Corbett jLl.200989

In press: J. Larkin, R.Chabay & C.Sheftic (eds) Computer assisted instruction and intelligent tutoring systems: Establishing communication and collaboration. Hillsdale, NJ: Erlbaum.

7 LISP Intelligent Tutoring System: Research in Skill Acquisition

Albert T. Corbett and John R. Anderson
Advanced Computer Tutoring Project, Carnegie Mellon University

The LISP Intelligent Tutoring System (LISPITS) is an instructional program that helps students learn to program in the computer language LISP. Specifically, the tutor helps students with homework exercises. In each exercise the student is given a written description of a short program to write and, as the student types the program at the terminal, the tutor monitors the student's performance and provides assistance when errors are made. The tutor currently covers the first twelve chapters of an introductory LISP text (Anderson, Corbett & Reiser, 1987) and includes approximately 240 exercises.

LISPITS is dubbed "intelligent" because it is capable of generating correct solutions to the exercises and of assisting students based on this capability. The decision to build the tutor was inspired in large part by the collection of papers in *Intelligent Tutoring Systems* (Sleeman & Brown,1982) describing several tutors that embody the general approach described below. The tutor was developed for two major purposes:

- To automate some of the advantages of a personal tutor, thereby making them more widely available
- To test the real-world applicability of a psychological theory of skill acquisition, called ACT* (Anderson, 1983) and described below.

LISPITS was first tested in the summer of 1983, and has been used since then as a research tool. It has also been used in teaching a LISP course in the psychology department at Carnegie Mellon University each term since the fall of 1984.

The principal goal of the tutor is to allow students to practice programming. The tutor is constructed to be used in conjunction with an introductory LISP text *Essential LISP* (Anderson et al., 1987), which describes the LISP language. After every major section in the text (roughly every two or three pages) there are coding exercises which require the students to write brief programs. Although these exercises can be done with LISPITS, they can also be done with paper and pencil or on a computer in the conventional mode of typing, executing and debugging solutions. The goal in developing LISPITS has been to optimize the time spent in doing these coding exercises. ~~This goal omits~~ several things a human tutor might do, ~~that~~ LISPITS does not do: LISPITS doesn't try to provide any initial exposition of LISP, it doesn't make any provision for the students to ask general questions about LISP, and it doesn't try to assess the student's understanding of LISP (except in the context of doing exercises). Rather, the tutor provides an environment, similar to an editor, in which the student can do the exercises. But unlike an editor, it ensures that the student's final solutions are correct. More specifically, the tutor provides feedback when the student makes a mistake and, if the student appears to be floundering, will provide the next correct step in the solution. Thus, exchanges between the student and tutor are in the context of doing exercises and arise only if the student is having difficulties. Although in principle a student might complete all the exercises without knowing the tutor is in the background, in actuality, even if the student makes no mistakes, the tutor occasionally presents a menu in order to clarify the student's responses.

Given th

In this chapter we discuss assessment studies of the tutor and our experiences with it in the classroom. We then present the central assumptions of the ACT* theory and how these assumptions motivate the tutor's design and implementation. We then describe the research we have conducted with the tutor. To provide context, however, we precede the main part of the chapter with a brief discussion of LISP and a picture of the student's interaction with the tutor.

LISP AND AN ILLUSTRATIVE INTERACTION WITH LISPITS

At this point we first give a brief description of the language LISP and then run through a hypothetical interaction with LISPITS to give some idea what it is like to use the tutor.

BRIEF DESCRIPTION OF LISP

LISP is an interpretive language (like BASIC), which means it is not necessary to compile LISP code before running it. Instead it is possible to "enter a LISP environment" on the computer, that is, run a program called a LISP interpreter, and just type LISP code. The interpreter will "evaluate" (execute) the code and return an answer. For example, in the following interaction, the LISP interpreter has put a prompt (=>) on the screen. The user has typed a line of code, indicating the addition of three numbers. The LISP interpreter executes the line of code and returns the answer, `12`.

```
=> (+ 3 4 5)
12
```

Function calls

Functions in LISP are generally designed to accept one or more "arguments" (input values) and return an answer. The expression `(+ 3 4 5)` in the example above is a function call. The function + is applied to three arguments, and an answer is returned. There are two important points to recognize concerning the structure of function calls:

1. LISP uses prefix notation the operator precedes all the argument values on which it operates.

2. A function call takes the form of a list: the operator, followed by its arguments, are all enclosed in a set of parentheses.

In evaluating a function call, LISP first evaluates each of the arguments and then applies the operator to them. For example, consider the following function call:

```
=> (+ 4 (* 10 5))
54
```

The user has called the function + with two arguments, but the second argument is itself a function call (* 10 5). Before LISP can apply +, it must evaluate this function call. The operator * performs multiplication, so (* 10 5) returns 50, and the call to + returns 54.

In the preceding example, LISP evaluates the numbers (4, 10, 5), returning their own values. LISP can also evaluate variables. For example, suppose the symbol `sym` has the value 5. Then, as above,

```
=> (+ 4 (* 10 sym))
54,
```

because LISP evaluates `sym` as 5. If one wants LISP to evaluate a symbol literally rather than as a variable, then the symbol is preceded by a single quotation mark (e.g., `'sym`).

Function definitions

Programming in LISP largely consists of defining new functions. New functions can be defined in LISP by means of the built-in function `defun`. This function takes three arguments:

1. The name of the function.

2. A list of variables called "parameters."

3. The body of the function describing what the function does.

INTERACTING WITH LISPITS TO SOLVE A PROBLEM

We now turn to the tutor LISPITS and how a student might work with it to define a LISP function.

A sample problem

Consider the following problem description:

Define a function called `pal` that takes a single list as an argument and returns a palindrome that is twice as long. A palindrome is a list that reads the same forward and backward. For example, `(pal '(a b c))` returns `(a b c c b a)`.

Note that the argument of `pal` begins with a single quote, meaning that LISP should not evaluate the symbol `(a b c)` before applying `pal` to it.

The student is being asked to define a function which, when provided a list, creates a new list with a specific structure. The student must use the existing function `defun` to define a new function `pal`. As described above, `defun` requires three arguments. The following solution satisfies the problem description:

```
(defun pal (origlist)
   (append origlist (reverse origlist)))
```

The first symbol following `defun` (its first argument) specifies the name of the new function, `pal`. The second symbol `(origlist)` is a list containing a single parameter `origlist`. This parameter is a variable; when `pal` is subsequently executed and given a list, such as `(a b c)`, the list will automatically be assigned to the variable `origlist`. The final argument of `defun, (append origlist (reverse origlist))`, is the "body" of the function definition. To code the body of the function the student must realize ~~that it~~ is ~~necessary~~ to generate a list which is a mirror-image reversal of the original list, code `(reverse origlist)` ~~of the first and~~ then merge the original list and this reversal into a new list, ~~(append origlist (reverse origlist)).~~ with the function append.

Interaction with LISPITS

Let's trace the process by which a student might generate this code. The following snapshot shows the terminal screen shortly after the student has begun the exercise.

Tutor Window
Define a function called `pal` that takes a single list as an argument and returns a palindrome that is twice as long. A palindrome is a list that reads the same forward and backward. For example, `(pal '(a b c))` returns `(a b c c b a)`.
Code Window
(defun <**name**> <parameters> <body>)

When students are working on an exercise, the terminal screen is divided in two, with a tutor window at the top of the screen and a code window at the bottom. The tutor communicates with the student by means of the tutor window; the problem description appears in this window at the beginning of an exercise and remains there except when the student makes a mistake. The code the student types appears in the code window.

In the preceding snapshot, the student has already typed a left parenthesis and `defun` and LISPITS has responded by putting up a template for the student. Specifically, it has provided a matching right parenthesis and put three goal symbols on the screen in angle-brackets. These angle-bracket symbols are not themselves LISP code, but stand for the three arguments for `defun` that the student needs to fill in. They represent remaining goals to be satisfied. LISPITS highlights the goal symbol which the student must work on next. LISPITS constrains the student to type the function definition from left-to-right and top-down, so the next symbol the student must replace with code is `<name>`.

As the student works on an exercise the tutor monitors the student's input on a symbol-by-symbol basis. LISPITS is intended to recognize any reasonable solution that conforms to both the problem specification and the stylistic guidelines specified in the text. In many exercises there are several possible solutions and in some cases LISPITS recognizes literally hundreds of acceptable variations . As long as the student remains on any reasonable solution path, the interactions with LISPITS is very similar to working with a programming editor. As the student types, the tutor replaces the goal symbols with the student's correct entries, and advances the cursor to the goal symbol that is to be expanded next.

After the student types `defun` and LISPITS has put up the three-part function-definition template, the tutor then highlights each of the three goal symbols in succession, for the student to fill in. The student would type `pal` over the first symbol `<name>`. Then the student would replace the symbol `<parameters>` with a list (i.e., parentheses enclosing a set of symbols for function parameters). As we have seen, the function `pal` takes a single argument, so only one parameter is required, and the student should type a list containing a single symbol here. There are many errors that students make at this point as they are first learning to code functions. One interesting error would be to type the parameter list `(list)`. Technically, this is is a correct step; in LISP the student can legally create a parameter called `list`. However, LISPITS would object to this step for two reasons. The following snapshot shows what the student's code would look like at this juncture and how LISPITS would respond. (In each of these snapshots, the specific input to which the tutor is responding is in bold typeface).

