THE AUTOMATED TUTORING OF INTRODUCTORY COMPUTER PROGRAMMING

The methodologies of artificial intelligence and the knowledge of cognitive psychology can be used to automatically generate fine-grain tutorial interactions, rather than having to program them manually.

JOHN R. ANDERSON and EDWARD SKWARECKI

Intelligent computer-assisted instruction (ICAI) or intelligent tutoring [14] has long been viewed as impractical and as something that only exists in the research laboratory. It was common to require a million-dollar machine to interact with one student, and often the response time was slow. The rate of development of intelligent educational software was typically much slower than the industry standard for conventional computer-assisted instruction (CAI). However, owing to dropping machine costs and increasing knowledge in the area of cognitive science that unites artificial intelligence and cognitive psychology, these obstacles are now being overcome. At Carnegie-Mellon University (CMU), we have built systems that are teaching introductory programming economically.

With the growing number of high-school and college students currently taking introductory computer programming courses, sizable amounts of faculty time are spent on course development and instruction. Not only does this put a strain on the already limited supply of computer science instructors, but as a result of the large lecture classes necessitated by limited teaching resources, students are not likely to get individualized attention and may learn the material poorly. Even in ideal circumstances, most of the student's learning occurs through hands-on programming experience outside

of class and not during lectures. There is considerable evidence [6] that human tutors greatly improve learning of such problem-solving skills. Significant potential for improving the quality of programming education and for reducing instructor work load lies in computer-based tutoring systems that can coach students in solving introductory programming problems.

The conventional CAI approach to this formidable task is to first determine how a good teacher would respond to each possible student action and then build a branching program with each response explicitly programmed. Conventional CAI systems, however, tend to be rather restrictive and to interact with the student only at a course-grain size in highly complex problem-solving domains such as computer programming. It is not possible by conventional means to build up to the fine-grained interactions desired in reasonable development time. (An industry standard has been set of spending between 100 and 1000 hours to program 1 hour's worth of computer-based instruction.) To economically construct more effective tutors, more powerful tools are necessary.

Our main premise is that, to construct systems capable of automatically generating effective tutorial dialogues, we need to incorporate precise and realistic models both of how students should program and how they actually program. The Lisp tutor [3, 11] reflects the most extensively developed demonstration of this approach. Prior to and during development of the Lisp tutor, we studied how students learn to program in Lisp [1, 2, 5]. Based on this

empirical research, we constructed a number of pre-
scriptive and descriptive simulation models of stu-
dent programming in introductory Lisp courses.
With these models, we were able to simulate the
desired behavior (when prescriptive) and the stu-
dents' actual behavior (when descriptive) to the
point of generating the symbols that constitute
the Lisp code in the order the students write these
symbols. Developed as production systems in a
production-system dialect called GRAPES [12], these
productions involve planning operations such as
those represented by the following production:[1]

```
IF    the goal is to code the body of
      a function involving an integer
      argument
THEN  try integer recursion and set
      subgoals to plan a terminating
      case and a recursive case
```

as well as coding productions such as

```
IF    the goal is to code a test to
      see if a value is equal to zero
THEN  write down a call to the func-
      tion ZEROP and set a subgoal to
      code the value to be given as an
      argument to the function
```

The first production initiates a plan for coding a
recursive function, and the second codes a test to
determine whether a value (a variable or the result
of a computation) is equal to zero. These productions
set subgoals, which can in turn set other subgoals,
and so on. Executing in sequence, productions such
as these can write an entire Lisp program.

We use these cognitive theories to simulate the
different ways students write programs and to model
what students are thinking while writing them. At
any point in the problem solving, production sys-
tems can generate a wide range of simulated next
steps. The on-line tutor tries to match this full set of
simulated next steps with the observed next coding
actions of the student. The sequence of production
firings that generates the student's actions is as-
sumed to represent what the student is thinking.
Using this assumption, the tutor can choose instruc-
tion appropriately. This process of following the stu-
dent's behavior is called *model tracing*. Its goal is to
use a rich interpretation of the student's cognitive
processes to direct the tutoring interaction.

To illustrate how this model-tracing methodology
can be applied to the tutoring of introductory pro-
gramming, we will describe the implementation of

the Lisp tutor and discuss plans for making that im-
plementation even more effective.

