

THE LISP TUTOR

BY JOHN R. ANDERSON AND BRIAN J. REISER

*It approaches the effectiveness
of a human tutor*

FOR THE LAST FIVE YEARS, we have been studying how students learn mathematical, logical, and programming skills. We have reached the point where we can develop computer-based tutors for such domains. This article discusses our work on a LISP tutor. LISP is one of the main programming languages of artificial intelligence (AI) and has gained importance with the rising prominence of AI.

Universities like ours, Carnegie-Mellon University (CMU), are seeing a rapidly increasing demand for courses in LISP. Many students here learn it as their first programming language. But LISP is quite difficult because of its symbolic nature and its use of recursion and because of the need to learn abstract AI programming techniques. We see a remarkable range of mastery from courses designed to teach LISP. Some students find that a single course serves as their entry into the world of AI; others leave feeling they have learned nothing.

Studies of students learning such cognitive skills have revealed that private tutoring appears to be much more effective than conventional

classroom instruction. Classroom learning involves listening to lectures, reading texts, and working alone on homework problems. Private tutoring provides the student with an experienced person to guide his reading and problem solving. In a comparison involving LISP, we found that students with private human tutors needed only 11 hours to learn as much as classroom students learned in 43 hours. In both situations most of the time was spent actually trying to write LISP programs rather than reading or reviewing the instruction. The major role of the tutor is to make the problem-solving episodes more effective learning experiences.

Educational psychologists have observed that private tutoring is an advantage with many different types of material. One study (see reference 1) compared students who spent the same amount of time learning—some with private tutors and some in the classroom—for two different subjects, probability and cartography. Ninety-eight percent of the tutored students did better on performance tests than the average classroom student did. Interestingly, the major benefit occurred with the poorer students. There was

relatively little advantage of private tutoring for the best students.

Our goal has been to develop a computer-based tutor that is as effective in teaching LISP as a human tutor. GREATERP (Goal-Restricted Environment for Tutoring and Educational Research on Programming) is an attempt to combine artificial-intelligence technology and a psychological theory of skill acquisition into an effective teaching device. This tutor is itself a large LISP program that runs under Franz LISP on VAXes. We have already begun field testing this tutor in CMU classrooms and have seen it lead college students to faster, more effective learning of LISP programming. In this article, we discuss how the tutor works, why it is effective, and the prospects for moving a version of it to personal computers.

GREATERP is only one of the tutors

(continued)

John R. Anderson is a professor of psychology and computer science at Carnegie-Mellon University. He has a Ph.D. from Stanford University. Brian J. Reiser holds a Ph.D. from Yale University and is doing postdoctoral research at CMU. Write to them at Carnegie-Mellon University, Department of Psychology, Schenley Park, Pittsburgh, PA 15213.

that we have developed at Carnegie-Mellon. The Advanced Computer Tutoring Project at CMU currently has tutors under development for high-school algebra and geometry, and we have plans to create tutors for calculus and other programming languages such as Pascal and Prolog. The LISP tutor provides an example of our approach in bringing AI techniques into educational-software development.

INTELLIGENT TUTORING

These tutoring projects are examples of a particularly promising approach to educational software called *intelligent tutoring* (see reference 2). Such systems differ from other AI approaches to education by their attempt to provide effective instruction in problem solving the way a human tutor does.

There are a number of components typically found in an intelligent tutoring system. First, there is a *domain expert*, which can actually solve the problems. (GREATERP contains a system that can write LISP functions from problem specifications.) Second, there is a *bug catalog*, which contains all the possible deviations a particular student can make from the ideal expert behavior. Third, there is a *tutoring module*, which constitutes an expert system for instruction. It is based on three sets of principles: one for determining from a student's behavior what he knows and what confusions or *bugs* he has, another for deciding when to interrupt him in the problem-solving process and what to say, and a third to figure out what problems a student should do and when he should advance to new material. Typically, these decisions are based on an internal model that the tutor keeps of the student's knowledge and difficulties so far, enabling it to tailor its instruction to each individual student.

Finally, the tutoring system must contain an interface for communicating with the student. Its construction requires human-engineering decisions concerning how to present information understandably, how to query the

student, how the student should enter answers, and what information should be maintained on the screen.

