COGNITIVE SCIENCE 13, 467-505 (1989)

Skill Acquisition and the LISP Tutor

JOHN R. ANDERSON
FREDERICK G. CONRAD
ALBERT T. CORBETT
Carnegie-Mellon University

An analysis of student learning with the LISP tutor indicates that while LISP is complex, learning it is simple. The key to factoring out the complexity of LISP is to monitor the learning of the 500 productions in the LISP tutor which describe the programming skill. The learning of these productions follows the power-law learning curve typical of skill acquisition. There is transfer from other programming experience to the extent that this programming experience involves the same productions. Subjects appear to differ only on the general dimensions of how well they acquire the productions and how well they retain the productions. Instructional manipulations such as remediation, content of feedback, and timing of feedback are effective to the extent they give students more practice programming, and explain to students why correct solutions work.

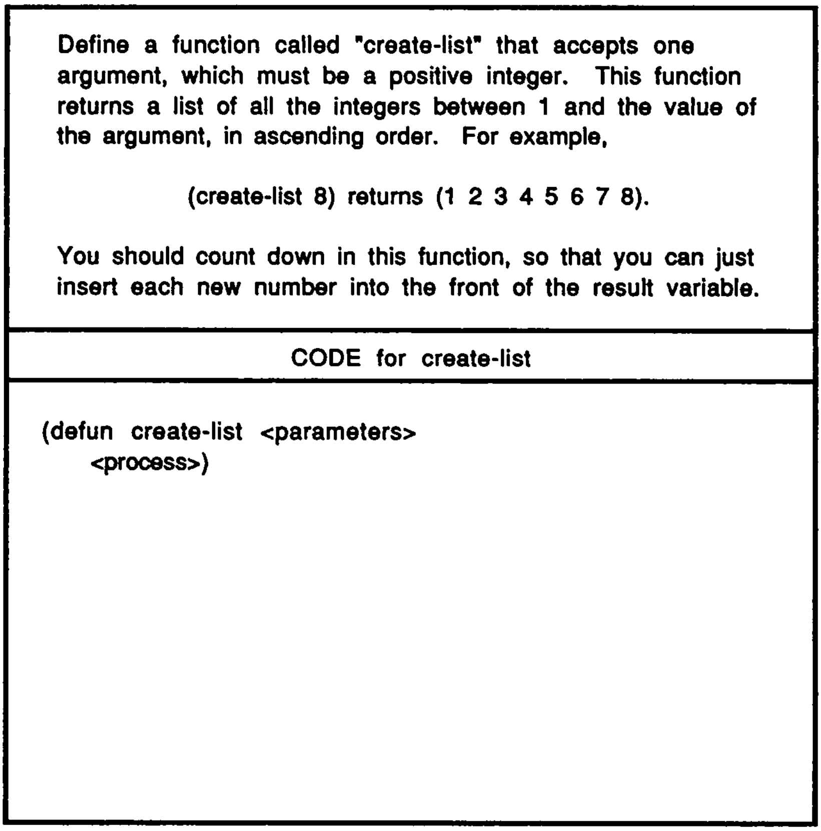
INTRODUCTION

If one observes a student learning LISP, the first impression one gets is of a very complex and perhaps chaotic phenomenon. It certainly stands as a challenge to a theory of skill acquisition to make sense of that behavior. An interesting possibility is that the apparent complexity is simply in the eyes of the beholder. Just as the random dot stereograms (Julesz, 1971), which encode simple geometric objects seem complex, so it might be that behind the complexity in LISP learning there is a very simple pattern of learning. All one would need would be the right filter to bring that pattern into focus. From early studies of LISP programming (Anderson, Farrell & Sauers, 1984), it seemed that this might be true. It was possible to argue, in specific cases of students solving a particular problem, that the complexity of LISP learning reflected the complexity of LISP itself and the complexity of the student's experiences with LISP, while the actual learning processes were simple. The argument took the form of simulating the learning that occurred in a specific coding episode. Unfortunately, such simulations of protocol

Correspondence and requests for reprints should be sent to John R. Anderson, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213-3890 studies only examine a small fraction of the curriculum by a few students. In order to confirm the hypotheses about how complex skills are acquired, a methodology facilitating the study of many students mastering the whole of a complex skill would be required.

