Intelligent Tutoring Systems

John R. Anderson, C. Franklin Boyle, Brian J. Reiser

Summary. Cognitive psychology, artificial intelligence, and computer technology have advanced to the point where it is feasible to build computer systems that are as effective as intelligent human tutors. Computer tutors based on a set of pedagogical principles derived from the ACT* theory of cognition have been developed for teaching students to do proofs in geometry and to write computer programs in the language LISP.

Computer systems for intelligent tutoring are being developed to provide the student with the same instructional advantage that a sophisticated human tutor can provide (*1, 2*). A good private tutor understands the student and responds to the student's special needs. From its beginnings, the computer has been viewed as capable of providing such instruction, thereby having the potential to improve the quality of education. Of particular importance is the improvement of training in the mathematics and science topics that are requisite for entrance to the scientific community and to the high-technology world.

There are now over 10,000 pieces of educational software available. Almost all of this software can be classified as computer-assisted instruction (CAI) in contrast to intelligent computer-assisted instruction (ICAI) or programs that simulate understanding of the domain they teach and that can respond specifically to the student's problem-solving strategies. A large fraction of CAI software is of low quality and accounts for much of the teacher disenchantment with the computer (*3, 4*).

There have been attempts to bring artificial intelligence techniques to bear in development of ICAI (*2, 5*), but this has been viewed as impractical and has been largely relegated to the research laboratory. One of the reasons was the high cost of ICAI. It was common to require a million-dollar machine to interact with one student, and often the response time of the machine was slow. A second reason was the large amount of time associated with creating educational software. It is thought to take 200 hours to create 1 hour's worth of conventional CAI, and the time associated with ICAI is thought to be an order of magnitude greater. Finally, there was no established paradigm for enabling students to acquire knowledge. Early ICAI efforts often were ill-focused attempts to interact intelligently with the student without any clear understanding of the impact of those interactions on learning.

These obstacles to past efforts at ICAI

The authors are on the staff of the Advanced Computer Tutoring Project, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.

are now being overcome. The cost of computing hardware is dropping rapidly. Soon personal computers will be able to provide intelligent tutoring. For instance, the personal computers envisioned for use at the Carnegie-Mellon campus (*6*) in 1986 will be adequately powerful. Furthermore, advances in artificial intelligence techniques have provided more efficient methods for achieving intelligent programs.

Advances in artificial intelligence and cognitive psychology have also meant real gains in the time to create instructional lessons. For instance, we can create instructional lessons at a rate that is faster than the 200 to 1 typically cited for conventional educational software. This is because ICAI can be generative; that is, it is not necessary to specify every interaction with the student, but only the general problem-solving principles from which these interactions can be generated.

Finally, advances in cognitive science have provided a theoretical basis for designing educational software that can be effective. We now have models of how successful students perform various cognitive tasks (*7, 8*). This enables one to be precise about instructional objectives for a particular course of study. Furthermore, current theories address the issue of how the student acquires new cognitive skills. The learning principles derived from these theories provide the direction needed in the design of instructional software. We have based our work on the ACT* theory of cognition (*9, 10*). In this article, we review briefly the assumptions from the ACT* theory that are relevant to the design of tutoring systems and then describe our approach to intelligent tutoring based on this theory. We present two examples of this work, a tutor for high school geometry and one for LISP programming.

The ACT-Based Approach to Intelligent Tutoring

The ACT* theory has been embodied in computer programs that simulate many aspects of human cognition. The ACT* theory, which is an attempt to identify the principal factors that affect human cognition and organize them into a complete cognitive theory, consists of a set of assumptions about a declarative memory and a procedural memory. We have found that only certain aspects of the theory are relevant to the tutoring of cognitive skills—in particular, the procedural assumptions.

The procedural component in the ACT* theory takes the form of a production system. A production system is a set of rules in which each rule represents a unit of a skill. Productions are used in many cognitive theories (*11, 12*). Much of human cognition appears to unfold as a sequence of actions evoked by various patterns of knowledge. These steps of cognition are given by rules that specify which actions to perform under a particular set of conditions. An English approximation of a production from one of the ACT* computer simulations for proving a theorem of geometry is the following:

IF the goal is to prove $\triangle UVW \cong \triangle XYZ$,
THEN set as subgoals to
1) prove $\overline{UV} \cong \overline{XY}$
2) prove $\overline{VW} \cong \overline{YZ}$
3) prove $\angle UVW \cong \angle XYZ$

This is a backward inference rule that embodies the side-angle-side rule of geometry. The rule says that when the goal is to prove a pair of triangles congruent, that goal can be achieved by trying to prove corresponding pairs of segments and their included angles congruent. The theory does not claim that the production exists in this form in the student's head, but rather that the student's thought processes follow these rules.

