Modelling Cognition
Edited by P. Morris
© 1987 John Wiley & Sons Ltd.

4

Cognitive principles in the design of computer tutors

JOHN R. ANDERSON, C. FRANKLIN BOYLE, ROBERT FARRELL and BRIAN J. REISER
Advanced Computer Tutoring Project, Carnegie-Mellon University

In this chapter a set of eight principles are derived from the ACT* theory for designing intelligent tutors: use production system models of the student, communicate the goal structure of the problem space, provide instruction on the problem-solving context, promote an abstract understanding of the problem-solving knowledge, minimize working memory load, provide immediate feedback in errors, adjust the grain size of instruction according to learning principles, and enable the student to approach the target skills by successive approximation. These principles have guided our design of tutors for LISP and geometry.

There has been and continues to be a great deal of hope for the role of computers in education (Cohen, 1982; Papert, 1980; Taylor, 1980). The actual record of accomplishment is still quite modest, however. Most computer education takes the form of simple drill and practice and is often not as effective as classroom drill and practice. There has always been the hope that artificial intelligence techniques would illuminate computer instruction. The buzz word of a decade ago was 'intelligent computer assisted instruction' or ICAI (Carbonnel, 1970). The relative lack of progress in that field led to a disenchantment with that expression, and we now see new descriptors. The basic problem with the earlier work was lack of a clear paradigm for bringing intelligence to bear in delivering instruction. A current belief is that the most promising paradigm for bringing intelligence to bear is to emulate a private human tutor. This is the intelligent tutoring paradigm (Sleeman and Brown, 1982).

The basic observation that motivates the intelligent tutoring approach is the great effectiveness of private human tutors over either classroom instruction or standard computer education. Bloom (1984) reports that 98 percent of students instructed by private tutors performed better than the average student in the classroom. In an experiment reported at the end of this chapter, we found private tutors were able to bring students to the same level of achievement in perhaps as little as a quarter of the time that was needed in the classroom. The hope of intelligent tutoring is to find some way of 'bottling' the skill of the human tutor and putting it in a computer tutor.

The most straightforward application of artificial intelligence techniques to intelligent tutoring would be an expert systems approach. In this approach, one would treat the human tutor as the expert whose knowledge has to be extracted and build an expert system to apply that knowledge. Work such as that of Stevens, Collins and Goldin (1982) on teaching topics such as rainfall seems to have this character. However, human tutoring does not have the characteristics of a domain that proves susceptible to the expert system methodology. It is not a clearly circumscribed knowledge domain, it is heavily dependent on natural language understanding, and human tutors show enormous individual differences in their tutoring styles. Thus, it seems unlikely that there is a well-defined expertise to be captured and emulated.

It is probably for this reason that most approaches to intelligent tutoring have not tried to really emulate human tutors (see the papers in Sleeman and Brown, 1982). Rather they have tried to identify abstract principles of effective tutoring and design tutors that embody these principles without concern with whether these tutors emulate humans. While the tutoring system is not itself an expert system, an intelligent tutor often has an expert system as a submodule for solving problems in the domain to be tutored. For instance, Brown, Burton and DeKleer (1982) include an expert circuit analysis system as part of their SOPHIE tutor for troubleshooting circuits. Such a system can reason correctly about the domain and thus provide the correct answer, which is, of course, an important piece of information in instruction.

We feel that the actual pedagogical design of the tutor must be based on detailed cognitive models of how students solve problems and learn. The state of the art in cognitive science has reached the point where we now are able to produce theories capable of applications to intelligent tutoring. In this paper we would like to develop a set of principles for computer tutoring based on the ACT* theory of cognition (Anderson, 1983). This theory has been developed at a sufficient level of detail that it is possible to produce simulations of the theory that actually solve problems the way students would solve problems and which learn from problem-solving much as students learn. These simulations form the core of our tutorial efforts because part of every tutor is what we call an *ideal* model, which models how successful students solve problems in the domain. Such models have an additional requirement not found in the expert systems of most tutors. Not only must they be able to solve the problems in the domain, they must be able to solve them the way students do. For instance, Clancey (1983) has found it necessary to rework mycin in order to build a tutor for medical reasoning.

The major portion of this paper will be devoted to identifying and justifying eight principles for doing intelligent tutoring. Then the last two sections of the paper will describe two tutors we have built partially embodying these principles. One is a tutor for the LISP programming language and the other is a tutor for high-school geometry. However, before embarking on any of this we should state a major limitation on the principles that we will articulate. We can only defend their application to a relatively restricted set of topics—those for which we can develop ideal student models. These include the domains of high-school and early-college math and introductory programming. These domains are relatively sparse in their importation of extra-domain knowledge, and thus the ideal model need only address domain knowledge. It is possible to develop much more articulate tutors for domains for which one has precise student models. If one does not know exactly what it is the student is supposed to do, it forces one to back off into a different tutoring strategy.

Throughout this paper we will be making reference to observations we have made of high-school students learning geometry and college students learning to program in LISP. We have observed four students spend approximately 30 hours studying beginning geometry and three students similarly spending 30 hours learning LISP. These sessions have all been recorded and have been subjected to varying degrees of analysis. Some of these analyses have been reported in a series of prior publications (Anderson, 1981a,b; 1983; Anderson, Farrell and Sauers, 1984; Anderson, Pirolli and Farrell, in press). This data base of protocols has served as a rich source of information about the acquisition of problem-solving skill and has heavily influenced design of our computer tutors. In addition to gathering this protocol data, we have performed a good number of more analytic experiments which we will refer to throughout this chapter.

REVIEW OF THE ACT* THEORY

Before turning to the cognitive principles it is useful to give a brief review of the ACT* theory on which they are based. This theory has been embodied in computer programs which have successfully simulated many aspects of human cognition (Anderson, 1983). The ACT* theory is an attempt to identify all the principal factors that affect human cognition and organize them into a complete cognitive theory. The ACT* theory is quite complex, consisting of a set of assumptions about a declarative memory and a procedural memory. However, it appears 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. Much of human cognition appears to unfold as a sequence of actions evoked by various patterns of knowledge. Productions attempt to capture this by pattern–action pairs describing individual steps of cognition. The actual productions are implemented in a computer system that simulates human cognition. An 'Englishified' version of a production from one of the ACT* simulations is:

```
     IF   the goal is to prove $\Delta UVW \cong \Delta XYZ$
   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 backwards chaining rule that embodies the SAS (side–angle–side) rule of geometry. If the goal is to prove two triangles congruent, it sets as a subgoal to prove two sides and an included angle congruent. One can also have forward inference rules such as:

```
     IF   the goal is to make in inference from the fact that $\overline{XY} \cong \overline{UV}$
          and it is true that $\angle XYZ \cong \angle UVW$
          and it is true that $\overline{YZ} \cong \overline{VW}$
   THEN   infer that it is true that $\Delta UVW \cong \Delta XYZ$
               because of the side-angle-side postulate.
```

As these examples illustrate, productions in the ACT* theory are *goal-factored*. That is, they make reference in their condition to a specific goal to which they are relevant. These productions can only be evoked when that goal is active.

The conditions of these productions are patterns that match to information being held in working memory. The ACT* theory assumes people have only a limited capacity to keep information active in working memory. It is possible that the capacity will be exceeded in a problem, and critical information for the matching of a production will be lost. This can result in the failure to execute the appropriate production or the execution of an inappropriate production. A good many errors of learners are due to working memory failures rather than failures of understanding (Anderson and Jeffries, in press).

Knowledge compilation

Key to any theory of tutoring are the learning mechanisms by which new procedures (productions above) are acquired. Knowledge compilation is a major mechanism for procedural learning in ACT*. Elsewhere (Anderson,

1983) I have argued that knowledge compilation and production strengthening account for all forms of procedural learning. There are two subprocesses involved in knowledge compilation. The first, proceduralization, creates productions that eliminate the retrieval of information by pattern matching of a production condition. Proceduralization builds that information into the proceduralized production. One example of this involves retrieval of information from long-term memory about LISP programming. In ACT* there is a production that will retrieve function definitions from long-term memory and apply them:

```
  IF   the goal is to code a relation defined on an argument
       and there is a LISP function that codes this relation
THEN   use this function with the argument
       and set a subgoal to code the argument
```

In this production, *relation* and *function* are variables which allow the production to match different data. The second line of the condition might match, for instance, 'CAR codes the first member of a list'. If this rule is proceduralized to eliminate the retrieval of the CAR definition, it becomes:

```
  IF   the goal is to code the first member of a list
THEN   use the CAR of the list
       and set as a subgoal to code the list
```

This is achieved by deleting the second clause that required long-term memory retrieval from the first production. In addition, the rest of the production is made specific to the relation *first element* and the function CAR. Now a production has been created which can directly recognize the application of CAR. This will result in a reduction in the amount of long-term memory information that needs to be maintained in working memory.

Composition involves collapsing a number of successful operators into a single macro-operator that produces the same overall effect as the sequence of individual operators. As an example of this from LISP, suppose one wanted to insert the first member of List1 into List2. Then the following two operators would apply in sequence:

```
  IF   the goal is to insert an element into a list
THEN   CONS the element to the list
       and set as subgoals to code the element
       and to code the list

  IF   the goal is code the first member of a list
THEN   take the CAR of the list
       and set as a subgoal to code the list
```

The first rule above would apply and bind ***an element*** to 'the first member of List1' and ***a list*** to 'List2'. The second production would apply and bind ***a list*** to 'List1'. A simple case of composition would involve combining these two productions together to produce:

```
   IF  the goal is to insert the first member of one list into another list
 THEN  CONS the CAR of the first list to the second list
       and set as subgoals to code the first list
       and to code the second list
```

Such composition would collapse repeated sequences of coding operations to create macro-operators. The result would be a speed up in coding because problems could be coded in fewer steps. McKendree and Anderson (in press) provide evidence for such speedup of frequently repeated combinations of LISP functions.

We have briefly reviewed four features of the ACT* theory (use of productions, goal structure, working memory limitations and knowledge compilation) that prove to be key to the cognitive principles that we will be describing.

STATUS OF THE COGNITIVE PRINCIPLES

Before describing our principles of computer tutoring, it is important for us to clarify the relationship among these principles, our ACT* theory, and various empirical results. Each principle is in fact derived from the ACT* theory. However, the derivation is not always transparent, and in fact it was often a discovery for us that ACT* implied these principles. This is frequently the relationship between a scientific theory and design principles based on it. The theory is typically cast to predict what the results will be of particular manipulations, not to predict what manipulations will achieve a particular optimal outcome. Also, these principles often have boundary conditions—it is not the case that in all circumstances a particular manipulation is optimal.

The ACT* theory has a fair degree of empirical support (e.g. Anderson, 1983) but, as with any psychological theory, hardly has the status of an established fact. Therefore, we would like to have more evidence for those principles than simply that they are implied by ACT*. We will also describe empirical evidence consistent with these principles. In fact, one of the major reasons for our interest in tutoring research is to gather further empirical evidence for these principles and hence for the ACT* theory on which they are based.

Principle 1: Represent the student as a production set

Probably the most important role for a cognitive theory in tutoring is to provide explicit process models of how the ideal student should behave and of how the current student is behaving. The process model of the ideal student allows one to be very precise about instructional objectives. The process model of the current student allows one to be very precise about the current state of the student and how he or she deviates from the desired state. The current student model also allows one to interpret the student's behavior.

The implications of the ACT* theory for student models are, of course, that they should be cast as production systems. Much of our success in constructing tutors is to be credited to this design choice. It is an interesting question what the evidence is for production systems. The hypothesis of a production system is too abstract to be put to direct empirical test. Rather, the evidence for production systems comes from their success as the basic formalism for developing explanations of human problem-solving (e.g. Newell and Simon, 1972). It is also the case that numerous other efforts in the general domain of intelligent tutoring have taken to representing the to-be-tutored skill as a production set (e.g. Brown and Van Lehn, 1980; O'Shea, 1979; Sleeman, 1982).

Production systems not only enable the system to follow student problem-solving, but the individual productions define an appropriate grain size for instruction. Each production is a package of knowledge that can be communicated to a student in one interaction. Basically, a tutoring system monitors whether a student has the correct form of each rule, and the system provides missing rules and corrects buggy rules. Also, as emphasized by Brown and Van Lehn (1980), student misconceptions or bugs can be organized as production rules which are perturbations of correct rules. For example, a buggy rule may be missing a condition, resulting in an over-general rule that applies in incorrect situations.

Principle 2: Communicate the goal structure underlying the problem-solving

According to the ACT* theory, and indeed most cognitive theories of problem-solving, the problem-solving behavior is organized around a hierarchical representation of the current goals. It is important that this goal structure be communicated to the student and instruction be cast in terms of it. Below we discuss the fact that it is not communicated in typical instruction for LISP or geometry and some of the unfortunate consequences of this fact.

Geometry

Figure 1 shows the two-column proof form that is almost universally used in geometry. It is basically a linear structure of pairs where each pair is a statement and justification. Typical instruction encourages the belief that the goal structure of the student should mimic this linear structure—that at any point in the proof the student will have generated an initial part of the structure and the current goal is to generate the next line of the structure.

Figure 1 also illustrates the typical instruction that is given about how to generate a proof. This is all the instruction a student receives from the textbook about how to generate a proof, and many teachers provide no more in their classroom instruction. Clearly, there is very little identification of the goal structure a student should assume in generating a proof.

Figure 2 illustrates the mental representation that we believe a successful student creates for the proof of a geometry problem. The conclusion to be

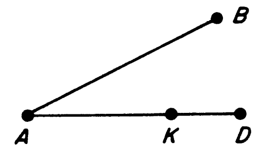
1-7 *Proofs in Two-Column Form*

You prove a statement in geometry by using deductive reasoning to show that the statement follows from the hypothesis and other accepted material. Often the assertions made in a proof are listed in one column, and reasons which support the assertions are listed in an adjacent column.

EXAMPLE. A proof in two-column form.

Given: $\overline{AKD}$; $AD = AB$

Prove: $AK + KD = AB$



Proof:

STATEMENTS	REASONS
1. AKD	1. Given
2. $AK + KD = AD$	2. Definition of between
3. $AD = AB$	3. Given
4. $AK + KD = AB$	4. Transitive property of equality

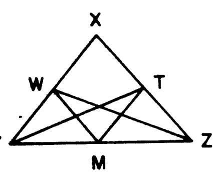
Some people prefer to support Statement 4, above, with the reason *The Substitution Principle*. Both reasons are correct.

The reasons used in the example are of three types: *Given* (Steps 1 and 3), *Definition* (Step 2), and *Postulate* (Step 4). Just one other kind of reason, *Theorem*, can be used in a mathematical proof. Postulates and theorems from both algebra and geometry can be used.

Reasons Used in Proofs

Given (Facts provided for a particular problem)
Definitions
Postulates
Theorems that have already been proved.

Figure 1. An example of the instruction about a two-column proof used in high-school geometry from Jurgensen *et al.* (1975).



GIVEN: $\overline{XY} \cong \overline{XZ}$, $\angle WMY \cong \angle TMZ$
M midpoint of $\overline{YZ}$

PROVE: $\overline{YT} \cong \overline{ZW}$

STATEMENT	REASON
M is midpoint of $\overline{YZ}$	Given
$\overline{YM} \cong \overline{MZ}$	Definition of midpoint
$\overline{XY} \cong \overline{XZ}$	Given
$\angle XYZ = \angle XZY$	Base angles of isosceles triangles
$\angle WMY \cong \angle TMZ$	Given
$\Delta WMY \cong \Delta TMZ$	Angle-side-angle (ASA)
$\overline{WY} \cong \overline{TZ}$	Corresponding parts
$\Delta WYZ \cong \Delta TZY$	Side-angle-side (SAS)
$\overline{YT} \cong \overline{ZW}$	Corresponding parts

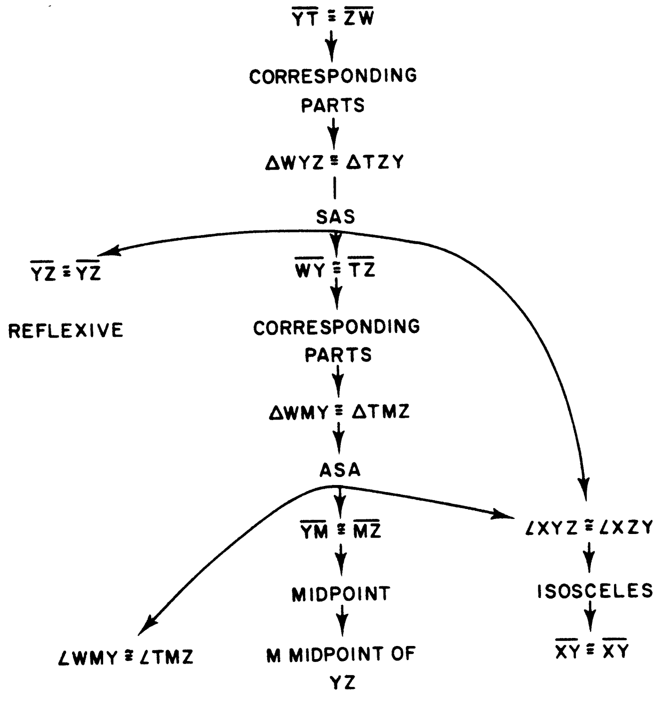


Figure 2. (a) A proof problem; (b) a representation of the logical structure of inferential support.

proven is related to the givens of the problem by a hierarchical structure. In that structure rules of geometry relate givens to intermediate statements and these statements to the conclusion. Hopefully, the readers will find nothing surprising in this representation of the logical structure of the proof, but it needs to be emphasized that conventional instruction does not communicate this structure and students hardly find it obvious when they first encounter such proofs (Anderson, 1981a,b).

Figure 3 illustrates the proof in Figure 2 embedded in a set of additional

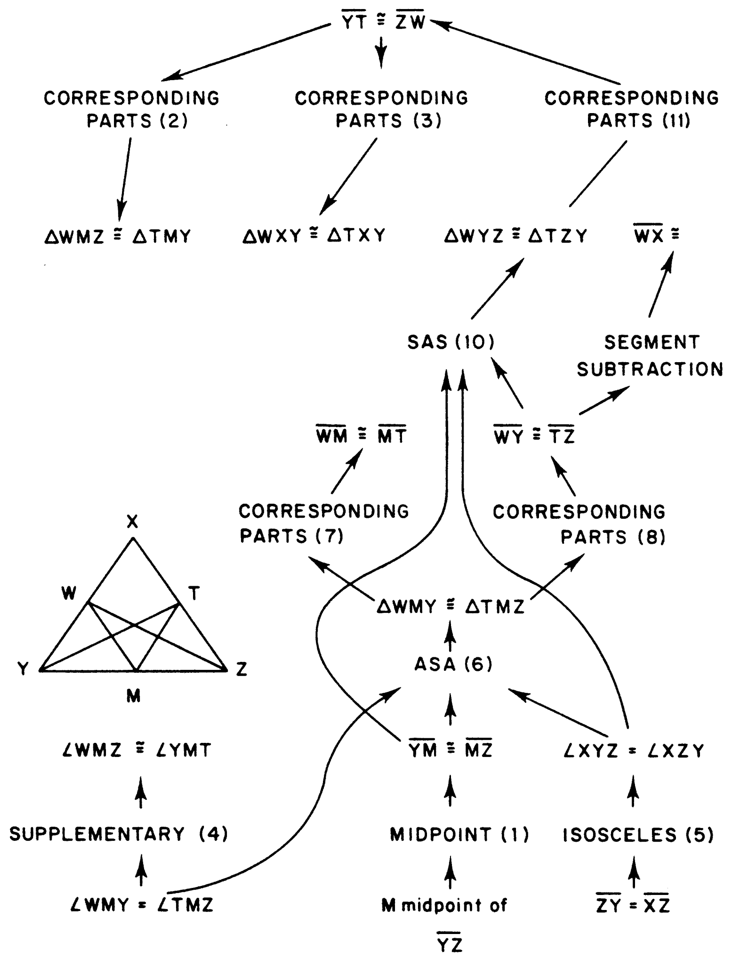


Figure 3. A representation of the order of inferences made in discovering the proof in Figure 2.

inferences generated by one of the authors (J.R.A.) in constructing the proof. The inferences are numbered in the order that they were made. The structure includes many inferences that were made in an attempt to construct the proof but were not part of the final proof. These extra inferences reflect some of the search that was involved in producing the final proof. Because standard pedagogy fails to communicate either the proof structure or the search based upon it, there is no way to provide instruction about this critical aspect of the learning process. This aspect is entirely a matter for the student to induce. Thus, typical instruction focuses only on the desired results without explaining the mental planning required to generate such a result. Given its complexity, it is no wonder that students have the difficulties they do with geometry proofs.

LISP programming

As in geometry, the goal structure underlying writing a LISP function does not correspond to the syntax of the problem solution and yet instruction is usually cast in terms of the syntax. Our studies indicate that generating a LISP program is largely a top-down planning process (Anderson, Farrell and Sauers, 1982; Anderson, Pirolli and Farrell, in press). The surface form of most programming languages involves a linear sequence of symbols. Thus, there is the natural danger of casting instruction in the terms of this linear structure. Fortunately, more enlightened instruction does emphasize a hierarchical, structured program. There is evidence that this is a better instructional mode (Shneiderman, 1980) although it is notoriously difficult to prove obviously correct hypotheses in this field with conventional experimental methodology (Sheil, 1981).

While structured programming is definitely a step in the right direction, it only ameliorates the basic problem. As Soloway, Bonar and Ehrlich (1983) show, the structured program itself is only a syntactic object which will have an imperfect correspondence to the structure of the programmer's plan. Consider an example we have studied at length (Anderson, Pirolli and Farrell, in press) from learning to program recursive functions, a recursive function that calculates the intersection of two lists:

```
(defun intersection (set1 set2)
  (cond ((null set1) nil)
        ((member (car set1) set 2)
            (cons (car set1)
                (intersection (cdr set1) set2)))
        (t (intersection (cdr set1) set2))))
```

From a syntactic point of view, this function consists of a conditional structure that is composed of three clauses, each consisting of a condition and an action.

The student is encouraged to believe that it is just a matter of 'programmer's intuition' that leads to the division of the conditional into the right set of three conditions and actions. Instruction is largely focused on explaining how the code works rather than on how to write it. As an instance of the kind of discussion that can be found in many texts, Siklossy (1976, p. 55) gives the following explanation of intersection:

> The simplest case occurs when one of the sets, say Set1, is the empty set (). The intersection is then the empty set. In the next simplest case, Set1 could be a set with only one element, for example (HAYADOIN). If the element HAYADOIN is a MEMSET of SET2, then the intersection is the set (HAYADOIN). On the other hand, if HAYADOIN is not a MEMSET of SET2, then the intersection is the empty set (). In the general case, we move down SET1 (taking CDR's) and accumulate those elements of SET1 which are also MEMSETs of SET2.

Clearly no algorithm is specified for deciding how to generally break such a function into cases.

However, there are very precise principles that underlie the division of this code into its components (Soloway, 1980; Rich and Shrobe, 1978). This is an example of what we call the CDR-recursion plan. This plan applies when an argument of the function is a list such as *set1*. The plan involves coding a terminating case for when the list is *nil* and a recursive case that relates the result of the function applied to the tail (*cdr*) of the list to the result of the function applied to the full list. In the code above, the first clause of the conditional implements the terminating case and the second and third clauses perform the recursive step. So, the plan that generated the conditional consists of two major subgoals, not three. However, it is necessary to break the recursive step into two subcases—one to deal with the situation where the first element is in the list and one where it is not. This division is dictated by a standard list search plan.

We have evidence (Pirolli and Anderson, in press) that students learn significantly faster when told about the CDR-recursion plan and encouraged to use it rather than having recursive evaluation explained to them. Our protocol studies (Anderson, Pirolli and Farrell, in press) show the extreme cost of not informing students of such goal structures. We have seen students work for ten hours with some popular LISP texts on problems like *intersection* and not figure out the CDR-recursion plan. Rather, they pay attention to surface characteristics of the examples and fail to identify the powerful principles for coding recursive functions. As a consequence, they were still floundering after the ten hours.

Summary

Traditional instruction does not explain the goal structure of the problem-solving plan nor the search involved in the problem-solving. Private human tutors often communicate this information as they direct and assist students' problem-solving. For instance, all three of the LISP tutors we observed in our protocols suggested dividing the code into terminating and recursive cases. A computer tutor should strive to communicate the goal structure to the student as this is absolutely critical to effective problem-solving.

Principle 3: Provide instruction in the problem-solving context

Students appear to learn information more effectively if they are presented with that information during problem-solving rather than during instruction apart from the problem-solving context. For example, providing a student with hints or answers to confusions is much more effective while the student is trying to solve a problem, as opposed to prior to the problem-solving or following it. That this should be true is a direct consequence of ACT*'s knowledge compilation mechanism. Because knowledge compilation only works on traces of problem solutions, productions are only acquired in the process of problem-solving; they cannot be learned in the abstract. In all of our studies of skill acquisition (see Anderson, 1981a,b), we have noted a great speedup after the first few problems. Detailed analyses of protocols suggest that students are compiling domain-specific productions from this experience with the first few problems, and that they did not have these productions prior to solving the first problems. Thus it seems that ACT* is correct in its basic claim that skill is only acquired by doing. Instruction should be most effective if given in the context of the problem-solving while the student is forming these productions.

In addition to this basic reason, there are four other reasons for believing instruction will be more effective when provided in context. First, there is evidence that memories are associated to the features of the context in which they were learned. The probability of retrieving the memories is increased when the context of recall matches the context of study (Tulving, 1983; Tulving and Thomson, 1973). An extreme example of this was shown by Ross (1984) who found that secretaries were more likely to remember a text-editor command learned in the context of a recipe if they were currently editing another recipe.

Second, it is often difficult to properly encode and understand information presented outside of a problem context. Thus the applicability of knowledge might not be recognized in an appropriate problem context. For instance, students may not realize that a top-level variable is really the same thing as a function argument in LISP even though they are obliquely told so. As another example, many students reading the side–angle–side postulate may not know what *included angle* means and so misapply that postulate.

Third, even if students can recall the information and apply it correctly, they are often faced with many potentially applicable pieces of information and do not know which one to use. We have frequently observed students painfully trying dozens of theorems and postulates in geometry before finding the right one. The basic problem is that knowledge is taught in the abstract and the student must learn the goals to which that knowledge is applicable. If the knowledge is presented in a problem-solving context its goal relevance is much more apparent.

Fourth, we do not want to overload the student by providing in advance every possible hint and explaining every possible pitfall. But we do not know in a classroom lecture or in writing a textbook what help the student will need. In fact, the students in listening to a lecture may not know what help they will need either. If we wait until the problems arise, we can provide just the information that is needed.

Private human tutors characteristically provide information in the problem-solving context. Some, but not all, of the tutors we observed gave almost nonstop comment as students tried to solve problems. They take great advantage of the multi-modality character of the learning situation—with the student solving the problem in the visual modality and their instruction in the auditory modality. Although it would be difficult for a computer tutor to be as interactive as these human tutors and take full advantage of multi-modal processing, we shall see that it is possible to partially achieve this by providing appropriate instruction tailored to the students' current goal context.

Principle 4: Promote an abstract understanding of the problem-solving knowledge

Students differ in the level of abstraction at which they bring knowledge to bear in problem-solving. Pirolli and Anderson (in press) compared the approaches of three students to writing recursive programs. One student wrote such programs by literally copying code from an example program. Another student focused on a more abstract representation of the syntax of the condition–action structure. A third student relied on an even more abstract representation of division into terminating and recursive cases. In terms of ability to transfer to coding other recursive functions, ACT* simulations of these three students learned the least in the first case and the most in the last case, as did the actual students. The reason for the ACT* behavior was that the most abstract representation corresponded best to the right problem-solving organization.

Again in geometry, student success is ordered by the level of abstraction at which they solve problems. For instance, some students will represent the side–angle–side postulate as involving only the lower left angle because that is the way it is illustrated in the book. Thus, they learn an overly specific rule, which leads to later difficulties in problem-solving.

It is much easier for a student to encode knowledge concretely but much less effective to do so. A good tutor should see to it that a student achieves the right level of abstraction. It should be noted, however, that this principle does not deny the usefulness of concrete examples. As noted with respect to Principle 3, it is much easier to understand an abstract principle when one sees it applied to a concrete example. The recommendation here is that students should be encouraged to produce the right abstract encoding of that concrete example. For example, in the case of recursion they should represent an example function according to its division into terminating and recursive cases.

Principle 5: Minimize working memory load

As mentioned earlier, a principal source of learner errors in ACT* is loss of critical working memory information. Anderson and Jeffries (in press) provide evidence that almost all of the errors in the first few hours of LISP are from this source.

A good human tutor can recognize errors of working memory and typically provides quick correction (McKendree, Reiser and Anderson, 1984). Tutors realize that there is little profit in allowing the student to continue with such errors. However, human tutors really have no easy means at their disposal for reducing the working memory load. This is one of the ways we think computer tutors can be an improvement over human tutors—one can externalize much of working memory by rapid updates in the computer screen. This involves keeping track of partial products and visually presenting the goal structures.

Working memory errors increase the time to solve a problem but also, according to ACT*, limit what can be learned from a problem. If students cannot hold the key factors to a solution in working memory, they cannot build them into the compiled productions.

Principle 6: Provide immediate feedback on errors

Novices make errors both in selecting wrong solution paths and in incorrectly applying the rules of the domain. Errors are an inevitable part of learning, but the cost of these errors to the learner is often higher than is necessary. They can severely add to the amount of time required for learning. More than half of our subjects' problem-solving sessions were actually spent exploring wrong paths or recovering from erroneous steps.

According to the ACT* theory, it is difficult for a student to learn the correct production from an episode involving applying the wrong production, applying a sequence of other productions predicated on the wrong production, hitting an impasse, evetually finding the difficulty, and correcting it. The student needs to represent in working memory all of this complex sequence of events in order to be guaranteed successful compiling of a correct production.

Obviously, representing so much information can pose a severe information-processing load. ACT* predicts best learning of the correct operator if students are told immediately why they are wrong and what the correct actions are.

Of course, the student can learn something from the error episode, if not what the correct production was. The student can learn how to diagnose and correct error states. For instance, one normally learns to debug programs by making errors in one's own program. Given that debugging is a valuable skill, time should be set aside to teach it. However, the prescription of the ACT* theory is that it should be taught in the same way as programming. That is, there should be an ideal cognitive model for debugging, and the student should be given immediate feedback when his or her behavior seriously deviates from the ideal model.

We have found our emphasis on immediate feedback to be the most controversial of our principles. One reason for the controversy is just a misunderstanding. This is the belief that we are advocating that it is not important to learn to identify and correct errors. The other reason reflects a fundamental disagreement. Many people have strong beliefs that we learn better when we discover our errors rather than when we are told about them. However, it needs to be emphasized that the importance of immediacy of feedback to skill acquisition is one of the best documented facts in psychology (e.g. Bilodeau, 1969; Skinner, 1958). Much of this research has been with somewhat simpler skills than the complex cognitive skills we have studied. However, we have shown the same principle in a complex problem-solving domain (Lewis and Anderson, in press). Subjects learned more slowly when they were allowed to go down erroneous paths and were only given feedback at delay. This is despite the fact that these students spent much more time in the learning situation than the immediate feedback students because it took them longer to solve the same number of problems.

Another cost of errors is the demoralization of the student. These are domains in which errors can be very frequent and frustrating. We believe that much of the negative attitudes and math phobias derive from the bitter experiences of students with errors. It is hard to convey on paper the emotional tone of some of the protocols we have gathered, but below are the excerpts from students struggling with LISP errors:

> ... No, I need another set of parentheses, and I think I want it around—but I can't do that—that's got to be—damn!—I think the first argument of CONST has to be a list ... and why is that? No—I don't need *const*!! What am I talking about?! I need to use Union! ... No, no! That didn't work before because ... oh (groan) ... let's see ... I don't know what I'd use the Union of! ...

> ... I don't know. I lost where I am. That's usually what happens when I do that—when I slow down and stop—'cause I forget what I am doing. Err ... It's taken that much time!! Ok. Geez! Ugh ... I guess I will have to go back to it again ...

. . . I'm just so slow today. Damn! . . . so that means if this element of *set1* is in *set2* . . . the value of *setdiff* is gonna be . . . the . . . the . . . Union of Cons? . . . Doesn't matter what I guess . . . So it's gonna be *setdiff* of . . . is it that simple!? . . . something with the *cdr* of *set1* with . . . the *cdr* of *set1* and *set2* . . . Let me see if that works . . . Is that right? If that's right, I'll be p . . . ed. God, I hate myself. I can't even think about it 'cause I'll be so p . . . ed. But I have got to think about it . . .

The potential advantages of a private tutor are clear in this regard. The tutor can prevent the student from wasting inordinate amounts of time searching wrong paths. The tutor can both provide immediate feedback when errors are made and point out to the students which aspects of the problem solution are correct and which are in error. Actual human tutors vary in how much feedback they give (McKendree, Reiser and Anderson, 1984). Typically, they point out immediately what they perceive to be slips. They may allow a more conceptual error to pass if they believe the subject may be able to detect it. However, they tend to be very concerned about the emotional tone of the learning situation, and if the subject is frustrated they will gently put the subject back on the right track.

Principle 7: Adjust the grain size of instruction with learning

The productions in the ACT* theory define the grain size with which a problem is solved. Because of the knowledge compilation process, this grain size will change with experience. As an example from LISP, a novice approaches each symbol in the definition syntax as a separate subgoal. Later the student writes out the whole structure as a unit. As another example, geometry students come to recognize that they can apply a whole sequence of inference steps. Clearly, if the tutor is going to be helpful, the tutor will have to adjust instruction with the growing grain size of the instruction.

Human tutors often adjust their instruction according to individual differences between students and advances in the student's performance during a lesson. One of the things the tutor does is to chunk the instruction, thus stopping the student before taking in too much material to practice. For example, on encountering a page of new LISP functions, one tutor stopped his student after the first function in order to insure that the student understood that function before continuing with the rest of the functions. Prior to this the student had had a difficult time understanding the basic LISP structures, and presumably the tutor calculated that simply reading through the descriptions of each of the functions would not be very effective until the student saw one of these functions working and was able to understand the basic idea of LISP operations.

We have also observed students getting into trouble when their tutors have made incorrect assumptions about the level of instruction appropriate. For example, one tutor we observed explained the details of the LISP read–

eval–print loop (the internals of LISP's interactive system) when the student did not understand about when function arguments needed to be quoted. Although the student appeared at first to have understood this explanation, he made several quote errors in later problems until the tutor then decided to work through an example concerning quotes in more detail.

Principle 8: Facilitate successive approximations to the target skill

Students do not become experts in geometry or LISP programming after solving their first problem. They gradually approximate the expert behavior, accumulating separately the various pieces (production rules) of the skill. It is important that a tutor support this learning by approximation. It is very hard to learn in a situation that requires that the whole solution be correct the first time. The tutor must accept partial solutions and shape the student on those aspects of the solution that are weak.

Generally, it is better to have the early approximations occur in problem contexts that are as similar to the final problem context as possible. It is a consequence of the ACT* learning mechanisms that skills learned in one problem context will only partially transfer to a second context. Students are learning features from early problems to guide their problem-solving operators. If these features are different from the final problem space the problem-solving rules that the students learn will be misguided. For instance, early problems in geometry tend to involve algebraic manipulations of measures. Consequently, the student learns to always convert congruence of segments and angles into equality of the measure of the parts. Later problems such as those involving triangle congruence do not involve converting congruence of sides and angles into equality of measures. Students frequently carry over their over-general tendency to convert and get into difficulties because of this. This is just what the ACT* learning process would do given this experience.

It is often extremely difficult for students starting out to solve the kind of problems that they will eventually have to solve. For instance, in geometry students cannot initially generate proofs. To deal with this, standard pedagogy often evolves intermediate tasks such as giving reasons for the worked-out steps of a proof. The problem is that the process of finding reasons for the steps of a proof is different from the process of generating that proof. In our ACT* simulations of these tasks, there is almost no overlap between the productions involved in reason-giving and proof generation. As another example, in programming students are often given practice evaluating recursive functions as preparation for writing recursive functions. Again, these are separate tasks. Both in the geometry case and the programming case we have shown that there is not much transfer from one task to another. Neves and Anderson (1981) found no transfer from ten days of reason-giving to proof generation. McKendree and Anderson (in press) found no transfer from four days' practice of function evaluation to the task of generating the LISP code.

The advantage of private tutors is that they can help the student through problems which are too difficult for the student to solve entirely alone. Thus, it is common to see a sequence of problems where the tutor will solve most of the first problem with the student just filling in a few of the steps; the tutor will help less with the second problem, etc., until the student is solving the entire problem. Consequently, the tutor can enable the student's early learning to be in problem contexts very similar to the more advanced problem contexts.

ISSUES OF HUMAN ENGINEERING

We have tried to achieve the principles enumerated above in our development of computer tutors. However, many of the problems that we face in creating actual computer tutors were not with seeing how these principles should apply but were at a level which might best be called human engineering. This refers to issues of designing the interface and the natural language dialogue so that information exchanges occur that satisfy our cognitive principles. If one cannot communicate the knowledge effectively, the cognitive bases for these systems will become lost. Our human-engineering efforts have been somewhat guided by what we know from cognitive psychology, somewhat guided by results in the literature on the human–computer interface, but largely the result of trial and error exploration. This human-engineering aspect is far from trivial. Before we are going to have a good theory of intelligent tutors, we will need a good theory of their human engineering.

In the remaining two sections of this paper we will describe our geometry tutor and our LISP tutor. We will try to show how these tutors approximate the design criteria we have set forth. We will also try to communicate some of our experience with the human-engineering problems.

THE GEOMETRY TUTOR

High-school geometry has all the characteristics of a topic which should be amenable to the intelligent tutoring approach. Of all the high-school math courses, it is most frequently rated as the least liked, although students who go on to have successful mathematical careers often rate it as their favorite (Hoffer, 1981). So there is a wide range of educational outcomes and a real need for improvement. The most difficult part of geometry is doing proofs. While proof generation in general is hard, high-school proof problems are within the range of artificial intelligence techniques. Therefore, it appears that intelligent methods might make an impact on this most difficult aspect of this most difficult of the high-school math subjects. Our geometry tutor (Boyle and Anderson, 1984) is focused on teaching students how to generate proofs.

A geometry proof problem as stated in a high-school geometry text (see Figure 1) consists of three ingredients—a diagram, a set of givens and a to-be-proven statement. Despite claims to the contrary, the diagram plays a critical role in many high-school geometry proofs. Frequently (although this not the case in Figure 1), the diagram is the only source of critical information about what points are collinear and which points are between which others. This information often is not provided in the givens. Most other information which can be read off the diagram, such as relative measure, is not to be taken as true in general. It is a rare high-school proof problem that involves constructions or creating new entities not in the diagram. Indeed, some geometry textbooks have a policy of never requiring the student to do proofs by construction although all conventional texts must use proof by construction in establishing theorems for the student. Therefore, to a good approximation, the student's task is to find some chain of legal inferences from the stated givens and the givens implicit in the diagram to the conclusion.

The ideal model

Consider the problem in Figure 4. There are a set of forward and backward deductions that can be made. Forward deductions take information given and note that certain conclusions follow. So, we can infer from the fact that the M is the midpoint of $\overline{AB}$ that $\overline{AM} \cong \overline{MB}$. We can also infer from the vertical angle configuration that $\angle AMF \cong \angle BME$. Backward inferences involve noting that a conclusion could be proven if certain other statements were proven. So, we could prove M is the midpoint of $\overline{EF}$ if we could prove $\overline{EM} \cong \overline{MF}$. We could prove the latter statement if we could prove $\overline{EM} \cong \overline{AM}$ and $\overline{MF} \cong \overline{AM}$.

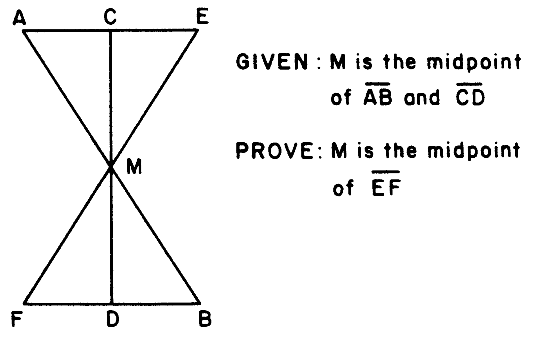

A
C
E
M
F
D
B
GIVEN : M is the midpoint of $\overline{AB}$ and $\overline{CD}$
PROVE: M is the midpoint of $\overline{EF}$

Figure 4. A relatively advanced proof problem for high school geometry.

As these examples illustrate, sometimes forward and backward inferences are part of the solution and sometimes they are not. In challenging problems the student cannot always know whether an inference is part of the solution. Rather, the student must make heuristic guesses about which inferences are likely. We saw this with respect to Figure 3.

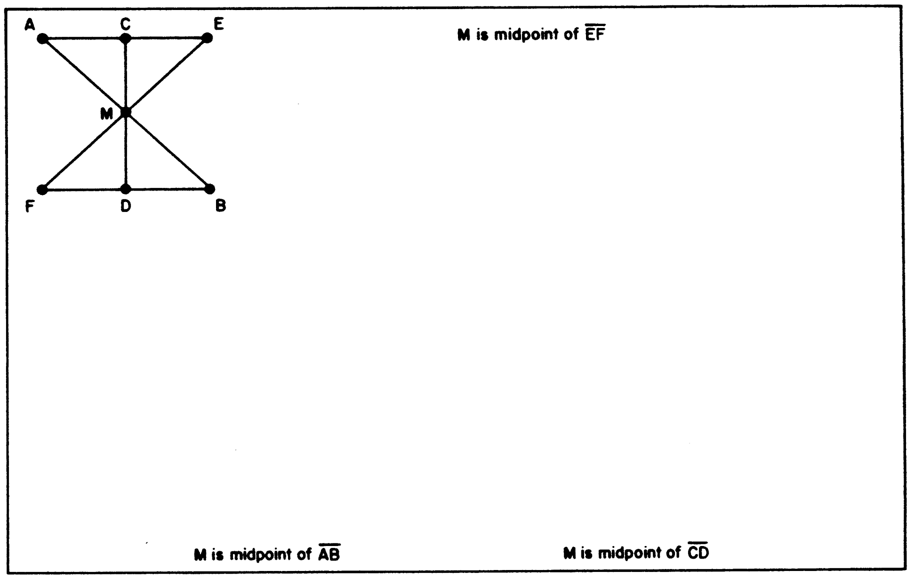
In our view, the ideal student extends the proof backward from the to-be-proved statement and forward from the givens until there is a complete proof. At each point the ideal student makes the heuristically best inference where this is defined as the inference most probably part of the final proof. 'Most probable' depends on some induction over the space of high-school problems. Currently, we have no formal definition of the probability that an inference will be part of a proof, but our intuitions are usually quite defensible. So, we create the ideal student model as a set of rules that seem heuristic. For instance, one rule is that when vertical angles are parts of to-be-proven congruent tirangles, infer that they are congruent but not otherwise. Basically, the rules all take the form of 'apply a particular rule of inference when such and such conditions prevail'. These conditions refer to properties of the diagram, givens, established inferences, and goals set in backward inference. These rules can be represented as production rules; for example:

```
  IF   $\Delta XYZ$ and $\Delta UVW$ are to-be-proven congruent triangles
       and $\overline{XYW}$ and $\overline{ZYU}$ are intersecting lines
THEN   infer $\angle XYZ \cong \angle UYW$ because of vertical angles.
```

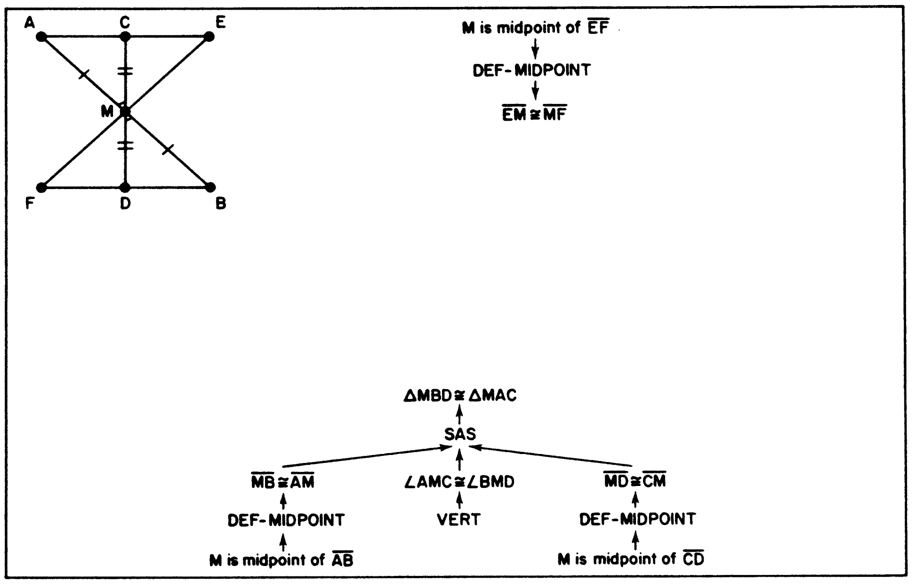
We have developed a set of 194 such production rules which seem to be sufficient for the problems in the first four chapters of Jurgensen *et al*. (1975), which is what our geometry tutor covers (about half of a high-school course). Many such rules can apply at any point in time, and the conflict-resolution principle selects the most highly rated rule to apply. Given this organization, we are able to solve all the problems in the text, develop proofs that strike us as the same as what we would do, and generate such proofs rapidly.

The proof graph

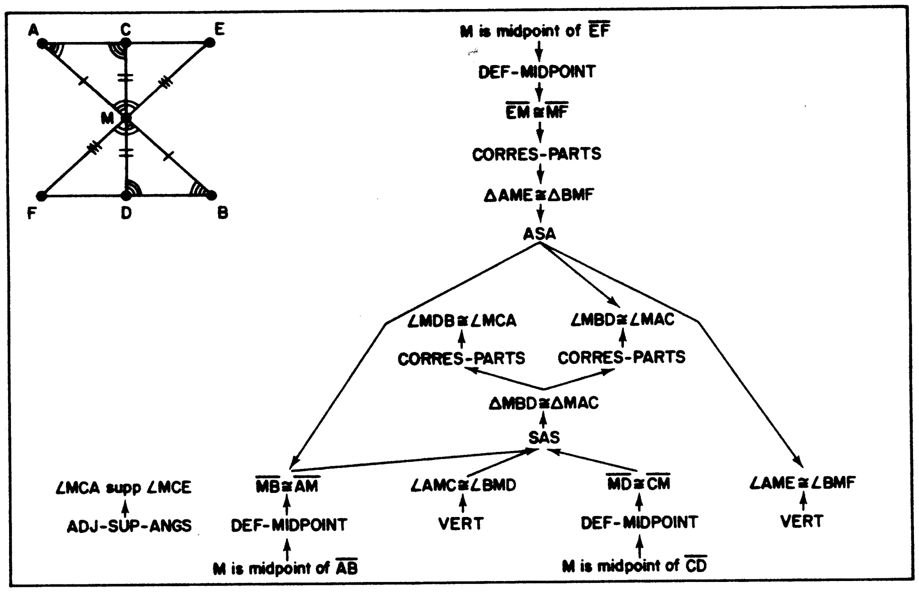
As noted earlier, standard instruction does a very poor job of communicating the goal structure to the student. Therefore, our first human-engineering problem was to find a way of communicating this information to the student. We decided to use a graphical formalism in which the to-be-proven statement was at the top and the givens were at the bottom. A proof is created as a logical network connecting the givens to the to-be-proven statement. The basic unit of this network is a structure connecting one or more givens to a conclusion through a rule of inference. Figure 5 shows four states of the proof network from beginning to end for the problem in Figure 4. The network can


A
C
E
M
F
D
B
M is midpoint of $\overline{EF}$
M is midpoint of $\overline{AB}$
M is midpoint of $\overline{CD}$

(a)


A
C
E
M
F
D
B
M is midpoint of $\overline{EF}$
DEF-MIDPOINT
$\overline{EM} \cong \overline{MF}$
$\Delta MBD \cong \Delta MAC$
SAS
$\overline{MB} \cong \overline{AM}$
$\angle AMC \cong \angle BMD$
$\overline{MD} \cong \overline{CM}$
DEF-MIDPOINT
VERT
DEF-MIDPOINT
M is midpoint of $\overline{AB}$
M is midpoint of $\overline{CD}$

(b)



(c)

Figure 5. Four states of the tutor screen in a students construction of a graph proof for the problem in Figure 5.

be grown from the bottom by forward inference or from the top by backward inference. As Figure 5 illustrates, it is certainly possible to generate inferences off the correct path. We have testimonials from three pilot students that they better understood both the structure of the proof and nature of our proof generation because of the graphical formalisms of the proof structure.

Interacting with the system

The basic cycle of interaction with the system is as follows. The student points to one or more statements on the screen from which he or she wants to draw an inference. Pointing causes these statements to blink. If at least one legal inference can apply to these statements, the system asks the student for the rule of inferent that applies. The student then types in the rule of inference. If it is applicable to these statements the system will prompt the student for the conclusion that follows. The student either types in the conclusion or points to it if it is on the screen. If correct the student can now initiate another cycle of interaction with the system, which will result in the posting of another deductive inference. This continues until a complete proof has been generated.

The human engineering of the system has involved a lot of trial and error to decide issues such as how to position the graph, what abbreviations to use, when to correct misspellings, how to let the student point, and how to relate the proof structures to the diagram. We have also found it useful to have the student spend an hour with a warm-up system that uses the same graphical conventions but with the familiar domain of arithmetic. The student can learn to use the system much more rapidly if this learning is decoupled from learning to do proofs in geometry. One problem with a complex screen is that the student may not notice when new information is added. We have found that color and motion are fairly effective ways of capturing attention. We we have taken to changing the colors of the windows, bringing up new windows in another color or blinking critical information.

For the accomplished geometry student, this system is a convenient and efficient vehicle for constructing proofs. It serves as an external memory so that forgotten information can be quickly recovered. It also catches slips of mind quickly. However, more is needed when dealing with a novice. The most basic problem for a student is not knowing how to proceed. There are a number of ways that our system helps students during problem-solving. Most directly the student can ask the system for help. Less directly, the student may choose to make inferences off nodes from which no inferences or no useful inferences follow. In either case, the system will provide the student with a hint in the form of suggesting the best nodes from which to infer.

The student may be uncertain about what inference rule to apply or how to apply the inference rule. He can manifest this again by asking for help or by inappropriately applying a rule. In this case windows can be brought up to display which rules of inference are currently applicable and to display a definition of each rule of inference. When the student displays a known bug, such as applying the side–angle–side postulate when the angles are not included by the sides, a window will appear explaining why the student's choice is not correct.

Students get in ruts in which they try to make an inference from an inappropriate set of statements over and over again or apply the same rule over and over again. In such cases, the system will interrupt and display what it regards as the next best inference step and interrogate the student to make sure that the step is understood.

An interesting property of the graphics screen is that the student has no access to solutions of previous problems. Therefore, it is very difficult to do any problem solution by copying parts of prior solutions—an unfortunate tendency of many students in the conventional classroom. When the students gets instruction, the instruction is about the inference rules in the form of general problem-solving operators. For instance, the system will give the following statement of the corresponding parts rule when it is evoked in backward inference to prove two sides congruent:

IF you want to prove the conclusion $\overline{UV} \cong \overline{XY}$
THEN try to prove the premise $\Delta UVW \cong \Delta XYZ$

along with a pattern diagram to help the student instantiate the abstract terms in this statement.

It is difficult to get access to high-school students, but we have looked intensively at three students working with the system—one with above-average ability, one of average ability and the other of below-average ability (as defined by their math grades). The below-average student came to us for remedial purposes having failed tenth-grade geometry. The other two students were eighth graders with no formal geometry experience. We can only report that the system works, i.e. all students learned with it and without great difficulty. We think they learned faster than with traditional instruction, but we have no way to document this belief. All students were able to do problems that local teachers consider too difficult to assign to their tenth-grade classes. They also all claimed to like the subject of geometry, which seems an important outcome given the negative ratings geometry typically gets.

Design principles in the geometry tutor

It is worth reviewing how the geometry tutor does and does not realize the design principles set forth in the beginning of this chapter.

1. As we have noted, we have an ideal model represented as a production system. We have yet to integrate a similar model of student bugs into the tutor.
2. The proof graph is an attempt to reify the goal structure and communicate it to the student.
3. The postulate, definition and theorems of geometry are taught in the context of their use in problem-solving.
4. The problem-solving is guided by abstract instruction, not by superficial properties of examples.
5. Working memory load is minimized in a number of ways. The proof graph is an attempt to represent subgoals. A color-coding scheme facilitates integration of the diagram and the abstract statements. This involves marking congruence on a diagram in the same color that they are displayed in the proof graph.
6. The system does provide immediate feedback on logical errors. It does not yet provide much strategic feedback about inferences that are logically correct but do not lead to proofs. We have observed students seriously flounder as a consequence. Providing such strategic advice is a current research goal.

7. Another deficit of the tutor is that it does not adjust grain size to reflect the level at which the student is working. Its grain size always corresponds to a single step of inference. However, beginning students need to be led through some inferences in mini-steps, while advanced students prefer to plan in multi-step inferences.
8. The system is a beautiful illustration of how a tutor can enable a skill to arise through successive approximation. Students start out relying on the tutor to provide almost all of the steps of the proof but reach the point where they are doing proofs completely on their own, proofs that schoolteachers consider too difficult to assign to conventional classes.

THE LISP TUTOR

Our work on the LISP tutor is based on earlier research (Anderson, Farrell and Sauers, 1984; Anderson, Pirolli and Farrell, in press) studying the acquisition of basic LISP programming skills by programming novices. Given standard classroom instruction, a programming novice typically takes over 40 hours to acquire a basic facility with the data structures and functions of LISP. At the end of this period they can write basic recursive and iterative functions. In this time the student has probably not written a LISP program more than three functions deep and still does not know how to use LISP for interesting applications.

We believe that the LISP tutor will be able to cover *the same material* in under 20 hours (we are currently over halfway there). After this point the tutor would step back and become an 'intelligent editor' which could help the student create programs and catch obvious slips, but would no longer instruct. Besides the desire to have a manageable project, we do not feel we are capable of modeling the problem-solving that occurs after learning the basics of LISP.

The ideal model

Brooks (1977) analyzed programming into the activities of algorithm design, coding and debugging. We are currently focusing on algorithm design and coding. Our past research on LISP involved creating simulations of both errorful and ideal students in the algorithm design and coding phases. Brooks characterized programming as first designing an algorithm and then converting it into code. However, the break is seldom so clean. The student alternates between algorithm design and coding, sometimes omitting the algorithm design altogether and going directly from problem statement to code. Often novices and experts differ as to whether there is a distinct algorithm design stage. One example we have studied involves writing LISP code to take a list and return that list with the last element removed. A number of experts generated (reverse (cdr (reverse lis))) immediately upon hearing the statement, whereas some novices went through a ten minute phase of means–ends analysis to come up with the algorithm.

The ideal model for code generation, both for experts and novices, involves a top-down generation of the code. Figure 6 illustrates the goal structure underlying the generation of the code for the function *powerset*, which takes a list and returns the list of all sublists. This is recognized as involving recursion on the list and subgoals are set to code the terminating step and the recursive step of the recursion. Both of these steps are broken down into algorithm design plus code generation. Writing the code for the recursive step involves writing a 'helping function' *addto* that will add an element to each list in a list of lists.

The actual code generated along with the goal structure in Figure 6 is:

```
(defun powerset (list)
     (cond ((null list) (list nil))
          (t (append (powerset (cdr list))
                    (addto (car list) (powerset (cdr list))))
          ))).
```

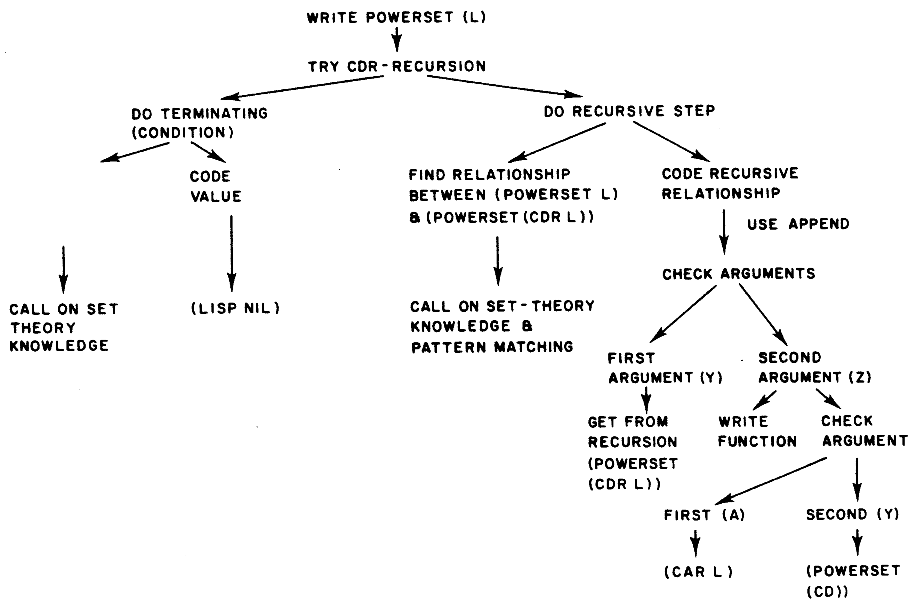

WRITE POWERSET (L)
TRY CDR-RECURSION
DO TERMINATING (CONDITION)
DO RECURSIVE STEP
CODE VALUE
FIND RELATIONSHIP BETWEEN (POWERSET L) & (POWERSET (CDR L))
CODE RECURSIVE RELATIONSHIP
USE APPEND
CHECK ARGUMENTS
CALL ON SET THEORY KNOWLEDGE
(LISP NIL)
CALL ON SET-THEORY KNOWLEDGE & PATTERN MATCHING
FIRST ARGUMENT (Y)
SECOND ARGUMENT (Z)
GET FROM RECURSION (POWERSET (CDR L))
WRITE FUNCTION
CHECK ARGUMENT
FIRST (A)
SECOND (Y)
(CAR L)
(POWERSET (CD))

Figure 6. A representation of the hierarchical goal structure underlying generation of the LISP function to calculate POWERSET.

There are a number of features to note about the goal decomposition shown in Figure 6. First, code is generated top-down. Second, there is alteration between algorithm design and code generation. Third, the encoding of the embedded function, *addto*, is postponed until the top function is coded.

It is difficult to develop ideal models for the algorithm design aspect of the problem-solving. Students can potentially bring any of their past experiences and prior knowledge to bear in designing an algorithm. With each problem we have to specially provide the ideal model with potentially relevant prior knowledge. So, for instance, to model the creation of a graph search function we had to provide the system with knowledge of the fact that paths in a graph can loop (Anderson, Farrell and Sauers, 1982).

Interacting with the LISP tutor

Figure 7 illustrates a typical state of the screen in interacting with the tutor. We used the hierarchical structure of LISP to support representation of the top-down structure of the programming activity. At all times during the problem the code window displays the part of the problem that has been coded. Placeholders are used to indicate the structures requiring top-down expansion: e.g.:

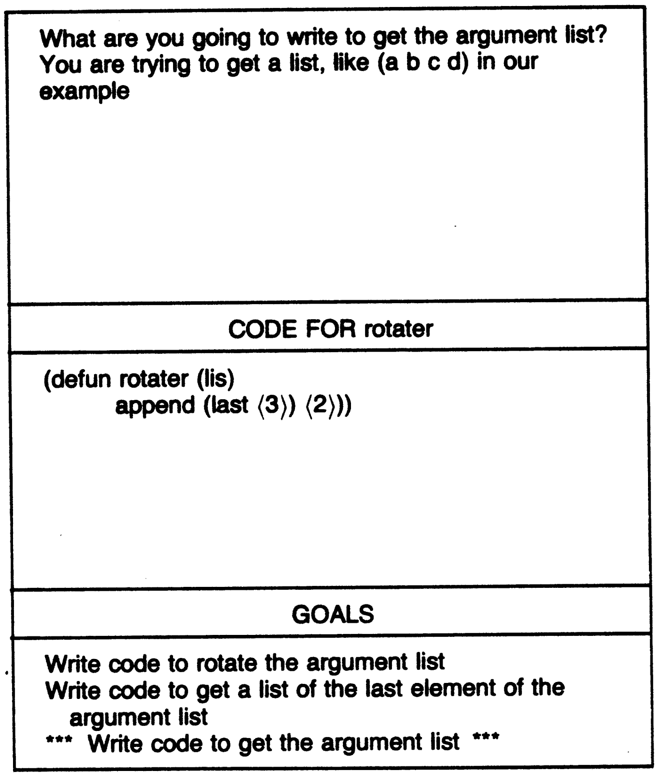
```
(defun rotator (lis)
          (append (last ⟨3⟩) ⟨2⟩))
```

where the symbols ⟨2⟩ and ⟨3⟩ denote the points for top-down expansion. The tutor moves the student to the next symbol for expansion, and the student types the code to replace this symbol. So, at this level the system is just a structured editor for creating the code.

In some cases, the hierarchical structure of the LISP code does not correspond to the goal structure. A good example of this was the intersection function discussed earlier where the COND structure had three clauses but the goal structure for CDR-recursion just involved one goal for doing the terminating case and one goal for doing the recursive step. In such cases we have the subject generate the code according to the underlying goal structure rather than the syntax of the LISP code. The placeholder symbols indicate the conceptual breakdown of the problem before they are transformed into LISP code. Thus, for instance, at one point in *powerset* the code is displayed

