Cognitive Modeling and Intelligent Tutoring*

John R. Anderson, C. Franklin Boyle, Albert T. Corbett and Matthew W. Lewis
Advanced Computer Tutoring Project, Carnegie-Mellon University, Pittsburgh, PA 15213-3890, USA

Introduction

Research on intelligent tutoring serves two goals. The obvious goal is to develop systems for automating education. Private human tutors are very effective [16] and it would be nice to be able to deliver this effectiveness without incurring the high cost of human tutors. However, a second and equally important goal is to explore epistemological issues concerning the nature of the knowledge that is being tutored and how that knowledge can be learned. We take it as an axiom that a tutor will be effective to the extent that it embodies correct decisions on these epistemological issues.

We chose intelligent tutoring as a domain for testing out the ACT* theory of cognition (Anderson [4]). It was a theory that made claims about the organization and acquisition of complex cognitive skills. The only way to adequately test the sufficiency of the theory was to interface it with the acquisition of realistically complex skills by large populations of students. When we read the *Intelligent Tutoring* book, edited by Sleeman and Brown [37], it became apparent that the authors in it were explicitly or implicitly performing such tests of theories of cognition and that it was an appropriate methodology for testing the ACT* theory.

The ACT* theory has been used to construct *performance models* of how students actually execute the skills that are to be tutored and *learning models* of how these skills are acquired. These two cognitive models are incorporated into our tutors and are used to interpret the student's behavior. A performance model consists of a set of correct and incorrect rules for performing the skill in question and is used in a paradigm we call model tracing. In this paradigm we

* This paper reports research supported by a number of grants: NSF Nos. MDR-8470337 and IST-8318629. ONR No. N00014-84-K-0064, and ARI No. MDA 903-85-K-0343.

Artificial Intelligence **42** (1990) 7–49
0004-3702/90/$3.50

compare the student's responses to the rules in the model in an attempt to follow in real time the cognitive states that the student goes through in solving a problem. The learning model consists of a set of assumptions about how the student's knowledge state changes after each step in solving a problem. This model is employed in knowledge tracing (as opposed to model tracing) to track the changes in the student's knowledge across problems. The information that results from knowledge tracing can be used to disambiguate alternative interpretations in model tracing and can be used for selecting problems to optimize learning.

We are currently working on tutors for beginning LISP programming [35], for proof generation in high-school geometry [8], and for solving algebraic manipulation and word problems [30]. These domains were selected because they involve the acquisition of well-defined skills and we can catch students at the point where they are just beginning to learn the skill. Our LISP tutor currently teaches a successful university-level course, our geometry tutor has completed two years of successful use in a local public high school, and the algebra tutor is being used in a local public high school in the 1987–1988 academic year. We believe that these tutors owe their success to the cognitive principles from which they were derived. However, it is not the case that the cognitive principles have remained unchanged in the face of these applications. In fact, we have found reasons to reject certain assumptions of the ACT* cognitive architecture and are working with a new architecture called PUPS (for PenUltimate Production System). So, even at this early stage of our endeavor, we have seen a fairly profitable flow of influence back and forth between the theory and the application.

This paper has three major sections. Section 1 describes the cognitive theory that serves as the basis for our tutoring endeavors. Section 2 describes the model-tracing methodology and how it derives from our cognitive theory. Section 3 discusses the issues that arise in implementing the model-tracing methodology.

1. The Cognitive Theory

In describing this cognitive theory, we want to make clear from the outset that we are not necessarily describing what is in our tutors. Instead, we are describing a theory that forms the basis for the tutors. If the mind functions according to our theory, then the tutors should prove to optimize the learning process. To derive predictions from our cognitive theory, we have developed a number of simulations of aspects of it. At some points the code in these simulations has been taken over whole cloth for the tutors, at other points it has influenced tutor code, and at other points the tutor code is just a derivation of the theory. Later we will discuss the tutor implementations. We will just outline the basic cognitive theory here. For details and empirical evidence the reader is referred to Anderson [4, 5] and Anderson and Thompson [13].

In both the PUPS theory and its ACT* predecessor, a fundamental theoretical distinction was made between declarative and procedural knowledge. This distinction borrowed its label from the distinction drawn in AI a decade ago (e.g., Winograd [38]) but has been fundamentally transformed to be a psychological distinction. Declarative knowledge is distinguished by the fact that the human system can encode it quickly and without commitment to how it will be used. Declarative knowledge is what is deposited in human memory when someone is told something, as in instruction or reading a text. Procedural knowledge on the other hand can only be acquired through the use of the declarative knowledge, often after trial and error practice, and is further characterized by the fact that it embodies the knowledge in a highly efficient and use-specific way. In the theory, procedural knowledge derives as a by-product of the interpretative use of declarative knowledge. We use the term knowledge compilation to refer to the learning process which creates the procedural knowledge.

1.1. Procedural knowledge: Productions

In the ACT* and PUPS theories, procedural knowledge is represented by a set of production rules that define the skill. Our goal in tutoring is basically to create experiences that will cause students to acquire the production rules which would be possessed by the competent problem solver. It would be worthwhile to examine some examples of productions that are used in our three domains of tutoring—i.e., LISP, geometry, and algebra.

1.1.1. *LISP*

Below are "Englishified" versions of a couple of the productions that are used in the LISP tutor:

IF the goal is to merge the elements of lis1 and lis2 into a list,
THEN use append and set as subgoals to code lis1 and lis2.

IF the goal is to code a function on a list structure and that function must inspect every atom of the list structure and the list structure can be arbitrarily complex,
THEN try car-cdr recursion and set as subgoals
(1) to figure out the recursive relation for car-cdr recursion
(2) to figure out the terminating cases when the argument is nil or an atom.

The first is a production that recognizes the relevance of a basic LISP function and the second is one that recognizes the applicability of a recursive programming technique. These and approximately 500 more production rules model an ideal student writing basic LISP code to solve problems that would appear in an introductory LISP textbook. These productions all have this goal decomposition

character of starting with some programming goal and decomposing it into subgoals until goals are reached which can be achieved with direct code. For an extensive discussion of a model of beginning LISP programming see [10].

1.1.2. *Geometry*

The character of the production rules underlying the geometry tutor are somewhat different. Below are two examples from the approximately 300 in that system:

IF the goal is to prove $\triangle XYZ \cong \triangle UYW$
and X, Y, W are collinear,
and U, Y, Z are collinear,
THEN conclude $\angle XYZ \cong \angle UYW$ because of vertical angles.

IF the goal is to prove $\triangle XYZ \cong \triangle UVW$
and $\overline{XY} \cong \overline{UV}$
and $\overline{YZ} \cong \overline{VW}$,
THEN set a subgoal to prove $\angle XYZ \cong \angle UVW$ so SAS can be used.

The first production makes a forward inference from what is known about a problem while the second makes a backward inference from what is to be proved. A proof is completed when a set of subgoals from the to-be-proven statement makes contact with a set of forward inferences from the givens of the problem. The production rules for forward and backward inference are contextually constrained. That is, they make reference not only to the information necessary for application of the rule but also to other information about the proof which is predictive of the aptness of that inference. Thus, for instance, the first rule not only makes reference to the collinearity information which is logically necessary for application of the vertical angle rule, it also makes reference to the fact that these angles are corresponding parts of to-be-proven congruent triangles. For more discussion of the nature of the ideal student model in geometry read [1, 8].

1.1.3. *Algebra*

The production system for the algebra tutor is again somewhat different in character from the production systems for LISP or geometry. Below are seven of the production rules involved in modeling the ideal student's knowledge of distribution:

IF the goal is to solve an equation with a subexpression of the form "coefficient(exp1 + exp2)",
THEN set as a subgoal to rewrite the equation with the subexpression distributed.

IF the goal is to rewrite an equation with a subexpression distributed,

THEN set as subgoals
(1) find the coefficient associated with the subexpression,
(2) multiply the parenthesized part by the coefficient,
(3) replace the subexpression by the product.

IF the goal is to find the coefficient of "term",
THEN the answer is 1.

IF the goal is to find the coefficient of "– term",
THEN the answer is -1.

IF the goal is to find the coefficient of "num term",
THEN the answer is num.

IF the goal is to multiply "num1" by "term1 + term2",
THEN set as subgoals
(1) to multiply term1 by num1,
(2) to multiply term2 by num1,
(3) to combine the two products.

IF the goal is to multiply an expression by a number,
THEN set as subgoals
(1) to find the coefficient associated with the expression,
(2) to multiply the coefficient by the number,
(3) to combine the product with the rest of the expression.

These rules would be invoked if, for instance, there were an expression of the form $\ldots 3(5x + 2) \ldots$ somewhere in the equation to be solved. The first rule recognizes the applicability of distribution and the second sets three subgoals to accomplish this. The third and fourth rules are special cases of extracting coefficients of 1. The fifth applies in this case and extracts the coefficient of 3. The sixth rule decomposes the distribution into two simpler multiplications. The final production sets the subgoals to extract the 5 from the $5x$, multiply 5 by 3, and then to combine the 15 with x.

The algebra rules highlight the issue of grain size which is also an issue for other production systems. We could have compacted all of these rules into a single production rule which recognized and applied distribution to the equation in one fell swoop (as, for instance, Sleeman [36] does). On the other hand, we could have broken each of these steps into multiple substeps. For instance, note that we do not decompose the process of calculating the product of num1 and num3 into a set of substeps as it might well be implemented cognitively. Our decision about the level at which to model the student was determined by pedagogical considerations. Students entering the algebra course usually have their multiplication skills well-learned and do not need to be tutored on these. In contrast, students do have problems with the subcomponents of distribution and so we need to separate these out for purposes of separate tutoring.

An implication is that the production rules that we use in the algebra tutor, and indeed in the other tutors, represent only upper levels of the skill. These productions set subgoals which are met by other productions whose action we do not bother to simulate. These include such things as the actual typing of answers into the computer. The assumption is that such productions, below the level that we are modeling, are well-learned.

While the production systems for the different domains do have some features in common, the production rules in each domain create different goal structures. Our learning theory would predict that the different task structures of the different domains produce different organizations of the production rules. Generating LISP code is a design activity and lends itself to a problem decomposition structure. The search character of generating geometry proofs produces an opportunistic structure in which there can be large switches of attention among parts of the proof. The linear structure of the algebra equations and the algorithmic character of algebra equation solving produces the symbol substitution character of the algebraic rules. One of the major functions of a tutor for a particular domain should be to communicate the ideal problem-solving structure of that domain.