One can also have forward inference rules such as:

IF the goal is to make an inference from the facts that $\overline{XY} \cong \overline{UV}$, $\angle XYZ \cong \angle UVW$, and $\overline{YZ} \cong \overline{VW}$,
THEN infer that $\triangle UVW \cong \triangle XYZ$ because of the side-angle-side postulate.

We have successfully used rules like these to simulate the sequence of the inferences (correct and incorrect) that students report making in trying to solve a geometry problem.

As these examples illustrate, productions in the ACT* theory are goal-directed; that is, their conditions include a specific goal. These productions can apply only when a goal is set. This goal-directed character of cognition proves to be the key to much of the tutoring effort. It is critical for the student to be aware of the goals to be set and achieved to solve a problem.

The conditions of these productions contain patterns that must match information held in the student's working memory. Working memory, according to the ACT* theory, stores what the problem-solver currently knows about the problem; furthermore, the capacity to maintain information in working memory is assumed to be limited. It is possible that the capacity required for the solution of a particular problem will be exceeded and thus that critical information for the matching of a production will be lost. This can result in the failure to execute the appropriate production, the execution of an inappropriate production, or an error in executing the production. Many errors of learners are due to failures of working memory rather than to failures of understanding (*13*).

A major effort in our tutoring work is therefore concerned with helping students to manage working memory load. This is accomplished by having the tutor encode on the computer screen much of the information that a student is likely to forget. This enables the student to solve the problem more easily and to learn from that problem-solving effort.

In the ACT* theory a learner becomes more skilled at a domain by acquiring new productions that encode special rules for solving problems in that domain. "Knowledge compilation" is the name given to the learning mechanisms by which new productions are acquired. We have used a computer implementation of this knowledge compilation mechanism to simulate the way students learn in a domain (*7*). The basic feature of this mechanism is that it provides new rules that summarize many of the productions for the solution of a problem in an episode of learning. Therefore, the next time the student encounters a similar problem-solving context, these new rules can produce a more efficient solution, one that involves less trial-and-error search.

The technical details of knowledge compilation are not important for our present purpose; what is important is to emphasize that new productions are formed only during problem-solving. This means that instruction is effective to the degree that it can be integrated with problem-solving. Therefore, in our tutoring programs, formal instruction is made part of the problem-solving rather than preceding the problem-solving.

We have briefly reviewed four features of the ACT* theory—use of productions, goal structure, working memory limitations, and knowledge compilation—that are the key to the tutoring efforts described below. Implications of this theory for tutoring include making the goal structure explicit, minimizing the working-memory load, and giving instruction in the problem-solving context. Another important implication of these principles is that students should

be given immediate feedback about their errors. This will make it easier for the student to integrate the instruction about errors into the new productions that they form.

These observations point to the value of a private tutor who can observe a student's problem-solving, provide the right instruction at the right moment, correct errors, and identify the problem-solving goals. There is evidence that private human tutors can be very effective at instruction in domains that have a significant problem-solving component. For instance, when we compared the teaching of programming by human tutors with classroom instruction of programming, we found a four-to-one advantage for the private tutor, as measured by the amount of time required for students to get to the same level of proficiency. Bloom (*14*) in his comparisons of private tutoring with classroom instruction of cartography and of probability found that 98 percent of the students with private tutors performed better than the average classroom student, even though all students spent the same amount of time learning the topics. The poorest students benefited most. There was little difference in the achievement levels of the best students under the two conditions.

From these general observations about the effectiveness of private tutoring and our own theory, we developed a general paradigm for providing students with individualized tutoring, which we call "model tracing." The model-tracing paradigm is built around having a model of specific productions for the correct solution of the problem by the student (called the "ideal model") and productions for the errors students can make (the bug catalog). The tutor infers which rule the student applied by determining which one matches the student's response. If it is a correct response, the tutor is quiet and continues to trace the student's problem-solving. If an incorrect response has been given, the tutor interrupts with appropriate remedial instruction. Other possibilities are that the student does not know what to do next or that the student's behavior matches no production, correct or incorrect. Usually, this occurs when the student is confused. We have found that the best thing to do in such situations is to tell the student what to do next. If this is explained properly, the student is often able to get back on a right track. In the next two sections we present the geometry and LISP tutors we have developed according to this model-tracing paradigm.