Until recently, intelligent tutoring was a topic for advanced research but didn't seem to offer a viable method for delivering educational software. This situation has changed for two reasons. First, an increasing expertise in cognitive psychology and artificial intelligence allows us to build such tutors much more efficiently and rapidly. For example, we can develop lesson material on our LISP tutor at a faster rate than that estimated for conventional educational software (200 hours per hour of instruction). Second, while an intelligent tutor requires a larger, faster computer, hardware costs are dropping to where it is becoming cost-effective to purchase the equipment required.

DESIGN OF THE LISP TUTOR

The goal underlying our design of the LISP tutor is simple: A student should be able to work on a problem in a "friendly" environment, as if he were using a smart, structured editor. However, whenever he makes a planning or coding error or asks for help, the tutor should provide helpful information that guides the student back to a correct path to the solution. In addition, we wanted our tutoring environment to represent the conceptual structure of programming problems better than a simple screen editor.

In order to monitor a student's progress and discover and instruct about errors, the tutor must be able to solve the problems the student is working on. Therefore, the first component in the LISP tutor is the *ideal model*, a simulation of the programming knowledge ideal students use in solving problems. This ideal model is based on a detailed theory of how students learn to program (see reference 3).

We used GRAPES (Goal-Restricted Production System, see reference 4) to represent the rules programmers have for solving problems. Each problem-solving rule is represented in the system as a production rule. Each pro-

duction rule contains an IF part, which is a set of conditions used to determine if the rule applies, and a THEN part, which specifies what to do in that situation. The following are English versions of two of the hundreds of GRAPES production rules known by the tutor:

- ```

IF the goal is to combine LIST1
 and LIST2 into a single list
THEN use the function APPEND and
 set as subgoals to code LIST1
 and LIST2

IF the goal is to check that a
 recursive call to a function will
 terminate and the recursive
 call is in the context of a MAP
 function
THEN set as a subgoal to establish
 that the list provided to the
 MAP function will always
 become NIL after some
 number of recursive calls

```

The first is a straightforward production rule about the use of the LISP function APPEND to make one list from two other lists. The second is a rather esoteric production rule that an advanced programmer might have. When the ideal model codes a LISP function, it applies many production rules like these to plan and then write the code. It also contains a large set of *buggy* rules that represent misconceptions novice programmers often develop during learning.

The ideal model represents the knowledge we want the student to acquire. But the tutor must also represent what he currently knows or does not know and his approach to each particular problem. The tutor follows the student as he types in his code, symbol by symbol, and tries to figure out what correct or buggy production rule would have led to that input. If the rule found is a correct one, then the tutor stays silent and waits for further input. If, on the other hand, the input is in error, the tutor interrupts with advice. Thus, as long as the student follows a path leading to a correct solution, the tutor stays in the background.

(continued)

## The LISP tutor works through the algorithm step by step.

The LISP tutor is designed to provide as much guidance as necessary. When it finds that the student is having difficulty coding a problem, it takes him from "coding mode" into "planning mode"; that is, the tutor works through the algorithm with the student, step by step, using an example. After the algorithm is constructed, the student can return to coding, presumably with a better idea of what he should do to get his code to work properly.

We have designed this tutor with a strong commitment to immediate feedback. As soon as the student makes a mistake, the tutor responds with an appropriate diagnostic message. Because a student can write his code a small piece at a time, the feedback appears as soon as one item is wrong. By contrast, in the standard learning situation a student only receives feedback after he codes the entire function—or set of functions—and tries to run it. There is considerable psychological evidence that humans learn better with immediate feedback.

The tutor also provides guidance by hinting toward the correct solution if the student is having difficulty. These hints take the form of queries and reminders about current goals. If necessary, the tutor can provide the next small piece of code so that the student can continue. This is done at the student's request or after he has made more than the maximum number of allowed errors—usually two—for that portion of code. The goal here is for the student to do as much of the work as possible. Students learn much more effectively by doing than by watching. By providing the next portion of code, the tutor enables the student to work through the rest of the problem in cases where he might otherwise have given up. As

a consequence, he can tackle more and more difficult problems.

A major design feature of the tutoring interface provides the student with a structured editor through which to enter code. This editor automatically balances parentheses and provides placeholders for the arguments of each function. For example, to write a function definition in LISP, you must use the function `defun` followed by the function name, a parameter list, and the function body. To begin, the student types a left parenthesis and the word `defun`. As soon as he types the space following that word, the tutor redisplay the code as

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

The symbols in angle brackets indicate arguments that must be coded. The tutor places the cursor underneath `<NAME>` and highlights it to indicate that the function name must be coded next.