1.2. Declarative knowledge: PUPS structures

According to our cognitive theory, knowledge is initially encoded declaratively in what we have come to call PUPS structures. At first these structures are used by weak problem-solving productions. As a result of this activity, the knowledge is converted into use-specific production form. PUPS structures are basically schema-like structures which are distinguished by the fact that they have certain special slots which prove critical to their interpretive application in problem solving. These include the function slot which serves to indicate the function of the entity represented by the structure, the form slot which indicates its form or physical appearance, and the precondition slot which states any preconditions that must be satisfied for that form to achieve that function. To illustrate such structures let us consider how an ideal student might encode the following fragment of text from the second edition of Winston and Horn [39, p. 24]:

The value returned by car is the first element of the list given as its argument.

```
(CAR '(FAST COMPUTERS ARE NICE))
  FAST
```

This Winston and Horn example is interesting because it contains a nice juxtaposition of some abstract instruction with a specific example. However, the PUPS encodings of the two (given below) are basically structurally isomorphic. The abstract encoding of **car** indicates in its function slot that it serves

as the function in the abstract LISP code represented by **car-structure**. The representation of **car-structure** shows in its form slot the abstract template for function calls involving **car** and in its function slot it specifies what these function calls calculate. The **example1** structure has the same form as **car-structure**, except that an argument is specified. The other PUPS structures encode that argument and the value returned by the example call.

car
ISA: function
FUNCTION: (function-in **car-structure**)
FORM: (text **car**)

car-structure
ISA: lisp-code
FUNCTION: (calculate-first **arg**)
FORM: (list **car arg**)

example1
ISA: lisp-code
FUNCTION: (illustrate **car**)
(calculate-first **lis**)
FORM: (list **car lis**)

lis
ISA: list
FUNCTION: (argument-in **example1**)
(hold (**fast computers are nice**))
FORM: (text '(**fast computers are nice**))

fast
ISA: atom
FUNCTION: (value-of **example**)
(first **lis**)
FORM: (text **fast**)

The structures above represent the outcome of successful encoding of the text; however, it should be stressed that there is a lot of room for "misunderstanding" (incorrect encoding). Clearly, a critical issue for learning is correct interpretation of the instruction. One problem with virtually all instructional material is that it omits many things that the student needs to know in order to perform the tasks, and the student is left to figure them out by trial and error experimentation. One of the payoffs in developing an ideal student model, even before it is used in tutoring, is that it provides a cognitive analysis of what the student really needs to know. Instruction can then be designed to communicate that. In our work we have found that instructional materials designed to communicate all the information in the ideal model (and to not waste prose

communicating non-information) are more effective than standard texts even without a tutor. This emphasis on economy and focus in instruction has been confirmed by a number of other researchers (Carroll [20], Reder, Charney and Morgan [34]). It is the motivation for our text on LISP (Anderson, Corbett and Reiser [9]).

However, we believe that it is not possible to avoid all or even most misinterpretations. In communicating unfamiliar material there is the inevitable difficulty of the student being weak on the key concepts. For instance, we have never observed a student go from reading any textbook on LISP to practicing that knowledge without errors. One important role for a tutor is to monitor for these errors of misunderstanding and correct them as they show up in the performance of a task.

1.2.1. *Interpretive use of declarative knowledge*

We assume that the declarative PUPS structures illustrated above are deposited in memory essentially as the product of language comprehension. It is important that the necessary structures get encoded correctly, but this is by no means the end state of the learning process. These structures do not directly lead to any performance and it is necessary to interpret them to get performance. This interpretive process is of high demand cognitively and is a major cause of slips in performance [11, 32]. Thus, it is important to create productions like the ones in the ideal model which will automatically apply the knowledge.

There is essentially a double loop of inefficiency promoted by interpretive use of declarative knowledge. The outer loop involves a search through the operations the student knows to find an appropriate next step. For instance, a student might search through all the postulates for proving the triangles congruent: side-side-side, side-angle-side, etc. While it is not possible to entirely avoid search, the productions in the ideal model have features built into them that greatly cut down on this search. The example productions we displayed earlier illustrate this in that they include heuristic tests that check the likelihood that a rule of inference would contribute to a final proof. The inner loop involves the analogical application of a declarative PUPS-structure representation of an operation to the problem at hand in order to produce a response. This analogical application of declarative knowledge is costly in terms of the amount of information that must be held in working memory. For instance, a great deal of prolonged effort can go into an attempt to map the general statement of the side-angle-side postulate to a specific problem [2]. Once the corresponding information is proceduralized, however, its application makes a much smaller demand on working memory.

1.2.2. *Analogy*

We have observed that the major way that students solve problems involving concepts is by analogy to examples of solutions involving these concepts. To

illustrate the analogy process, suppose the student has the goal of getting the first element of the list (*A B C*). This is represented by the PUPS structures below:

goal1
ISA: lisp-code
FUNCTION: (calculate-first **lis2**)
FORM: ?

lis2
ISA: list
FUNCTION: (hold (**A B C**))
FORM: ?

As is typically the case in the PUPS representation of a problem-solving situation we have PUPS structures with functions represented but forms empty. The goal is to devise a form that satisfies each functional specification. Both of the required forms can be calculated by analogy to the earlier PUPS structures created from comprehension of the Winston and Horn instruction. Using **example1** as the source for the analogy and **goal1** as the target, PUPS creates the following analogy:

function(example1):form(example1)::function(goal1):?

In solving this analogy, **lis** from **example1** is mapped to **lis2** from **goal1** and the specification (LIST **CAR lis2**) is created for the **goal1** form slot. A similar analogy between **lis** and **lis2** leads to the description (LIST '(**A B C**)) for the form slot of **lis2**. This constitutes a solution to the problem.

1.3. Knowledge compilation

What we have just described is a solution by analogy for a specific example problem. Such analogical reasoning is not optimal for problem solving, however, because it is costly to compute the mapping, and because it will only work when there is an example at hand. Therefore, knowledge compilation tries to analyze the essence of the analogical solution and generate a production rule that can produce the solution at will. Basically, it does this by looking at the problem states before and after generating the analogical solution and creating a production rule that maps one onto the other. Essential to knowledge compilation is diagnosing what was critical in the before situation and what is critical in the solution. This depends on the semantics of the PUPS structure. The result of the compilation process for this example is a production with one variable (=list) that can bind to any list:

IF the goal is to get the first element of =list,
THEN type (car =list).

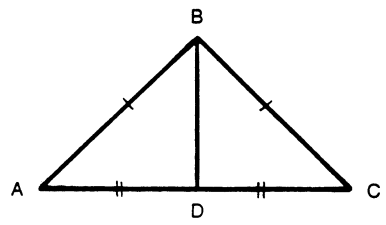
The knowledge compilation process that produced this has to know about the correspondences computed in calculating the analogy. Thus, this learning mechanism has built into it knowledge of how PUPS structures are interpreted in analogy.

A second thing knowledge compilation will do is eliminate some of the relatively blind search that characterizes early problem solving. Consider the diagram in Fig. 1, which shows a problem that appears early in the geometry problem sequence. The student is given that two sides of the triangles are congruent and must try to prove that the triangles are congruent. At this point the student has only been taught the side-side-side and side-angle-side postulates for proving triangles congruent. One student, not atypical, was observed to (1) try side-angle-side but fail because there is not an angle congruence; (2) try side-side-side but fail because only two sides are given as congruent; (3) apply the definition of congruence to infer that the measure of $\overline{AD}$ is equal to the measure of $\overline{CD}$; (4) apply the reflexive rule to infer $\overline{AD}$ is congruent to itself; and, (5) finally, apply the reflexive rule, to infer that $\overline{BD}$ is congruent to itself. This last step was the key one that allowed the student to apply the side-side-side rule to achieve his goal. It seemed that the subject engaged in an almost random search of legal operators until he came across one that was useful.

Knowledge compilation creates rules that skip over the steps that were not relevant to the final solution and tries to produce a rule that connects key features in the original situation with the ultimately useful operator. The rule that should be produced in this case is:

IF the goal is to infer $\triangle XYZ \cong \triangle UYZ$,
THEN infer $\overline{YZ} \cong \overline{YZ}$ because of the reflexive property of congruence.

Note this rule is not specific to the solution of this problem by side-side-side



Given: $\overline{AB} \cong \overline{BC}$
$\overline{AD} \cong \overline{CD}$
Prove: $\triangle ABD \cong \triangle CBD$

Fig. 1. A problem that occurs early in the problem sequence used with the geometry tutor.

nor to the fact that there are already two sides proven congruent. This is what we noted of our subject: He emerged from this episode with a tendency to infer that the shared side of two triangles is congruent to itself whenever he set as his goal to prove these triangles congruent.

This geometry example illustrates the general features of learning from search: If the student applies a number of operators and some of the operators prove successful—in the geometry example a number of inferences were applied and one was part of the final proof—then some knowledge may be proceduralized while additional declarative structures may be formed that encode how the operators achieved their successful function. With subsequent practice these additional declarative structures can lead to the formation of more productions. It is critical that the students properly encode their experience and this is again where tutors can be critical—by assuring the proper encoding of the experience. So, for instance, in the reflexive case discussed above, if the student represented the function of the rule as establishing side-side-side, he would have created too specific a rule. On the other hand, if he represented it as just making a legal inference, he would have created too general a rule.

1.4. Strengthening

In addition to knowledge compilation, there is a simple strengthening of declarative and procedural knowledge with use. As knowledge becomes strengthened it comes to be applied more rapidly and reliably. There is ample empirical evidence for such a simple learning process in humans although its exact nature is in some dispute [2]. The major implication of a strengthening-like process for tutoring concerns the introduction of new knowledge. As the execution of acquired knowledge becomes more proficient there is more capacity left over to properly process the new knowledge.

1.5. Other learning mechanisms?

An important characteristic of this model is what it does not contain. Unlike the ACT* line of learning theories there are no inductive learning mechanisms that automatically compare the current situation with past situations and try to form generalizations and discriminations about when rules will and will not apply. This is not to say that subjects do not engage sometimes in inductive behavior as a conscious problem-solving activity—they certainly do. Rather the claim is that there is not an automatic learning mechanism of the status of compilation and strengthening. Generalizations and discriminations are declarative knowledge structures produced by problem-solving productions rather than productions produced by automatic learning mechanisms. There is a fair amount of evidence that people are aware of their inductive generalizations

and discriminations (Lewis and Anderson [29], Dulany, Carlson and Dewey [24]).

This has major implications for instruction. Rather than leaving students to induce generalizations and discriminations from carefully juxtaposed examples, which would have been the pedagogical implication of ACT*, one should simply tell the student what the critical features are. Thus, if a student is overusing the vertical angle inference, he should be told the circumstances under which he wants to use it. This is not to argue that examples are not important, but they should be annotated with information about what they are supposed to illustrate.

2. Converting Theory to Tutoring: Model Tracing

This theory of knowledge acquisition is radical in the juxtaposition of its simplicity and its claim to completeness. To review, learning in the theory involves:

(1) acquisition of new declarative knowledge by the processing of experience through existing productions (e.g. for language comprehension);
(2) application of declarative knowledge to new situations (i.e., situations for which productions do not exist) by means of analogy and pure search;
(3) compilation of domain-specific productions;
(4) strengthening of declarative and procedural knowledge.

Probably there is little controversy that these things (or things very similar to them) are involved in knowledge acquisition, but the issue is whether these assumptions are sufficient to account for all knowledge acquisition. The question is how to put that theory to test. As argued in detail elsewhere [6] our tutoring work is a methodology for testing the theory. Since the design of the tutors is based on the theoretical analysis, the success of the tutors, as measured by post tests and total learning time, is one test of the theory. Moreover, one can ask whether detailed analyses of the student's interaction with the tutor accord with theoretical predictions.

The simplicity of the underlying theory maps onto a rather straightforward tutoring methodology that we call model tracing. The theory provided us with a *performance model* which specifies how a student's knowledge state will map onto performance on a particular problem. The performance model can be used to interpret the student's performance on a particular problem. Instruction is generated to address any confusions that the student is interpreted as showing and to keep students on a correct solution path. In addition, a *learning model* which specifies how the student's knowledge state will change as a result of problem-solving experiences can be used to trace the student's knowledge state over time. Problems and accompanying instruction are selected to practice the student on productions that are diagnosed as weak or missing in

the student's knowledge state.[1] Given this structuring of the learning situation, we trust the automatic learning mechanisms in (1)–(4) above to move the student forward on an optimal learning trajectory. In the following sections we will give some examples of this model-tracing methodology. Then, we will discuss some issues in implementing it.

2.1. The LISP tutor

The LISP tutor is based on our earlier efforts to model learning to program in LISP [10]. Appendix A contains a dialogue with a student coding a recursive function to calculate factorial. This does not present the tutor as it really appears. Instead, it shows a "teletype" version of the tutor where the interaction is linearized. In the actual tutor the interaction involves updates to various windows. In the teletype version the tutor's output is given in normal type while the student's input is shown in bold characters. These listings present "snapshots" of the interaction; each time the student produces a response, we have listed his input along with the tutor's response (numbered for 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 boldface type). For instance, in episode (2) he has added "zero" as an extension of "(defun fact (n) (cond ((".