The development of a computer-based tutor for teaching introductory programming was partly motivated by the desire to place LISP learning under systematic analyses. The tutor has been in use teaching undergraduates at Carnegie-Mellon University since the fall of 1984. The tutor provides students with a series of programming exercises and gives help when needed as the students generate solutions. The tutor has been shown to produce significant performance enhancement. In two evaluation studies, students completed the exercises with the tutor in one-third to three-quarters the time required by control groups who completed the exercises on their own. In addition, students using the tutor scored at least as well on tests as the control group, and in one of the studies scored about one letter grade higher on the final test. There is evidence from the lab, however, that the tutor is not as effective as a human tutor. Thus, while the LISP tutor is effective, it is by no means utopian.

The major motivation for developing the tutor was to shed light on issues concerning the nature of cognition. At the heart of the tutor is a cognitive model of the knowledge an ideal student would employ in doing the coding exercises. As described below, this knowledge is represented in the form of if-then rules (productions) for code generation. When provided problem descriptions analogous to those given the student, the model can generate a step-by-step solution to the exercises required of the students. The model also contains incorrect coding rules that generate errors which actual students are likely to make in various contexts. The tutor provides assistance to students essentially by running the model in synchrony with the student, comparing the student's response at each step to the relevant correct and incorrect rules and responding accordingly. Elsewhere (Anderson, Boyle, Corbett, & Lewis, in press; Anderson & Reiser, 1985; Corbett & Anderson, in press) the LISP tutor has been described in detail, including the philosophy of its design, and assessments of its instructional effectiveness. This article is concerned with examining its contributions as a tool for cognitive science research. Our goal is to assess the adequacy of the ACT* theory (Anderson, 1983), as embodied in the student model, in describing the behavior of students learning LISP. The basic instructional strategy of the tutor is to get the student to mimic the steps of the ideal production model. It was by no means obvious that such an instructional strategy would work, and its success serves as a general confirmation of the ACT* theory. The goal of this article is to provide a more specific accounting of the behavior of students with the tutor, and determine if these behavioral details correspond to the ACT* theory.



Define a function called "create-list" that accepts one argument, which must be a positive integer. This function returns a list of all the integers between 1 and the value of the argument, in ascending order. For example,

(create-list 8) returns (1 2 3 4 5 6 7 8).

You should count down in this function, so that you can just insert each new number into the front of the result variable.

CODE for create-list

```
(defun create-list <parameters>
   <process>)
```

Figure 1. The appearance of the tutor screen at the beginning of a coding problem

An Example Interaction with the LISP Tutor

While it is not the intention of this article to go into the details of the tutor's implementation, it will be useful to display one sample of what it is like to interact with the tutor. Figure 1 depicts the terminal screen at the beginning of an exercise. The screen is divided into two windows, and the problem description appears in the "tutor window" at the top of the screen. As the student types, the code appears in the "code window" at the bottom of the screen. This exercise is drawn from Lesson 6, in which iteration is being introduced. Students are familiar with the structure of function definitions by this point, so the tutor has put up the template for a definition, filling in **defun** and the function name for the student. The symbols in angle brackets represent code components remaining for the student to supply. The tutor places the cursor over the first symbol the student needs to expand, <PARAMETERS>.

As the student works on an exercise, the tutor monitors the student's input, essentially on a symbol-by-symbol basis. As long as the student is on some reasonable solution path, the tutor remains in the background and the interface behaves much like a structured editor. The tutor expands templates for function calls, provides balancing right-parentheses for students, and advances the cursor over the remaining symbols which must be expanded. If the student makes a mistake, however, the tutor immediately provides feedback and gives the student another opportunity to type a correct symbol. When the student types another response, the feedback is replaced either by the problem description (if the response is correct) or another feedback message (if the student makes another error). The tutor will also provide a correct next step in a solution, along with an explanation if the student appears to be floundering,[1] or if the student requests an explanation.

Table 1 contains a record of a hypothetical student completing the code for the exercise. This table does not attempt to show the terminal screen as it actually appears at each step in the exercise. Instead, it shows an abbreviated "teletype" version of the interaction. As described above, while the student is working, the problem description generally remains in the tutor window, while the code window is being updated on a symbol-by-symbol basis. Instead of portraying each update to the code window in the interaction, the table portrays nine key "cycles" in which the tutor interrupts to communicate with the student. At each of these enumerated cycles the complete contents of the code window are shown, along with the tutor's response. The tutor's response is shown below the code to capture the temporal sequence of events; on the terminal screen, the tutor's communications would appear in the tutor window above the code. In each cycle all the code which the student has typed since the preceding key cycle is shown in boldface. However, in each case, the tutor is responding specifically to the last symbol the student typed.