THE LISP TUTOR

The sidebar (on the following pages) contains a
student–tutor dialogue for coding a recursive func-
tion to calculate factorial.[2] These listings present
"snapshots" of the interaction; we listed the stu-
dent's input and the tutor's response (numbered for
convenience). The total code is shown as it appears
on the screen. The student added only what is dif-
ferent from the previous code (shown in boldface
type).[3]

The student started out by typing "(defun" on
the first line. The template for defining a function
then appeared:

(defun <NAME> <PARAMETERS> <BODY>)

The terms in angle brackets denote pieces of code
the student must supply to define the function. The
student should fill in the <NAME> slot with the
name of the function, the <PARAMETERS> slot with
the parameters of the function, and the <BODY> slot
with the actual code. At this point in the simulation
model, there is just one production that could fire.

```
IF    the goal is to code a Lisp
      function
THEN  write DEFUN and set subgoals to
      code the name of the function,
      its parameters, and its body
```

The tutor assumes that this production represents
the student's intentions and the student's subgoals
are the tutor's subgoals. Thus, the < > slot holders
in the code represent the subgoals of the simulation
model.

On Line 2, the student filled in the function name
and the parameters, and has begun specifying the
body of the function. Note that all parentheses are
automatically balanced by the tutor, and syntax is
checked.

The code for the body of the function factorial
consists of a conditional structure with two clauses.
The first clause tests for the terminating case (when
the parameter is zero) and will return 1. The second
clause is an *else* clause and will call the function
recursively. Although the student had some diffi-
culty with the syntax of the conditional tests in
Lines 2 and 3, the terminating case for the factorial
function was coded correctly. Typically, students

---

[1] These productions are given in the text with what we have called an
"Englishified" representation. The actual productions coded in GRAPES are
more precise and harder to read.

[2] 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, and the student's
input is shown in bold characters.

[3] For instance, in Line 3 the student has added "**zero**" as an extension of
"(defun fact (n) (cond ((."

A Student–Tutor Dialogue for Coding a Recursive Function to Calculate Factorial

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

```
1.  (defun <NAME> <PARAMETERS> <BODY>)
```

```
2.  (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.

```
3.  (defun fact (n)
        (cond ((zero <ACTION>)
              <RECURSIVE-CASE>))
```

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

```
4.  (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
        multiplying 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**

have little difficulty with terminating cases, but
have great difficulty with recursive cases. The dia-
logue after Line 4 illustrates how the tutor guides
the student through a design of the recursive func-
tion. The tutor helps the student construct examples
of the relationship between fact (n) and fact
(n − 1) and then identify the general relationship.
Figure 1, page 846, shows the screen image at a criti-
cal point in the design of this function.
   The dialogue after this point shows two errors stu-
dents commonly make in defining recursive func-
tions. The first error, in Line 5, is to call the function
directly without combining the recursive call with
other elements. The second error, in Line 6, is to call
the function recursively with the same argument
rather than a simpler one.
   After coding the function, the student goes to the
Lisp window to experiment. The student is required

to trace the function, as the recursive calls embed
and then unravel. Figure 2, page 846, shows the
screen image at this point with the code on top and
the trace below it.

Features of the Model-Tracing Methodology
The example in Figure 2 illustrates a number of
features of the model-tracing methodology:

• The tutor constantly monitors the student's prob-
  lem solving and provides direction whenever the
  student wanders off the path.
• The tutor provides help with the overt parts of the
  problem solution and the planning. However, to
  address the planning, a mechanism was intro-
  duced in the interface (in this case, menus) to
  allow the student to communicate the planning
  steps.

```
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.

```
5.   (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.

```
6.   (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.

```
7.   (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 one from a number.

```
8.   (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
```

- The interface handles details such as syntax checking, which are irrelevant to the problem-solving skill at hand.
- The interface is highly reactive in that it responds to every symbol the student enters.

Mechanics of Model Tracing

Within the tutor is a production system consisting of hundreds of ideal and buggy rules. We have already seen examples of ideal rules. Below is an example of the buggy production rule that generates the feed-back that follows the code in Line 2.

```
IF   the goal is to test if a value
     is equal to zero
THEN use the function EQUAL and set
     as subgoals to code the value
     and zero
```

Associated with the error in Line 5 is the following buggy production:

```
IF   the goal is to perform an opera-
     tion on a recursive call to a
     function
THEN use that function and set as
     subgoals to write the arguments
     to that function
```

Both of these buggy productions are attached to natural-language templates that are instantiated to provide the feedback that follows the error.