Tutor Window
Remember that you are trying to get the parameter list of the function here. You should not be calling the function list. If you were thinking of using list as a parameter name, it is a bad idea because you might get confused between the function and the parameter.
Code Window
(defun pal (list) <process>)

Whenever the student makes a mistake, the tutor immediately provides feedback and, if the tutor can diagnose the nature of the error, it provides an explanation. After the student presses the return key, the erroneously typed symbol is erased and the student is given chance to type a correct symbol. The student then types a new symbol. If this response is correct, the problem description reappears. If the response is again erroneous, another feedback message appears. If the student repeatedly makes errors at a particular step that the tutor cannot diagnose or, if the student is repeating the same type of diagnosed error, the tutor will intervene to prevent floundering and provide the student with a correct next step along with an explanation, then allow the student to proceed.

The message in the preceding sample tutor display is typical of the general framework for providing feedback on errors. It reminds the student of the current goal and tries to explain why the student's code does not accomplish the goal. As the feedback suggests, LISPITS does not just point out syntactic or algorithmic errors, but also enforces certain stylistic constraints. While `list` can be used as a parameter, it is not good style to do so since it is also the name of a function in LISP, which can cause confusion. The other interesting point about this error is that there is another interpretation. The student may have intended to call `list` as a function here, perhaps because he/she forgot the parameter list and began work on the body of the function. Since there are two explanations of the student's behavior and LISPITS cannot distinguish between them, it says something about both interpretations.

In the next snapshot, we assume that the student has typed the parameter list correctly, with a single parameter `origlist`. This parameter represents the list to be given to the function `pal`. The student now makes a mistake in beginning the body of the function. The Tutor Window shows LISPITS' response.

Tutor Window
Yes, you will need to use the function REVERSE, but not yet. Remember that you need to make a new list that combines the original list with its reverse, so first you will need a list combining function.
Code Window
(defun pal (origlist)   **(reverse)**

As the feedback suggests, the current goal is to code a function that will build a list, but the student's code, `reverse` will not accomplish that goal. However, LISPITS recognizes that `reverse` does accomplish a subgoal of the current goal. It tells the student that it is premature to be working on that subgoal and tries to explain what the current goal is.

In the next snapshot the student seems to be working on the correct goal, but has called the wrong function to accomplish it:

Tutor Window

You will need to make a list, but LIST is the wrong combining function. If you LIST (a b c) with its reverse, you will get ((a b c) (c b a)) instead of (a b c c b a).

Code Window

```
(defun pal (origlist)
    (list)
```

In this message, LISPITS confirms that the student is working on the correct goal, but tries to show, by means of the example in the problem description, why the student's code won't satisfy the goal.

In the following cycle the student has the right idea, but has made a typing error. LISPITS has a simple spelling corrector which detects the likely error and queries the student.

Tutor Window

Define a function called pal that takes a single list as an argument and returns a palindrome that is twice as long. A palindrome is a list that reads the same forward and backward. For example, (pal '(a b c)) returns (a b c c b a).

Code Window

```
(defun pal (origlist)
    (append orglist <list2>))

Did you mean origlist?
```

In the next snapshot the cursor is on the symbol `<list2>` and the student has asked for an explanation of what to do next.

Tutor Window

Here we can see that the argument list needs to be reversed. REVERSE will flip around the order of the elements in a list. For example, if we apply REVERSE to (a b c) we get (c b a).

Code Window

```
(defun pal (origlist)
    (append origlist <list2>))
```

LISPITS tells the student what the correct next step is and will put the correct symbol into the code for the student.

Finally, in the next snapshot the student has completed the code correctly.

Tutor Window

Define a function called pal that takes a single list as an argument and returns a palindrome that is twice as long. A palindrome is a list that reads the same forward and backward. For example, (pal '(a b c)) returns (a b c c b a)

Code Window

```
(defun pal (origlist)
    (append origlist (reverse origlist)))
```

After completing each exercise the students have an opportunity to try out the code they just wrote. When students complete an exercise, they enter a LISP window that gives them access to the a LISP Interpreter. Students can experiment in the LISP window as they choose; the only constraint is that they successfully call the function they have just defined at least once. The following snapshot shows a the result of successfully calling the newly defined function `pal` on the list `'(d e f)`.

LISP Window

```
=> (pal '(d e f))
(d e f f e d)
```

LISPITS: ASSESSMENT AND CURRENT USE

Two studies of LISPITS were conducted early in its development to assess its effectiveness (Anderson, Boyle, & Reiser, 1985; Anderson & Reiser, 1985). The first study compared three groups of students, all novice programmers, who learned LISP by reading a text and doing a standard set of exercises. One group performed this task with the assistance of a human tutor, a second group used LISPITS to do the exercises, and the third group performed the task on their own, doing the exercises entirely in the LISP window. There were no differences among the three groups on a post-test, but there were substantial differences in how quickly they worked through the exercises. The students working with a human tutor completed the task in approximately 12 hours, those working with LISPITS finished in about 15 hours, while those working entirely on their own required approximately 28 hours.

The second study assessed LISPITS in the context of a class. The students in this study had taken a prior programming class (in Pascal). These students attended lectures on LISP and performed the same set of exercises. Half the students did the exercises with LISPITS, while the other half did the exercises on their own in the LISP window. The students working with LISPITS took 30% less time to complete the exercises and scored 43% higher on a posttest. Thus, while LISPITS is not as effective as a human tutor, it leads to strong performance gains in comparison with students doing exercises on their own.

Currently LISPITS is used to teach a self-paced course in LISP offered each term in the Psychology Department at Carnegie Mellon. Most of the work in the course involves completing LISPITS' twelve lessons. Two mechanisms are built into the program to ensure that students are learning the material. First, LISPITS assesses students' performance as they complete exercises and provides additional exercises to students who are having difficulties (this mechanism is described in more detail below). Second, after every other lesson LISPITS gives an on-line quiz, in which students are asked to perform four programming exercises without tutoring assistance. Students must pass this quiz before they can move on. If they flunk a given quiz twice, LISPITS has them repeat the material in the two preceding lessons. If they flunk the quiz two more times, students are required to see the instructor before proceeding. In addition to these internal mechanisms, there are external mechanisms to ensure that students are able to function in LISP without the assistance of LISPITS. First, they are asked to do a handful of exercises entirely outside the tutoring program with a standard LISP interpreter and a typical screen editor. These non-tutor assignments culminate in a final project that typically involves writing a program that employs artificial intelligence search techniques to perform a problem solving task. In addition, students typically take paper-and-pencil midterm and final exams involving programming exercises.

THEORETICAL PRINCIPLES UNDERLYING LISPITS

The goal of this research project has been to develop a programming tutor on the basis of principles derived from the ACT* theory of cognition (Anderson, 1983). ACT* is a general theory of cognitive processing, and as such is necessarily complex, but only some of its assumptions are directly relevant to the tutor. Table 1 contains a summary of the relevant assumptions from the ACT* theory, with corresponding tutoring principles derived from these assumptions (Anderson, Boyle, Farrell, and Reiser,1987).

Table 1: ACT* Assumptions and Related Principles for a Computer-Implemented Tutor

	ACT* Assumptions	**Corresponding Tutoring Principles**
1.	Problem-solving behavior is goal driven.	Communicate the goal structure underlying the problem-solving task.
2.	Declarative and procedural knowledge are separate. The units of procedural knowledge are IF-THEN rules called productions.	Represent the student's knowledge as a production set.
3.	Initial performance of a task is accomplished by applying weak (general) procedures to declarative knowledge structures. Task-specific productions arise by applying weaker productions to declarative knowledge. These task-specific productions underlie more efficient performance.	Provide instruction in the problem-solving context; let student's knowledge develop through successive approximations to the target skill. Provide immediate feedback on errors.
4.	As a result of additional practice, productions can be chained together into larger-scale productions.	Adjust the step size of instruction as learning progresses.
5.	The student maintains the current state of the problem in a limited-capacity working memory.	Minimize working memory load.

These principles generally governed the development of LISPITS and other programming and mathematics tutors developed by Anderson's group (cf. Anderson, Boyle, & Yost, 1985; Lewis, Milson, & Anderson, 1987; Anderson, Boyle, Corbett, & Lewis, in press). In this section we will elaborate the assumptions of the model and the tutoring principles from Table 1.

BEHAVIOR IS GOAL-DRIVEN (PRINCIPLE 1)

Like most theories of problem solving, ACT* assumes that behavior is goal driven (Miller, Galanter, & Pribram, 1960; Newell, Shaw, & Simon, 1958). Problem solving requires that the goal specified in the problem description (e.g., writing a program) be decomposed into subgoals which can either be satisfied directly or, in turn, decomposed. Thus, to a first approximation, acquiring a skill such as LISP coding requires the student to learn both (1) operations that can accomplish specific goals and (2) how to decompose high-level goals into subgoals that can be satisfied by known operations.

