Logic and Computers at Georgetown

ABSTRACT March 1990

Steve Kuhn, Brendan Henry and Sharon Pizzo

At Georgetown University we have begun to use microcomputers to help in teaching logic courses. We believe that the programs we have written and planned have some novel features that may be of interest to others in the academic community. Our efforts can be divided into two parts--a derivation checker (for which we have a reasonable implementation running) and a truth checker (which is now only partially implemented).

The derivation checker

The derivation checker is written for the system in Lemmon's *Beginning Logic*, although it could be easily adapted to other systems. It was written in Borland Corporations's *Turbo Prolog*. Georgetown students, although very bright, are not especially attracted to the idea of using computers. (We have, for example, several hundred pre-law students, but only three or four computer science majors.) Furthermore, as might be expected, the students who have the most difficulty with the logic course, tend to be the ones most uncomfortable with computers. Thus it was a high priority for us to design the program so that it would be easier to write derivations on the machine than to use paper and pencil, and so that most of the benefits could be enjoyed with minimal instruction. Some of its special features are the following:

1. Automated line entry.

A derivation line consists of a *line number*, the *formula* of that line, its *rule* of justification, its *source* (i.e., the line number or numbers to which the rule was applied to get the formula), and the *assumptions* on which the line rests. In our program the line number is provided automatically and the user is prompted to enter each of the other parts in turn. Any of these parts, however, may be left to the program to provide. For example, instead of typing out a long formula the user can just indicate that she wants to do a modus ponens on lines 6 and 20. This feature is particularly easy to implement in prolog--we merely leave unbound all the parts of the derivation line that the user does not provide. As a result, if the user does not provide enough information to uniquely determine the line, the program will make the first match it can. If the user merely specifies that &-elimination is to be performed, for example the program always delivers the left conjunct of the topmost conjunction. If a formula is entered with no rule, source, or assumption it gets treated as an assumption. The lazy user quickly learns which match is made when he provides insufficient information. In our next version we intend to allow the user the option of rejecting a match, so that the computer will search for another way to satisfy the user's intentions.

We believe that this flexibility in line entry has advantages for both the quicker and slower student. The quicker student is relieved of the tedium of recording information that is obvious to him. The slower student is relieved of the necessity to "get everything perfect" while building confidence that he understands the basic ideas. He can also use this feature as self-teaching device by silently guessing, say, the assumptions to carry down and then checking to see whether the machine delivers what he expects. This would seem to be a useful mean between reading a derivation in the text (where the immediate accessibility of all information preempts a serious consideration of alternatives and may engender an illusion of understanding) and writing out all the details oneself (which seems to require that the student already know the principles he is trying to learn). It might be thought that our automated line entry makes things too easy. If the user were to enter just the formula and assumptions, for example, he would be asking the machine to do his work--he is turning a theorem checker into a theorem prover. In practice, however, when a match is not obvious the backtracking needed to find it quickly generates a stack overflow. The program traps this error and the user gets a message asking him to redo the line, providing more information. Still, we have come to feel that for some students at some stage in learning there may be virtue in requiring that all information be provided. Our next version will contain a menu item allowing the user to disable automated line entry.

2. Sequent bank

Lemmon thinks of a derivation of A from Γ as a proof of the sequent $\Gamma \vdash A$. His system includes a shortcut rule that permits use of substitution instances of previously proved sequents. For example, if P has been previously derived from PvQ and -Q and the current derivation contains formulas Sv-T and –T then, by the shortcut rule, one can enter S with appropriate assumptions. Our checker allows the user, when he has finished a derivation, to add a result to the "sequent bank" where it will be available for use in connection with Lemmon's shortcut rule. The sequent bank initially includes all the results in the text with Lemmon's numbering. For the more useful sequents we have added our own mnemonic labels. Entering "SI" when prompted for a rule opens a window containing the sequent bank. The user can scroll through the sequents and, with a single keystroke, select the one he wishes to use. This is considerably more efficient than thumbing through the text. Alternatively, the shortcut rule can be invoked by citing the sequent directly by number or label without opening the special window. Sequents added by the user are added to Lemmon's list sequentially and the user is permitted to give them mnemonic labels.

The sequent bank provides an extremely simple means of tailoring this program to other systems. For as long as other rules are not improper (i.e., as long as they are truth-preserving) they can be viewed as instances of Lemmon's SI rules. For example, to tailor the program for an axiomatic system one need only add to the sequent bank the sequent $\vdash A$ for each axiom A and the additional sequent $A, A \rightarrow B \vdash B$ for modus ponens. To take full advantage of this idea, of course, requires that there be a simple way of disallowing the use of all of Lemmon's rules other than SI. But the ability to selectively "disable" rules would be a useful feature even for teaching the Lemmon system and one quite easy to incorporate in our program. Future versions will do so. Thus instructors with no programming experience should be able to easily adapt our program to other systems.