This structured editor relieves students of the burden of balancing parentheses and checking syntax. It enables them to focus on the more conceptually difficult aspects of LISP. Our results show that this leads to faster learning of these major techniques and skills—with no deficit in syntax knowledge. Students removed from the tutor perform as well as or better on all aspects of coding, including algorithm design, memory for LISP functions, and syntax, than those conventionally taught.

The editor also facilitates communication between the student and the tutor. Our studies of interfaces have shown that in the normal question-and-answer format of most educational software, the tutor and the student can easily get "out of sync" on complex problems, where the student is not sure what part of the problem the tutor is talking about. In the LISP tutor, the student types directly into the code, replacing one of the placeholders, and thus it is always clear what part of the problem is being coded. Furthermore, these symbols help to communicate the conceptual structure of the programming prob-

lem. For example, when the student types the iterative construct `prog`, the tutor provides the template for iteration:

```
(prog <LOCAL VARIABLES>
 <INITIALIZATIONS>
 <BODY>
 <REPEAT>
)
```

This template helps to structure the problem into a list of local variables, initializations of those variables, code for the program body (i.e., the repeated actions), and a return to the start of the loop. In many cases a symbol is expanded into more detailed symbols; for example, the `<BODY>` is coded as two portions: a `<TERMINATING CASE>` and the `<UPDATING CODE>`.

When an error arises or the student requests assistance, the tutor constructs an English explanation based on templates associated with each production rule. These explanation templates allow the tutor to describe an error or provide a hint by using a general rule and making reference to the specific problem being coded. However, writing programs to *understand* natural language is an enormously difficult and expensive task. In fact, students' descriptions of their algorithms are often hard for even human tutors to understand. Therefore, when the student is working with the tutor to design an algorithm, rather than having him type in English responses to the tutor's questions, we provide a menu of choices for his responses. This menu is constructed from the English descriptions of the correct and buggy production rules under consideration.

### LEARNING LISP WITH THE TUTOR

Currently the LISP tutor contains approximately 325 production rules about planning and writing LISP programs and 475 buggy versions of those rules. It is effective in diagnosing and responding to between 45 and 80 percent of the student's errors, depending on the complexity of the

(continued)

Table 1: The LISP curriculum.

1. Basic LISP functions
2. Defining new function
3. Conditionals and predicates
4. Structured programming
5. Progs, input/output, and evaluation
6. Integer-based iteration
7. Integer-based recursion
8. List-based recursion
9. List-based iteration
10. Advanced recursion
11. Advanced programming constructs: Maps, Dos, Lets
12. Property lists and arrays
13. Search techniques
14. An advanced problem: Waterjug
15. List structure and destructive functions
16. Macros and advanced function types
17. Advanced topic: Pattern Matching
18. Advanced topic: Implementing Production Systems

Listing 1: The dialogue between student and computer tutor to define the function `rightp`.

Define `rightp`. It takes 3 arguments, `side1`, `side2`, and `side3`, which represent the lengths of the three sides of a given triangle, where `side3` is the longest side. `rightp` tests whether the three sides form a right triangle. In a right triangle, the sum of the squares of `side1` and `side2` is equal to the square of `side3`. However, your function should allow for a measurement error of 2 percent. In other words, your function should return `t` if the sum of the squares of the first two sides is within 2 percent of the square of the third side.

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

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

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

2. (defun `rightp` (`side1 side2 side3`)  
(`lessp` (`difference`) <NUM2>))

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

3. (defun `rightp` (`side1 side2 side3`)  
(`lessp` (`abs` (`difference` <NUM3> <NUM4>))  
<NUM2>))

Which do you want to do first?

PRESS: IF YOU WANT TO:

- 1 Write code to get the sum of the squares of the first two sides.
- 2 Write code to get the square of the third side.

Menu choice: 1

4. (defun `rightp` (`side1 side2 side3`)  
(`lessp` (`abs` (`difference` (`plus`) <NUM4>))  
<NUM2>))

(continued)

## Each lesson takes from one to four hours to complete.

lesson and the amount of testing we have conducted. The curriculum we have designed is shown in table 1. The system successfully tutors students through problems on lessons 1 to 10, each lesson taking from one to four hours to complete. Each topic involves a small instructional booklet and many problems practicing the skills taught in that lesson. Our goal in designing these booklets was to keep the written instruction to a minimum. There is considerable evidence that written technical instruction is most effective when it is brief (see reference 5).