DECLARATIVE VS. PROCEDURAL KNOWLEDGE (PRINCIPLE 2)

A second fundamental assumption, again shared by many cognitive theories, is that a distinction must be drawn between declarative and procedural knowledge. Students are assumed to acquire declarative knowledge of programming by reading or listening to lectures. Declarative knowledge can be readily learned but, in isolation, does not lead to behavior. Instead, performance requires procedural knowledge. It is assumed in ACT*, as in many other psychological models, that this knowledge is represented in the form of IF-THEN rules called productions. When a production is triggered (by the satisfaction of its IF clause), behavior results (through the execution of its THEN clause).

For example, one of the first declarative facts a student learns about LISP might be rendered in English as:

The function + takes one or more numbers as arguments and returns the sum of those numbers.

This declarative knowledge could be employed in different types of tasks, for example:

A Code Generation Task:

Write a function call that will add the numbers 10, 20, 30 and 40.

A Code Evaluation Task:

Compute the answer LISP would return if given the expression `(+ 3 4 5)`.

The assumption of the ACT* theory is that the declarative knowledge alone is not sufficient to perform either of these tasks. Instead, procedural knowledge is required; productions must execute (or "fire") to lead to any behavior such as writing code or evaluating code. In terms of the ACT* model, the ultimate goal in learning a programming language is to acquire productions that can be employed to perform these operations, for example:

A Code Generation Production:

IF the goal is to add together a set of numbers
THEN code a call to + and set goals to code the arguments.

A Code Evaluation Production:

IF the goal is to evaluate a function call of the form
`(+ <arg1> ... <argn>)`
THEN evaluate each of the n arguments and generate the sum of
the resulting numbers.

Note that each of these rules refers to the student's current goal and what to do to satisfy the goal. It is an important assumption of the theory that a production will lead to behavior only in the context of a particular goal. If the current goal is to write code, the second production cannot execute, while if the current goal is to evaluate a LISP expression, the first production cannot execute.

ACQUIRING PROCEDURAL KNOWLEDGE (PRINCIPLES 3 AND 4)

When the student first codes a call to +, it is not by firing a production like the one described above. Instead, the student executes a more general production for coding function names that refers to declarative knowledge structures, for example,

IF the goal is to perform an operation and the appropriate
function name is in declarative memory
THEN code a call to that function and set a goal to code its
arguments.

It is only as a result of successful applications of such general productions to specific declarative knowledge that the more specific coding production for + is acquired.

Once goal-specific productions have been formed, additional practice leads to two additional types of learning. First, productions are strengthened with additional practice. That is, with practice, they come to execute more quickly and reliably. Second, with practice, sequences of productions can be composed into larger-scale productions. One of the first obvious candidates for this composition process that arises in learning LISP concerns the operation of returning the second element of a list. LISP provides a function, `car` which returns the first element of a list, e.g., `(car '(a b c))` returns `a`. LISP also provides a complementary function, `cdr`, which takes a list as an argument and removes the first item, e.g., `(cdr '(a b c))` returns `(b c)`. In order to return the second item in a list, it is necessary to remove the first item, by applying `cdr` to the list and then to return the first element of the resulting list with `car`. Thus, `(car (cdr '(a b c)))` returns `b`. When students first solve this problem (in Lesson 1) presumably they trigger two separate productions that code `car` and `cdr`. However, with experience, they are assumed to develop knowledge characterized by the following composed production:

IF the goal is to code a function call that returns the second element of a list,
THEN code a call to `car` on the result of a call to `cdr` and set a goal to code the argument to `cdr`.

WORKING MEMORY (PRINCIPLE 5)

According to ACT*, the current state of a problem is stored in a limited capacity working memory. It is the current state information that can satisfy the IF clause of a production and so cause it to execute. Implementing the THEN clause of a production ordinarily causes a change in working memory so that some new productions are satisfied.

In the early phases of learning, before efficient productions have been built, relatively large amounts of information must be stored in working memory over relatively long periods of time. As more efficient productions are built, the load on working memory decreases.

TUTORING PRINCIPLES

These fundamental assumptions give rise to the tutoring principles summarized in Table 1 and govern the general nature of LISPITS. The text enables the student to encode declarative knowledge structures and, as Tutoring Principle 3 suggests, the tutor concentrates on giving the student the opportunity to practice the skill and build productions. It is certainly likely that the student will not, by studying the text, completely and correctly encode the necessary declarative knowledge to complete the exercises in the tutor. In fact, empirical estimates of the probability that the student will correctly encode the declarative knowledge to construct various production rules ranges from 15% to 90% with a mean of about 60%. However, the assumptions underlying the tutor suggest that the most efficient course of action is to give students the opportunity to write code, and to repair declarative knowledge as the need arises in the course of coding.

Of course, within this general framework, it is necessary to make various decisions about the specifics of the interface. Again, theoretical assumptions suggest goals to strive for (although not necessarily how to obtain them). For example, in problem-solving, the process of recognizing which productions will satisfy the current goal relies heavily on working memory. This is particularly true in the early stages of problem solving, when larger declarative knowledge structures must be maintained in working memory to allow more general productions to execute. As a result, it should be optimal to reduce the load of non-essential information in working memory. Indeed, research by Anderson & Jeffries (1985) suggests that many errors made by novices learning to program are slips that result from the loss of information from their working memory. This research examined a simple code generation task (among other tasks) involving list operations and found that when more complicated arguments were involved, students were more likely to make errors in selecting the correct function to accomplish a goal. The fact that subjects were making errors inconsistently, i.e., making errors in some situations and not in others, suggests that the errors were slips rather than systematic misconceptions. The fact that errors occurred more often in more complex exercises suggests that the slips may have resulted from working memory overload. The tutor reduces working memory load largely by providing templates for function calls. By providing balancing right parentheses these templates reduce the burden of syntactic considerations in coding. By providing angle-bracket symbols for arguments these templates also provide external cues concerning pending goals (and simultaneously satisfy Principle 2 in Table 1).

One of the tutoring principles in Table 1 has proven quite controversial and deserves special discussion. This is the principle of immediate feedback. This principle is derived from both theoretical and practical considerations. Theoretically, it is assumed that procedural learning is based on a memory trace of the students' practice. If a student makes an error while coding a function and must go back and self-correct then, first, the memory trace is more complicated and, second, it reflects error repair rather than correct generation. These two factors combine to decrease the potential for successful learning. First, it is less likely that a correct production will be formed from the more complicated trace. Second, even if the correct production is formed, it is less likely that the student will recognize its goal at the next opportunity without going through the same error and repair process.

There are two practical reasons for preferring very immediate feedback over more delayed feedback. First, if feedback is to lead to a useful production, it is necessary that it be provided in the context of the appropriate active goal and working memory state. If feedback is delayed it may be less likely that the appropriate context will be reinstated, rendering the feedback less useful. Lewis and Anderson (1985) have provided some evidence of a weaker effect of delayed feedback in an adventure game which is formally equivalent to solving equations. A second reason for providing immediate feedback is that once a student has branched onto an erroneous path, the error can be compounded by additional errors and the student can spend a lot of time and become quite distressed trying to recover. Since the tutor is unable to follow a student once he/she has diverged from a successful path, the simplest solution to this problem is to immediately bring the student back to a correct path. The adequacy of the immediate feedback principle is an open issue and we will return to it in the final experiment discussed in this chapter.

Of course, if the tutor provides feedback on an error, the consequence can only be to repair declarative knowledge and not to encode a production for doing it correctly in the first place. However, the student is given the opportunity to try again at the point of the error, so the possibility remains of generating the correct symbol and perhaps encoding the corresponding production. If the tutor tells the student what code to generate, for example, "code a call to + here," the production a student is likely to form is

IF the goal is to code what the tutor tells me to and
the tutor says to code "+"
THEN code +.

This production is not likely to be generally useful so, in responding to errors, the goal of feedback in the tutor is to provide a reminder of the current goal and why the student's code does not achieve the goal. It is not to tell the student what the correct code is. The tutor only gives the student the correct code when the evidence is that the student is lost.

MODEL TRACING: IMPLEMENTING LISPITS

To construct a tutor that performs according to the specifications in the preceding section, it is necessary to have at all times a pattern against which the students' behavior can be measured. That is, as the student generates a LISP program step-by-step, the tutor must have information that allows it to recognize whether each successive step is on the path to a successful solution. One way to accomplish this task is to provide the tutor with a catalog of possible solutions for each exercise. This approach would allow it to recognize whether each symbol of code does or does not match a known solution. It could recognize correct steps, could have relatively simple rules for deciding if a mistake was close to, or far from, being correct, and could readily tell the student steps that would work. The difficulty arises in trying to explain steps to students, in terms of underlying goals and constructs. A given function always performs the same operation, so it is possible to describe to the student *what* the next step in the algorithm is but, with this approach, it is not possible to say *why* that step is appropriate. This difficulty with a solution catalog might be surmounted, e.g., by providing hand-coded explanations for each solution or, perhaps, by providing the tutor with a description of each exercise and with rules for inferring, in relation to the description, the purpose of each symbol of code. However, we have taken a different approach with LISPITS. Instead of providing LISPITS with a set of solutions for each exercise, we provide LISPITS with rules that allow it to generate solutions.