```
(defun powerset (lis)
      (cond ⟨TERMINATING-CASE⟩
         ⟨RECURSIVE-CASE⟩))
```

Sometimes, the student has to branch into algorithm design where the problem-solving will not correspond to the hierarchical structure of the code. In this case the tutoring window is used to work out a successful algorithm.


What are you going to write to get the argument list?
You are trying to get a list, like (a b c d) in our example

CODE FOR rotater

```
(defun rotater (lis)
       append (last ⟨3⟩) ⟨2⟩))
```

GOALS

Write code to rotate the argument list
Write code to get a list of the last element of the argument list
*** Write code to get the argument list ***

Figure 7. A typical state of the tutor screen in interacting with a student. The tutor's query is displayed at the top of the screen. The 'code' window displays the current code, and the 'goals' window summarizes the problem solving so far and indicates the current goal.

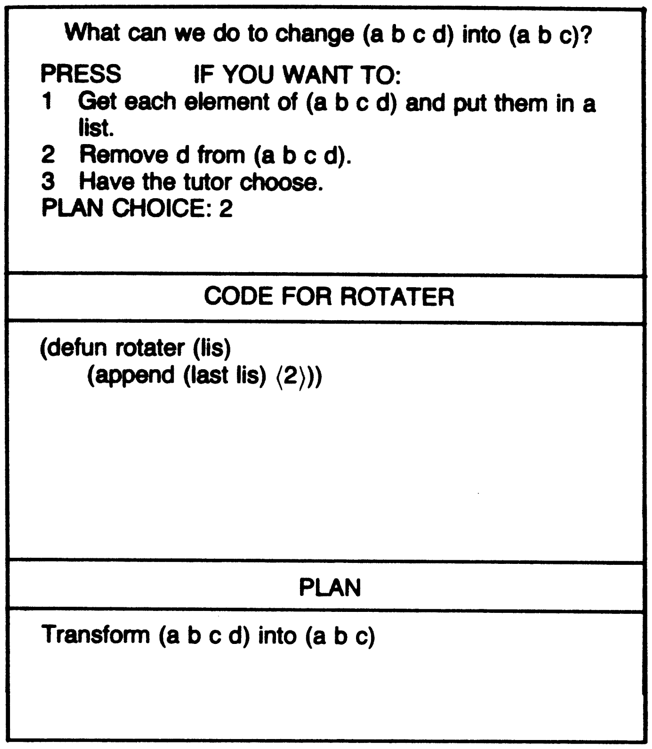
Figure 8 illustrates the tutor in the algorithm-design mode. The final product of the algorithm design ing will be some specific code that must be mapped into the abstract code of the function. So, for instance, if the student needs to design an algorithm for returning a list of all but the last of the *lis*, we will have him work out a solution for a concrete instance of *lis*, say lis = (a b c d). The final product in this case would be:

(reverse (cdr (reverse '(a b c d)))

which can be mapped into the abstract function to produce:

(reverse (cdr (reverse lis)))

Again, when doing the recursive step for *powerset* (see Figure 6) with the a



What can we do to change (a b c d) into (a b c)?

PRESS IF YOU WANT TO:
1 Get each element of (a b c d) and put them in a list.
2 Remove d from (a b c d).
3 Have the tutor choose.
PLAN CHOICE: 2

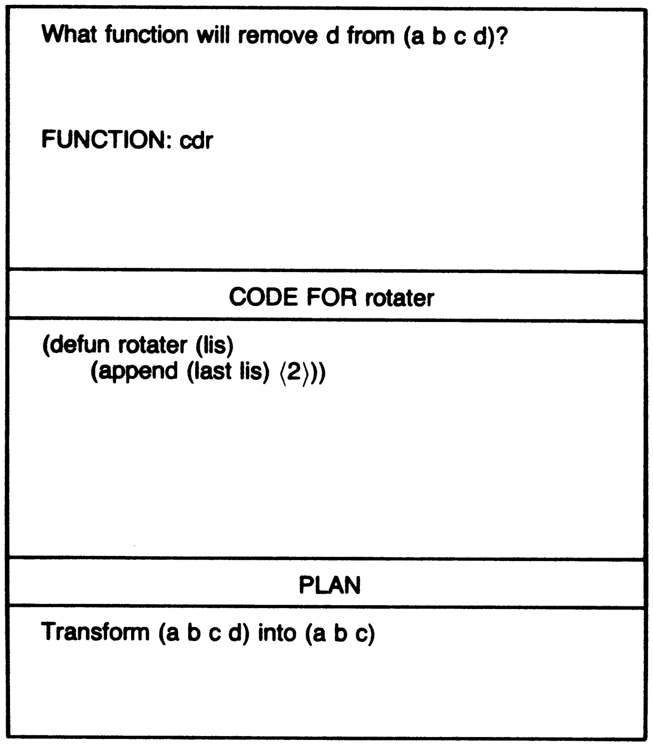
CODE FOR ROTATER

```
(defun rotater (lis)
     (append (last lis) ⟨2⟩))
```

PLAN

Transform (a b c d) into (a b c)

b



What function will remove d from (a b c d)?

FUNCTION: cdr

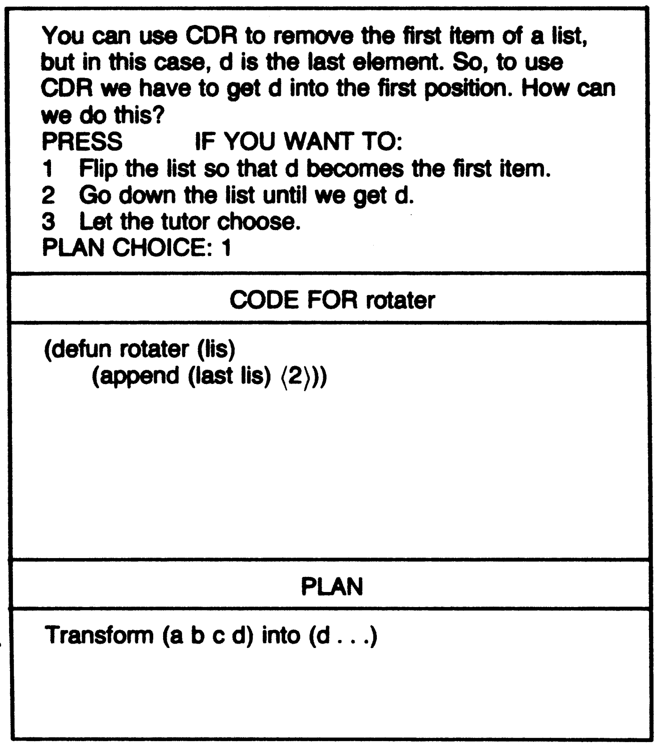
CODE FOR rotater

```
(defun rotater (lis)
     (append (last lis) ⟨2⟩))
```

PLAN

Transform (a b c d) into (a b c)

c



You can use CDR to remove the first item of a list, but in this case, d is the last element. So, to use CDR we have to get d into the first position. How can we do this?

PRESS IF YOU WANT TO:

1 Flip the list so that d becomes the first item.
2 Go down the list until we get d.
3 Let the tutor choose.

PLAN CHOICE: 1

CODE FOR rotater