The Geometry Tutor

The geometry tutor (*15*) is based on our earlier work on the problem-solving strategies underlying the generation of proof in geometry (*16*). This tutor is based on a number of principles derived from our learning theory—the use of an ideal model, use of a proof graph to represent problem structure, instruction in context, and immediacy of feedback.

The Ideal Model for Generating a Geometry Proof

Figure 1a illustrates a geometry proof problem as it is initially presented to a student by the tutor. This problem is considered relatively complex for high school students. In it the student has to prove the statement printed at the top of the screen and is given the statements at the bottom of the screen ("M is midpoint of $\overline{AB}$" and "M is midpoint of $\overline{CD}$"). As in high school geometry textbooks, the student is allowed to assume that any points that appear collinear are collinear, but nothing else can be assumed from the diagram.

At any point in the solution of the problem shown in Fig. 1, a number of inferences can be made. For instance, from the given fact that M is the midpoint of $\overline{AB}$, it is possible to infer that $\overline{AM} \cong \overline{MB}$. It is also possible to infer that $\angle AMF \cong \angle BME$ because they are vertical angles. The possible inferences can be ordered according to aptness, the first of the above inferences being apt in this context, but the second one not.

In this type of problem-solving, students also reason backward from a statement to be proved to statements that will prove them. Thus, a student can reason backward from the goal of proving M is the midpoint of $\overline{EF}$ to the subgoal of proving $\overline{ME} \cong \overline{MF}$ by applying the definition of midpoint. It is then possible to reason backward from this subgoal. For instance, the student might reason backward from the goal of proving $\overline{ME} \cong \overline{MF}$ to the subgoal of proving $\triangle AME \cong \triangle BMF$ by applying the rule that corresponding parts of two triangles are congruent if the triangles are congruent. Alternatively, a student might reason backward from the goal of proving $\overline{ME} \cong \overline{MF}$ to the subgoals of proving $\overline{ME} \cong \overline{AM}$ and $\overline{AM} \cong \overline{MF}$ with the intention of using the transitive property of congruence to deduce that $\overline{ME} \cong \overline{MF}$. Again, these backward inferences can be ordered as to their aptness with the first two inferences being quite apt in this context, but the last one not.

The aptness of an inference is not an absolute property of the rule of geometry that authorizes it. Instead, as indicated above, the aptness of an inference depends on the context in which it occurs. As another example, in this problem it is not strategic in reasoning forward to make the inference that $\angle AMF \cong \angle BME$. However, another inference about vertical angles, $\angle AMC \cong \angle BMD$ is quite apt, particularly after the student establishes that $\overline{AM} \cong \overline{BM}$ and $\overline{MC} \cong \overline{MD}$. Then the congruences of the two pairs of sides and the congruence of the angles can be used to show that $\triangle AMC \cong \triangle BMD$ by the side-angle-side postulate.

Thus, our ideal model for generating proofs in geometry involves both forward and backward inference rules with contextual restrictions. The following rule of forward inference makes use of the congruence of two vertical angles. This conclusion will then enable a side-angle-side inference to be made.

IF	$\overline{XY} \cong \overline{UY}$ and $\overline{YZ} \cong \overline{YW}$ and there are triangles $\triangle XYZ$ and $\triangle UYW$ where X, Y, and W are collinear points and U, Y, and Z are collinear points
THEN	infer $\angle XYZ \cong \angle UYW$ by vertical angles.

As an instance of a contextually bound backward rule, consider the following:

IF	the goal is to prove two lines parallel and there is a transversal
THEN	set as a subgoal to prove that alternate interior angles are congruent.

The ideal model contains 200 such rules ordered according to aptness. The model executes the best inference rule that applies in a situation, whether that is a backward or a forward rule. This system generates proofs for all of the problems in the high school geometry topics we have been working with, and the proofs are like those generated by human subjects. Not all of the inferences the system makes are part of the final proof, but when it deviates from the final proof, it deviates in the way we have observed in human subjects.

In summary, the ideal model is an effective and human-like proof system that contains a set of rules for making the most reasonable inference in a particular context. This ideal model defines what we are trying to get the student to emulate.