In the first of the cycles displayed, the student has typed in the parameter list and has called **loop** in order to iterate. The tutor reminds the student that it is necessary to create some local variables before entering the loop.

In the second cycle, the student has called **let** and is about to create a local variable. The template for numeric iteration calls for two local variables in this function, so the tutor puts up a menu to clarify which variable the student is going to declare first.

In the third cycle, the student has coded an initial value which would be correct if the function were going to count up. However, this exercise is intended to give the student practice in counting down, so the tutor interrupts the student.

[1] A student is judged to be floundering at a step in the solution if he/she repeats the same type of error three times or makes two mistakes that the tutor does not recognize.

TABLE 1
A Teletype Depiction of a Hypothetical Student Completing the Code
for the Function **create-list** with the Tutor

The Problem Description

Define a function called "create-list" that accepts one argument, which must be a positive integer. This function returns a list of all the integers between 1 and the value of the argument, in ascending order. For example,

(create-list 8) returns (1 2 3 4 5 6 7 8).

You should count down in this function, so that you can just insert each new number into the front of the result variable.

Cycles in which the tutor communicates with the student

1. *(defun create-list* **(num)**
```
      (loop)
```
You will need local variables in this function. So before you code anything else, there is a particular LISP operator you need to code.

2. *(defun creat-list (num)*
```
      (let ((  )
        <BODY>))
```
Which of the following do you want to do?

```
PRESS:          IF YOU WANT TO
1         Initialize a variable to hold the list of digits between
        1 and the value of the parameter.
2         Initialize a counter variable to keep track of how
        many times we go around the loop.
3         Have the tutor choose.
```

Menu choice: 2

3. *(defun create-list (num)*
```
      (let ((count 1)
            <OTHER-INITALIZATIONS>)
        <BODY>))
```
In this problem we won't want to start counting at the same value each time we call the function, so we can't initialize the counter to a constant like 1. How is the starting value being communicated to the function?

4. *(defun create-list (num)*
```
      (let ((count num)
            (numlist (list num)))
           (lop ))
```
Did you mean loop (y or n)? y

5. *(defun create-list (num)*
```
      (let ((count num)
            (numlist (list num)))
          (loop
            (cond ((equal count 1) numlist))
            <LOOP-ACTIONS2>)))
```
You do want to return the variable "numlist" here, but just typing it into the code won't accomplish that. What LISP operator will allow you to return something from a LOOP?

(continued)

TABLE 1 (Continued)

```
6. (defun create-list (num)
       (let ((count num)
             (numlist (list num)))
         (loop
           (cond ((equal count 1) (return numlist)))
            <LOOP-ACTIONS2>)))
```

In numeric iteration the first thing you should do after the exit test is update the counter variable. You can use the function SETQ to do this.

```
7.     (defun create-list (num)
           (let ((count num)
                 (numlist (list num)))
             (loop
               (cond ((equal count 1) (return numlist)))
               (setq count (1- count))
                (cons )))
```

You've got the right idea, since you'll need to use the function "cons", but remember you are trying to insert the new value of the counter into the list. What LISP function allows you to assign the results of this function call to a local variable?

```
8. (defun create-list (num)
       (let ((count num)
             (numlist (list num)))
         (loop
           (cond ((equal count 1) (return numlist)))
           (setq count (1- count))
           (setq numlist (list)))
```

If you LIST together arguments like 7 and (8) you get (7 (8)). What you want is (7 8). Try another function that combines things together.

```
9. (defun create-list (num)
       (let ((count num)
             (numlist (list num)))
         (loop
           (cond ((equal count 1) (return numlist)))
           (setq count (1 count)
           (setq numlist (cons count numlist)))))
```

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

THE LISP WINDOW

```
=> (create-list 10)
(1 2 3 4 5 6 7 8 9 10)

=> next
```

In the fourth cycle, the student has made a typing error which the tutor recognizes, and in the fifth cycle the student is attempting to return the correct value from the loop, but has forgotten to call **return.**

In the sixth cycle, the cursor is on the symbol <LOOP-ACTIONS2> and the student has asked the tutor for an explanation of what to do next. The tutor tells the student what the current goal is and what symbol to type next in order to accomplish the goal. In addition, the tutor puts the symbol, **setq**, into the code for the student.