In the first line, when the subject typed "(defun", the template

(defun ⟨name⟩ ⟨parameters⟩ ⟨body⟩)

appeared. The terms in angle brackets (⟨ ⟩) denote pieces of code the student will supply. The subject then filled in the ⟨name⟩ slot and the ⟨parameters⟩ slot and had started to fill in the ⟨body⟩ slot. Parentheses are automatically balanced and syntax is checked. The motivation here is to remove from the student some of the cognitive load required for checking low-level syntax and to enable the student to focus on higher-level coding problems.

Although the student has some difficulty with the syntax of the conditional tests in episodes (1) and (2), he basically codes the terminating case for the factorial function correctly. Typically, we find students have little difficulty with terminating cases but have great difficulty with recursive cases. Therefore, after episode (3) the tutor interrupts the student to see if the student understands how to write the recursive code. When the student's answer to the first question indicates lack of knowledge, the tutor starts a dialogue to guide the student through a design of the recursive function. Basically, it leads the student to construct a couple of examples of the relationship between fact (n) and fact (n − 1) and then gets the student to identify the general relationship.

[1] The actual learning system is not simulated in our tutor—rather we more directly (and much more efficiently) calculate its implications.

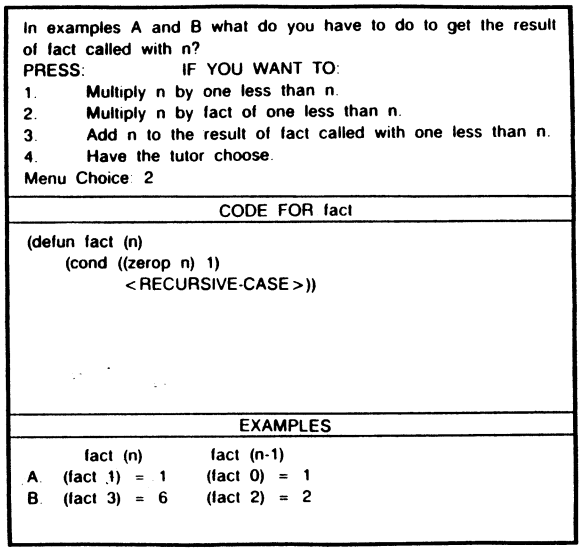


Fig. 2. The screen configuration before episode (4) in Appendix A.

Figure 2 shows the screen image at a critical point in the design of this function.

The dialogue after this point shows two errors that students make in defining recursive functions. The first, in episode (4), is to call the function directly without combining the recursive call with other elements. The second, in episode (5), is to call the function recursively with the same argument rather than a simpler one.

After the student finishes coding the function, he goes to the LISP window and experiments. He is required to trace the function, and the recursive calls embed and then unravel. Figure 3 shows the screen image at this point with the code on top and the trace below it.

This example illustrates a number of features of our tutoring methodology.

(1) The tutor constantly monitors the student's problem solving and provides direction whenever the student wanders off one of the correct solution paths.

(2) The tutor tries to provide help with both the overt parts of the problem solution and the planning. However, to address the planning a mechanism had to be introduced in the interface (in this case menus and short answers) to allow the student to communicate the steps of planning.

(3) The interface tries to eliminate aspects like syntax checking, which are irrelevant to the problem-solving skill being tutored.

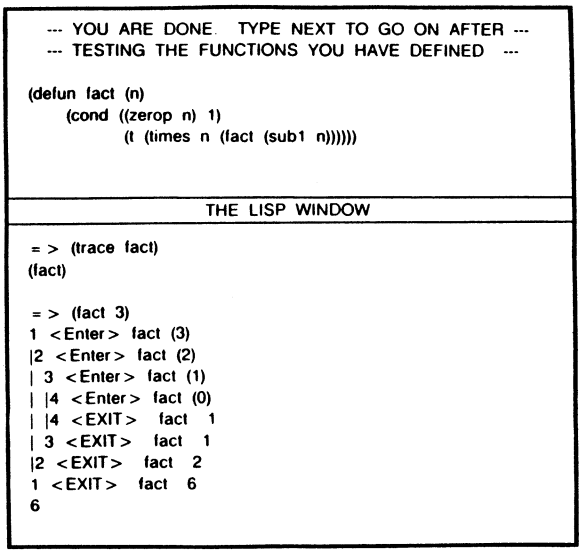


Fig. 3. The screen configuration at the end of the dialogue in Appendix A.

(4) The interface is highly reactive in that it makes some response to every symbol the student enters.

It is interesting to note the contrast between the LISP tutor and the PROUST system of Johnson and Soloway [27]. That system provides feedback only on residual errors in the program and does not try to guide the student in the actual coding. One technical consequence is that the PROUST system has to deal with disentangling multiple bugs. Since the LISP tutor only corrects errors immediately, the code never contains more than one bug at a time.

2.2. The geometry tutor

The geometry tutor is similarly based on our earlier work studying geometry problem solving (Anderson [1–3]). Figure 4 illustrates how a problem is initially presented to a student. At the top of the figure is the statement the student is trying to prove. At the bottom are the givens of the problem. In the upper left corner is a problem diagram. The system prompts the student to select a set of statements using a mouse. Then the system prompts the student to enter a rule of geometric inference that takes these statements as premises. When the student has done so, the system prompts the student to type in the conclusion that follows from the rule. The screen is updated with each step to indicate where the student is. The sequence of premises, rule of inference, and conclusion completes a single step of inference. Figure 5 illustrates the screen

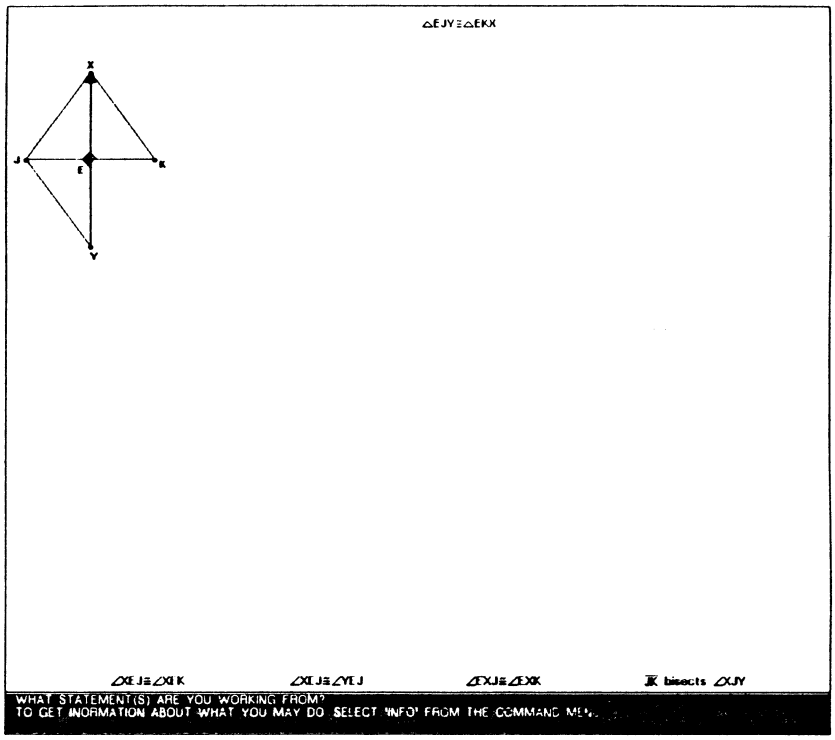


Fig. 4. An initial screen configuration with the geometry tutor.

at the point where the student has selected the definition of bisector to apply to the premise $\overline{JK}$ bisects $\angle XJY$ but has not yet entered the conclusion. A menu has been brought up at the left of the screen to enable the entry of the conclusion. It contains the relations and symbols of geometry. By pointing to symbols in the menu and to points in the diagram, the student can form the new statement $\angle XJK \cong \angle KJY$. We find it useful to have the student actually point to the diagram to make sure the student knows the reference of the abstract statements.

Figure 6 shows the geometry diagram at a still later point. The student has completed the bisector inference and added a plausible transitivity inference, but one that proves not to be part of the final proof. At this point the student begins to flail and has tried a series of illegal applications of rules, the most recent being application of angle-side-angle (ASA) to the premises $\angle EJX \cong \angle EJY$ and $\angle EXJ \cong \angle EXK$. The tutor points out that ASA requires three premises, and so it clearly is inappropriate. Since the student is having so much difficulty, the tutor points the student to the key step in solving this problem: To prove $\triangle EJY \cong \triangle EKX$ one will have to prove $\triangle EJY \cong \triangle EJX$ and

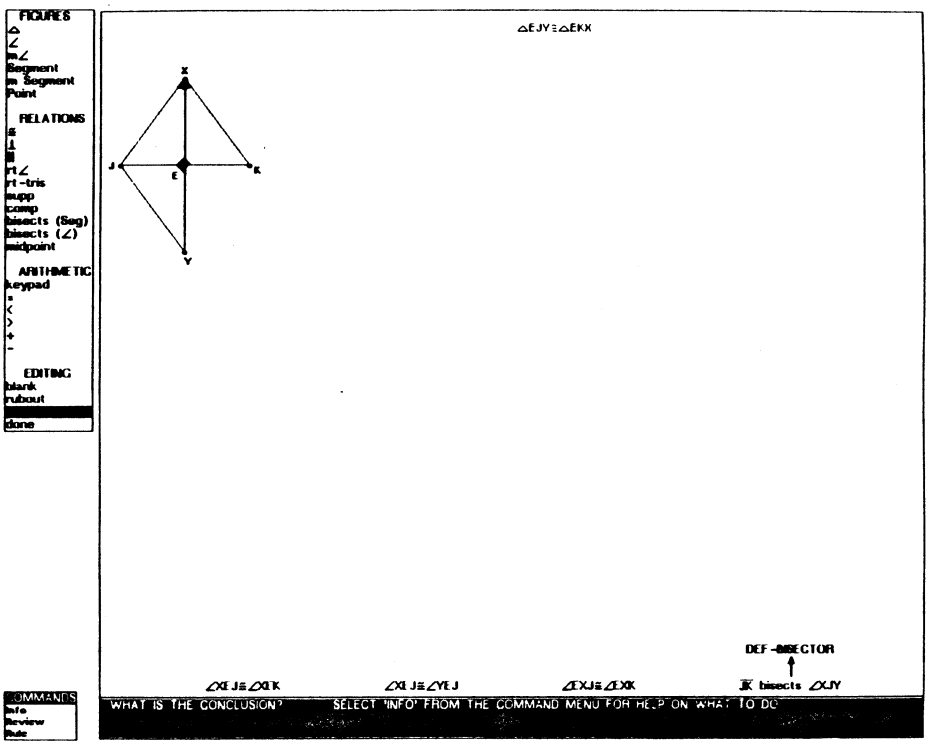


Fig. 5. The screen configuration after the student has selected the premises and the rule and is about to enter the conclusion.

$\triangle EJX \cong \triangle EKX$ and then apply transitivity. The tutor indicates this key step by boxing the conclusion. Thus, the student is asked to use backward inference to enter a rule and a set of premises from which the conclusion logically follows. If necessary, the tutor can step the student through how transitivity of the two triangle congruences will enable the conclusion to be proven. The student then will have the task of proving the two triangle congruences.

Figure 7 shows the state of the diagram at a still later point where the student has proven one of the triangle congruences while the other remains to be proven. It nicely illustrates how students can mix reasoning forward from the givens and reasoning backwards from the conclusions.