```
(defun rotater (lis)
    (append (last lis) ⟨2⟩))
```

PLAN

Transform (a b c d) into (d . . .)

Figure 8(a, b, c). Three stages of the tutor screen during the tutoring of the algorithm. Note the description of the algorithm as applied to a concrete example. The 'plan' window summarizes the current plan for the example.

specific list (A B C), the final result in the algorithm window would be:

```
(powerset '(a b c)) =
(append '(() (c) (b) (b c)) '((a) (a c) (a b) (a b c)))
and
(powerset '(b c)) = (() (c) (b) (b c))
```

We feel that the code window and the algorithm window do a good job of communicating the hierarchical structure of the programming activity as well as the separate status of algorithm design.

We feel we have been successful in structuring the interface for the input of code. However, generating and understanding a dialogue with the student about algorithms are much more difficult. The most obvious method is to have the student type in an English description of his algorithm and for our tutor to try to understand that. The program has only the task of categorizing the student's description into one of the algorithm categories it is prepared to process. Even with this considerable constraint, we have only had modest success at language comprehension. One of the frequent student complaints about earlier versions of our tutor is its inability to understand algorithm descriptions. In part this is due to the tutor's limited natural language understanding and in part this is because students often only have vague ideas of algorithm choices. Our solution to both difficulties has been to implement a menu system to replace the need for natural language parsing.

Our menus are generated dynamically from the instantiations of productions in the ideal model. The menus contain English descriptions of all the algorithmic variations, correct and buggy, that we are prepared to process. Menu selection is technically simpler than language comprehension. It is an issue for further research as to which is more effective. We were surprised by how well students appear to adapt to menu selection. This may be simply that it is much easier to pick a menu entry than to generate a description.

As in the geometry system, there are general help facilities so that the student can bring up information. Again, the information is given in terms of abstract problem-solving operators. For example, rather than showing how *cons* is used in a specific program, *cons* might be explained: 'If you want to insert an element into a list use the functions *cons*' with an example to instantiate the explanation.

Novices are prone to a substantial set of misconceptions and slips in writing LISP programs. One of the strengths of the system is that we have created recognizers for a great many of the possible bugs and provided for appropriate feedback on the errors. In our pilot study, we have discovered a number of other stereotypic errors which we have also entered into the system with appropriate feedback. We are able to recognize 80 per cent of the errors our students make once we have the errors carefully catalogued.

The rightp problem

Table 1 presents a dialogue which is focused on teaching a student how to decompose a problem into a set of subfunctions. It is impossible in this limited space to present the tutor as it really appears. Instead, Table 1 is essentially a 'teletype' version of the student–tutor interactions. The actual interactions involve use of several windows displayed on the terminal screen: a code window in which the student types the code, a tutoring window in which tutoring feedback is displayed, and a goals window in which reminders are displayed about the current portion of the problem. In the 'teletype' versions that we present, the tutor's output is given in normal roman characters, while the student's input is shown in bold characters. This table presents 'snapshots' of the interaction—each time the student provokes a response from the tutor we have listed the student's input and the tutor's response (numbered for

Table 1 rightp

Define rightp. It takes three arguments, side1, side2 and side3, which represent the lengths of the three sides of a given triangle, where side3 is the longest side. The function rightp tests whether the three sides form a right triangle. In a right triangle, the sum of the squares of side1 and side2 is equal to the square of side3. However, your function should allow for a measurement error of 2 percent. In other words, your function should return *t* if the sum of the squares of the first two sides is within 2 percent of the square of the third side.

For example (rightp 3 4 5.01) = *t*, because 3 squared + 4 squared = 25, which is within 2 percent of 5.01 squared.

1. **(defun rightp (side1 side2 side3) (cond**))

Yes, you could use COND here, but there is a much easier way to test if the sum of the squares of the first two sides is within 2 percent of the square of the third. You can use a predicate, since we only have one test to do.

2. *(defun rightp (side1 side2 side3)*
(lessp (difference) ⟨*NUM2*⟩))

You want to code the amount of difference between the sum of the squares of the first two sides and the square of the third side, and you do NOT care which of these two numbers is larger. So we want to take the absolute value of their difference.

3. *(defun rightp (side1 side2 side3)*
(lessp **(abs (difference** ⟨*NUM3*⟩ ⟨*NUM4*⟩))
⟨*NUM2*⟩))

Which do you want to do first?

PRESS: IF YOU WANT TO:
1 Write code to get the sum of the squares of the first two sides
2 Write code to get the square of the third side

Menu choice: 1

4. *(defun rightp (side1 side2 side3)*
(lessp (abs (difference **(plus**) ⟨*NUM4*⟩))
⟨*NUM2*⟩))

No. You should create a helping function for this code, rather than coding it directly. Our function, rightp, is now four levels of embedding deep, and the relation we are about to code—the sum of squares of two numbers—would be a useful helping function. Thus, let us code the sum of the squares of the first two sides using a helping function.

5. *(defun rightp (side1 side2 side3)*
(lessp (abs (difference **(sumsq** ⟨*ARGUMENTS*⟩) ⟨*NUM4*⟩))
⟨*NUM2*⟩))

What do you want the helping function sumsq to do?

PRESS: IF YOU WANT TO:
1 Takes two arguments, the square of the first side and the square of the second side. Computes their sum.
2 Takes two arguments, the first side and the second side. Computes the sum of the squares of these numbers.

Table 1 (*continued*)

3 Have the tutor choose.
Menu choice: 1
No. We would not need a helping function JUST to compute the sum of two numbers. You should create your helping function so that it does more than just a sum.
It would be better to define sumsq so that it takes two arguments, in this case, [1] the first side, and [2] the second side, and computes the sum of the squares of these numbers.
Menu choice: 2

6. *(defun rightp (side1 side2 side3)*
(lessp (abs (difference (sumsq **side1 side2) (times)***))*
⟨NUM2⟩))
You should use a helping function that you have already coded, square. Since this function computes the square of a number, you can use it to code the square of the third side.

7. *(defun rightp (side1 side2 side3)*
(lessp (abs (difference (sumsq side1 side2) **(square side3))**
(plus)*))*
I do not understand that, Rob. How are you going to code 2 percent of the square of the third side?
[EXPLAIN]
OK. I will explain what I wanted. You could either code this directly or create a helping function here to code 2 percent of the square of the third side. Let us create a helping function just to make things easier. We will call this helping function percentage. We will define percentage when we are finished coding rightp.
What do you want the helping function percentage to do?
PRESS: IF YOU WANT TO:
1 Takes two arguments, one-hundredth of 2 and the square of the third side. Computes their product.
2 Takes two arguments, 2 and the third side. Computes the given percentage of the square of its second argument.
3 Takes two arguments, 2 and the square of the third side. Computes the given percentage of its second argument.
4 Takes one argument, the square of the third side. Computes 2 percent of its argument.
5 Have the tutor choose.
Menu choice: 3
Very good.

8. *(defun rightp (side1 side2 side3)*
(lessp (abs (difference (sumsq side1 side2) (square side3)))
(percentage 2 (square side3))*))*
OK. Now we have finished coding the function rightp. Let us now code the helping function(s) we used but have not yet defined.
Which do you want to do first?

Table 1 (*continued*)

PRESS: IF YOU WANT TO:
1 Define the helping function sumsq
2 Define the helping function percentage

Menu choice: 1

Define the helping function sumsq. It takes two arguments. (num1 num2) are the suggested names for the arguments, but you may name them whatever you wish. The function sumsq should compute the sum of the squares of num1 and num2.

9. **(defun sumsq (s y)**
(plus (square s) (square y)))

OK. Now we have finished coding the function sumsq. Let us now code the helping function(s) we used but have not yet defined.

Define the helping function percentage. It takes two arguments. (pcnt total) are the suggested names for these arguments, but you may name them whatever you wish. Percentage should compute pcnt percent of total.

10. **(defun percentage (x y)**
⟨PROCESS⟩)
[EXPLAIN]

OK. I will explain what I wanted. Let us figure out how to code *x* percent of the second number. How can you code *x* percent of the second number?

PRESS: IF YOU WANT TO:
1 Multiply the second number by one-hundredth of *x*.
2 Multiply the second number by *x*.
3 Divide the second number by 100.
4 Have the tutor choose.

Menu choice: 1

Good.

11. *(defun percentage (x y)*
(times *⟨NUM1⟩ ⟨NUM2⟩))*

What do you want to do first?

PRESS: IF YOU WANT TO:
1 Write code to get one-hundredth of the first number.
2 Write code to get the second number.

Menu choice: 1

12. *(defun percentage (x y)*
(times **(quotient × .01)** *⟨NUM2⟩))*

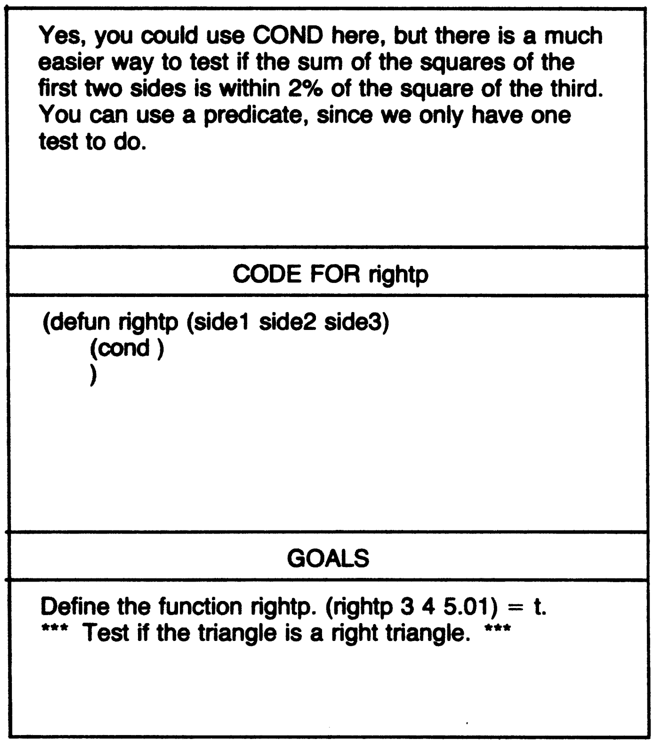
No, that is not the right number. How are you going to specify 100 in LISP?

13. *(defun percentage (x y)*
(times (quotient × **100)** y))

OK. Now we have finished coding the function percentage. There are no more helping functions to define.

convenience.) The total code as it appears on the screen is shown, although the student has added only what is different from the previous code, shown in bold face type. For example, in line 2 of Table 1 the student has typed '**(lessp (difference**' as an extension of the code in line 1. It is important to point out several things about these dialogues. First, the tutor balances parentheses and provides placeholders (e.g. ⟨CASE⟩, ⟨NAME⟩, ⟨NUM⟩) for portions of code that must be expanded. Second, although we are showing in some cases large portions of code being added from one line to the next, the tutor has understood each separate word or symbol the student has entered, and would have responded with an error message if it were necessary for any of those items.

Although the frequency of errors in this dialogue is somewhat greater than for the typical student, the dialogue is a good illustration of the various ways that the tutor responds to student errors and requests for assistance. As can be seen, the tutor allowed the student to input information until the student typed **cond**, at which point the system pointed out that a conditional structure



Yes, you could use COND here, but there is a much easier way to test if the sum of the squares of the first two sides is within 2% of the square of the third. You can use a predicate, since we only have one test to do.

CODE FOR rightp