Altogether the tutor has over 1200 productions (correct and buggy) to model student performance in our lessons, which cover all the basic syntax of Lisp, design of iteration and recursive functions, use of data structures, and means–ends planning of code.

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

```
                CODE FOR fact
```

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

```
                EXAMPLES
```

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

FIGURE 1. The Screen Configuration before Line 5 in the Sidebar

```
- - -  YOU ARE DONE. TYPE NEXT TO GO ON  - - -
- - -  AFTER TESTING THE FUNCTIONS YOU   - - -
- - -  HAVE DEFINED                      - - -
(defun fact (n)
      (cond ((zerop n) 1)
            (t (times n (fact (sub1 n)))))))
```

```
                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
```

**FIGURE 2. The Screen Configuration at the End
of the Dialogue in the Sidebar**

ISSUES RAISED BY THE LISP TUTOR

We think the Lisp tutor project was very successful. We produced a viable piece of software that teaches a one-semester course at CMU and is now available as a commercial product. Comparisons with other teaching modes have shown that students do better with the Lisp tutor than they do in courses where they simply write programs in ordinary Lisp environments, although they do not do as well as they would if they had a private human tutor. Typical results show students scored one letter grade higher on final written tests when they worked with the tutor. Having established a functioning tutoring paradigm, the issue now is how to embellish the tutor so that it can match or surpass the effectiveness of a human tutor. Although the current system, running on DEC MicroVaxes, is cost-effective in an environment such as CMU, we would like to work toward a more efficient and affordable system. Below we discuss a number of dimensions for improvement.

Interaction

The interaction style with the Lisp tutor is quite restrictive. First, coding must be top-down. It is not possible to generate a piece of code "inside-out" and insert it in a slot. Second, the program is generated in a strictly left-to-right manner. Students cannot code the body of a loop before the initialization, for instance. Third, the tutor reacts to each symbol as the student types it. A student cannot create and edit a sequence of symbols before getting feedback from the tutor on any of the symbols.

Each of these features in the tutor has pedagogical motivations. There are reasons to believe that, on average, top-down and left-to-right programming is in fact the correct style. Immediate feedback on decisions is usually best. There are technical motivations as well: Such a tutor can be implemented more efficiently than a more flexible tutor. However, the pedagogical considerations are only approximate—there are situations where the correct problem-solving strategy would be inside-out and where it would be wiser to delay feedback.

For example, one of the problems of immediate feedback in programming is that the student might not have established enough context in the program to explain why a particular piece of code is wrong. One of the advantages of the Proust system [10] is that it waits until the program is complete to give feedback. However, delayed feedback proves difficult in terms of pedagogical effectiveness. What is needed is a system that can choose more strategically when to give feedback.

Another less-than-perfect aspect of the Lisp tutor is that it is limited to entering into rather restrictive dialogues. Improving the dialogue capability would improve instruction. To produce high-quality dialogue, however, considerable amounts of computation are required; and this conflicts with the need for rapid response.

Efficiency

We must distinguish between the interpretive Lisp tutor and the compiled Lisp tutor in discussing the tutor's efficiency. In the interpretive tutor, the production system simulates the student at instruction time. In contrast, the compiled tutor's production system generates all possible simulated behaviors in advance and records them in a data structure for use during tutoring. The original interpretive Lisp tutor is still used to develop new instructional material. The compiled tutor is used to deliver actual instruction. For a similar idea, see Sleeman [13].

Since student modeling is a significant source of computation, both a 50 percent time and space savings are achieved with the compiled tutor. It may seem odd that there is a space savings associated with compiling these problem solutions because all problem solutions must be explicitly represented rather than only having them implicitly in the tutor. However, representing 1200 productions and their partial instantiations is also space-expensive in the GRAPES production system, which has a pattern-matching algorithm modeled after the RETE algorithm, which explicitly trades space for time [8].

We do not believe that the strategy of advance compiling of the intelligence that goes into instruction has been exploited to its full potential in the current tutor.

Software Engineering

The Lisp tutor compares favorably with standards for developing educational software. We built an intelligent computer-assisted system for an entire introductory programming course with the time investment typically associated with development of conventional CAI. The system delivers 30–40 hours worth of instruction with the investment of about three person-years (i.e., 6000 hours). Moreover, we are currently producing remedial material at the rate of 1 hour's worth of remediation for each week's work (40 hours) from nonspecialists (i.e., undergraduates without artificial intelligence training).