3. Convenient deletions

When one has made a mistake, one can choose to begin the derivation again, begin from a specified line, redo the current line. A relatively simple feature that would give the machine an advantage over paper and pencil would be to allow the user to delete a previous line, understanding that this would cause all subsequent lines depending on that line to be removed and the remaining lines, sources and assumption numbers to be renumbered appropriately. Future versions will incorporate this feature.

4. Helpful error messages

The program recognizes and calls attention to some common mistakes. For example, if the user affirms the antecedent when he intends to do modus ponens, the fallacy is pointed out. More such special messages will be added as we discover common errors. One interesting possibility is to allow the machine to help here by recording the faulty inferences it refuses to accept.

5. Guidance on strategy

In future versions, we intend that the program will offer some guidance on strategy. In natural deduction systems, some of the most common strategic errors have to do with inappropriate assumptions. In Lemmon's system there are three rules by which assumptions can be eliminated: conditional proof, reductio ad absurdum, and v-elimination. Each of these is associated with a particular derivation strategy. It is useful practice never to add a new assumption without knowing how that assumption is to be eliminated. We intend to that, when the strategy guidance feature is active, entering a new assumption will cause a special window to open. The user will be prompted for a rule (by which the assumption is to be eliminated) and a subgoal (whose derivation is necessary for that elimination to occur) and from these the program will compute an intended result. (For example, if the user assumes **P** thinking of the rule **RAA** his subgoal is a contradiction and the intended result is -P.) When the user has verified this information, it will be stored and the window closed. If the user then succeeds in deriving the subgoal from the extra assumption a tone will remind him that he has done so. The user will also be able to ask on any line to be reminded about why the assumptions of that line were introduced. Doing so will display the windows associated with these assumptions.

The truth checker

We believe that a weakness of many introductory logic texts, including *Lemmon*, is the relatively little attention paid to semantic matters. At Georgetown we try to supplement the text with some discussion of models and truth. It would be useful to us for our program to provide help in this part of the course as well. The program we envision will enable the user to construct finite models and check the truth of a formula in a given finite model. (Infinite models would seem to raise special problems for computerized logic teaching.) We plan several independent components.

1. Pictures to models

To inculcate the distinction between objects in the world and their names and descriptions in language, we take as individuals odd extended ASCII characters (and perhaps concatenations of these). The formal language, of course, uses more familiar characters--lower case letters from the middle of the alphabet are proper names, lower case letters from the end of the alphabet are individual variables and capital letters from the middle of the alphabet are predicate letters. A *picture* is just an arrangement of individuals on the screen. We represent a *model* by a collection of windows, one containing the individuals in the "domain" and each of the others providing the interpretation for a single non-logical symbol. Windows for proper names contain single individuals; those for unary predicates contain sets of individuals; those for binary predicates contain sets of pairs of individuals and so forth. Our program's first semantic component will attempt to help the user understand the relation between an informal interpretation of a predicate letter (as an open sentence of English) and its formal interpretation in a model (as a set of tuples). For example, given a picture, and the open sentence "x is to the left of y and y is to the left of z" the user will be expected to move the cursor successively to three symbols in the picture satisfying that description. When she has done so, that triple will be added to the appropriate window and she will be prompted to look for other appropriate triples. Alternatively, the user can be given a picture, a model and informal interpretations for some names and predicates and asked whether the model correctly represents the picture.

2. Game theoretic interpretation of quantifiers

Given a model and a prenex formula, the user plays a game against the computer to see whether the formula is true in the model. Play starts with the leftmost quantifier of the formula. An existential quantifier indicates that it is the user's turn, and a universal quantifier that it is the computer's turn. A turn consists in a selection of an individual from the domain. The user tries to make the formula true and the computer tries to make it false. The user can win only if the formula is true. If the user loses, he may ask to take back his moves. If he is convinced that winning is impossible he may guess that the formula is false. If he is wrong, the computer will show him the strategy he missed.

3. Validity and satisfiability

Given a formula and a "standard" picture (i.e., a list of available individuals) the user will try to construct models that satisfy or falsify the formula. The method of construction is similar to that described in 1 above. For each name the user points to an individual. For each predicate he points to a series of tuples. When an individual pointed to is added to a name or predicate window it is also added to the domain window if necessary. When the model is constructed the program checks to see whether it does the job and the user is given appropriate comments. Alternatively, the user can be presented with model and formula and asked to evaluate truth.

Comments welcome.

Dept of Philosophy
Georgetown University
Washington, D.C. 20057