Figure 8 shows the completed proof in which there is a graph structure connecting the givens to the to-be-proven statement. Students find such representations of proof solutions enlightening in two ways. First, it enables them to appreciate how inferences combine to yield a proof, something they tend not to get from the traditional two-column formalism. Second, the search inherent in proof generation is explicitly represented. So, for instance, students

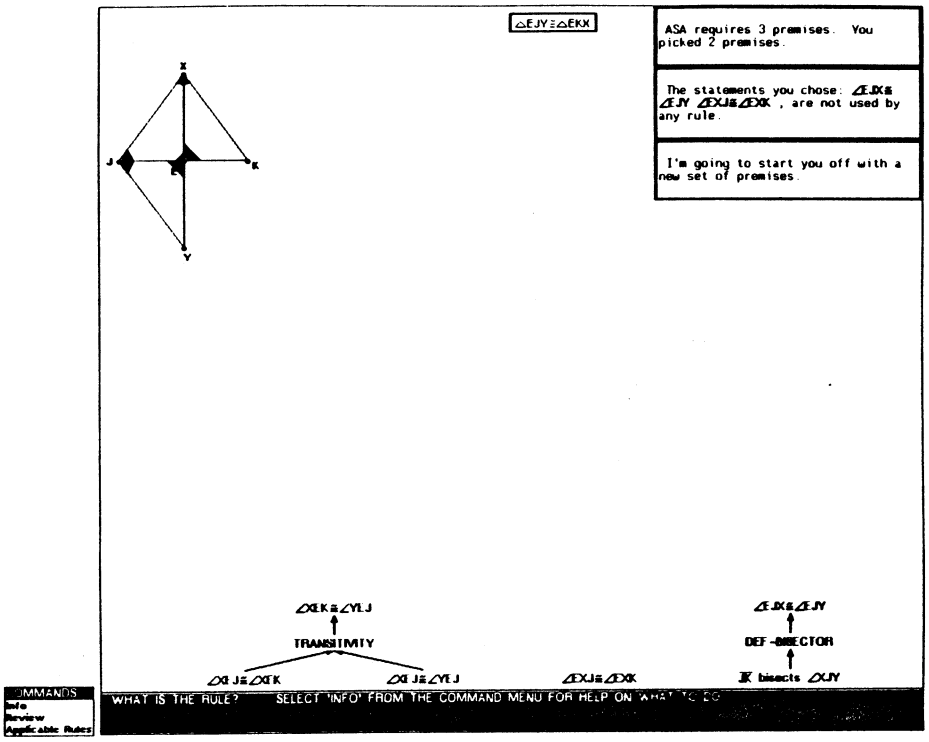


Fig. 6. The student has just tried to apply angle-side-angle to the two premises $\angle EJX \cong \angle EJY$, $\angle EXJ \cong \angle EXK$.

can immediately identify inferences, such as the angle transitivity inference, which are off the main path.

Much of our work on the geometry tutor has been concerned with principles for providing immediate feedback. We will postpone discussion of these principles until the next section on issues involving the model-tracing methodology. The example in Figs. 4–8 illustrates what Brown has referred to as reification. The proof graph makes concrete two abstract features of problem solving in geometry—the logical relationships among the premises and conclusions and the search process by which one hunts for a correct proof. Normally, students have a great deal of difficulty with both of these constructs. By creating an external referent in the form of a proof graph we facilitate instruction about these abstract concepts. Students unanimously report that they prefer this proof graph structure to the more traditional two-column proof form. They typically justify their preferences with the assertion that it is "easier to do a proof" with this formation.

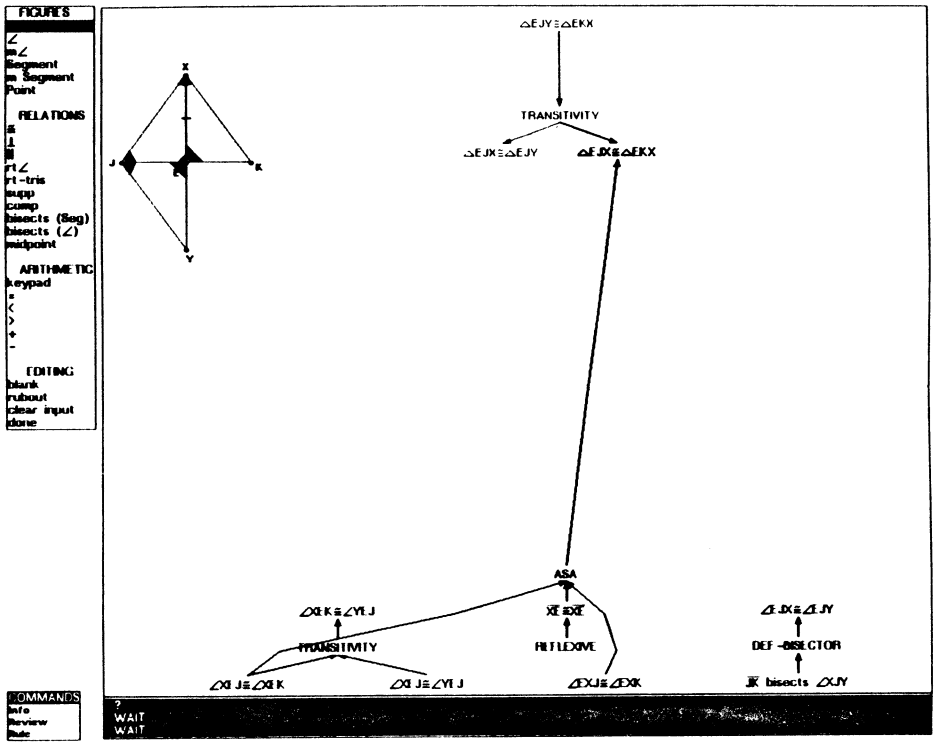


Fig. 7. The student has succeeded in proving one of the two requisite triangle congruences.

2.3. The algebra tutor

The algebra tutor (Lewis, Milson and Anderson [30]) is a more recent endeavor of ours and does not have the prior history of domain study. It reflects an attempt to see how well the methodology that we have developed transfers to a new domain.

So far we have developed a curriculum which takes students through a review of pre-algebra, and through linear and quadratic equations. Figure 9 shows the initial state of the screen at the beginning of solving a linear equation. To the left is a solution window in which the student is going to develop the solution. There are three areas to the right. At the top, there is a blackboard on which the tutor posts messages to the student. In the middle, there is a calculator scratchpad in which the student can perform arbitrary manipulations. At the bottom, there is a menu of choices which the student can point to in order to interact with the tutor. The menu varies; in Fig. 9 it largely consists of options for piecing together a solution to the problem. The student has the option of directly writing in the answer without showing intermediate

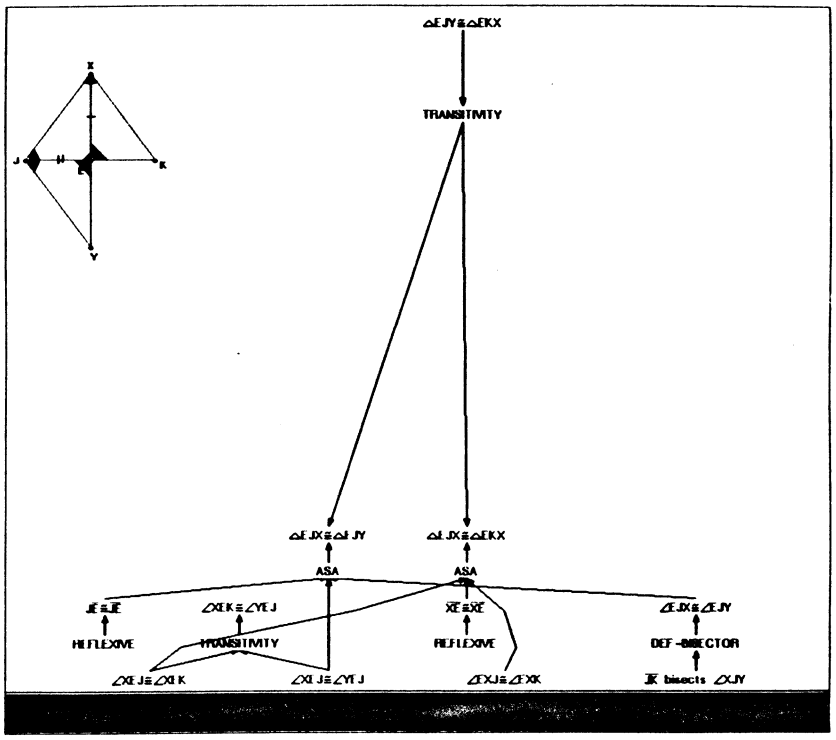


Fig. 8. The proof of the problem is now complete.

results. The student can also decompose the solution into a number of substeps which can be indicated to the tutor by selecting the operations item in the menu.

A new menu replaces the one at the bottom left of Fig. 9 when the student selects "operations". This new menu is shown in Figure 10. The change is that a new menu has come up with possible operations that might be performed on equations. The correct choice at this point would be "cleanup" which refers to eliminating parentheses and collecting like terms. Figure 11 shows the contents of the solution window after the student selects "cleanup." A cleanup form has appeared and the student must figure out what arguments to pass to this operation and what the result will be. The student can point to the equation $3-3(x-4)=-x$ in the solve line above and it will appear as an argument to cleanup. This is one example of many where we try to minimize the number of operations that the student must perform.

Just as the student can decompose "solve-equation" into a number of substeps so the student can decompose "cleanup" into a number of substeps. The first substep for "cleanup" is "distribute." Again, "distribute" can be

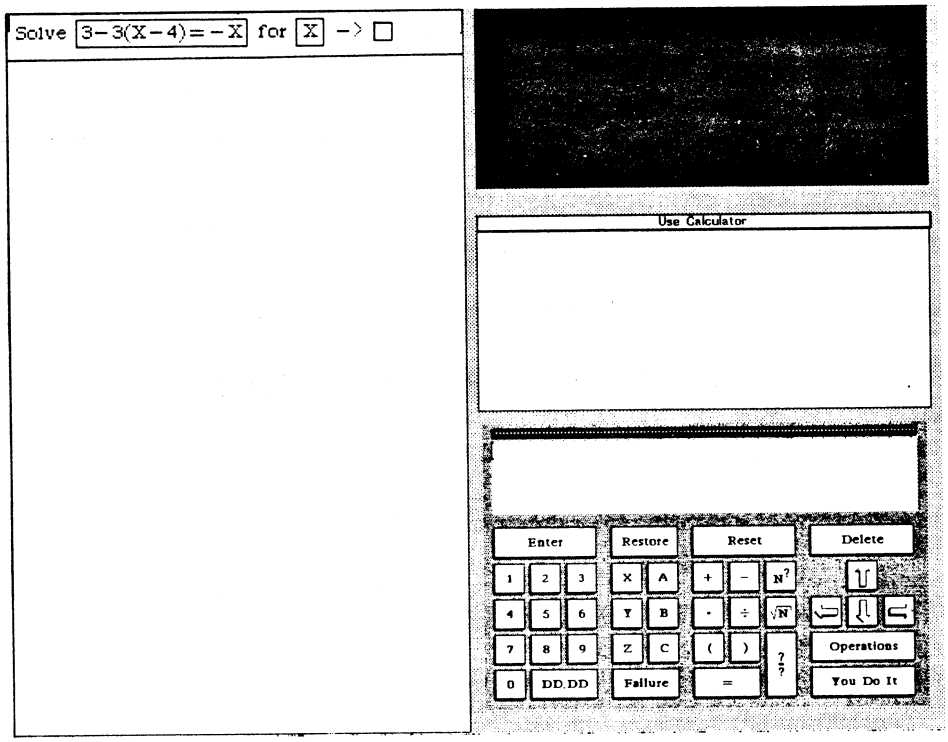


Fig. 9. The algebra tutor's interface as it appears at the beginning of a problem. On the left is the solution window; at the top right is a blackboard for posting messages to the student; at the middle right is a calculator scratchpad where the student can perform primitive operations; and the bottom is a menu of choices for communicating with the tutor.

decomposed into a number of substeps and the first substep is "get-coefficient." Figure 12 shows the screen image with these substeps embedded. Note how the tutor embeds boxes on top of boxes to indicate levels of embedded goals.