In this approach LISPITS is provided a set of general rules for writing LISP code and a specification of each exercise. From these, it attempts to model the steps that a student might take in solving a problem. Thus, while the student is working, the tutor in lock-step simulates the steps that a knowledgeable student could take in writing the code. In addition, it models errors that students make at each step on the basis of known misconceptions. By comparing the students' response to the set of possible legal actions and the set of known erroneous actions, the tutor is able to recognize whether the student is on a correct solution path, appears to be suffering from a known misconception, or has typed something unrecognizable. This set of correct and incorrect rules for writing LISP programs is referred to as the student model and is described in more detail in the following section. We refer to this process of comparing the student's steps in writing a program to the steps generated by the student model as **model tracing**.

THE STUDENT MODEL

The student model that underlies the LISP tutor is partly descriptive and partly prescriptive. That is, it is derived in part from observations of students learning LISP (Anderson, Farrell, & Sauers, 1984; Pirolli & Anderson, 1985) and in part from analysis of requisite knowledge for LISP programming and considerations of good programming style. As described earlier, procedural knowledge of how to write LISP code is modelled by a set of productions. Each production is essentially an IF-THEN rule. An English translation of a typical production rule that students learn in Lesson 1 would be:

IF the goal is to form a list by inserting an item at the beginning of an existing list
THEN code a call to the function `cons` and set subgoals to code the item and the list.

A more complex rule that is encountered in a later lesson is:

IF the goal is to code a function that takes a list and the function must access every atom in the list structure and the list structure can be arbitrarily complex
THEN code `cond` to implement car-cdr recursion and set subgoals to code terminating cases and to code recursive cases.

The complete set of correct rules for writing code is referred to as the ideal student model and represents the instructional objectives of the text and tutor. The student model also includes a set of incorrect rules that reflect known misconceptions and are collectively referred to as the bug catalog, or the set of buggy rules.

In actually modelling student behavior, the production system is given a specification of the LISP program to be written and a goal is set to write the program. The problem description, which is analogous to the English description provided the student, is loaded into the production system's working memory. At each step in generating a program, the production system examines its correct and buggy rules and decides which ones could execute according to the

IF clauses satisfied by the current problem state, i.e., by the the current goal and the current contents of working memory. This set of eligible rules is termed the conflict set and will always contain one or more correct rules and will generally contain one or more buggy rules. Only one of these eligible rules in the conflict set can actually be triggered, and the next step is to decide which one best describes the action taken by the student. LISPITS does this by comparing the symbol the student types to the symbol that would be generated by each of the rules in the conflict set.

If the student's input matches the symbol generated by a correct rule, then that rule is triggered. As a result, the student's symbol is added to the permanent problem solution, one or more new goals may be set, and information may be added to working memory. At that point, a new goal is activated (drawn from the list of unsatisfied goals) and the cycle is repeated. If the student's input matches the symbol that would be generated by one of the buggy rules, or does not match any rule in the conflict set, then no rule is triggered. Instead, the tutor provides a feedback message and gives the student another chance. If the student then types a correct symbol, a correct rule is then triggered as described above. If student repeatedly types symbols that either do not match any rule in the conflict set or match the same buggy rule, then the tutor intervenes to prevent floundering. This is done by triggering one of the correct rules in the conflict set and providing an explanation of why that rule is appropriate to accomplish the current goal. ~~starting from the current problem state~~. AI-check this

This cycle of activating goals, generating conflict sets, and matching the student's inputs continues until the exercise is completed.

WHY MODEL TRACING?

As suggested earlier, given our goal of providing assistance to students as they work on programming exercises, it is necessary to provide the tutoring program with either a set of exercise solutions or a mechanism for generating solutions. In model tracing, we provide LISPITS with a mechanism for generating solutions; more specifically, we provide a mechanism that is intended to model the student's thought processes in completing the exercises.

As mentioned earlier, the chief reason that model tracing is useful in tutoring is that there can be different underlying reasons for employing a given LISP function in different contexts. For example, the function `setq` always has the effect of assigning a value to a variable, but there are different underlying goals that can be achieved by performing this action. One distinction concerns the type of variable being assigned a value: we could be assigning a value to a global variable so that it is available to any function we define, or we could be updating a local variable that is only needed within a specific function. The reason for the assignment can also vary: we could be storing intermediate results within a loop, or we could be storing the result of a complex operation to avoid having to repeat it.

In the student model different production rules govern the coding of a given symbol (such as `setq`) for different purposes. As a practical matter, this allows LISPITS to generate contextually appropriate explanations, since we can associate a contextually appropriate explanation with each production. When the student model is running through an exercise, contextually appropriate rules will be available at each goal and, if an explanation is required, a contextually appropriate one is automatically produced.

The student model is intended to be more than a convenient formalism for generating explanations, however. It is intended to be a psychological model of the knowledge the student is acquiring. Indeed, examination of the log files of the students' interactions with LISPITS supports the hypothesis that different rules generate the same symbol in different contexts (Anderson, in press; Conrad & Anderson, in press). As described in the following section, LISPITS maintains a model of how well each student understands how to program in LISP. To the extent that different rules govern the coding of a single symbol in different contexts, it is important to represent and track the rules separately to maintain an accurate model of the student's knowledge. Again, the student model allows us to do this. It provides us with a list of the rules to track and, as the student model is running, ensures that the student's responses at each goal are associated with the appropriate production rules.

In addition, student modelling also allows us to automates the initial stage of data analysis. When students interact with LISPITS it maintains a timed-stamped log file of the student's inputs and ~~the~~ LISPITS' responses. Since LISPITS is modelling the student's behavior, it can conveniently identify and store the underlying rules which theoretically give rise to each of the student's responses.

Finally, there is at least one beneficial side effect of model tracing. Development of the student model is a useful aid in curriculum development, since it forces us to think in specific terms about what a student needs to learn.

MODEL TRACING AND MASTERY LEARNING

When the tutor was first used in teaching LISP to programming novices, it became clear in posttests that an appreciable minority of the students were not learning LISP well enough. In a rare case or two, it turned out the student was just asking the tutor for an explanation at each step and having the tutor generate the code. In most cases, however, even though students were doing most of the work themselves in getting through the exercises, they had trouble doing exercises on their own.

The current configuration of the course, as described in an earlier section, in large part reflects this observation. We instituted a handful of non-tutor exercises and frequent quizzes to provide the students as well as the instructor a measure of the student's understanding. Moreover, quiz results are used to implement a mastery learning paradigm: students cannot proceed to later lessons until they have passed each quiz.

In addition to these revisions in the course structure, we introduced a degree of mastery learning within LISPITS itself. We modified LISPITS so that it monitors the student's performance in completing exercises and tailors the exercise sequence to accommodate the student's rate of progress. Specifically, the tutor maintains a model of the student's knowledge state in the course of proceeding through the twelve lessons. The model of each student is essentially an overlay model that consists of a list of the production rules in the ideal student model. For each rule LISPITS maintains an estimate of the probability that the student has learned the rule. By examining log files of students who have used LISPITS to see what percentage of them use a given rule correctly the first time, we can compute an estimate of the average probability that students will learn that rule just from reading the text. These estimates are initially assigned to the production rules when an overlay model is first created for an individual student. Then the probability estimate for a rule is updated when the student has the opportunity to employ the rule. The new estimate of whether the student has learned the rule depends on whether the student responds correctly or makes an error and is computed according to a simple two-state learning theory[1] and Bayesian statistics.

---

[1]A two-state model assumes that a production rule can be in exactly one of two states in memory, either the student has learned it or not. ACT* is not a two-state model, and there is ample learning data that support either multi-state or

Of course, once these probabilities are computed, it is a separate issue how to use them. As previously mentioned, we have opted for a mastery-based system of learning. We have adopted a probability value of 0.95 as the criterion for concluding that the student knows a rule. Each time a new set of rules is introduced in the text and the student is ready to do exercises, the tutor presents a set of required exercises that draw on the new rules. After the student has completed the required exercises, the tutor reviews all the rules in the student's knowledge model to see if any fall below the 0.95 criterion. If so, the tutor selects the best exercise to practice the rule(s). An optimal practice exercise is defined as one in which the student knows 90% of the rules required to write the code but needs to learn the remaining 10% of the rules, and LISPITS selects an exercise that comes closest to this ratio. Only after the student has brought all rules above the 95% criterion does the tutor allow the student to move on to the next section in the text.

We refer to this process of monitoring and remediating the student's knowledge as **knowledge tracing**. As a final fail-safe measure in knowledge tracing, if the student completes many exercises in a lesson (operationally defined as twice the number of required exercises) without completing the lesson, the tutor requires the student to see the instructor before proceeding. This allows us to deal with the rare student who has fundamental misconceptions. A later section describes assessment of this knowledge tracing.