In the seventh cycle, the tutor recognizes that the student is computing the new value for the result variable, but has forgotten that the new value must be assigned to the variable with **setq**. In the eighth cycle, the student has gotten mixed up on the appropriate combiner function to use in updating the result variable. The tutor tries to show, by means of an example, why **list** doesn't perform quite the right operation and another combiner is needed.

Finally, in the ninth cycle, the student has completed the code. Note that, for illustration sake, this interaction shows students making rather more errors than they usually do. Typically, the error rate is about 15% while it is approximately 30% in this dialogue.

After each exercise, the student enters a standard LISP environment called the LISP window. Students can experiment in the LISP window as they choose; the only constraint is that they successfully call the function they have just defined (which the tutor has loaded into the environment for them).

ACT* AND TUTORING

The focus in analyzing student interactions with the LISP tutor here will be on their implications for the ACT* theory (Anderson, 1983, 1987b). While the ACT* theory is quite complex, there are basically three claims in it that are significant in the instruction of a skill like LISP programming. These assumptions are discussed in this section, along with their consequences for instruction.

1. Production Rule Representation of a Skill

According to the ACT* theory, a skill like LISP programming can be represented as a set of independent production rules. So, for instance, consider the following piece of LISP code which creates a function that inserts the second element of one list at the beginning of another list:

```
(defun insert-second (lis1 lis2)
       (cons (car (cdr lis1)) lis2))
```

The following are the production rules that would apply in coding this function:

```
p-defun
      IF the goal is to define a function
  THEN code defun and set subgoals
             1. To code the name of the function.
```

```
            2. To code the parameters of the function
            3. To code the relation calculated by the function

p-name
     IF the goal is to code the name of the function
        and =name is the name
   THEN code =name

p-params
     IF the goal is to code the parameters of the function
        and the function accepts one or more arguments
   THEN create a variable for each member of the set
        and code them as a list within parentheses

p-insert
     IF the goal is to insert one element into a list
   THEN code cons and set subgoals
            1. To code the element
            2. To code the list

p-second
     IF the goal is to get the second element of a list
   THEN code car and set a subgoal
            1. To code the tail of the list

p-tail
     IF the goal is to code the tail of a list
   THEN code cdr and set a subgoal
            1. To code the list

p-var
     IF the goal is to code an expression
        and a function parameter has the expression as a value
        and =name is the name assigned to that parameter
   THEN code =name
```

About 500 such production rules have been created to encode the skill of programming in LISP. The 500 rules define a prescriptive model of how the student should solve programming problems. It is called the *ideal student model.* A major fraction of tutor development has gone into developing such rules. It is a difficult task to define a set of rules that will solve a large class of problems. It amounts to solving a subset of the automatic programming problem (Barr & Feigenbaum, 1982) with the added constraint that the solution be in a form that can be realized in the human head.

2. Declarative Origins of Knowledge

Although the theory holds that the skill knowledge of an experienced student is represented as a set of production rules, it is not the case that the skill begins in this form. According to the theory, one cannot present these production rules to the student and expect the student to encode them directly

TABLE 2

Function Calls	Value Returned	Operation
(car '(c d f))	c	Return the first element in the list
(cdr '(c d f))	(d f)	Return the list with the first element removed
(cons 'c '(d f))	(c d f)	Insert the first argument at the beginning of the second argument
(list 'c '(d f))	(c (d f))	Make a list out of the arguments

as production rules. Instead, relevant information must be initially encoded in declarative knowledge structures by the student. Here, for instance, is the instruction presented in Anderson, Corbett, and Reiser (1987) relevant to the production rule p-tail above:

> The function cdr accepts one argument, which must be a list, and returns the tail of the list. That is, it returns a version of the list with the first element deleted. (p. 10)

Table 2, also drawn from Anderson, Corbett, and Reiser, summarizes the English description and provides an example function call for **cdr** and three other elementary list-processing functions. An informal observation is that students refer to the examples in this table a great deal more than they do to the English instruction.

In any case, students must encode such information in a declarative representation of what a function does and use it to guide their programming. In the ACT* theory there is a process called knowledge compilation which converts the initial interpretive use of declarative knowledge into a procedural production-rule form. Thus, the ACT* theory of learning is one of learning by doing—the only way one gets knowledge into its ultimate procedural form is by practicing the operations which these production rules will implement.