Figure 13(a) shows the screen image after the student has completed all of

EQUATIONS	EXPRESSIONS
FRACTIONS	NUMBERS
Add to equation	Cleanup
Collect Constants	Collect Like Terms
Constants Other Side	Distribute
Isolate Solve Var	Multiply Equation
Simplify Equation	Solve
Undo addition	Undo all operations
Undo multiplication	Variables One Side
You Do It	

Fig. 10. The new menu that appears when the student selects the "operations" system in Fig. 9.

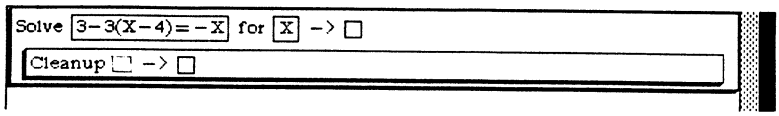


Fig. 11. The solution window after "cleanup" is chosen from Fig. 10.

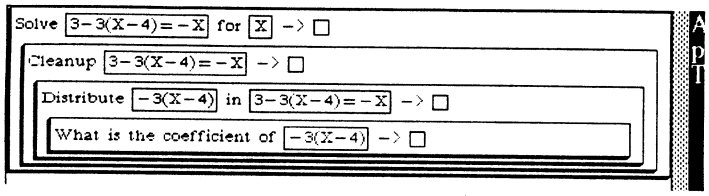


Fig. 12. A later state of the solution after the student has selected "distribute" as a substep to "cleanup" and "get-coefficient" as a substep to "distribute."

the substeps of cleanup. The screen maintains two levels of completed substeps. Thus, the student can see that cleanup was solved by a "distribute" followed by a "collect." Having finished the cleanup substep the student has turned to the next major step in solving the problem which is to move the variable terms to one side. The student has chosen to try to answer this directly rather than pursue it in substeps. Unfortunately, he has made the classic sign error and entered $15 = -4x$. The tutor recognizes this error and enters a remedial message on the blackboard. This error message is illustrated in Fig. 13(b).

Figure 14 shows the final solution window when the problem is solved. The student finished the move-variables goal and started on the isolate goal.

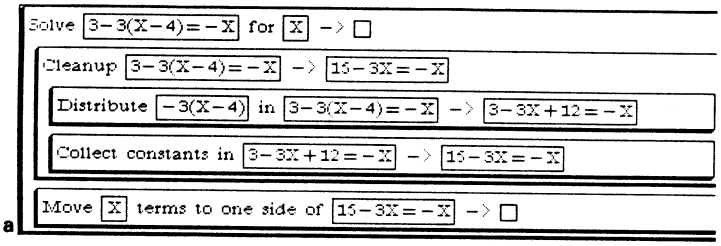


Fig. 13. (a) A later state of the solution window after the student has gone on to try the substep of "move-variable-to-one-side" after completing the "cleanup" step. (b) The error message given to the student who enters $15 = -4x$ as the result for the last step in (a).

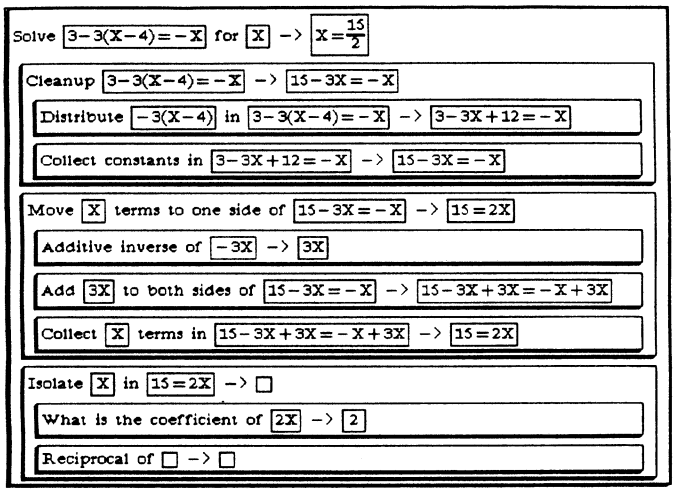


Fig. 14. The final state of the solution window after solving $3 - 3(x - 4) = -x$.

However, in the midst of solving the isolate goal the student saw the answer and chose what is called the popout option. This lets him put in the answer to the isolate goal without filling in the suboperations. Finally the student posted that result as the answer to the top goal of solving the equation.

At all points there is an option on the current menu called "you do it." If the student selects this when an argument or a goal is required, the tutor describes the argument or goal. If the student selects "you do it" when a result is required, the tutor will decompose the task of obtaining the results into a set of substeps unless the result comes from a primitive goal. In this case the tutor just gives the student the result.

As in the case of the other tutors, the algebra tutor moves the student along towards a solution. The one thing unique about the algebra tutor is our policy of decomposing a result calculation into substeps recursively until primitive steps are reached. Our informal observation is that this seems to be pedagogically effective in that it enables the tutor and student to determine the locus of a misconception.

2.4. Summary of the tutors

Underlying each of the tutors is an ideal model of how students should solve the respective problems and a model of how students err. The error model is used to recognize and remediate errors. The ideal model is used to guide students along a correct solution path if necessary. This combined use of the ideal and error model (together called the generic model) is what defines the model-tracing methodology—the tutor traces out the path the student tries to

take through the generic model and insists that the student stay on a correct path.

The major activity of the tutor is monitoring students' problem solving. We attempt to create highly interactive interfaces that quickly let the students know when their solutions deviate from ideal solution behavior and just where they deviate. This kind of instructional environment has a highly procedural flavor and contrasts with the more abstract and declarative instruction in some tutoring efforts (e.g., [22]). This reflects fundamental differences in the nature of the knowledge to be communicated and how that knowledge is communicated. Our current tutors are focused on helping the procedural component of learning although we are currently considering extending them to declarative instruction where we might adopt a methodology more like Collins, Warnock and Passafuime [22].

2.5. Evaluating the model-tracing methodology

A critical issue is whether the model-tracing methodology really works in improving the learning of the subject domains. A general problem with work in intelligent tutoring is that it has tended to progress without any empirical feedback as to whether the proposed mechanisms work. What feedback there is has been largely anecdotal. We have been able to perform some systematic tests of the effectiveness of our tutors which we will briefly report here. After reviewing these evaluations we will discuss issues concerned with why such summative evaluations are not completely satisfying and we will try to identify further research directions.

2.5.1. *LISP*

The LISP tutor was systematically evaluated in a course we taught in the fall of 1984. A class of 20 students was divided into two groups counterbalanced according to statistics such as math SATs and prior computing experience. All students attended the same lectures and did the same problems as homework. One group of students did these problems with the LISP tutor and the other group did them in the standard FRANZLISP environment. There was a proctor available to all students to answer questions. The proctor spent most of his time with students who lacked the tutor. We estimate that perhaps 5% of these students' time was spent with the proctor. Thus, we have a fairly controlled comparison of a group of students working with the tutor and a group of students learning in a fairly representative college environment with perhaps a little more access to human help than is typical.

The LISP curriculum taught by the tutor at that time was a subset of the curriculum currently taught. It involved the following nine lessons: an introduction to basic LISP functions, defining new functions, conditionals and predicates, structured programming, input-output, integer-based iteration, integer-based recursion, list-based recursion, and advanced recursion. Figure 15

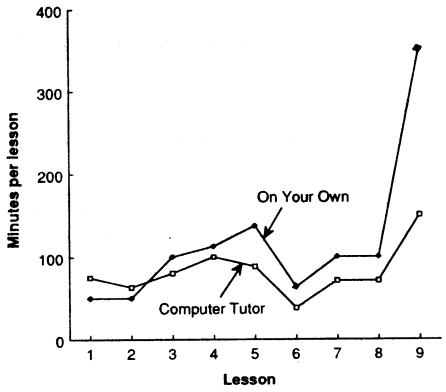


Fig. 15. Comparison of time spent per lesson by subjects learning LISP in the tutor and control groups.

shows the amount of time spent solving the problems in each lesson with and without the tutor. As can be seen, there is very little difference between the conditions over the first four lessons. In fact, students take less time without the tutor on the first two lessons. One reason tutored subjects might spend more time is that the tutor was somewhat slow—a condition that has been improved upon in succeeding years. On the other hand, students are taking more than twice as long without the tutor on the ninth lesson which involves programming some difficult recursive functions. The basic result seems to be that the tutor reduces time on tasks where there is considerable difficulty and search in finding a correct solution. Certainly, subjects in the on-your-own condition spent a lot of time on lesson 9 trying solutions that they had to completely abandon.

Students in the class took two paper-and-pencil exams which provide an assessment of performance outside the learning environment. There was no difference between the two groups on an exam after lesson 6, but a statistically significant advantage for the tutored group appeared following the ninth lesson. Tutored students scored 43% higher on the final exam. All students were required to do a final project without the tutor which involved writing a program to solve the waterjug problem. Tutored students received 10% higher grades, but this did not approach statistical significance.

2.5.2. *Geometry*

The geometry tutor has been used for two years in a local high school. This first year (1985–1986) was devoted primarily to observing its reliability in a classroom setting. The tutor was used throughout the third quarter of the academic year by four classes. Two of these classes consisted of students who were in the regular academic track, one class consisted of students placed in

the "scholars" track on the basis of their high level of performance in past math courses, and one class of students consisted of students designated "gifted" on the basis of IQ scores. The results of this evaluation were very encouraging. Students were enthusiastic about using the tutor and all three groups of students showed statistically significant improvement in test scores between a pre-test and a post-test on geometry proofs. The mean score of the students in the regular academic track rose from 44 points on the pre-test to 54 points on the post-test (out of a maximum score of 80 on each test). The mean for the "scholars" track rose from 57 to 63 and the mean for the "gifted" class rose from 55 to 72 points. About 10 points on this scale corresponded to a letter grade difference, so average performance increased a minimum of about a half-grade difference, to as much as a grade and a half across the three groups, although the differences in degree of improvement across classes is not significant.

Thus, the initial use of the tutor proved successful, but what is lacking in this evaluation is an appropriate measure of how well comparable groups of students would have done without the tutor. So, a controlled comparison of tutor and non-tutor classes was carried out in the second year. In this comparison, the same teacher taught five classes which varied in student ability level and whether the tutor was present or not. We also had the teacher instruct one class of larger size in which two students worked with each tutor. In comparisons of classes of similar size and ability, tutored students scored 64 points (out of 80) on the post-test while non-tutored students scored 48 points. This difference was statistically significant.

We ran a regression analysis of the difference among students on the post-test using as predictor variables IQ, grade in the prior year's algebra class, whether the student used the tutor, and finally, if the tutor was used, whether there was a 2–1 ratio of students to tutors. There were significant effects of algebra grade, tutor and student–tutor ratio. Students scored an average of 7 points higher on the post-test with each higher algebra grade, 14 points higher if they used the tutor, but 10 points lower if they were in a 2–1 ratio of students to tutors. This indicates that the tutor has the same predictive impact as a two-letter grade difference in algebra. It also indicates that the tailoring of instruction for a particular student is important.

2.5.3. *Algebra*

Our evaluation of the algebra tutor is more preliminary since we have been using it in laboratory situations only and have not introduced it in the classroom. So far we only have evidence that it produces learning. That is to say, all students who work with it know more about algebra than before they started. We have no evidence that it is better than comparable time spent without the tutor. We hope to address this question in a classroom test in the 1987–1988 school year.