```
(defun rightp (side1 side2 side3)
     (cond )
     )
```

GOALS

Define the function rightp. (rightp 3 4 5.01) = t.
*** Test if the triangle is a right triangle. ***

Figure 9. Screen configuration after line 1 in Table 1.

was not necessary. The actual screen image at this point in the interaction is shown in Figure 9. Line 2 illustrates that the student next types **(lessp (difference** and receives a hint which causes the student to correctly insert the absolute value function (**abs**) before **difference** in line 3. Note that the tutor presents a menu when it is uncertain what the student will do next. For instance, since the arguments to **difference** in **rightp** can be in either order, the tutor needs to know which the student will type next, and it asks the student via a menu after line 3.

After line 4 we see the tutor give the student information about when it is useful to code a separate helping function. The tutor queries the student after line 5 to make sure the student and tutor agree on what that helping function will compute. This is an example of the planning mode in the tutor. In this case, the student has a mistaken idea about what the subfunction should compute, which is then rectified by the tutor.

After line 7 the tutor detects that the student's code will not achieve the goal. However, this input does not match any of the buggy rules in the model, so the tutor provides minimal feedback: it indicates that it cannot understand the input, and queries the student to remind him about what he should be trying to code. This hint is not enough for the student who asks for an explanation by hitting a special key, whereupon the tutor helps the student specify another helping function *percentage*.

The final form of **rightp** is displayed in line 8. In line 9 the student defines the helping function **sumsq** without error and goes on to defining *percentage* in line 10. He is stuck as to what to do after typing the function body and requests an explanation. The tutor helps the student refine his algorithm. After this the student defines percentage with one error in line 12.

After defining **rightp** and its helping functions, the LISP tutor puts the student into a real LISP environment where the student can experiment with the functions that have just been defined, and can also try variations on those functions, perhaps to see what type of error would be produced by a function the student had in mind but was prevented from coding by the tutor. After the student has experimented to his satisfaction, the tutor provides the next problem in the lesson.

Evaluation

Because of the availability of the college population, we have been able to evaluate the quality of our LISP tutor more carefully than we have the geometry tutor. We have run a fairly robust version of the tutor that took ten students through basic LISP functions and data structures, function composition, function definition, predicates, conditional expressions, auxiliary functions and recursion. In a questionnaire administered to classes of students of comparable background, students reported an average of 43 hours spent studying this material, attending class and (the bulk of the time) solving problems. We ran two comparison groups of ten subjects each. One group, the human-tutor group, had private human tutors to help them work through the material. The other group, the 'on-your-own group', read the instructional material written for the other groups and solved the same sequence of problems on their own with help from a proctor only when they really got stuck (average amount of help = 6 mins/hour). In all three groups the majority of the time was spent solving problems, not reading instructions. Subjects solved 20 problems not involving function definition and 38 function definition problems spread over six lessons, one per day.

Not all subjects were able to solve the recursion problems in the allotted time, so we extrapolated how long it would have taken them to finish. Averaging the actual and extrapolated times, subjects took 11.4 hours with a human tutor, 15 hours with the computer tutor and 26.5 hours in the on-your-own condition. The difference between the computer-tutor condition and the human condition is not statistically reliable, but both are faster than the on-your-own condition.

We compared their performance on a test just before the recursion exercises. In a series of small problems the human-tutor subjects got 56 percent correct, the computer-tutor subjects got 65 percent correct and the on-your-own subjects got 64 percent correct. There are no significant differences. We also asked them to recall all the functions they could and describe what these functions did with a simple example. Subjects in the human-tutor condition recalled 22.1 functions; in the computer-tutor condition, 19.3 functions; and in the on-your-own condition, 19.3 functions. In terms of percentage of these functions given correct definitions, the results were human tutor, 82 percent; computer tutor, 77 percent; and on-your-own, 75 percent. Again there are no statistically significant differences.

To summarize, both human and computer tutors take significantly less time to bring subjects to the same criterion as the on-your-own condition. However, there is no difference in final level of knowledge. It is interesting that the on-your-own estimate (26.5 hours) is less than the class estimate (43 hours). This may reflect an overestimate by class students or the better design of our instructional material.

Cognitive principles in the design of the LISP tutor

As with the geometry tutor, it is worth reviewing how the LISP tutor does and does not achieve the cognitive-design principles set forth earlier:

1. In this system we represent both the ideal student model and the bugs as production rules.
2. We try to reify the coding goal structure in the code and placeholder

symbols. We have been less successful with communicating the goal structure behind the algorithm design but do try to communicate it by the hierarchy of menus.

3. All of the instruction about LISP is provided in the context of writing LISP code.
4. As with the geometry tutor, we try to focus the student on the critical abstract properties of problems and not on their superficial properties. For instance, error messages are general descriptions of the relevant issues.
5. There are a number of ways we try to minimize working memory. The annotated code that we display is an attempt to represent the goal structure. When students are working on examples all relevant information in these examples is kept displayed in an example window.
6. The system does provide immediate feedback on errors.
7. As with the geometry tutor, a serious deficit of this tutor is that it does not adjust grain size with the student's development.
8. As for the issue of successive approximation, the LISP tutor has had one success case and one failure case. The failure case was from a demonstration in the summer of 1984. Many students complained that they were moved onto more advanced topics before mastering easier ones. Partly in response to that we instituted more practice problems for a demonstration in the fall of 1984. In this case, students were able to successively approximate more and more advanced problem-solving skills.

FINAL CONCLUSIONS

This is a report of work in progress. We have yet to get our tutors for geometry or LISP into their final states. We have yet to completely formalize our general methodology for creating tutors. Our evaluation is still preliminary, both of the general methodology and of the specific tutors. However, we believe that the results are sufficiently encouraging to present at this time. The basic result is that it does seem possible to create computer tutors that are capable of making a major improvement in the education for at least some topics. An important additional observation is that the computer technology to deliver these tutors is rapidly becoming economically feasible. For example, the personal computers soon to be distributed to each undergraduate student at Carnegie-Mellon University are powerful enough to run our tutors.

REFERENCES

Anderson, J. R. (1981a). Tuning of search of the problem space for geometry proofs. In *Proceedings of IJCAI-81*, pp. 165–70.

Anderson, J. R. (1981b). *Acquisition of Cognitive Skill*. ONR Technical Report 81-1. Carnegie-Mellon University, Pittsburgh. Pa.

Anderson, J. R. (1983). *The Architecture of Cognition*. Cambridge. Mass.: Harvard University Press.

Anderson, J. R., Farrell, R. and Sauers, R. (1982). *Learning to Plan in LISP*. ONR Technical Report ONR-82-2. Carnegie-Mellon University.

Anderson, J. R., Farrell, R. and Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, **8**, 87–129.

Anderson, J. R., and Jeffries, R. (in press). *Novice LISP Errors: Undetected Losses of Information from Working Memory. Human–Computer Interaction*.

Anderson, J. R., Pirolli, P. and Farrell, R. Learning recursive programming. In forthcoming book edited by Chi, Farr and Glaser.

Bilodeau, I. McD. (1969). Information feedback. In E. A. Bilodeau (ed.) *Principles of Skill Acquisition*. New York: Academic Press.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective a one-to-one tutoring. *Educational Researcher*, **13**, 3–16.

Boyle, C. F., and Anderson, J. R. (1984). Acquisition and automated instruction of geometry proof skills. Paper presented at the 1984 AERA meetings.

Brooks, R. E. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man–Machine Studies*, **9**, 737–51.

Brown, J. S., Burton, R. R. and DeKleer, J. (1982). Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III. In D. Sleeman and J. S. Brown (eds) *Intelligent Tutoring Systems*. New York: Academic Press, pp. 227–82.

Brown, J. S., and Van Lehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, **4**, 379–426.

Carbonell, J. R. (1970). AI in CAI: An artificial intelligence approach to computer-aided instruction. *IEEE Transactions on Man–Machine Systems*, **11**, 190–202.

Clancey, W. J. (1983). The epistemology of a rule-based expert system—a framework for explanation. *Artificial Intelligence*, **20**, 215–51.

Cohen, V. B. (1982). Computer software found weak. *New York Times* (20 April), C4. Summary of a research.

Halasz, F., and Moran, T. P. (1982). *Analogy Considered Harmful*. Technical Report. Proceedings of the Human Factors in Computer Systems Conference, 15–17 March. Gaithersburg, Md.

Hoffer, A. (1981). Geometry is more than proof. *Mathematics Teacher*, **1981** (January), 11–18.

Jurgensen, R. C., Donnelly, A. J., Maier, J. E. and Rising, G. R. (1975). *Geometry*. Boston, Mass.: Houghton Mifflin.

Kant, E., and Newell, A. Problem solving techniques for the design of algorithms. *Proceedings of the symposium on the Empirical Foundations of Information and Software Science*. Atlanta. (Ga.)

Lewis, M., and Anderson, J. R. (in press). The role of feedback in discriminating problem-solving operators. *Cognitive Psychology*.

McKendree, J., and Anderson, J. R. (in press). Frequency and practice effects on the composition of knowledge in LISP evaluation. In J. M. Carroll (ed.) *Cognitive Aspects of Human–Computer Interaction*.

McKendree, J., Reiser, B. J. and Anderson, J. R. (1984). Tutorial goals and strategies in the instruction of programming skills. *Proceedings of the Sixth Annual Conference, Cognitive Science Society*. Boulder, Colo.

Neves, D. M., and Anderson, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (ed.) *Cognitive Skills and their Acquisition*. Hillsdale, NJ: Erlbaum.

Newell, A., and Simon, H. (1972). *Human Problem Solving*. Englewood Cliffs. NJ: Prentice-Hall.

O'Shea, T. (1979). A self-proving quadratic tutor. *International Journal of Man–Machine Studies*. **11**(1), 97–124.

Papert, S. (1980). *Mindstorms*. New York: Basic Books.

Pirolli, P., and Anderson, J. R. (in press). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*.

Rich, C., and Shrobe, H. (1978). Initial report of a LISP programmers' apprentice. *IEEE Trans. Soft. Eng*., **SE-4:6**, 456–66.

Ross, B. H. (1984). Remindings and their effects in learning a cognitive skill. *Cognitive Psychology*, **16**, 371–416.

Sheil, B. A. (1981). The psychological study of programming. *Computing Surveys*, **13**, 101–20.

Shneiderman, B. (1980). *Software Psychology*. Cambridge. Mass.: Winthrop.

Siklossy, L. (1976). *Let's Talk LISP*. Englewood Cliffs, NJ: Prentice-Hall.

Skinner, B. F. (1958). Teaching machines. *Science*, **128**, 889–977.

Sleeman, D. (1982). Assessing aspects of competence in basic algebra. In D. Sleeman and J. S. Brown (eds) *Intelligent Tutoring Systems*. New York: Academic Press.

Sleeman, D., and Brown, J. S. (eds) (1982). *Intelligent Tutoring Systems*. New York: Academic Press.

Soloway, E. (1980). *From Problems to Programs via Plans: The Context and Structure of Knowledge for Introductory LISP Programming*. COINS Technical Report 80-19. University of Massachusetts at Amherst.

Soloway, E., Bonar, J. and Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, **22**, 853–60.

Stevens, A., Collins, A. and Goldin, S. E. (1982). Misconceptions in student's understanding. In D. Sleeman and J. S. Brown (eds) *Intelligent Tutoring Systems*. New York: Academic Press.

Taylor, R. (1980). *The Computer in the School: Tutor, Tool, Tutee*. New York: Teachers College Press.

Tulving, E. (1983). *Elements of Episodic Memory*. London: Oxford University Press.

Tulving, E., and Thomson, P. M. (1973). Encoding specificity and retrieval processes in episodic memory. *Psychological Review*, **80**, 352–73.

Van Lehn, K. (1983). *Felicity Conditions for Human Skill Acquisition: Validating an AI-based Theory*. Technical Report CIS-21, Xerox Parc. Palo Alto, Calif.