Most of the time in any lesson is spent going through problems. Listings 1 and 2 illustrate sample dialogues in which the tutor monitors the student's progress through simple problems. This does not present the tutor as it really appears. Instead, it shows essentially *teletypewriter* versions of the student-tutor dialogue. The actual interaction involves the use of several display windows: a *code window* in which the student types the code, a *tutoring window* in which tutoring feedback is displayed, and a *goals window* in which reminders are presented.

In the teletypewriter versions the tutor's output is given in normal type characters, while the student's input is shown in bold characters. These listings present "snapshots" of the interaction; each time the student provokes 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 example, in line 2 of listing 1 he has typed (`lessp` (`difference` as an extension of the code in line 1.

It is important to note several things

(continued)

## LISP TUTOR

No. You should create a helping function for this code, rather than coding it directly. Our function, `rightp`, is now 4 levels of embedding deep, and the relation we are about to code—the sum of squares of two numbers—would be a useful helping function. Thus, let's code the sum of the squares of the first two sides using a helping function.

```
5. (defun rightp (side1 side2 side3)
 (lessp (abs (difference (sumsq < ARGUMENTS >) < NUM4 >))
 < NUM2 >))
```

What do you want the helping function `sumsq` to do?

PRESS: IF YOU WANT TO:

- 1 Take 2 arguments, the square of the first side and the square of the second side. Compute their sum.
- 2 Take 2 arguments, the first side and the second side. Compute the sum of the squares of these numbers.
- 3 Have the tutor choose.

Menu choice: 1

No. We wouldn't need a helping function *just* to compute the sum of two numbers. You should create your helping function so that it does more than just a sum.

It would be better to define `sumsq` so that it takes two arguments, in this case, [1] the first side, and [2] the second side, and computes the sum of the squares of these numbers.

Menu choice: 2

```
6. (defun rightp (side1 side2 side3)
 (lessp (abs (difference (sumsq side1 side2) (times)))
 < NUM2 >))
```

You should use a helping function that you have already coded, `square`. Since this function computes the square of a number, you can use it to code the square of the third side.

```
7. (defun rightp (side1 side2 side3)
 (lessp (abs (difference (sumsq side1 side2) (square side3))) (plus)))
```

I don't understand that, Rob. How are you going to code 2 percent of the square of the third side?

### [EXPLAIN]

OK. I'll explain what I wanted. You could either code this directly or create a helping function here to code 2 percent of the square of the third side. Let's create a helping function just to make things easier. We'll call this helping function `percentage`. We will define `percentage` when we are finished coding `rightp`.

What do you want the helping function `percentage` to do?

PRESS: IF YOU WANT TO:

- 1 Take 2 arguments, one hundredth of 2 and the square of the third side. Compute their product.
- 2 Take 2 arguments, 2 and the third side. Compute the given percentage of the square of its second argument.
- 3 Take 2 arguments, 2 and the square of the third side. Compute the given percentage of its second argument.
- 4 Take 1 argument, the square of the third side. Compute 2 percent of its argument.
- 5 Have the tutor choose.

Menu choice: 3

Very good.

```
8. (defun rightp (side1 side2 side3)
 (lessp (abs (difference (sumsq side1 side2) (square side3)))
 (percentage 2 (square side3))))
```

(continued)

about these dialogues. First, recall that the tutor balances parentheses and provides placeholders (e.g., <CASE>, <NAME>, <NUM>) for portions of code that need ex-

panding. Second, despite the large portions of code added from one line to the next, it understands each separate word or symbol the student enters, and it responds with an error

message if necessary for any of those items.

### THE rightp PROBLEM

Listing 1 presents a dialogue from lesson 4 that focuses on decomposing a problem into a set of subfunctions. Although this dialogue contains more errors than the typical student makes, it is a good illustration of how the tutor responds to errors and requests for assistance. The student is allowed to input information until he types cond, at which point the tutor tells him that a conditional structure is not necessary. (The actual screen setup is shown in figure 1.) Next he types (lessp (difference (line 2) and receives a hint; he correctly inserts the absolute value function (abs) before difference in line 3. Note that the tutor presents a menu when it doesn't know what the student will do next. For instance, since the arguments to difference in rightp can be in either order, it needs to know which one will be typed next, and it asks him via the menu following line 3.