It is a fair summary of this evaluation research to say that our tutors do help. This is a far from obvious outcome. When we started out it was a radical proposal to assert that we could get students to improve their performance by forcing them to follow the steps of our cognitive models. Clearly, if our cognitive models did not have some substantial truth to them, we would have failed miserably. Thus, this positive evaluation outcome is general support for our theoretical position.

Nonetheless, it is hardly satisfactory evidence. We do not really know what features of our tutors produced these positive outcomes nor do we know how optimal our tutors are. The LISP and geometry tutors produce an improvement of about one standard deviation in classroom performance, whereas human tutors are known to produce an improvement of two standard deviations [16]. It is unclear whether performance of human tutors is achievable by computer tutors. Some of the benefit of human tutors might be due to affective reactions to human interaction. In addition, some of the benefits may depend on an ability to process natural language questions and answers that exceed the level that is practically obtainable in computer tutors. On the other hand, there is no reason to believe that human-tutor performance defines an upper bound since humans almost certainly are not always optimal in their decisions. The basic point is that we need to begin to do systematic studies of design variations on our tutors to determine which features of the tutor are critical to our positive results, which are neutral, and which may be preventing us from achieving even more positive results. Such research would also be more illuminating with respect to the underlying theoretical issues. We are just embarking on such a program of research.

3. Implementing the Model-Tracing Methodology

A major prerequisite to implementing a model-tracing tutor is to create all the production rules that will be involved in the tracing. A significant subtask here is adding an adequate set of buggy rules to the student model in order to be able to account for the errors we see. In our experience the best we have been able to do is to account for about 80% of the errors—the remaining being just too infrequent and too removed from the correct answer to yield to any analysis. One approach to coding the systematic errors has been simply to observe the errors students make with our tutor, try to understand their origin, and code the inferred buggy productions one by one into the system. In more recent work such as in our algebra tutor we are trying to generate these errors on a principled basis in a fashion similar to the notable work on subtraction [17, 19] and on algebra [31]. For instance, a frequent source of errors in algebra is forgetting to perform a necessary substep in the calculation.

Given a production set which can model the range of behaviors we see in our students, our tutor design then can be decomposed into three largely indepen-

dent modules. There is the student module which can trace the student's behavior through its nondeterministic set of production rules. There is the pedagogical module which embodies the rules for interacting with the student, for problem selection, and for updating the student model. The separation between student module and pedagogical module is similar to the separation of instruction from the expert system in a number of tutoring systems, including those of Brown, Burton, and de Kleer [18] and Clancey [21]. Finally, there is the interface which has the responsibility of interacting with the student. As a software engineering issue, these three components can be developed separately with the pedagogical module taking responsibility for controlling the interaction among the three modules—getting interpretations and predictions from the student module and making requests to the interface to present information to the student or to get information from the student. While each module is complex, dividing a major software project into three independent components is a big step in the direction of tractability. Much of the subsequent discussion will be organized around issues involving each of the components.

3.1. The student module

The basic responsibility of the student module is to deliver to the pedagogical module an interpretation of a piece of behavior in terms of the various sequences of production rules that might have produced that piece of behavior. The obvious methodology for doing this is to run our nondeterministic student model forward and see what paths produce matching behavior. While there are complexities and efficiencies that have been added to this basic insight this is the core idea. The rest of the discussion of the student model is concerned with issues raised in trying to implement this core idea.

3.1.1. *Nondeterminacy*

Nondeterminacy in the production sequence is a major source of problems in implementing the model-tracing methodology. We face nondeterminacy whenever multiple productions in the student module produce the same output. (For instance, in the algebra tutor the student says he wants to apply distribution, and there are multiple possible distributions in the equation.) A special case of this is when productions produce no overt output as when a student is doing some mental calculating or planning. What to do in the case of such planning nondeterminacy is an interesting question. The set of potential paths can explode exponentially as the simulation goes through unseen steps of cognition. Also, the potential for actually effectively tutoring these steps is weakened the greater the distance between the mental mistake and the feedback on that decision. Therefore, one is naturally tempted to query the student as to what he is thinking—that is, to force an association of some

output with the mental steps. On the other hand, it is difficult to design an interface which can trace planning in a way that does not put an undue burden on the student. Students often resent even giving vocalized answers to the question "what are you thinking about" and there is reason to believe such simultaneous report generation may interfere with the problem solving [25]. The sample LISP tutor interaction that we traced earlier, in which the student and tutor work through a recursive plan for factorial is one instance of our effort at plan tracing. While we have some evidence that such interactions help, students report that they do not like being slowed down by having to go through such interactions.

Another example of the problem created by nondeterminacy is that misunderstandings and slips can often produce the identical behavior. For instance, students can confuse CONS and LIST in programming either because they really do not understand the difference or as a result of a momentary lapse [11]. The student model must be capable of delivering both interpretations to the tutor, leaving to the tutor the task of assessing the relative probability of the two interpretations and deciding what remedial action should be taken.

3.1.2. *Production system efficiency*

A major complication we face when we try to trace a student's problem solving is that running a production system in real time can create serious problems. Students will not sit still as a system muddles for minutes trying to figure out what the student is doing. They will not pace their problem solving to assist the diagnosis program. Interestingly, our observation has been that human tutors have problems with real-time diagnosis and one of the dimensions on which human tutors become better with experience is real-time diagnosis.

Production systems, for all their advantages, are by and large not the fastest way to solve problems. The inherent computational problems of production systems are exacerbated in tutoring for a number of reasons:

(1) The grain size of modeling is often smaller than would be necessary in expert system applications, and the complexity of the production patterns required to expose the source of student confusions is often considerable.

(2) The system has to consider enough productions at any point to be able to recognize all next steps that a student might produce. This contrasts with many applications where it is sufficient to find a production that will generate a single next step.

(3) Often it is not clear which of a number of solution paths a student is on and the production system has to be used nondeterministically to enable a number of paths to be traced until disambiguating information is encountered.

The production systems we have produced have all involved variations on the RETE algorithm developed by Forgy [26] for pattern matching which has

supported many of the OPS line of production systems. However, we have not had good success with simply using OPS as our expert system because the pattern matching for each domain has special constraints upon which we have had to try to optimize. Anderson, Boyle and Yost [8] discuss this issue for the domain of geometry.

A major issue in designing the pattern matcher for a domain is to decide how much detail of the actual problem should be represented. For instance, if one is developing an algebra tutor it is useful to have different representations for the following two expressions during the early stages of teaching factoring:

$$2AB + 4A\,,$$
$$2BA + 4A\,.$$

There is evidence that the first expression can be more easily factored into $2A(B+2)$ than can the second expression: Commutativity of multiplication is not automatic in many students, and the common factor of $2A$ might not be seen in the second expression above. On the other hand, when we look at students who have mastered algebra and are learning calculus, it is no longer necessary to represent the distinction between these two forms. This means that in calculus we can use certain "canonicalizations" that simplify the pattern matching and reduce the number of productions.

The computational cost associated with implementing such production systems has a space as well as a time dimension. The number of productions can be on the order of thousands to tutor a domain and the RETE algorithm can be space expensive storing partial products of pattern matching.

Of course, it is an open question just how efficient in time and space we can make our production system implementations. In their current form they are just within the threshold of acceptability, which is to say students are barely satisfied with the performance of a machine like a Dandetiger with over three megabytes of memory. However, there are reasons for us not to be satisfied with this performance. In the first place such machines are still a good deal beyond the range of economic feasibility. Secondly, efficiency issues impact on the range of topics we handle. This manifests itself in a number of ways:

(1) Problems tend to become more costly as they become larger even if the larger problems involve the same underlying knowledge. This is because production system working memory tends to increase, as does the nondeterminism. Therefore, there is an artificial size limit on the problems we tutor students on.

(2) Progress into more advanced topics is as much limited by dealing with the added computational burden posed by these topics as with adequately understanding and modeling the domain.

(3) The actual tutoring interactions become limited by the need to reduce nondeterminacy. For instance, some of our tutors force a particular interpreta-

tion of the student's behavior on the student, rather than waiting until the student generates enough of the solution to eliminate the ambiguity.

3.2. Compiling the model tracing

If one looks at all possible sequences of productions that can be generated in any of our models, one finds that it defines a problem space of finite cardinality. That cardinality can be quite large, but often simply because we are looking at different permutations of independent or nearly independent steps in a problem solution. This suggests that if we are clever in our representation of the problem space we need not dynamically simulate the student in order to interpret him. Rather, we can generate the problem space beforehand and just use the student's behavior during problem solving to trace through this pre-completed problem space. Given the cost of real-time simulation with a production system, this seems that it might be a worthwhile step. In one case, we obtained a 50% performance improvement in our LISP tutor by a partial implementation of this step. In a major project just completed, we used this technique to transfer the geometry tutor from the Dandetiger to the Macintosh and got a significant improvement in performance.

There are other advantages to having the complete problem space compiled in advance of the actual tutoring session. This makes it easy for the tutor to look ahead and see where a step in the problem solution will lead. Often in a proof tutor, a production rule will be favored by the ideal model but in fact not lead to a solution. For instance, there are geometry problems where even experts make certain inferences which do not end up as part of the final proof. It is the sort of heuristic inference which is successful nine times out of ten but is not useful one time in ten. If the tutor recommended dead-end steps just because the ideal model makes them, the student would quickly loose faith in the tutor. Human tutors also tend to look ahead to make sure that their recommendations lead somewhere.

3.3. The pedagogical module

One interesting observation about our overall tutoring framework is that it is possible to decouple the pedagogical strategy from the domain knowledge.[2] Domain knowledge resides in both the student model and the interface. It is the pedagogical module that relates the two and which controls the interaction. This module does not really require any domain expertise built into it. It is concerned with (1) what productions can apply in the student model, not the internal semantics of the productions; (2) what responses the student generates and whether these responses match what the productions would generate, not

[2] This point was demonstrated much earlier in GUIDON (Clancey [21]). Our tutoring rules were much influenced by the Clancey GUIDON rules.

what these responses mean; and (3) what tutorial dialogue templates are attached to the productions, not what these dialogues mean.

We are in fact working on a new PUPS-based tutor [12] which is a limited realization of this idea. It is concerned with tutoring three programming languages—LISP, PASCAL, and PROLOG. We hope to build student models for different programming domains independent of tutoring strategy and to build different tutors to implement variations on tutoring strategy independent of domain. Specific tutors can be generated by crossing the tutorial module with the domain module without tuning one to another.

There are theoretical reasons for believing that we can create domain-free tutoring strategies and that the optimal tutoring strategy will be domain-free. Our theory of human skill acquisition leads us to believe that the basic learning principles are domain-free. The optimal tutoring strategy would simply optimize the functioning of these learning principles.

However, in our current running systems we have built a separate tutor for each domain. While it is not the case that the tutoring strategies they implement are identical, they are quite similar and we have claimed publicly that they are attempts to embody a strategy based on the ACT learning theory [7]. It is useful to identify what the features of the common tutoring strategy are and what the variations on the strategy could be. It will become clear that, when we look at any dimension of tutoring, there are conflicting considerations as to what the optimal choice should be.

3.3.1. *Immediacy of feedback*

The policy on immediacy of feedback is well illustrated by the LISP tutor. The LISP tutor insists that the student stay on a correct path and immediately flags errors. This minimizes problems of indeterminacy. There are a number of reasons for desiring immediacy of feedback besides this technical one. First, there is psychological evidence that feedback on an error is effective to the degree that it is given in close proximity to the error [14, 29]. The basic reason for this is that it is easier for the student to analyze the mental state that led to the error and make appropriate correction. Second, immediate feedback makes learning more efficient because it avoids long episodes in which the student stumbles through incorrect solutions. Third, it tends to avoid the extreme frustration that builds up as the student struggles unsuccessfully in an error state.