Nonetheless, the design of the tutor has unnecessarily slowed down development and frustrated our efforts to produce optimizations of the code. Components of the code are unnecessarily intertwined creating the need for a great deal of coordination. We would like to improve the modularity of the tutor to speed the development cycle.

THE PUPS TUTORING ARCHITECTURE

We are currently working on what we have dubbed the PUPS Tutoring Architecture (PTA), which is an attempt to address the issues raised by the Lisp tu-

tor. We have two major goals. First, we want to base it on the PUPS production system [4], which is currently our best approximation to the architecture of human problem solving and which has a great deal more flexibility in its flow of control than the GRAPES system used in the Lisp tutor. Using PUPS, the tutor can follow a student who chooses to program in a non-left-to-right manner. Unlike the strict left-to-right coding order in the Lisp tutor, at any given time in the PUPS tutor the student can choose to expand any nonterminal symbol in the partially completed program. Second, we would like to generalize our experience with the Lisp tutor to tutor an introduction to any programming language. The languages targeted for demonstration are Lisp, Prolog, and Ada. Rather than creating a series of independent tutors, we have begun to devise a prototypical system architecture that can be instantiated to produce efficient tutors for different languages.

Overview

The PTA differs from the Lisp tutor in several ways. The PUPS architecture strives for maximum modularity to increase maintainability and to cleanly partition the features specific to each language being taught from the general-purpose tutoring apparatus. In addition, the compile-and-tutor strategy is more fully exploited to produce a faster interactive system. Finally, the PTA contains a more flexible internal knowledge representation to facilitate interface improvements such as delayed-feedback interaction style.

Experimental versions of almost all of the components of the PTA have been implemented, although we have not integrated them or developed a curriculum that can be used in an actual instructional setting. Figure 3, page 848, illustrates the organization of the PTA. The PTA contains two major components: a *tutoring engine* and a *problem solver*. The tutoring engine interactively assists the students step-by-step as they write programs. To determine the correctness of a student's actions and to decide which feedback messages to provide, the tutoring engine refers to a data structure called the *solution trace*, constructed in advance by the problem solver. The problem solver generates the solution trace for each programming exercise by running a PUPS simulation of the ideal and buggy student behavior involved in completing that exercise and by attaching the appropriate tutorial actions to that trace.