After line 4 the tutor gives the student information about when it is useful to code a separate helping function. It queries him after line 5 to make sure they agree on what that helping function will compute. This is an example of the tutor's planning mode. In this case, the student is mistaken about what the subfunction should do; the tutor corrects him. After line 7 it determines that his code will not achieve the goal. However, this input does not match any of the buggy rules in the ideal model, so the tutor provides minimal feedback: it indicates that it cannot understand the input and queries the student to remind him about what he should be trying to code. This hint is not enough for the student, who asks for an explanation by hitting a special key, whereupon the tutor helps him specify another helping function, percentage.

The final form of rightp is displayed in line 8. In line 9 the student defines the helping function sumsq without error and goes on to define percent-

(continued)

OK. Now we have finished coding the function rightp. Let us now code the helping function(s) we used but haven't yet defined.

Which do you want to do first?

PRESS: IF YOU WANT TO:

- 1 Define the helping function sumsq.
- 2 Define the helping function percentage.

Menu choice: 1

Define the helping function sumsq. It takes 2 arguments. (num1 num2) are the suggested names for the arguments, but you may name them whatever you wish. sumsq should compute the sum of the squares of num1 and num2.

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

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

Define the helping function percentage. It takes 2 arguments. (pcnt total) are the suggested names for these arguments, but you may name them whatever you wish. percentage should compute pcnt percent of total.

10. (defun percentage (x y)
  - <PROCESS>

#### [EXPLAIN]

OK. I'll explain what I wanted. Let's figure out how to code x percent of the second number.

How can you code x percent of the second number?

PRESS: IF YOU WANT TO:

- 1 Multiply the second number by one-hundredth of x.
- 2 Multiply the second number by x.
- 3 Divide the second number by 100.
- 4 Have the tutor choose.

Menu choice: 1

Good.

11. (defun percentage (x y)
  - (times <NUM1> <NUM2>))

What do you want to do first?

PRESS: IF YOU WANT TO:

- 1 Write code to get one-hundredth of the first number.
- 2 Write code to get the second number.

Menu choice: 1

12. (defun percentage (x y)
  - (times (quotient x .01) <NUM2>))

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

13. (defun percentage (x y)
  - (times (quotient x 100) y))

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

age in line 10. He is stuck as to what to do after typing the function body and requests an explanation. The tutor helps him refine his algorithm. After this he defines percentage with one error in line 12.

After the student defines `rightp` and its helping functions, the LISP tutor puts him into a real LISP environment where he can experiment with them and try variations. After he experiments to his satisfaction, the tutor

provides the next problem in the lesson.

### THE fact PROBLEM

Listing 2 illustrates how the tutor guides the coding of a recursive function such as finding the factorial of a number, an early problem in lesson 7. Although the student has some difficulty with the syntax of the conditional test in lines 1 and 2, he basically codes the terminating test correctly. Typically, we find students have little difficulty with terminating cases but great difficulty with recursive cases. The dialogue after line 3 shows how the tutor guides the student through the design of the recursive function. It leads him to construct examples of the relationship between `fact (n)` and `fact (n - 1)` and then asks him to identify the general relationship. Figure 2 shows the screen image at a critical point in the design of this function.

The dialogue after this point shows two classic errors students make in defining recursive functions. The first in line 4 is to call the function directly without combining the recursive call with other elements. The second, in line 6, 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 with it. He must trace the function and the recursive calls embed and unravel. Figure 3 shows the screen image at this point, with the code on top and the trace below it.

### EVALUATING THE LISP TUTOR

We should emphasize that the tutorial knowledge in these interactions was not especially handcrafted for these problems; it shows the general approach used by the tutor. That is the power of an intelligent tutoring system. Once we have a system that can tutor recursion problems, for example, we can easily add problems on that topic. Furthermore, although some testing is necessary to build our catalog of student errors and misconceptions, once coded, the tutor can

(continued)