The Proof Graph and the Goal Structure

It is important to communicate to the student the logical structure of a proof and the structure of the problem-solving process by which a proof is generated. Figure 1, a to c, illustrates the proof graph that we have developed for this purpose. The proof graph is shown at the beginning of a geometry proof, in the middle of the proof, and at the end of that proof. Figure 1a illustrates the initial presentation of the problem. The student can reason forward from the given conditions and backward from the statement to be proved. The student adds to the graph by a combination of pointing to statements on the screen and by typing information. Each logical inference involves a set of premises, a reason, and a conclusion. Reasoning forward, the student points to the premises, types the reason, and points to the conclusion (if it is already on the screen) or types it. For instance, the student might point to the premise "M is midpoint of $\overline{AB}$," type the reason "definition of midpoint," and type the conclusion "$\overline{AM} \cong \overline{MB}$." Reasoning backward, the student points to the conclusion, types the reason, and then provides the premises.

Figure 1, b and c, shows some of the possible states in the development of a proof. The student is finished when there is a set of logical inferences connecting the given statements to the statements to be proved. Figure 1b illustrates how inferences can be made from the top and the bottom to meet in the middle. Figure 1c shows how the screen looks when a student achieves a final proof; this student made some inferences that were not part of the final proof.

One function of this formalism is to illustrate the structure of a complete proof. High school students typically do not understand how the steps of a proof fit together and find this structure helpful. The proof graph also concretely illustrates critical features of the problem space—namely, that inferences can be made in both forward and backward modes, that there are points at which the student must choose among several inference rules, and that the ultimate goal is a well-formed logical structure.