Key Features

It may seem that we are striving for contradictory goals in designing the PTA—more generality, intelligence, and flexibility on the one hand and greater
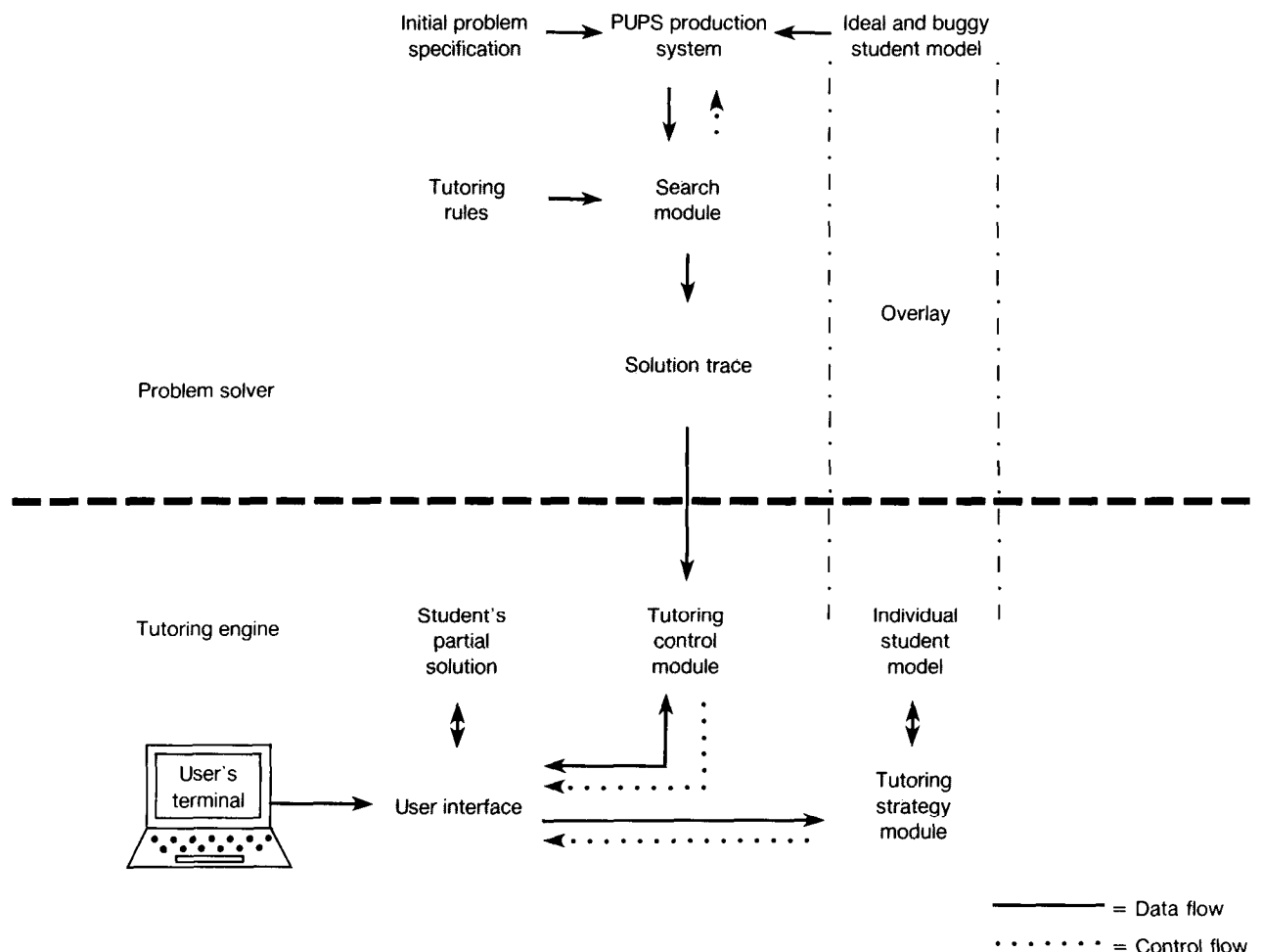
FIGURE 3. The Overall Structure of the PUPS Tutoring Architecture

efficiency and economy on the other. However, the key insight is that we do not have to achieve these two goals *simultaneously*. The partition in Figure 3 between the problem solver and the tutoring engine allows us to achieve the two *separately*. It no longer matters if the system spends minutes calculating exactly how to respond to a particular error because this is done off-line during the construction of the solution trace where real-time computation is not an issue. Also, using an expensive machine to generate the solution trace is not a drawback because only one such machine is needed. Once constructed, the solution trace can be copied and distributed for use by students on very economical computers.

The principal concept in the implementation of the PUPS tutor is the solution trace. It is an economical representation of possible code a student might type, with buggy code tagged with precomputed tutorial interactions as well as an abstract specification of the set of possible correct solutions a student might generate to solve a problem. The solution trace is abstract in the following senses:

- If there are multiple ways to achieve coding goals 1 and 2 and these ways are independent, it does not represent all combinations of the two sets of code.

- It does not explicitly represent the order of logically unordered portions of code.

- It does not represent the order in which the student actually enters the code.

The size of the solution trace is not trivial, but its size is small relative to the number of possible student interactions. To represent a given program used in the Lisp tutor, we often have to store upwards of a hundred abstract programming options that can be combined to generate thousands of concrete programs that can be programmed in literally millions of orders. For instance, the very simple factorial program (sidebar, pp. 844–845) has 8 abstract programs reflecting decisions such as whether to count up or down and whether to code the function recursively or iteratively. These 8 abstract programs map into 64 actual programs, ignoring things such as different

choices for variable names. There are thousands of ways, aside from the basic left-to-right coding, the student could write each of these programs.

The solution trace is a *code-centered* representation because it is built around the actual code that the student might write. The code-centered solution trace offers a compromise between standard CAI and standard ICAI. As with standard CAI, the tutorial interactions are already represented before the tutoring interaction, and they do not have to be dynamically computed. This yields computational efficiency and economy. However, as with ICAI we do not have to go through the laborious effort of constructing the tutorial interactions by hand. This speeds software development. The system's intelligence lies primarily in its ability to automatically generate tutorial interactions. Another advantage over standard CAI is the abstractness of the code-centered representation, which eliminates the need to specify all possible interactions. Thus, it is possible to follow the student in detail down many possible paths and to be highly flexible about what the student is allowed to do—these have always been goals of ICAI.