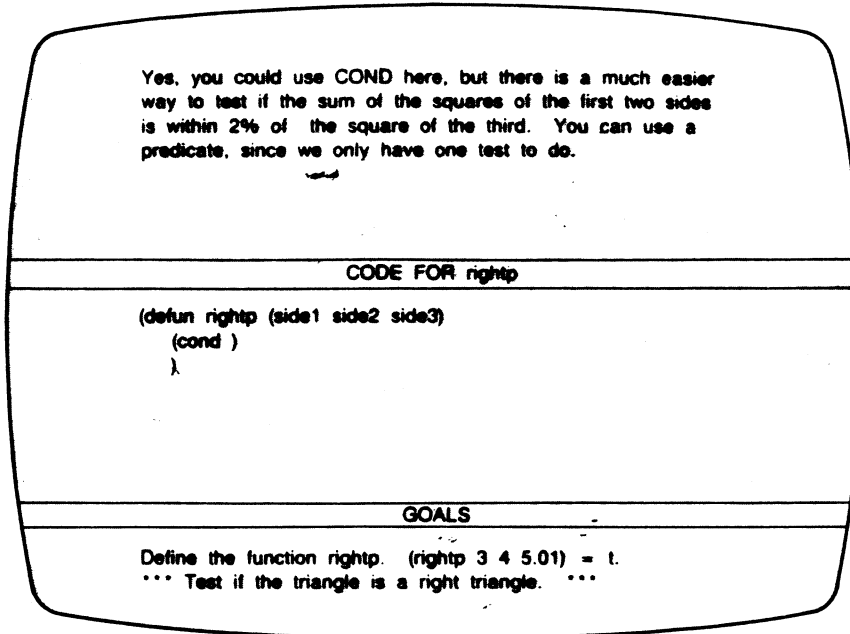


Figure 1: The screen configuration after line 1 in listing 1.