However, we have discovered a number of problems with the use of immediate feedback:

(a) The feedback has to be carefully designed to force the student to think. If at all possible, the feedback should be such that the student is forced to calculate the correct answer rather than just being given the answer [14]. It is important to learning that the student go through the thought processes that generate the answer rather than copy the answer from the feedback.

(b) Sometimes students would have noticed the error and corrected it if we just gave them a little more time. Self-correction is preferable when it would happen spontaneously. As we know, people tend to remember better what they generate themselves [4, Chapter 5].

(c) Students can find immediate correction annoying. This is particularly true of more experienced students. Thus, novice programmers generally liked the immediate feedback feature of our LISP tutor whereas experienced programmers did not. While our goal is not to produce positive affective response, it probably does have some impact on learning outcome.

(d) Often it is difficult to explain why a student's choice is wrong at the point at which the error is first manifested because there is not enough context. To consider a simple example, compare a student who is going to generate (append (list *x*) *y*) where (cons *x y*) is better. It is much easier to explain the choice after the complete code has been generated rather than after "(append" has been typed.

There is no reason why the model-tracing paradigm commits us to immediate feedback, although as noted there are psychological reasons for choosing it. One of the variations we would like to explore in the LISP tutor is a system that gives feedback after "complete" expressions like (append (list *x*) *y*). This will give the student some opportunity for self-correction and also provide a larger context for instruction. On the other hand the distance between error and feedback will still be limited. For more discussion of the different feedback options in the LISP tutor see [23].

3.3.2. *Sensitivity to student history*

By and large the only student model we use is our generic model which is a composite of all correct and incorrect moves that a student can make. At each point in time we are prepared to process all the production rules that we have seen any student use, correct or buggy. If students make an error we give the same feedback independent of their history. The only place we show sensitivity to student history is in presenting remedial problems to students who are having difficulties. It is relatively easy to implement a generic student model, and the question is whether there is any reason to go through the complexity of tailoring the model to the student.

There is one aspect of this generic student model which derives from our theory of skill acquisition and another aspect which does not. The aspect that is theoretically justified is the belief that there are not different types of students who will find different aspects of a problem differentially hard. That is, our theory does not expect individual differences or traits in learning, beyond some overall difference in ability or motivation. The theory implies that all people learn in basically the same way. Of course, it is an open question whether there is empirical evidence for the theory on this score. In our own research it does appear that students differ only in a single dimension of how well they learn

[6]. Despite valiant searches we have yet to find evidence that one set of productions cluster together as difficult for one group of students while a different cluster of productions are difficult for another group of students.

The aspect of a generic model that does not derive from the theory is the assumption that past history of use with a rule implies nothing about the interpretation of a current error. We have evidence that different subjects continue to have trouble with specific different rules. (This is to be contrasted with a trait view that says there is a nonsingleton set of productions that a number of subjects will have difficulty with.) If the student has had a past history of success on a rule it is more likely that error reflects a slip, rather than some fundamental misunderstanding. Currently, our tutors treat all errors as if they reflected fundamental misconceptions and offers detailed explanation, but the better response sometimes would be simply to point the error out.

3.3.3. *Problem sequence*

The existing tutors implement a mastery model for controlling the selection of problems to present to the student. They maintain an assessment of the student's performance on various rules and have knowledge of what problems exercise what rules. The tutors will not let the student move on to problems involving new rules until the student is above a threshold of competence on the current rules. If the student has not demonstrated mastery, the tutor will select additional problems from the current set which exercise rules on which the student is weak.

While such a mastery policy for problem sequence may seem reasonable and there is evidence in the educational literature for its effectiveness [16], it is interesting to inquire as to its underlying psychological rationalization. Why not go onto new problems while the student is weak on current knowledge and teach both the new knowledge and the old weak knowledge in the context of the new problems? Fundamentally, the mastery policy rests on a belief in an optimal learning load—that if we overload a student with too many things to learn, he will learn none of them well. On the other hand, students are advanced to new material at some point when further training on the old material could have improved their performance even more. So there is a countervailing assumption about diminishing returns—that at some point the gain in improving performance on old rules is not equal to gain in learning new rules.

Our choice about exactly where to set the mastery level has been entirely ad hoc. In the ACT and PUPS theories working-memory load affects learning and problems pose less load as they become better learned. However, these processes are not specified in a way that enables us to define an optimal next problem. The issue of problem sequence and mastery levels remains to be worked out in a model-tracing paradigm.

3.3.4. *Declarative instruction*

A student's first introduction to the knowledge required to solve a class of problems is typically not from the tutor; rather it is declarative instruction typically provided in a texbook or lecture. How should this declarative instruction be formulated to make it maximally helpful in learning the skill? Given our analysis of learning by analogy, instruction should take the form of examples appropriate for mapping into problem solutions. Given our PUPS structures, it is not enough that the student simply have the form slots of these structures properly represented; it is critical for successful learning that the student have properly represented the function of these structures and any prerequisites to these structures achieving their functions. For example, Pirolli and Anderson [33] showed that, while all students learn recursive programming by analogy to existing programs, what determines how well they learn is how well they represent how these programs achieve their function. Basically, students often understand an example only superficially and thus emerge from analogy with mischaracterizations of the range of problems for which the structures in the example are appropriate.

In our efforts to create textual instruction to go along with our tutors, we have focused on the issue of giving good examples for purposes of mapping and trying to assure that the student achieves the right encoding of the example. Indeed, we have produced a LISP textbook [9] which consists mainly of carefully crafted examples with explanation aimed at promoting the right encoding. However, what is missing is interactive instruction to assure that the students have encoded the example correctly.

3.4. The interface

One might have thought that the discussion to this point would complete the description of our tutoring systems. We have stated how a tutor models a student and how it uses that model to achieve pedagogical goals. However, the discussion is abstract and leaves completely unspecified what the student actually experiences, which is the computer interface. We have learned that design of the interface can make or break the effectiveness of the tutor. Below are just a few examples:

(1) Early in the history of the LISP tutor we had a system in which the student entered code in a buffer and then dispatched the contents of that buffer to appear in a code window. Students get confused with the system because they were frequently working on one goal while the tutor was processing a different goal. We changed this to a system where one typed the new code right into the old code and these confusions disappeared.

(2) An early version of the algebra tutor had a system in which students entered a next equation, the tutor figured out what steps they engaged in, and

tried to give appropriate feedback and point them back to the right track. The problem was that the students' error might well have occurred at some intermediate step that the students were no longer fixated upon (e.g., adding two fractions in the course of moving a number across the equation sign). It was very difficult to communicate to the student what the problem was. We introduced the system described earlier in this paper in which the student actually stepped through the microsteps of the transformation in a relatively painless way with the system. The tutor could flag the errors as they occurred and these miscommunications disappeared.

(3) We used to have our students type in geometry statements through a typical keyboard. Given the rather special syntax of geometry statements, students would often enter basically correct statements in syntactically incorrect form. After entering a syntactically incorrect statement, the system would tell them it could not understand what they meant. This response by the system often caused them to doubt their understanding of the problem. To remedy this we introduced a real-time parser which flagged them as soon as they entered a character which would make their expression syntactically illegal. Again our difficulties disappeared.

(4) The graphical structure we use to represent geometry statements (Figs. 4–8) seems to be the key to enabling students to understand the structure of a proof even though it is essentially isomorphic in logical structure to a linear proof. The graphical structures make explicit the logical relationships students would have to infer.

(5) In all of our tutors it seems critical to spend considerable time fashioning the English to make it as brief and as understandable as possible. If students face great masses of hard-to-understand prose, they will simply not process the message.

(6) Performance on the LISP tutor seemed to improve when we introduced a facility to bring up the problem statement at any point in time, and when there is room on the screen, the problem statement is now automatically displayed. Performance in the geometry tutor seemed to improve when we introduced a facility for bringing up statements of geometry postulates at will.

(7) One of the major disadvantages of all of our tutors compared to human tutors is that, at least so far, they use only the visual medium. This means that students must move their eyes from the problem to process the textual instruction. In contrast, with a human tutor, the student can listen to the tutor while continuing to look at the problem and even have parts of the problem emphasized through the tutor's pointing.

These observations illustrate two general points about interface design for tutors:

(a) It is important to have a system that makes it clear to a student where he

or she is in the problem solution and where their errors are (observations (1)–(3)).

(b) It is important to minimize working memory and processing load involved in the problem solving (observations (4)–(7)).

While one wants an interface with these properties, it is important that the interface itself be easy to learn and use. One does not want the task of dealing with the interface to come to dominate learning the subject material. An easy interface is one that minimizes the number of things to be learned and minimizes the number of actions (e.g., keystrokes, mouseclicks) that the student has to perform to communicate to the tutor. Its learnability is enhanced if it is as congruent with past experience as possible. It should also have a structure that is as congruent as possible with the problem structure. Finally, the actions should be as internally consistent as possible.

So there are clearly an important set of criteria that our tutoring efforts place on interface design. The problem is that criteria like "minimize working-memory load" or "make learning the interface easy" are not generative. To date we have dealt with interface design on an intuitive basis and on a trial and error basis. We are always left to wonder whether there is some new insight about interface design that would dramatically enhance the achievement gains displayed by a particular tutor.

4. Conclusions

What we have described is a theoretical framework for our tutoring work and some experiences based on that framework. Both the tutors and the theory are evolving objects and so it is not the case that the current embodiments of our tutors reflect all of the current insights of our theory. Still there is an approximation here and it is worthwhile to ask to what degree our tutoring experience confirmed the theory.

The first observation is that students do seem to learn from the tutors. We think this is quite a remarkable fact and not something that we had really believed would work so well when we set out to build these tutors. We have taken cognitive models of the information processing, embedded them in instructional systems, and nothing has fallen apart. They can embody substantial amounts of material, can be developed in feasible time, run within acceptable bounds of efficiency, and are robust in their behavior. The evaluations of the tutor clearly indicate that they are better than standard classroom instruction. This feasibility demonstration gives some credence to the general theoretical framework in which the tutors were built.

It is worth noting here that conventional computer-based instruction usually produces less than half of a standard deviation of improvement [15, 28]. Such instruction involves handcrafted interactions with the student in contrast to our tutors in which the interactions are generated from general principles.

It is a separate question of whether the students behave and learn with the tutors as the theory would predict. This is a difficult question to assess because the theory is probabilistic and does not specify in advance values such as the probability of encoding a production; rather these probabilities must be estimated from the data. It is also difficult because the theory only makes predictions given students' encodings of the instruction and of the problem, and students clearly vary in how they encode this information. Nonetheless, what analyses we have done do seem to confirm the theory. Figure 16 presents an analysis of some data from the LISP tutor that monitors time to type in units of code that correspond to the firing of individual productions. So for instance typing "(cons" corresponds to the firing of a production that recognizes the applicability of the CONS function. We have plotted average times associated with the firing of productions learned in lessons 2, 3, and 5 as a function of the number of times students used the production in the lesson. What these times correspond to psychologically is somewhat complex because they include a lot of low-level interactions with the tutor. However, they should reflect the

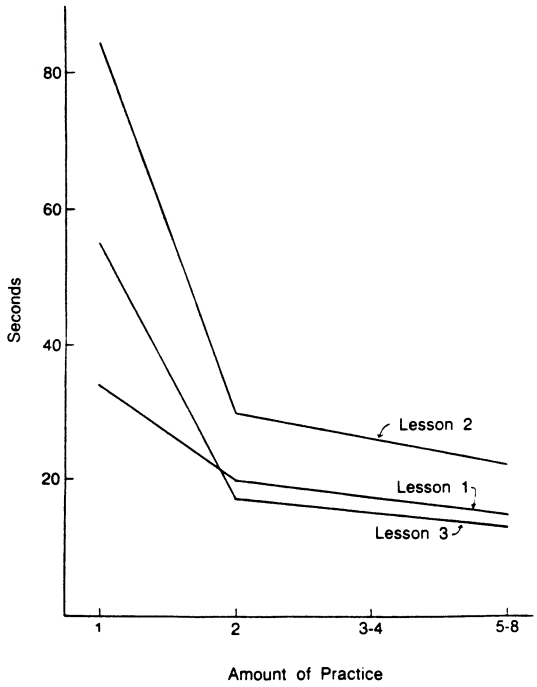


Fig. 16. Plot of learning data from the LISP tutor. Time to code the LISP symbols that are introduced in the first three lessons is plotted as a function of practice. The coding of one of these symbols corresponds to the firing of a production in the student model.

learning of the production or productions associated with the execution of the code. These learning curves have just the shape we would predict. There is a big drop in coding time from first application of a production to second as the knowledge is compiled into production form and a gradual speedup after that as the production rules become strengthened.