There are clear pedagogical implications of this initial stage of using declarative knowledge. One is that one should carefully fashion it so that the target productions will be compiled. One method of accomplishing this goal would be to guide carefully the students' interpretation of the instruction and extrapolation of this instruction to the target problem. A tutor which would do this might look much like the Socratic WHY tutor of Stevens and Collins (1977). However, developing such tutorial interactions is not what this project has focused on, and it remains a future research goal. Instead of engaging the student in a dialogue, the tutor gives the student the opportunity to practice coding and comments on the understanding which the student demonstrates. This approach has proven successful.

Part of the reason for ignoring the acquisition of declarative knowledge is the intimate connection of this with comprehension of natural language which would raise a host of complexities. It certainly is possible that careful fashioning of the instruction that precedes practice might have substantial pedagogical benefit. The working assumption here is simply that the student emerges from this instruction with a probability of having extracted the correct information and the tutor takes it from there.

3. Tuning of the Skill
One's instructional objectives are not completed when the student has formed production rules to embody the skill. It could be the case that these rules are incorrect, overly specific, or overly general. Thus, one has to monitor the student's performance to determine if the rules are correct and show the student what is correct if the student is in error. It is also the case that as these productions are practiced they can increase in strength so that they will apply more readily and rapidly. Thus, the tutor tries to monitor how well students are doing on individual productions and selects problems to practice rules on which the tutor judges a student to be weak. Much of the effectiveness of the LISP tutor appears to be because of its ability to monitor student performance on specific productions, which allows it to respond immediately to specific errors and to give individualized practice.

Skill Acquisition is Simple
Perhaps the most interesting point about the ACT* theory of skill acquisition is that there is nothing more to skill acquisition than envisioned under assumptions 1-3 above. Thus according to the ACT* theory, the process of acquiring a complex skill like LISP programming is very simple in and of itself. All the complexity is due to the structure of the domain, reflected in the structure of the productions, and not in the learning process. If it can be confirmed that the theory is accurate in its analysis of the learning of LISP, this would provide very substantial support for the ACT* theory generally.

The research to be reported in this article is aimed at putting to test assumptions 1 and 3 above; by its structure, the tutor does not permit the analysis of assumption 2. The absence of any analysis of part 2 necessarily limits conclusions of simplicity. It will be argued that learning is simple, *after* the initial declarative information is incorporated.

Although much of the data will be presented in summary form only, an attempt has been made to be exhaustive in presenting all the features of student behavior with the LISP tutor that have been identified in this project. This has the cost of not focusing on just the most interesting results. However, if one wants to conclude that skill learning is simple, all the known trends should at least be mentioned with nothing held back from the reader. It is only in the context of an extensive effort to identify complexity in learning, that the conclusion of simplicity becomes compelling.

PRODUCTION RULES AS THE UNITS OF SKILL

In this section, a number of analyses of student interactions with the LISP tutor will be detailed. It is worth identifying at the outset how this data was analyzed in order to shed light on the purported production rules. The data from the LISP tutor comes in as a stream of keystrokes and responses by the tutor. This data can be partitioned into cycles in which (1) the tutor sets a coding goal (i.e., places a cursor over goal symbol on the screen); (2) the student types a unit of code corresponding to a production firing (generally a single atom or "word" of code); and, (3) the tutor categorizes the input as correct or incorrect (or as a request for help) and responds accordingly. If the response is correct, the tutor will set a new goal in the next cycle. If it is incorrect, the tutor provides feedback and resets the same goal in the next cycle. If the student asks for an explanation or appears to be floundering at the goal, the tutor will provide the correct answer and set a new goal in the next cycle.

Thus, consider the example of insert-second above, and imagine that the student has just typed **cons**. At this point the screen would look like this:

```
(defun insert-second (lis1 lis2)
    (cons <elem1> <elem2>))
```

In the following cycle the tutor would place the cursor over the goal symbol <elem1>, the student would type code, for example, "(car" and when the student has typed the final space after car, the tutor would evaluate the input and respond. It is of interest here to extract two measures of production firings from this data: time and accuracy. Firing time is measured only for goals in which the student's first response is correct. The measure of firing time is the time from when the tutor is ready to accept input (cursor over <elem1> in the above example) to when the student has completed input (the final space bar in the above example). Two measures of firing accuracy have been extracted: (1) the probability that a student responds correctly in his/her first attempt at a goal, and (2) the number of extra attempts (cycles) required to achieve a correct answer at a goal. The second measure will be largely used since it proves to be more sensitive (often initial errors are just slips while repeated errors are signs of real difficulty). As a rule of thumb, the number of extra attempts is about one and a half times the number of errors.