The code-centered representation can be thought of as an annotated answer sheet. The tutoring process involves uncovering portions of this sheet in response to the student's actions. Each time the student performs a legal step, the corresponding portion of the solution is displayed. When the student makes an error, the tutor prints a suitable message stored at the location in the program where the error was anticipated. Since each piece of code is associated with the name of the production rule that produced it during the student simulation, the tutor can interpret each uncovering operation as a coding action. The student's partial solution thus forms an overlay of the code-centered solution trace. Since editing actions reported by the user interface map directly onto commands to uncover specific portions of the solution trace, the interactive stage of student interpretation handled by the tutoring engine becomes simpler.

Current Implementation Efforts

We have recently begun to implement the PUPS architecture and have developed a rudimentary tutor for small subsets of Ada, Lisp, and Prolog. We plan to implement our tutoring engine using an existing structure editor from the MacGnome programming environment [7] recently constructed at CMU for the Apple MacIntosh. Using an efficient conditional display mechanism [9] to implement our answer-uncovering scheme, we hope to construct a highly modular and affordable MacIntosh tutoring engine that can be made available to students.

CONCLUSION

This article has focused mainly on the technical progress that has been made while implementing intelligent tutors for computer programming. However, it should be stressed that studying students working with these tutors is leading to a better understanding of the programming process itself. Essentially, intelligent tutors serve as tools for collecting data about programming behavior and for producing experimental manipulations. Given that the effectiveness of these tutors is predicated on our understanding the cognitive processes involved in programming, we expect our research with them to lead to improved tutoring based on a deeper understanding of programming.

REFERENCES
1. Anderson, J.R., and Jeffries, R. Novice LISP errors: Undetected losses of information from working memory. *Hum.-Comput. Interaction 1*, 2 (1985), 107–131.
2. Anderson, J.R., and Kessler, C.M. A model of novice debugging in LISP. In *Empirical Studies of Programmers*, E. Soloway and S.S. Iyengar, Eds. Ablex, Norwood, N.J., 1986.
3. Anderson, J.R., and Reiser, B.J. The LISP tutor. *Byte 10*, 4 (Apr. 1985), 159–175.
4. Anderson, J.R., and Thompson, R. Use of analogy in a production system architecture, 1986. Unpublished manuscript.
5. Anderson, J.R., Farrell, R., and Sauers, R. Learning to program in LISP. *Cognitive Sci. 8*, 2 (Apr.–June 1984), 87–129.
6. Bloom, B.S. The 2 Sigma Problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educ. Res. 13* (1984), 3–16.
7. Chandhok, R., Garlan, D., Goldenson, D., Tucker, M., and Miller, P. Programming environments based on structure editing: The GNOME approach. In *Proceedings of the 1985 National Computer Conference* (July). IFIPS Press, 1985.
8. Forgy, C.L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell. 19*, 1 (1982), 17–37.
9. Garlan, D. Flexible unparsing in a structure editing environment. Tech. Rep. CMU-CS-85-129, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa., Apr. 1985.
10. Johnson, M.L., and Soloway, E. PROUST: An automatic debugger for Pascal programs. *Byte 10*, 4 (Apr. 1985), 179–190.
11. Reiser, B.J., Anderson, J.R., and Farrell, R.G. Dynamic student modelling in an intelligent tutor for LISP programming. In *Proceedings of IJCAI-85* (Los Angeles, Calif.). IJCAI, 1985, pp. 8–14.
12. Sauers, R., and Farrell, R. GRAPES user's manual. Tech. Rep., Psychology Dept., Carnegie-Mellon Univ., Pittsburgh, Pa., 1982.
13. Sleeman, D. Inferring student models for intelligent computer-aided instruction. In *Machine Learning*, R.S. Michalski, J.G. Carbonnel, and T.M. Mitchell, Eds. Tioga, Palo Alto, Calif., 1983.
14. Sleeman, D., and Brown, J.S., Eds. *Intelligent Tutoring Systems*. Academic Press, New York, 1982.

Authors' Present Address: John R. Anderson and Edward Skwarecki, Carnegie-Mellon University, Pittsburgh, PA 15213.