, these rules do not appear in logic text books, and so we will not require Test 3. An example of such a rule is A&(B v $-$B)/A&($--$ C v $-$ C).

9. There is no need, in this context, to assign special weight to disjunction.

10. Complexity here, defined as before: number of characters, with the disjunction operator getting triple weight.

James W. Garson and Paul Mellema, Department of Philosophy, University of Notre Dame, Notre Dame, Indiana 46556