What is happening during the period of time attributed to a production? It is clearly not just a single correct production rule firing. There must be the setting of subgoals to type the individual characters, and the actual typing of these characters. Moreover, students can delete characters in order to correct mistypings, or even change their minds about the correct code unit to type. The tutor will also intervene to block syntactically illegal characters. Thus, the time for these segments will involve much more than simply the time for the target production to fire. The target production just sets the top level organization for the episode. However, it is the rule of interest, because it represents the new chunk the student must learn. Also, since typing and interacting with the tutor presumably represent skills at a relatively high asymptotic level of proficiency, learning the coding rules accounts for much of the variation in performance across segments.

Having segmented the student protocol into such production units, one can then begin to analyze various statistics associated with each unit. This requires aggregating events involving the same production. When all production firings in an exercise are collapsed, the data does not appear particularly systematic. As an example, Figure 2 plots the average coding time and error rate for each of the first six coding problems in Lesson 3 from data collected in the academic year 1985–1986. As can be seen, there is not much of a systematic trend, consistent with one's first impression that the LISP data is chaotic. The critical question is whether one begins to see systematic trends when the data is partitioned not by exercise, but by coding opportunity for the individual production rules. Figure 2 plots average times for the first occurrence of each production, the second occurrence, and so on. As can be seen, there is now a very systematic learning curve. To the extent that such systematic trends are seen and to the extent they are interpretable, this will be evidence for the psychological reality of production rules. A further issue which will be discussed in a later section is whether this level of aggregation, across all production rules, hides any systematic trends. If it does, and if these trends are not predicted by the ACT* theory, this would be important evidence against the theory.

Learning

One of the first analyses completed looked for learning trends within the LISP tutor. This question was first examined in data collected from 34 students who learned LISP from the tutor in the spring of 1985. Figures 3 and 4 present one relevant analysis for Lessons 2, 3, and 5, from that course. For each lesson, the fate of new production rules has been examined as they are practiced across opportunities in the lesson (i.e., rules which had not appeared in earlier lessions).[2] Figure 3 plots the number of errors the student makes per goal (this measure is bounded above by three in the LISP tutor), and in Figure 4 firing time has been plotted. Both dependent measures in the figures are plotted on a log scale. Along the abscissa of each graph is log practice. Most production rules do not have as many as eight opportunities for practice, which is why later trials have been aggregated.

The individual lessons show some variability but the overall trend is quite clear. There is a linear relationship between log performance and log practice, at least for the second, and following opportunities. Such linear functions on log-log scales imply a power function relationship between performance and practice—a relation which is typically found in learning research (Newell & Rosenbloom, 1981). There is some indication that the first point may be

[2] Lesson 1 is excluded because students received special help with their first few problems, and Lesson 4 is excluded because there are very few new productions.