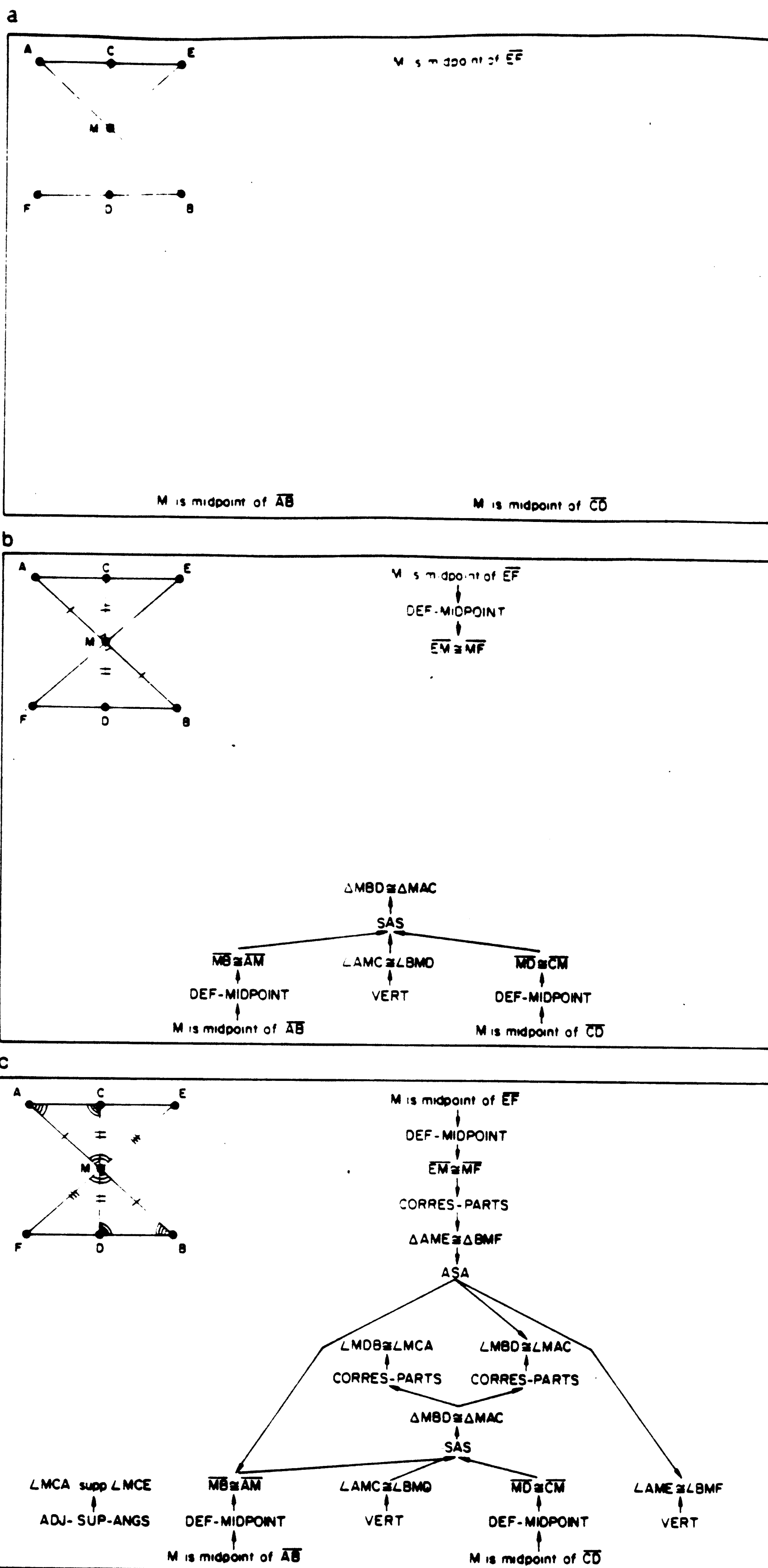

Fig. 1. (a) The geometry tutor's initial representation of the problem; (b) a representation in the middle of the problem; and (c) a representation at the solution of the problem; SAS, side-angle-side.

Instruction in Context

All of the instruction with the geometry tutor is provided in the context of solving problems. Only one concept (like the side-angle-side rule) is introduced at a time and it is accompanied by problems

that make use of the concept. The tutor does not allow a student to move to new concepts until he or she shows mastery of the current concepts. This instruction mode differs from instructional modes in which lectures are separate from problem-solving. Our knowledge compilation theory implies that it is critical for instruction and problem-solving to be closely juxtaposed.

Immediacy of Feedback

The fourth feature of the geometry tutor is that it provides immediate feedback on the student's problem-solving efforts at each step. Whenever the student makes an incorrect inference, the system responds by identifying the error in the student's logic. For instance, when the student tries to use the side-angle-side rule but chooses an incorrect pair of angles, the tutor will point this out to the student. When the student makes an inference that is logical but is not on a path that leads to a proof (as determined by our ideal model), the tutor allows the student to explore that path until he or she appears to be lost. Then the tutor intervenes and points to a correct path.

Assessment of the Geometry Tutor

Three students have now learned geometry by use of the tutor in various stages of its development. One student was of above-average ability, one of average ability, and one of below-average ability (as defined by their grades in mathematics courses). The below-average student came to us for remedial work, having failed tenth-grade geometry. The other two were in the eighth grade and had no formal training in geometry. All learned geometry successfully and were solving problems more complex than are usually assigned in the schools. After it was over, all claimed to like geometry, which is encouraging since classroom geometry is usually the least liked of all school subjects (*17*). An experimental classroom in a Pittsburgh public high school that has a set of powerful computer workstations will test the geometry tutor in the fall of 1985. We are also working toward establishing the geometry tutor on more economical machines. Growing understanding about how to optimize code and the increasing power of personal computers should make this feasible.

a

```
I assume you are trying to test if the number argument
is equal to 0   There is a special function for testing
equivalence to 0   Try using that function instead of EQUAL

                    CODE FOR fact
(defun fact (n)
     (cond ((equal )  <ACTION>)
              <RECURSIVE-CASE>))

                        GOALS
Write code to get the value of fact
Code the terminating case
*** Code the condition for the terminating case ***
```

b

```
In examples A and B what do you have to do to get the result
of fact called with n?
PRESS              IF YOU WANT TO
1        Multiply n by one less than n
2        Multiply n by fact of one less than n
3        Add n to the result of fact called with one less than n
4        Have the tutor choose
Menu Choice  2

                    CODE FOR fact
(defun fact (n)
     (cond ((zerop n) 1)
              <RECURSIVE-CASE>))

                      EXAMPLES
      fact (n)         fact (n-1)
A   (fact 1) = 1     (fact 0) = 1
B   (fact 3) = 6     (fact 2) = 2
```

Fig. 2. (a) An early point in writing the code for factorial; (b) the tutor guiding the student in designing the algorithm for factorial.