SKILL ACQUISITION RESEARCH WITH LISPITS

At this point we would like to describe some research we have conducted with LISPITS. An earlier section provided a comparison of students using LISPITS with students who used only a standard LISP programming environment to do exercises. The results do indicate that LISPITS lets students learn at least as much LISP in a considerably shorter time. The following studies address more detailed questions about what features of LISPITS are useful, and what ~~what~~ might be eliminated or changed.

---

continuous-strength memory models. However, the estimates of all such models are highly correlated and the data generated by the tutor is not sufficiently powerful to distinguish among them. Thus, we have opted for the computationally simplest model.

The first study examines whether practice in evaluating sample programs transfers to skill in writing programs. The outcome of this study has practical pedagogical implications and is of theoretical interest, since it investigates the possible transfer between two types of procedural skill based on common declarative knowledge structures. The remaining studies examine two aspects of feedback: content and control. In the first two of these studies, we examine the effectiveness of the explanations LISPITS by comparing the standard tutor with provid... a version that notifies students of errors but does not try to explain them. In the final section, we address some practical issues that arise in implementing the tutor and describe some preliminary results obtained with a version of LISPITS in which the student controls the timing of feedback.

DOES PRACTICE IN CODE EVALUATION IMPROVE CODE GENERATION?

The first pedagogical issue we addressed with the tutor concerns the assumption that time should be concentrated on practice in writing programs rather than in other activities. In particular, we examined whether better performance in *writing* programs can be produced by practice in *evaluating* programs (i.e., executing them in the mind to predict their results). Textbooks and instructors ordinarily assume that an effective way to introduce new programming constructs is to show sample code and to explain how the code is executed, i.e., to evaluate the code. An interesting question is whether such practice in evaluating code contributes substantially to skill in generating programs. ACT* would suggest that practicing evaluation would provide relatively little benefit in learning to generate code. At least three studies have confirmed this prediction over two restricted areas of LISP, basic LISP functions (McKendree & Anderson, 1987; Kessler, 1988) and recursive functions (Pirolli, 1985).

We used the tutor to investigate the issue systematically across the eleven lessons that existed in the fall of 1985. In this experiment, all students went through the tutor and took a paper and pencil test after the sixth lesson and after the final lesson. Half the students received practice in evaluating sample functions in addition to performing the tutor's programming exercises. Specifically, before starting each section of code generation exercises, the tutor presented students one or two code evaluation exercises. In each evaluation exercise the students were presented with a function definition and a sample call to the function. By pressing special keys the student indicated how LISP would sequentially evaluate the function definition in the course of executing the sample call. To solve an evaluation exercise, students indicated not only the order in which the expressions were evaluated, but also the result of each evaluation step. Students continued through this evaluation process until a final result was obtained. As the students worked the tutor provided immediate feedback on errors concerning both evaluation sequence and results.

Table 2 presents the results of this experiment. This table contains one measure of performance from the tutor exercises and one posttest performance measure. The measure of performance in completing the LISPITS exercises is average production execution time for goals at which the student's first response is correct. The operational definition of execution time is the average time it takes students to respond to each of the angle-bracket goal symbols in defining functions. Each such step the student takes corresponds to executing a production in the student model, so this measure is theoretically a measure of the time for a student to make an inference corresponding to the execution of one production. As can be seen, evaluation practice had no impact on coding time for the tutor exercises. Average production execution time is essentially the same for the two conditions.

	With Practice on Evaluation	With No Practice on Evaluation
LISPITS Programming Exercises		
Average Time for One Production Execution	12.5 sec	12.0 sec
Percent Correct on Posttest		
Coding	60%	62%
Evaluation	66%	56%
Debugging	61%	65%

Table 2: Transfer from Code Evaluation to Programming

The posttest score is percent correct across the midterm and cumulative final exam. As indicated in the table there were three types of exercises on the exams: coding exercises in which the student wrote function definitions, evaluation exercises similar to the ones administered by the evaluation tutor, and debugging exercises, in which students were asked to repair buggy programs. As can be seen, students in the two groups obtained virtually the same score on the coding exercises; if anything the students who used the evaluation tutor scored slightly lower in these exercises. Students who practiced evaluation exercises did perform slightly better on the posttest evaluation exercises, 66% vs 56%. Thus, the data suggests that students who practiced evaluating code were acquiring some knowledge that was not acquired by the students who did not practice evaluation. However, there is no indication that this knowledge transferred to writing code, nor to debugging code. While there is common declarative knowledge underlying coding, evaluation and debugging, this result supports the assumption that employing that knowledge in the pursuit of one type of goal will have minimal impact on the pursuit of different goals.

EFFECTS OF EXPLANATORY FEEDBACK AND REMEDIAL PRACTICE

In a pair of studies conducted over the past year we evaluated the effectiveness of the tutor's feedback and the tutor's ability to provide remedical practice based on knowledge tracing. In the case of feedback, we were interested in whether LISPITS' explanations are helpful, or whether students might benefit from generating their own explanations when the tutor has identified an error. To evaluate the effectiveness of explanations, we simply shut them off for half the students. The tutor continued to interrupt as usual if an error occurred, and continued to provide correct answers under the standard rules when the student appeared to be floundering. However, the tutor no longer attempted to explain the nature of errors nor to explain correct answers it provided. That is, if the student made an error, the tutor interrupted immediately but just responded, "That doesn't seem to be correct." When the tutor provided a correct answer, it simply told the student what code to write, but not why, e.g., "You need to call the function APPEND here."

To assess the effectiveness of knowledge tracing, and its individually tailored remediation practice, we shut it off for half the subjects. These students completed just the required exercises in each section. While the tutor continued to update estimates of the students' understanding, it did not provide any additional practice exercises with the aim of remediating weaknesses in the student's knowledge. The manipulation of explanations and remediation were crossed in these studies, yielding a two-by-two between subjects design. By crossing the manipulations we can observe whether the presence of either explanations or remediation alone compensates for the absence of the other.

We ran two replications of this experiment: Study 1 in the spring of 1987 covered Lessons 1-3, 5, and 6; Study 2 in the fall of 1987 covered all 12 lessons. Table 3 presents the results of both experiments.

Table 3 presents, for Study 1, two measures of performance in completing the LISPITS programming exercises: average production execution time for goals at which the student's first response is correct and average number of errors in responding at each goal. These LISPITS performance measures are derived from the required exercises, which are common to all groups. The posttest score is percent correct on a cumulative final exam. As can be seen, there is virtually no effect of either manipulation on posttest performance. An analysis of variance confirmed that the two main effects and interaction were non-significant.

Exercises: Production Firing Time (sec) and Errors per Goal

		Study 1 Explanations: Yes	Study 1 Explanations: No	Study 2 Explanations: Yes	Study 2 Explanations: No
Remedial Practice:	Yes	14.9 sec 0.23 errors	13.6 sec 0.28 errors	10.4 sec 0.13 errors	11.1 sec 0.18 errors
	No	15.5 sec 0.15 errors	22.8 sec 0.38 errors	12.4 sec 0.18 errors	10.6 sec 0.26 errors

Posttest Performance (Percent Correct)

		Study 1 Explanations: Yes	Study 1 Explanations: No	Study 2 Explanations: Yes	Study 2 Explanations: No
Remedial Practice:	Yes	80%	79%	95% (quizzes) 87% (exam)	94% (quizes) 82% (exam)
	No	85%	85%	86% (quizzes) 65% (exam)	88% (quizes) 79% (exam)

Table 3: Performance based on combinations of Error Explanation and use of Knowledge Tracing with its Remedial Practice.

The manipulations of explanations and remediation did have measurable effects on students' performance with LISPITS, however. Averaging over subjects who did and did not receive remediation, subjects who received explanations made fewer errors per goal than subjects in the other group, 0.19 errors/goal vs. 0.33 errors/goal. This effect of explanations was significant in an analysis of variance $F(1,21) = 6.58$, $p < .05$. Neither the main effect of remediation on this measure of accuracy nor the interaction of explanations and remediation were significant. Thus, students who received explanations from LISPITS made fewer errors in completing the tutor exercises.

Knowledge tracing remediation did influence production execution time, however. Students who received remedial exercises responded more quickly at each goal, 14.3 seconds/production vs. 19.2 seconds/production. This difference was significant in an analysis of variance, $F(1,21) = 7.97$, $p < .05$. Production execution times were also faster for subjects who received explanations, 15.2 seconds/production vs. 18.2 seconds/production, although this effect was only marginally significant, $F(1,21) = 3.11$, $p < .10$. Finally, the interaction of explanations and remediation was significant in this analysis, $F(1,21) = 6.16$, $p < .05$. As can be seen students who received neither explanations nor remediation were particularly slow in responding.

The pattern of effects on performance with the tutor is understandable. First, if the explanations are useful, then we would expect students to make fewer second and third errors at a goal after having received an explanation on the first error. In that case, we would expect fewer errors per goal as observed. Second, students in the remediation condition performed more total exercises with LISPITS than did students in the non-remediation condition. This fact is likely to influence response speed, even though the measure is based just on the required exercises common to all groups. Once the first few required exercises have been encountered, students in the remediation condition will begin doing remedial exercises with the result that, when any subsequent required problem is encountered, students in the remedial groups will have performed more prior exercises. Since performance speed generally decreases with time on task, it follows that students in the remedial condition would on average perform the required exercises more quickly.