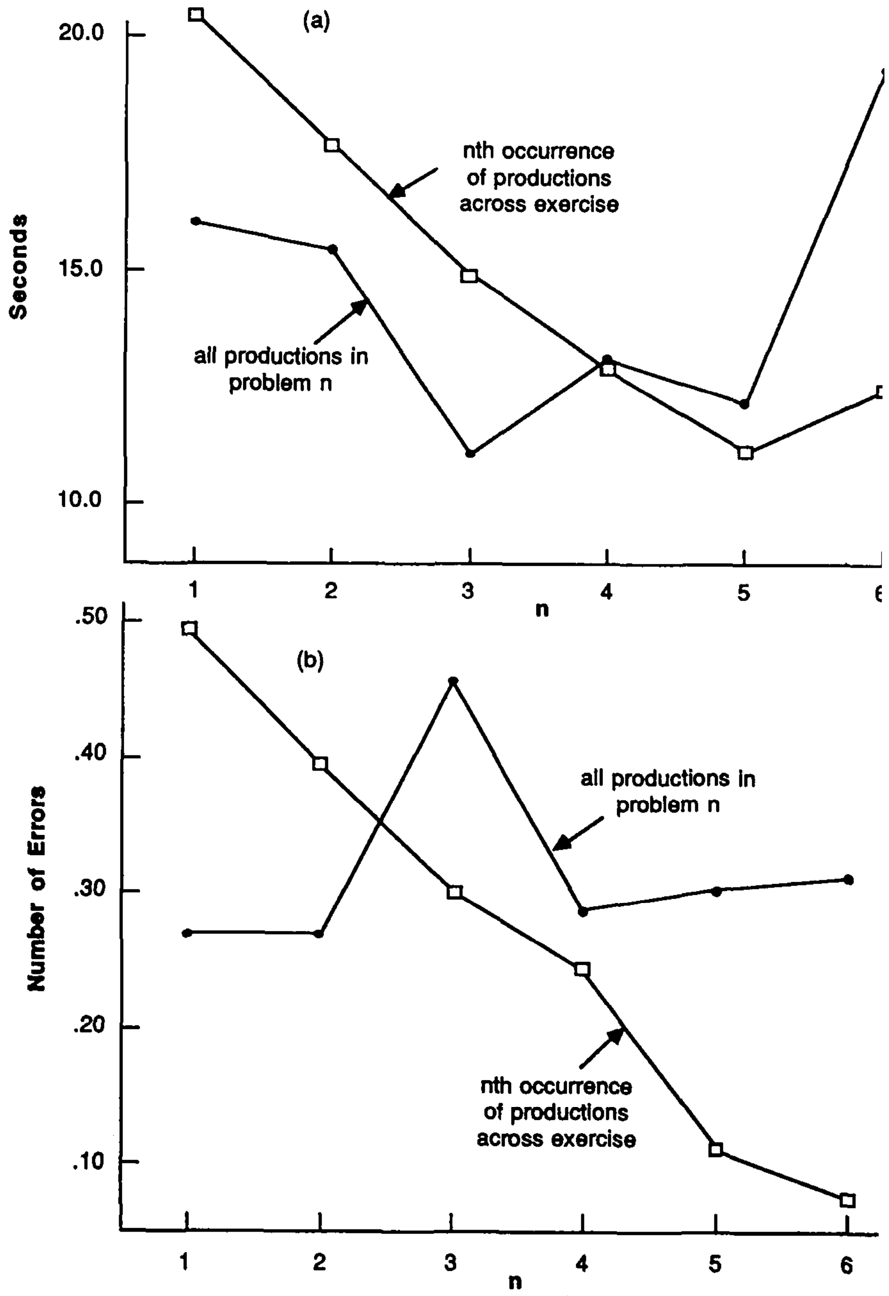


Figure 2. Mean time per production (part a) and mean error rate (part b) in Lesson 3 as function of serial position. The contrast is the regularity of the data when it is aggregate over productions that appear in the nth problem versus productions that appear for th nth time

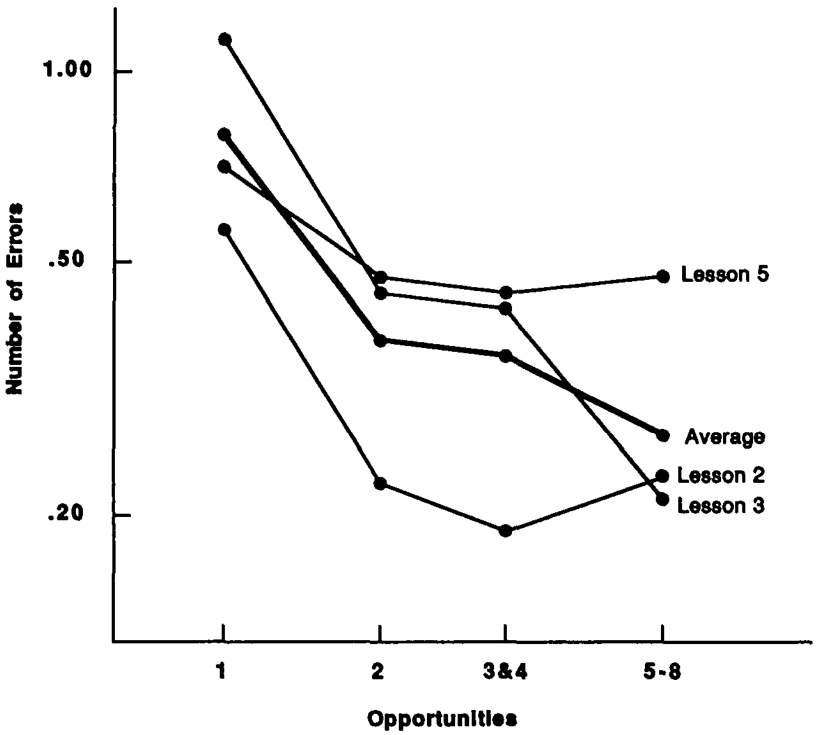


Figure 3. The number of errors per production made by students as a function of the amount of practice in the lesson in which the productions are introduced

off the linear relationship. It appears that the improvement from first to second trial may be greater than would be predicted extrapolating backwards from the rest of the curve.[3] According to ACT*, the large improvement from the first to second opportunity reflects the compilation of the production rule following the first opportunity.