The LISP Tutor

The LISP tutor, which is used to teach basic programming constructs for the writing of LISP programs, has been used to teach introductory programming at Carnegie-Mellon University. The tutor is based on studies of how students learn programming (*7, 18*). As was true of the geometry tutor, the design of the LISP tutor was guided by an ideal model for LISP programming, an interface for communicating the goal structure of programming tasks, instruction in context, and immediacy of feedback.

The Ideal Model for LISP Programming

We have developed a production system capable of simulating the way that good students code introductory-level LISP programs. Currently, the tutor's ideal model contains approximately 325 production rules, including the two productions shown here:

IF the goal is to multiply NUMBER1 by NUMBER2
THEN use the function TIMES and set as subgoals to code NUMBER1 and NUMBER2

IF the goal is to code a recursive function, FUNCTION, with an integer argument, INTEGER
THEN use a conditional structure and set as subgoals
1) to code the terminating case when INTEGER is 0, and
2) to code the recursive case in terms of FUNCTION (INTEGER − 1).

The first rule codes use of the basic LISP function for multiplication. The second, a more advanced production, defines a recursive function dealing with integers. The rule sets a first subgoal to code the terminating case for the recursion. For example, to code a factorial function, the first subgoal generates 1 as the value of 0!. The other subgoal defines the recursive case in terms of a function call with one less than the integer. Thus, the factorial of n is computed from the factorial of $n - 1$; that is, $n! = n \times (n - 1)!$.

Both of these productions involve setting subgoals. LISP code is generated by decomposing goals into subgoals, and these into further subgoals, until goals are set that can be directly achieved. Students are taught to program according to the goal decomposition methods in the tutor's ideal model.

The Tutorial Interface

A major design feature of the interface has been to provide the student with a structured editor with which to enter code. The structured editor automatically balances parentheses and provides placeholders for the arguments of each function. For example, to define a LISP function, one specifies the function "defun," the name of the function, a parameter list, and the function body. To begin, the student types a left parenthesis and the word "defun." At that point the tutor redisplays the code as

(defun <NAME> <PARAMETERS>
<PROCESS>)

The symbols in brackets indicate arguments that must be coded. The tutor places the cursor underneath the symbol <NAME> and illuminates it to indicate that this symbol must be coded next.

This structured editor relieves students of the burden of balancing parentheses and checking syntax, thus enabling them to focus on the aspects of LISP that are conceptually more difficult. Our results demonstrate that enabling students to pay more attention to the central conceptual issues in programming leads to faster learning of these major skills, with no deficit in the student's knowledge of syntax. In addition, the structured editor facilitates communication between the student and the tutor. The student types directly into the code, replacing one of the placeholding symbols, and thus it is always clear which part of the problem is being coded. In the question-answer format of most educational software, the student can easily get "out of synch" with the tutor when the student is not sure which part of the problem the tutor is discussing or querying.

The Goal Structure of LISP Programming

The tutor has been designed to communicate the conceptual structure of programming problems. This is accomplished in part by using the placeholders to provide a template for the rest of the problem solution. The tutor also communicates the goal structure in its guidance in planning LISP programs. When requested or when the student encounters sufficient difficulty, the tutor initiates a planning mode in which it leads the student through the design of an algorithm to accomplish the current portion of the problem. Thus, the student learns how a complex problem can be broken down into simpler problems to be solved. In both coding and planning modes, a special goals window reminds the student about the current goal in solving the problem.

Instruction in Context

As in the geometry tutor, instruction is provided in the context of solving problems. After each new concept is introduced, the student is given a number of problems designed to put that concept to use. The instruction can then be tailored to the difficulties encountered and can be provided while the student is trying to apply new skills rather than merely reading about them.

Immediacy of Feedback

Like the geometry tutor, the LISP tutor provides immediate feedback and guidance on incorrect and nonstrategic steps. In addition to the correct production rules in the ideal model, the tutor contains a bug catalog, a collection of 475 rules that represent errors made by novice programmers. The tutor compares each item of code entered by the student to determine which correct or incorrect production rule led to that input. If the input matches a correct rule, the tutor is silent and waits for further input. If the input is diagnosed as an error, the tutor interrupts with advice. Thus, the feedback is immediately provided, and necessary instruction can be given both in general terms and in the context of the current problem.

The tutor also curtails unnecessary floundering by providing guidance of various sorts. The student can request clarification of the current part of the problem and can also ask for the next step in the solution. In addition, if the student has sufficient difficulty in coding a particular part of the problem, the tutor will intervene. If the current portion of code is complex, the tutor initiates a planning mode for designing an algorithm. If the current part of the problem is more straightforward, the tutor provides the next step, setting the student back on one of the correct paths to a solution.