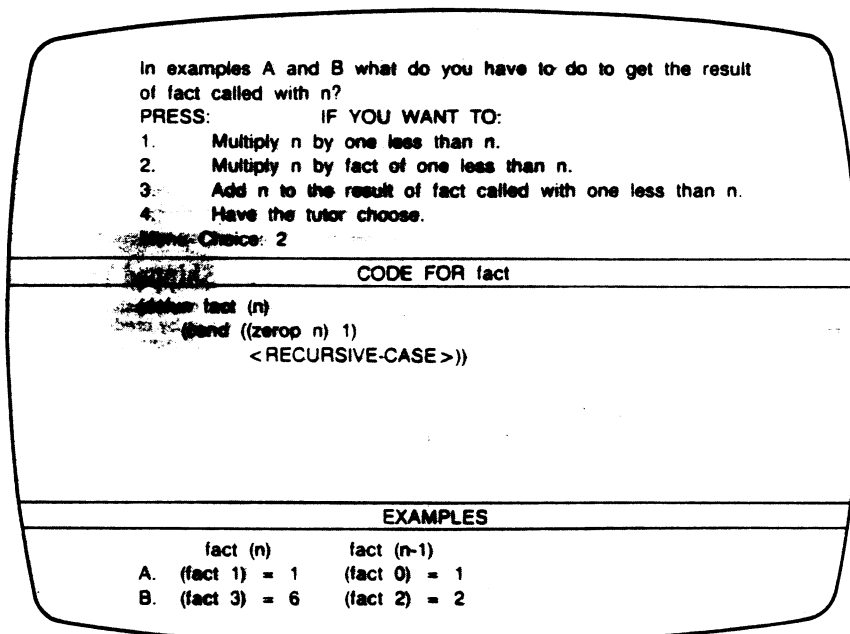


Figure 2: The screen configuration before line 4 in listing 2.

```

-- 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 3: The screen configuration at the end of the dialogue in listing 2.

lead students through a variety of correct solutions and respond to a wide range of errors. Thus, it is generative in the sense that it creates instructional interactions rather than using preprogrammed sequences.

Generally, students are happy with the tutor and rate it better than learning experiences they have had in other introductory programming courses. Their major complaints are that it uses too many menus and is sometimes slow. We are implementing optimizations and improvements to resolve both these problems. Interestingly, students are more appreciative of the tutor's guidance and more tolerant of its speed as the lessons increase in difficulty.

We have run one evaluation of the tutor, comparing it to experienced private human tutors and "on-your-own" condition. We found that the advantage of private tutoring—human or computer—over the on-your-own condition increased as the material became more difficult. The students finished lesson 1 basically with the same speed and success in all three conditions. However, those in the on-your-own condition often had great difficulty getting through the recursion material; in fact, some did not finish the material in the allotted time. We had to extrapolate how long it would have taken them to finish it. Averaging the actual and extrapolated times, the students took 11.4 hours with the human tutors, 15 hours with the computer tutor, and 26.5 hours "on-your-own" to learn six lessons. Polls of students in a traditional classroom suggest that it takes over 40 hours to cover the material in that situation.

We tested the three groups just before the recursion exercises. We found essentially identical performances even though they were learning at different rates. In another evaluation, we found students working with the tutor learned nine lessons of material faster and performed better on competence tests than students learning without the tutor. Thus, the basic conclusion is that the human

(continued)

## Subscription Problems?



### We want to help!

*If you have a problem with your **BYTE** subscription, write us with the details. We'll do our best to set it right. But we must have the name, address, and zip of the subscription (new and old address, if it's a change of address). If the problem involves a payment, be sure to include copies of the credit card statement, or front and back of cancelled checks. Include a "business hours" phone number if possible.*

**BYTE**  
 Subscriber Service  
 P.O. Box 328  
 Hancock, NH 03449



**Listing 2:** *The dialogue involved in defining the function fact, which computes the factorial of the number given.*

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

1. (defun fact (n)
 

```
(cond ((equal) <ACTION >)
 <RECURSIVECASE >))
```

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 >)
 <RECURSIVECASE >))
```

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

3. (defun fact (n)
 

```
(cond ((zerop n) 1)
 <RECURSIVECASE >))
```

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

(continued)

What is the value of (fact 2)?

Answer: 2

### EXAMPLES

- |                         |                      |
|-------------------------|----------------------|
| <b>fact</b> (n)         | <b>fact</b> (n - 1)  |
| A. ( <b>fact</b> 1) = 1 | ( <b>fact</b> 0) = 1 |
| B. ( <b>fact</b> 3) = 6 | ( <b>fact</b> 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 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 <NUM> <NUM2>))))

What do you want to do first?

- PRESS:            IF YOU WANT TO:
- 1            Write code to get the number argument.
  - 2            Write code to get the recursive part.

Menu choice: 1

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 1 from a number.

8. (defun fact (n)
   
    (cond ((zerop n) 1)
   
        (t (times n (fact (sub1 n))))))

out of range for the general public. The amount of computational power required is considerable. Even in future, more efficient implementations we estimate a minimum of 1 megabyte of memory will be necessary to tutor one student on one machine. By the late 1980s the computational power to deliver such instruction should be more generally available. For instance, in conjunction with IBM, CMU is planning to have on campus by 1986 a personal computer with 1 megabyte of memory capable of 1 million instructions per second. Such a machine will be more than adequate to implement the LISP tutor.

These technological trends encourage optimism about the future of intelligent tutoring efforts, of which the LISP tutor is one. We hope that, with continued research in domains such as high-school mathematics and college-level programming, we will soon establish the conceptual foundations to use the computational power that will be available. The prospect is great of providing every student with the educational benefits of a private human tutor. When this happens, the consequences for American education will be nothing short of revolutionary. ■

### ACKNOWLEDGMENT

This research is supported by Office of Naval Research under Contract No. N00014-84-0064. We would like to acknowledge the considerable contributions of Robert Farrell, Elliot Jaffe, Beth Marvel, and Peter Pirolli to the research on the LISP tutor.

### REFERENCES

1. Bloom, B. S. "The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring." *Educational Researcher*, 13, 1984, page 3.
2. Sleeman, D., and J. S. Brown, eds. *Intelligent Tutoring Systems*. New York: Academic Press, 1982.
3. Anderson, J. R., R. Farrell, and R. Sauer. "Learning to Program in LISP." *Cognitive Science*, 8, 1984, page 87.
4. Sauer, R., and R. Farrell. *GRAPES User's Manual*. Technical Report ONR-82-3. Pittsburgh: Carnegie-Mellon University, 1982.
5. Carroll, J. M. "Minimalist Training." *Datamation*, November 1984, page 125.

tutor is still best, the computer tutor not that far behind (and constantly improving), and the traditional on-your-own condition much worse.

### TODAY AND TOMORROW

Currently, we get satisfactory performance of the tutor on a VAX 725 with

3 megabytes of memory devoted to a single student. Our planned optimizations should enable it to handle a pair of students simultaneously running on the same machine. These implementations are cost-effective in expensive private schools like CMU or in industry; however, they are clearly