Another observation is that the knowledge acquired does seem to have the expected range of application. For instance, in the geometry tutor students are able to apply new combinations of the inference rules that they have learned to solve a new problem as long as the contextual heuristics that recommend the application of these rules are ones they have already encountered. On the other hand if they are presented with a problem that can be solved by the knowledge which they have been taught, but which requires applying a new contextual heuristic to see the applicability of that rule, students experience great difficulty.

Appendix A

Define the function fact. It takes a number argument, n and computes the factorial of n, which is the result of multiplying $n \times n-1 \times n-2 \times \cdots \times 1$. For instance, (fact 3) $= 3 \times 2 \times 1 = 6$. By definition, the factorial of 0 is 1.

(1) **(defun fact (n)**
(cond ((equal) ⟨*ACTION*⟩)
⟨*RECURSIVE-CASE*⟩))

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

What code can you write to test if the condition for terminating case is true? Try writing a predicate.

(2) (*defun fact* (*n*)
(*cond* ((**zero** ⟨*ACTION*⟩)
⟨*RECURSIVE-CASE*⟩))

Did you mean zerop (y or n)? **y**

(3) (*defun fact* (*n*)
(*cond* ((**zerop n**) **1**)
⟨*RECURSIVE-CASE*⟩))

Is the value of fact in the recursive cases calculated by:

PRESS:	IF YOU WANT TO:
1	calling fact recursively with n
2	multiplying n by fact of n minus 1

3 getting 1 2 . . . n and then multiply the results
4 have the tutor choose

Menu choice: **3**

That is a reasonable way to think of doing factorials but it is not a plan for a recursive function. Since you seem to be having trouble with the recursive cases let us work through some examples and figure out the conditions and actions for each of these cases.

The recursive relationship is the relationship between fact called with n and fact called with:

PRESS: IF YOU WANT TO:
1 n minus 1
2 n equal to 0
3 n plus 1
4 have the tutor choose

Menu choice: **1**

We will consider some concrete examples. We will characterize the recursive relationship between fact called with 1 and fact called with 0.

What is the value of (fact 1)?
Answer: **1**

What is the value of (fact 0)?
Answer: **1**

EXAMPLES

	fact (n)	fact (n − 1)
A.	(fact 1) = 1	(fact 0) = 1

We will consider some more concrete examples. We will characterize the recursive relationship between fact called with 3 and fact called with 2.

What is the value of (fact 3)?
Answer: **6**

What is the value of (fact 2)?
Answer: **2**

EXAMPLES

	fact (n)	fact (n − 1)
A.	(fact 1) = 1	(fact 0) = 1
B.	(fact 3) = 6	(fact 2) = 2

Since there is only one recursive case characterized by the examples you have just worked out, the condition in this case can be just coded as T. We will just focus on characterizing the action in this case.

In examples A and B what do you have to do to get the result of fact called with n?

PRESS:	IF YOU WANT TO:
1	multiply n by one less than n
2	multiply n by fact of one less than n
3	add n to the result of fact called with one less than n
4	have the tutor choose

Menu choice: **2**

We have now worked through a series of examples that illustrate the recursive relation.
We will now return to the goal of trying to produce LISP code that specifies the recursive relation.

(4) (*defun fact* (*n*)
(*cond* ((*zerop n*) *1*)
(t (fact))))

You will have to use fact at some point but right now you want to multiply.

(5) (*defun fact* (*n*)
(*cond* ((*zerop n*) *1*)
(*t* **(times n (fact n)))))**

You cannot just code n at this point. What you want to do is subtract one from n.

(6) (*defun fact* (*n*)
(*cond* ((*zerop n*) *1*)
(*t* (*times n* (*fact* **(difference))))))**

Difference would work in this case but it is more appropriate to use the function sub1 when you are subtracting 1 from a number.

(7) (*defun fact* (*n*)
(*cond* ((*zerop n*) *1*)
(*t* (*times n* (*fact* **(sub1 n))))))**

— YOU ARE DONE. TYPE NEXT TO GO ON AFTER —
— TESTING THE FUNCTIONS YOU HAVE DEFINED —

THE LISP WINDOW

⇒(*trace fact*)
(fact)
⇒(*fact 3*)
1 ⟨Enter⟩ fact (3)

```
|2 ⟨Enter⟩ fact (2)
|  3 ⟨Enter⟩ fact (1)
|  |4 ⟨Enter⟩ fact (0)
|  |4 ⟨EXIT⟩ fact 1
|  3 ⟨EXIT⟩ fact 1
|2 ⟨EXIT⟩ fact 2
1 ⟨EXIT⟩ fact 6
6
```

REFERENCES

1. Anderson, J.R., Tuning of search of the problem space for geometry proofs, in: *Proceedings IJCAI-81*, Vancouver, BC (1981).
2. Anderson, J.R., Acquisition of cognitive skill, *Psychol. Rev.* **89** (1982) 369–406.
3. Anderson, J.R., Acquisition of proof skills in geometry, in: J.G. Carbonell, R. Michalski and T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983).
4. Anderson, J.R., *The Architecture of Cognition* (Harvard University Press, Cambridge, MA, 1983).
5. Anderson, J.R., Production systems, learning, and tutoring, in: D. Klahr, P. Langley and R. Neches (Eds.), *Production System Models of Learning and Development* (MIT Press, Cambridge, MA, 1987) 437–458.
6. Anderson, J.R., Analysis of student performance with the LISP tutor, in: N. Fredericksen, R. Glaser, A. Lesgold and M. Shafto (Eds.), *Diagnostic Monitoring of Skill and Knowledge Acquisition* (Erlbaum, Hillsdale, NJ, 1989).
7. Anderson, J.R., Boyle, C.F., Farrell, R. and Reiser, B.J., Cognitive principles in the design of computer tutors, in: P. Morris (Ed.), *Modelling Cognition* (Wiley, New York, 1989).
8. Anderson, J.R., Boyle, C.F. and Yost, G., The geometry tutor, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 1–7.
9. Anderson, J.R., Corbett, A.T. and Reiser, B.J., *Essential LISP* (Addison-Wesley, Reading, MA, 1987).
10. Anderson, J.R., Farrell, R. and Sauers, R., Learning to program in LISP, *Cognitive Sci.* **8** (1984) 87–129.
11. Anderson, J.R. and Jeffries, R., Novice LISP errors: Undetected losses of information from working memory, *Human-Computer Interaction* **22** (1985) 403–423.
12. Anderson, J.R. and Skwarecki, E., The automated tutoring of introductory computer programming, *Commun. ACM* **29** (1986) 842–849.
13. Anderson, J.R. and Thompson, R., Use of analogy in a production system architecture, in: A. Ortony et al. (Eds.), *Similarity and Analogy* (to appear).
14. Anderson, R.C., Kulhavy, R.W. and Andre, T., Conditions under which feedback facilitates learning from programmed lessons, *J. Educ. Psychol.* **63** (1972) 186–188.
15. Bangert-Drowns, R.L., Kulik, J.A. and Kulik, C.C., Effectiveness of computer-based education in secondary schools, *J. Comput.-Based Educ.* **12** (1985) 59–68.
16. Bloom, B.S., The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring, *Educ. Researcher* **13** (1984) 3–16.
17. Brown, J.S. and Burton, R.R., Diagnostic models for procedural bugs in basic mathematical skills, *Cognitive Sci.* **2** (1978) 155–192.
18. Brown, J.S., Burton, R.R. and de Kleer, J., Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III, in: D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982) 227–282.
19. Brown, J.S. and VanLehn, K., Repair theory: A generative theory of bugs in procedural skills, *Cognitive Sci.* **4** (1980) 379–426.

20. Carroll, J., Designing minimalist training materials, Research Rept. 46643, IBM Watson Research Center, Yorktown Heights, NY (1985).
21. Clancey, W.J., Tutoring rules for guiding a case method dialogue, in: D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982) 201–225.
22. Collins, A.M., Warnock, E.H. and Passafiume, J.J., Analysis and synthesis of tutorial dialogues, in: G.H. Bowen (Ed.), *Advances in Learning Motivation* (Academic Press, New York, 1975).
23. Corbett, A.T. and Anderson, J.R., Problem compilation and tutoring flexibility in the LISP Tutor, International Conference on Intelligent tutoring Systems (submitted).
24. Dulany, D.E., Carlson, R.A. and Dewey, G.I., A case of syntactical learning and judgment: How conscious and how abstract? *J. Experimental Psychol. General* **113** (1984) 541–555.
25. Ericsson, K.A. and Simon, H.A., *Protocol Analysis: Verbal Reports as Data* (MIT Press, Cambridge, MA, 1984).
26. Forgy, C.L., Rete: A fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence* **19** (1982) 17–37.
27. Johnson, L. and Soloway, E., Intention-based diagnosis of programming errors, in: *Proceedings AAAI-84*, Austin, TX (1984).
28. Kulik, C.C., Kulik, J.A. and Shwalb, B.J., The effectiveness of computer-based adult education: A meta-analysis, *J. Educ. Comput. Res.* **2** (1986) 235–252.
29. Lewis, M.W. and Anderson, J.R., Discrimination of operator schemata in problem solving: Learning from examples, *Cognitive Psychol.* **17** (1985) 26–65.
30. Lewis, M.W., Milson, R. and Anderson, J.R., Designing an intelligent authoring system for high school mathematics ICAI: The TEACHERS APPRENTICE Project, in: G. Kearsley (Ed.), *Artificial Intelligence and Instruction: Applications and Methods* (Addison-Wesley, Reading, MA, 1990).
31. Matz, M., Towards a process model for high school algebra, in: D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982).
32. Norman, D.A., Categorization of action slips, *Psychol. Rev.* **88** (1981) 1–15.
33. Pirolli, P.L. and Anderson, J.R., The role of learning from examples in the acquisition of recursive programming skill, *Can. J. Psychol.* **39** (1985) 240–272.
34. Reder, L.M., Charney, D.H. and Morgan, K.I., The role of elaborations in learning a skill from an instructional text, *Memory and Cognition* **14** (1986) 64–78.
35. Reiser, B.J., Anderson, J.R. and Farrell, R.G., Dynamic student modelling in an intelligent tutor for LISP programming, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 8–14.
36. Sleeman, D., Assessing aspects of competence in basic algebra, in: D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982) 185–199.
37. Sleeman, D. and Brown, J.S. (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982).
38. Winograd, T., Frame representation and the declarative procedural controversy, in: D. Bobrow and A. Collins (Eds.), *Representation and Understanding* (Academic Press, New York, 1975).
39. Winston, P.H. and Horn, B.K.P., *LISP* (Addison-Wesley, Reading, MA, 3rd ed., 1984).