The type of feedback and guidance provided by the tutor can be seen in Fig. 2, a and b. In this example the student is writing a recursive function to code the factorial of a number. Figure 2a presents an early stage in that interaction in which the student receives a hint that another LISP function is more appropriate than the one he or she used. In Fig. 2b, the tutor helps the student design an algorithm after he or she had difficulty in coding the recursive case. At the bottom of the screen the student has worked out some examples of the relation between fact (n) and fact ($n - 1$), and he or she is being asked to generalize that relation.

Evaluation of the LISP Tutor

We have completed two studies of the LISP tutor in action. One, completed in the summer of 1984, compared ten students learning LISP from the computer tutor, ten learning LISP from a human tutor, and ten doing all their problem-solving on their own (which is the normal situation). All three groups of subjects read the same instructional material and worked on the same problems. The human-tutored subjects took 11.4 hours, the computer-tutored subjects took 15.0 hours, and the subjects on their own took 26.5 hours to cover this material. The difference between the two conditions in which the students were tutored was not significant, but both were significantly faster than the students learning on their own. The three groups performed equally well on tests of their LISP knowledge. However, this result may be deceptive because a number of subjects learning on their own did not finish the more difficult lessons as a result of the amount of time they had spent on the earlier lessons. Thus, our test scores for that condition are based on only the best subjects.

In the fall of 1984 we assigned ten students to the computer tutor and ten

students to learning on their own. Both groups got the same lectures and read the same material. All students completed the lessons. However, students working with the computer tutor spent 30 percent less time doing the problems associated with the lessons and scored 43 percent better on the final exam than the students working on their own.

Overall Assessment

Research on intelligent tutoring is now on the threshold of a methodology that will make qualitative changes in our ability to instruct students on topics that many students find difficult. This results from a fortuitous convergence of technological advances in the availability of computational power and scientific advances in the understanding of cognition. We have focused on the consequences of intelligent tutoring methods for pedagogy. However, we should stress that data collected in these pedagogical experiments advance the science of human cognition.

References and Notes

1. J. S. Brown and J. Greeno, in *Research Briefings, 1984* (National Academy Press, Washington, D.C., 1984).
2. D. Sleeman and J. S. Brown, Eds., *Intelligent Tutoring Systems* (Academic Press, New York, 1982).
3. *New York Times* (20 April 1984), summary of a research report by V. B. Cohen, p. C4.
4. E. B. Fiske, *New York Times* (9 December 1984).
5. J. R. Carbonnell, *IEEE Trans. Man-Machine Systems* **11**, 190 (1970).
6. D. Osgood, *Byte* **9**, 162 (June 1984).
7. J. R. Anderson *et al.*, *Cognitive Sci.* **8**, 87 (1984).
8. J. Larkin, J. McDermott, D. P. Simon, H. A. Simon, *Science* **208**, 1335 (1980).
9. J. R. Anderson, *The Architecture of Cognition* (Harvard University Press, Cambridge, Mass., 1983).
10. ACT* is the most recent in a series of theories which have been denoted by the acronym ACT. The acronym stands for Adaptive Control of Thought, a name that has only historical significance.
11. J. S. Brown and K. VanLehn, *Cognitive Sci.* **4**, 379 (1980).
12. D. Sleeman, in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown, Eds. (Academic Press, New York, 1982).
13. J. R. Anderson and R. Jeffries, *Hum. Comput. Interact.*, in press.
14. B. S. Bloom, *Educ. Researcher* **13**, 3 (1984).
15. C. F. Boyle and J. R. Anderson, "Acquisition and automated instruction of geometry proof skills", paper presented at the American Education Research Association meetings, New Orleans, April 1984.
16. J. R. Anderson, *Proceedings of IJCAI-81* [International Joint Conference on Artificial Intelligence (1981)].
17. A. Hoffer, *Math. Teach.* **11**, 18 (1981).
18. J. R. Anderson, P. Pirolli, R. Farrell, in *The Nature of Expertise*, forthcoming book, M. Chi, M. Farr, R. Glaser, Eds.

19. Supported by contract N00014-84-K-0064 from the Office of Naval Research and by grant IST-8318629 from the National Science Foundation.