Two other results concerning the measures of performance with LISPITS are perhaps less predictable. Consider first the interaction of explanations and remediation on production execution time. Apparently the presence of explanations roughly compensates for the extra practice received in the remediation condition. Reduced practice only leads to reduced speed of production execution when explanations are shut off. Perhaps the most surprising result, however, is that while the explanation and remediation manipulations affected tutor performance, there were no long-term effects on posttest performance. Students' performance on the posttest was independent of the version of LISPITS used.

The results for Study 2, also presented in Table 3, are slightly different. There are again two measures of performance with LISPITS: production execution time for goals at which the student's first response is correct and average number of errors per goal. (These analyses were performed just on the five lessons that also appeared in the Study 1). The table also contains two posttest performance measures. One is percent correct across a midterm exam and a cumulative final exam, both paper-and-pencil tests. The other measure is percent correct across six on-line quizzes presented by the tutor.

As can be seen in Table 3, the manipulation of explanations had little impact on either posttest score. In this experiment, unlike the previous one, there was an effect of knowledge tracing on posttest measures. When knowledge tracing was shut off, students scored lower on both the quizzes, 87% vs. 95%, and on the tests, 72% vs. 85%. The effect of remediation on quiz scores was significant, $F(1,29) = 6.48$, $p < .05$, while the effect on final exam scores was marginally significant, $F(1,29) = 2.97$, $p < .10$. The main effect of explanations was non-significant in the analyses of both the quizzes and final exams, although the interaction of explanations and remediation was marginally significant in the final exam analysis, $F(1,29) = 3.18$, $p < .10$. Examination of Table 3 indicates that there may be less effect of the remediation manipulation in the No-Explanation condition.

The two manipulations had weaker effects on the measures of performance with LISPITS than in the previous experiment. The explanation manipulation had no reliable effect on production execution speed and only a marginal effect on the number of errors. Subjects who received explanations made fewer errors per goal than those who did not receive explanations, 0.16 vs. 0.22, $F(1,29) = 3.08$, $p < .10$. The manipulation of remediation also had no reliable effect on production execution speed, but had a marginal effect on errors per goal. Students made fewer errors in the remediation condition, 0.16/goal vs. 0.22/goal, $F(1,29) = 2.99$, $p < .10$. Finally, the interaction of the two manipulations was non-significant in both analyses.

In summary, there is no indication in either study that explanations are having an effect on degree of learning as measured by posttest scores. There is evidence that knowledge tracing has an impact on posttest scores in the second experiment; students seem to learn the material better when knowledge tracing is turned on. This effect is not obtained in the first experiment, but this inconsistency is not entirely surprising. So far, no research limited to the first part of the LISPITS curriculum has led to posttest performance differences. In particular, the first assessment study of LISPITS described earlier, which covered just the first eight lessons, led to total time differences but no posttest differences among students in the three conditions. In the second assessment study, differences in posttest performance did not appear on the midterm, but only appeared on the cumulative final exam.

Concerning measures of performance with LISPITS, again the pattern is not entirely consistent across the two studies. The explanation manipulation did have an impact on accuracy in both studies, although the effect is marginal in the second one. Knowledge tracing chiefly affects performance time in the first study but affects accuracy in the second study.

The absence of an explanation effect on the posttest and the marginal effect on tutor performance in the second study is interesting. It may suggest that most of the effectiveness of LISPITS is obtained just by interrupting students immediately after an error and simply providing the correct next step when students are floundering. The tutor's efforts to provide explanations in those situations may not be helpful. One possible explanation of the small effect of shutting off explanations is that students simply aren't reading the explanations when they are presented. One way to investigate this issue is to ask: If a student makes a mistake that LISPITS can diagnose and explain to the student, what is the probability that the student will get the answer correct on the next attempt? If the explanations were effective, we would expect that students who received the explanations would be less likely to make another mistake at the same goal. Table 4 provides evidence on this point for Lessons 1-3, 5, and 6 in the Study 2.

Probability of Immediate Error Correction (LISPITS Exercise in Study 2)

		Explanations: Yes	Explanations: No
Remedial Practice	**Yes**	0.88	0.65
	No	0.70	0.62

Table 4: Effect of Explanation on probability of immediate correction of an error.

This table displays the probability that students take a correct step at a goal immediately after making a diagnosed error at the goal (i.e., a mistake for which the tutor could provide an explanation, but does so for only half the students). As can be seen, students who received explanations are more likely to make a correct response immediately after an error. Thus, there is some internal evidence that the students are reading the explanations. However, there is little evidence of a long-term effect of this feedback.

MODEL TRACING AND STUDENT-CONTROLLED FEEDBACK

As described earlier, LISPITS constrains students to type in their code in a strict top-down, left-to-right order and provides error feedback on a symbol-by-symbol basis. The first characteristic may appear to be a superficial property of the program's interface, and the second characteristic may appear to be purely a tutoring principle. However, these characteristics are closely tied to two characteristics of the student model. The input-order constraints are tied to assumptions built into the production system concerning active goals. While there is generally more than one unsatisfied goal pending at any point during problem solving, only one goal can be active at each step in problem solving.[2] Additional assumptions concerning which of the pending goals will be selected at each step give rise to strict top-down, left-to-right behavior in the student model, which in turn governs the tutor's interface (see Corbett, Anderson, & Patterson, 1988, for more details). Symbol-by-symbol feedback is tied to the grain-size of the productions in the student model. The student's knowledge of LISP is modelled at the finest grain-size that has meaning in LISP—roughly the level of individual symbols. Any finer grain-size would begin to model typing skill. The immediate feedback principle specifies that feedback should be presented right after an erroneous production is triggered, but it is the grain-size

[2]This theoretical assumption has been relaxed since work on the tutor was initiated. A new production system, in which all pending goals are active in each cycle, reflects this modification in the theory.

of the tutor which dictates the symbol-by-symbol feedback presented by the tutor.

Given this relationship between the student model and the tutor's behavior, modifications in coding order or feedback units would require modifications in the student model and not just in the interface and tutorial components of the program. This is an important point since the student model is fairly large, consisting of approximately 1200 rules. Fortunately, model tracing can be implemented by means of a technique called problem compilation that facilitates such modifications. This technique is described in the next section.

MODEL TRACING AND PROBLEM COMPILATION

We originally adopted problem compilation as a means to speed up the tutor. Running a production system, such as the student model, imposes high computational demands on the computer because of the large amount of pattern matching required in determining the conflict set at each goal. As a result, the tutor in its initial form had trouble keeping up with students as they worked through an exercise. Fortunately, model tracing does not actually require execution of the production system student model on-line with the tutor. The reason is that the dynamic behavior of the student model in writing a program can be captured in a data structure which contains all the information required for tutoring. For example, consider a function called `insert-two` that takes three arguments, the third of which must be a list, and inserts the first two arguments at the beginning of the third. For example, given the arguments `'x`, `'(y z)`, and `'(a b c d)`, `insert-two` would return `(x (y z) a b c d)` Two possible definitions of the function are:

```
(defun insert-two (item1 item2 alist)
   (cons item1 (cons item2 alist)))
(defun insert-two (item1 item2 alist)
   (append (list item1 item2) alist))
```

Figure 1 displays a tree structure that represents the possible steps the student model would take in coding this function. Each node (ellipse) in this tree represents a goal that is set in writing the code. The link descending from each node represents a production execution and ends at the symbol the production generates, e.g., `defun` is the symbol generated from the goal "code function." Links descending from the code symbol represent goals set by the production. The top goal in the tree represents the initial goal of defining the function `insert-two`. The link leading from that goal represents a production that generates the symbol `defun` and sets three goals: to code the function name, the parameter list, and the body of the function. All three of these goals must be satisfied. The first two of these goals are satisfied by productions that code the name `insert-two` and parameter list `(item1 item2 alist)` respectively and do not set any subgoals. As indicated by the arc labeled "OR", the third goal, coding the function body, can be satisfied by either of two productions. One of these productions generates the symbol `cons` and sets two subgoals. The other production generates the code symbol `append` and also sets two subgoals. Each of these subgoals is satisfied by additional productions and so on.

In essence this data structure contains the information we need to know in order to tutor: the goal that is set at each step in a solution and the production(s) that satisfy the goal (including the buggy productions not depicted in the figure). We can provide LISPITS a data structure analogous to the hierarchical goal tree in Figure 1 by running the student model through the exercise ahead of time, allowing it to pursue every path it can find that leads to a correct solution, and saving the relevant information. Thus the high computational demands of recognizing appropriate productions at each step in problem solving are met in advance and the results of the process are stored in a structure that can be processed quickly. (See Sleeman, 1983, for a description of a similar process.)

While problem compilation was initiated to speed LISPITS' performance, it also has a beneficial side effect: it allows us to conveniently modify the tutor's behavior without actually rewriting the production rules. Note that the result of the problem compilation is a complete specification of the steps at the smallest possible grain size of every acceptable solution to an exercise. Once this data structure exists, it is relatively easy to write a program that can track the students' behavior through the tree, even if they deviate from a top-down left-to-right coding order. We can also modify the functional grain-size of productions. Thus, we can modify the assumptions of the student model, without actually modifying the original production system, writing new rules, and regenerating solutions.

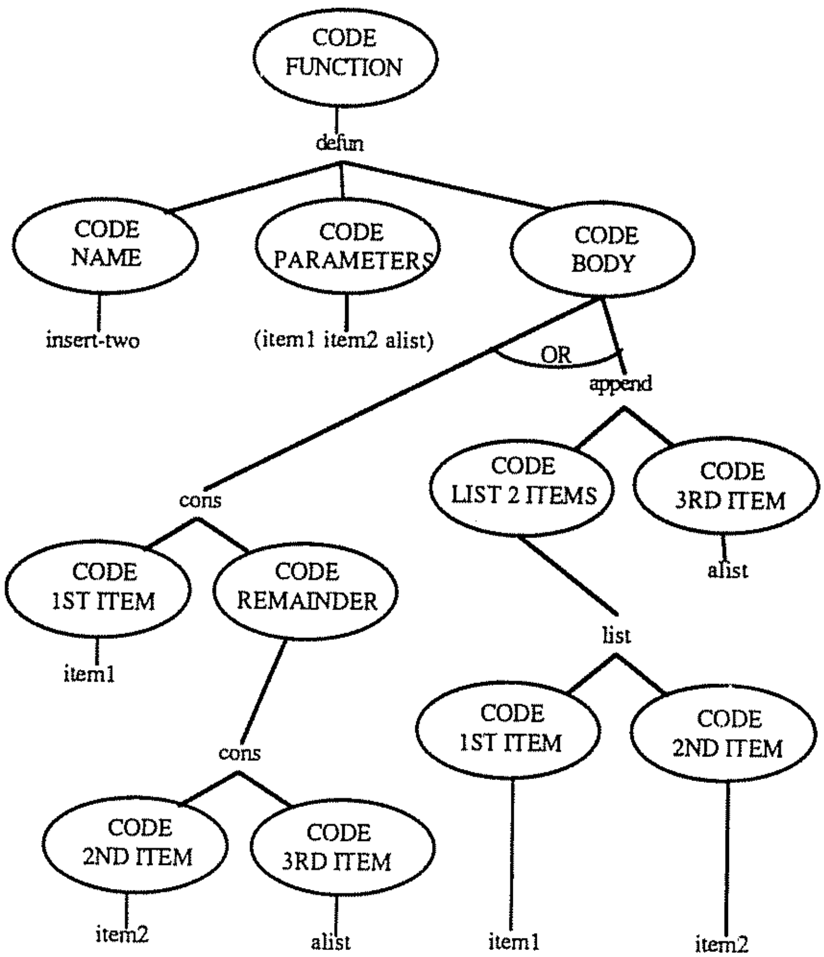

Figure 1: The hierarchical goal structure underlying the definition of `insert-two`. Each elliptical node represents a goal. The link descending from each goal represents a production execution. Each production link leads to ~~either(a)~~ the code symbol it generates, ~~or (b) to~~ additional goals that are set by the production.

then to any

4/17/4, to author

MODIFYING THE TUTORIAL INTERACTION: ENHANCED STUDENT CONTROL

It is not coincidental that we have discussed problem compilation in the context of input order and feedback. Perhaps the most frequent complaint we hear from students concerns symbol-by-symbol feedback. Another, less frequent complaint, concerns the input-order constraint. As a result, we have taken advantage of problem compilation to develop a version of LISPITS that gives the student complete control over both the order in which code is generated and the points at which feedback is provided (Corbett et al., 1988).

In this student-controlled version of LISPITS, the student is provided with a true editor for entering code. More specifically, it is a structured editor that provides function templates and balancing right parentheses much like the interface in the immediate-feedback version. The editor ensures that the student generates legal (syntactically correct) code. However is does not (and cannot) ensure that the student's program satisfies the exercise description. Thus, students can generate code in any order, including right-to-left and ~~and~~ bottom-up. Unlike the immediate-feedback version, students can also modify parts of the code they have already completed. While the student is working with the editor, the model tracing/tutoring component of the program is disengaged and never interrupts. However, the student is provided with three commands to request tutoring. One of these commands asks the tutor to check over the entire solution as it currently stands. When the student selects this command, the tutor checks the code in the same top-down left-to-right order as it ordinarily would in the immediate feedback version. If the code is correct and complete, the tutor recognizes that the student is done. If the existing code is correct but is not a complete solution, the tutor tells the student that the code is correct so far and returns editing control to the student. However, if an error is detected, the tutor stops and gives the same error feedback as in the immediate-feedback version and deletes the incorrect symbol from the code. Any code which follows the error remains unanalyzed and is moved from the code window to a separate buffer on the screen to emphasize to the student that that code has not been checked. (This code can readily be transferred back to the code window as the student sees fit.) After performing these actions, the program returns editing control to the student.

The two other tutoring commands allow the student to request either a goal hint or the correct code symbol at any goal, just as in the immediate feedback version. If one of these commands is selected, the tutor begins by checking over

4/17/4, to author

any code in the student's solution that precedes the goal in question. If an error is found, the program provides feedback on that error (to avoid discussing the goal specified by the student in an erroneous context). Otherwise, the tutor provides the same information on the requested goal as it would in the immediate feedback version.

TESTING STUDENT-CONTROLLED FEEDBACK

Our study with the student-controlled tutor covered the first two lessons. Half the students completed the two lessons with the new student-controlled version, while the other half used the standard immediate-feedback version. Knowledge tracing was shut off, so all students completed the the same set of exercises across the two lessons. After completing the lessons, the students took a cumulative quiz.

Assessment Measures

Two measures of student performance are of interest for pedagogical as well as theoretical reasons: posttest performance and total time to complete the exercises. This control manipulation had no effect on posttest scores; both groups scored 83% correct on the quiz. However, the manipulation did affect the time required to complete the exercises. Students using the immediate feedback version required an average of 2.9 minutes to complete each exercise, while students using the student-controlled version required an average of 4.3 minutes. This difference is significant, $t(30) = 3.9$, $p < .001$.

Processing Measures

The total-time difference in completing the exercises may in part reflect the fact that the editor in the student-controlled condition is intrinsically more difficult to use than the constrained interface in the immediate feedback condition. In examining the log files, there are indeed occasions in which the student seemed to struggle with the interface. However, examination of the log files also suggests a second reason that students may be taking longer in the student-controlled condition. Specifically, in the student-controlled condition the tutor is detecting an average of 0.83 errors per exercise per student, while in the immediate-feedback condition the tutor is detecting 1.15 errors. This difference is reliable ($t(30) = 2.48$, $p < .05$) and suggests that, in the student-controlled condition, students are detecting and correcting their own errors, which also may contribute to total processing time.

4/17/4, to author

Error Detection and Code Revision

There is an alternative explanation of this difference in error detection rate: subjects could be more cautious and simply make fewer errors in the student-controlled condition. To obtain firmer data on this point, we performed a detailed analysis of the editor actions performed in Lesson 2 by subjects in the student-controlled condition. The results of this analysis are summarized in Table 5

Total Tutor-Detected Errors:	165
Total Student-Initiated Revisions to Erroneous Code:	86
Changes of Erroneous Code	
To Correct Code:	49
To other Erroneous Code:	37
Changes of Correct Code	
To Other Correct Code:	9
To Erroneous Code:	12

Table 5: Student Controlled Feedback: Error Detection and Code Revision

This table shows the total number of errors detected by the tutor across the sixteen subjects in the student-controlled condition for the seven exercises in Lesson 2. It also shows the total number of code revisions made by these sixteen subjects across those seven exercises. As indicated, students detected and revised a total of 86 errors, while the tutor detected 165 Of the errors detected and revised by students, 57% are corrected (49 out of 86) while the remaining 43% are changed to a different error. In addition, students went back and changed correct symbols in the code 21 times. Forty-three percent of these changes resulted in alternative correct code, while the remaining 57% of the modifications introduced errors. Thus, students do rethink and revise code when given an opportunity that is not afforded them by the immediate feedback tutor. However, only about half (44%) of the revisions are constructive, that is, move toward a correct solution. The remaining non-constructive modifications are inefficient in the sense that they take time without yielding any progress to a successful solution. We don't know if there is a cognitive benefit of such non-constructive changes, although the posttest results don't suggest that subjects in the student-controlled condition learned the material more effectively.

A final code-revision issue we addressed concerns the relative position in the code of the revisions students make. Specifically, when a revision occurred, we looked at the structural relation between the goal the student had been working on prior to the revision and the goal at which the revision was made. Gray and Anderson (in press) investigated the code revisions students made in writing fairly complex iterative LISP functions and found that students are likely to make revisions only at the goal they are working on (or have just completed) or at a superordinate goal in the tree structure. To exemplify this result, consider the function `insert-two`. Suppose a student has just satisfied the goal "CODE 2ND ITEM" at the lower left in Figure 1 by typing the symbol `item2` in the body of the function. At this point the code window would look like this:

```
(defun insert-two (item1 item2 alist)
   (cons item1 (cons item2 <list2>)))

(the symbol <list2> denotes the pending goal of
coding some list as the second argument of cons).
```

[The text should remain with these lines – they are not part of the example)

page 37 →

Gray and Anderson's results suggest that the symbol the student is most likely to change at this point is `item2`, which has just been typed. The only other symbols the student would be likely to immediately change are `defun` and the two instances of `cons`, since these symbols were generated at the goals CODE FUNCTION, CODE BODY, and CODE REMAINDER, which are superordinate to CODE 2ND ITEM in the tree diagram. In Gray and Anderson's analysis the student is much less likely to revise the function name, parameter list or reference to `item1` in the body of the function since these symbols were generated at goals that are not superordinates of CODE 2ND ITEM.

Structural Relation of Revised Goal to Current Goal	Number of Revisions	Probability that Revision Corrects Error
Same	61	0.51
Superordinate	31	0.42
Other	15	0.33

Table 6: Position of Code Revisions with Student Controlled Feedback

An inspection of the Lesson 2 log files in this study confirmed this prediction, as shown in Table 6. Of the 107 revisions students made, 57% were at the current goal, 29% were at superordinate goals and only 14% were at other goals. Analysis also indicated that the probability a revision is productive (i.e., corrects an error) varies with the relative position of the revision. Of the revisions at the current goal, 51% corrected an error, whereas only 42% of the revisions at superordinate goals and 33% of revisions at other goals corrected errors.

Immediate Feedback and Input Order

Examination of the log files in the student-controlled condition also allows us to evaluate students complaints concerning input order and immediate feedback. For example, concerning feedback, students asked LISPITS to check over their solutions 661 times across the two lessons. Only 15 of these requests came when the student had a partial solution In the other 646 cases the student had complete, though not necessarily correct, code. Students also asked for goal hints or explanations a total of 72 times. Necessarily the student's code is incomplete when such requests are made but, even when these 72 requests are included, students are asking for help with partial code only 12% of the time. Thus, students' behavior in this experiment is consistent with previous complaints about step-by-step immediate feedback. When students are in control they not only do not request immediate feedback, they seldom ask for feedback until they are done. Moreover, when their code is not channeled into a correct solution path by the tutor, students do take advantage of the opportunity to rethink and revise their answers. The pattern of results concerning feedback on partial solutions may change in later lessons as exercises become more complex. However, the present results indicate that students would be happy with a tutor that only provides feedback on complete code, as in the case of PROUST (Johnson & Soloway, 1985; Sack & Soloway, this volume).

Finally, we examined the editor interactions in Lesson 2 to evaluate student complaints concerning input order. Across all students and exercises in Lesson 2 there were about 400 goals that could have been solved in a bottom-up rather than top-down order. (That is, there were about 400 non-terminal goals which entailed subgoals.) Among those 400 opportunities, there were only 5 occasions in which students coded bottom-up rather than top-down. In addition, there were about 450 occasions in which students could have generated code in a right-to-left fashion, rather than left-to-right. On only one occasion did a student take advantage of this opportunity. Thus, despite student complaints concerning coding order (which actually occur frequently in the context of Lesson 2 exercises), students took little advantage of the opportunity to deviate from top-down and left-to-right coding.

CONCLUSION

The goal of the LISPITS project is to create an environment in which students can practice LISP programming productively. More specifically, our goal has been to monitor students' performance in programming exercises, to recognize errors as they occur, and to provide feedback on these errors. The project has adopted a student-modelling approach to this task that is based on a cognitive model of (1) the programming knowledge we want the students to acquire and (2) misconceptions that may arise. By applying this model of ideal and buggy programming knowledge to the exercises that the students perform, we can generate a dynamic model of correct and incorrect steps to solve the exercises and we can use this dynamic model to assess the students' behavior and provide feedback.

This general approach has proven quite successful. In the assessment research to date, students using LISPITS complete the coding exercises substantially more rapidly than those working on their own, although not as fast as students working with a human tutor. Furthermore, students using LISPITS perform as well or better on posttests.

Current research focuses on the control and timing of feedback and the effectiveness of explanations. Much of LISPITS' effectiveness apparently can be attributed to the immediate feedback it provides on errors. When students are given control of feedback timing (and input order) exercise completion times increase by 50%. Moreover, research on LISPITS' feedback suggests that the explanations it provides have no impact on posttest scores and relatively little consistent impact on performance in completing the exercises. This suggests that it is the immediate noting of errors and the immediate provision of correct answers when the student shows signs of floundering that are critical in LISPITS' success. Nevertheless, students' most frequent complaint concerns immediate feedback. As a result, much of our current research involves resolving this conflict between the efficiency that is gained with immediate feedback and the students' desire for greater control.

Simultaneously we are examining other factors that govern the effectiveness of explanations. Some of these factors are internal to LISPITS, e.g., the tutor's ability to determine what instruction the student needs and how to present it clearly. Other important factors, however, may be external to LISPITS. For example, the components of the overall curriculum, including the quality of the text and non-tutor programming experience, may have a strong impact on the usefulness of explanations the tutor provides. Finally, the student's perception of control over the tutoring interaction may also have a strong bearing on the effectiveness of explanations the tutor attempts to provide.

NOTE

This research was supported by the Office of Naval Research contract N00014-87-K-0103.

REFERENCES

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. R. (in press). Analysis of student performance with the LISP tutor. In N. Fredericksen, R. Glaser, A. Lesgold & M. Shafto (Eds.), *Diagnostic monitoring of skill and knowledge acquisition*. Hillsdale, NJ: Erlbaum.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (in press). Cognitive modelling and intelligent tutoring. *Artificial Intelligence*.

Anderson, J. R., Boyle, C. F., Farrell, R., & Reiser, B. J. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), *Modelling cognition*. New York, NY: Wiley.

Anderson, J. R., Boyle, C. F., & Reiser, B. J. (1985). Intelligent tutoring systems. *Science, 228*, 456-462.

Anderson, J. R., Boyle, C. F., & Yost, G. (1985). The geometry tutor. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 1-7). Los Angeles, CA.

Anderson, J. R., Corbett, A. T., & Reiser, B. J. (1987). *Essential LISP*. Reading, MA: Addison-Wesley.

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

Anderson, J. R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction, 22*, 403-423.

Anderson, J. R., & Reiser, B. J. (1985, April). The LISP Tutor. *Byte, 10* (4), 159-175.

Conrad, F. G, & Anderson, J. R. (1988). The process of learning LISP. *Proceedings of the 10th Annual Conference of the Cognitive Science Society* (pp. 454-460). Montreal, PQ.

Corbett, A. T., Anderson, J. R., & Patterson, E. G. (1988). Problem compilation and tutoring flexibility in the LISP tutor. *Proceedings of ITS-88 The International Conference on Intelligent Tutoring Systems* (pp. 423-429). Montreal, PQ.

Gray, W., & Anderson, J. R. (in press). Change episodes in coding: When and how do programmers change their code. In G. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop*. Norwood, NJ: Ablex.

4/17/4, to author

Johnson, W. L., & Soloway, E. (1987). PROUST: An automatic debugger for Pascal programs. In G. Kearsley (Ed.), *Artificial intelligence and instruction: Applications and methods.* Reading, MA: Addison-Wesley. (Originally published in 1985 in *Byte, 10*, (4), 179-190)

Kessler, C. (1988). *Transfer of programming skills in novice LISP learners.* Ph.D. Dissertation, Carnegie-Mellon University.

Lewis, M. W., & Anderson, J. R. (1985) Discrimination of operator schemata in problem solving: Learning from examples. *Cognitive Psychology, 17*, 26-65.

Lewis, M. W., Milson, R., & Anderson, J. R. (1987). The TEACHERS APPRENTICE: Designing an intelligent authoring system for high school mathematics. In G. Kearsley (Ed.), *Artificial intelligence and instruction: Applications and methods.* Reading, MA: Addison-Wesley.

McKendree, J., & Anderson, J. R. (1987). Effect of practice on knowledge and use of basic LISP. In J. M. Carroll (Ed.), *Interfacing thought.* Cambridge, MA: MIT Press.

Miller, G. A., Galanter, E., & Pribram, K. H. (1960). *Plans and the structure of behavior.* New York, NY: Holt, Rinehart & Winston.

Newell, A., Shaw, J. C., & Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review, 65*, 151-166.

Pirolli, P. L. (1985). *Problem solving by analogy and skill acquisition in the domain of programming.* Ph.D. dissertation, Carnegie-Mellon University.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skill. *Canadian Journal of Psychology, 39*, 240-272.

Sleeman, D. H. (1983). Inferring student models for intelligent tutor-aided instruction. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine learning.* Palo Alto, CA: Tioga.

Sleeman, D. H., & Brown J. S. (1982). *Intelligent tutoring systems.* New York, NY: Academic Press.

4/17/4, to author