Whereas Figures 3 and 4 provide an analysis of how learning progresses within a lesson, Figure 5 provides an analysis of what happens to production rules across lessons. This figure tracks performance on production rules in the lesson in which they are introduced (referred to as the "original" lesson) and in the immediately subsequent lesson (referred to as the "subsequent" lesson). (Productions are included in this analysis only if they occur at least twice in each of the respective lessons). In this figure, students' performances (time and errors) were plotted the first and last times they coded a produc-

[3] One can raise questions about just how to place the first trial on these graphs. The issue is how to assign a measure of prior practice to the first trial (and subsequent trials). One might argue that it has 0 trials prior experience, not 1. On the other hand, one might argue that prior study gives it a prior practice greater than 1.

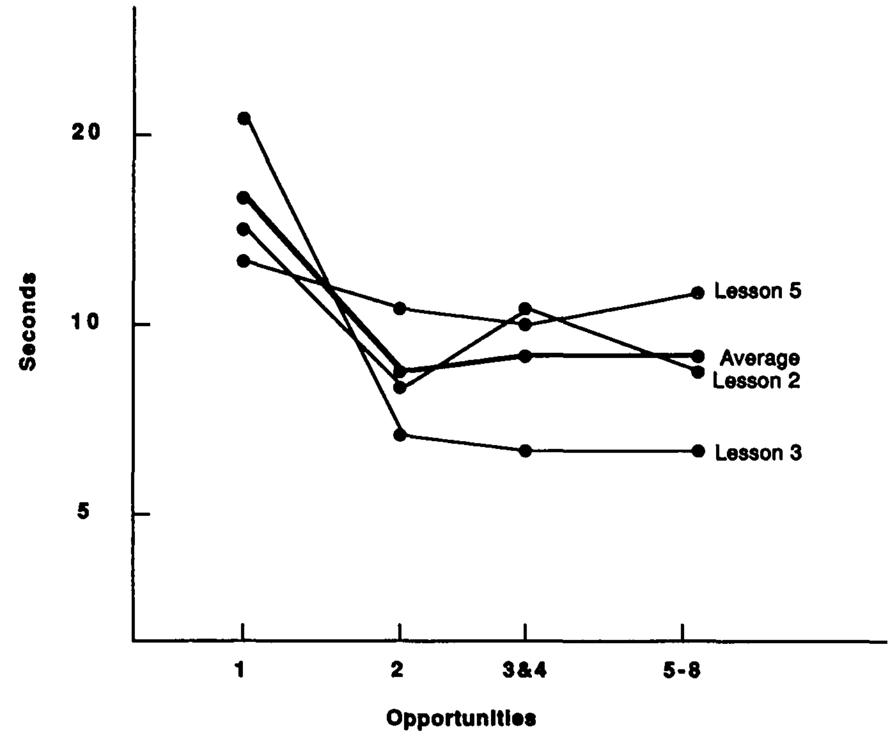


Figure 4. Time for correct coding per production as a function of the amount of practice in the lesson in which the productions are introduced

tion in the original lesson, and the first and last times they coded the production in the subsequent lesson. Rapid improvement is seen in the original lesson, some forgetting between lessons (which average a week apart), and further learning within the subsequent lesson.

The regularity of the data in Figures 3 through 5 is remarkable. Except for the possibility of a discontinuity from the first performance to later performances, there is nothing in the data that would not be expected from what is known about human learning in general. The learning trends are exactly what would be expected from an ACT* analysis. Of course such regularity and predictability supports the proposition that the production rules are the right units of analysis.

One might wonder if a production rule analysis is required to bring out the trends in the data. Perhaps these learning regularities are not defined on production rules but rather on something correlated with production rules. An obvious hypothesis is surface code. Thus, it might not be p-insert that is improving in performance but rather simply the typing of the LISP function **cons** that corresponds to it. Fortunately, the LISP tutor offers some opportunities to separate out these two possible explanations. This is because in

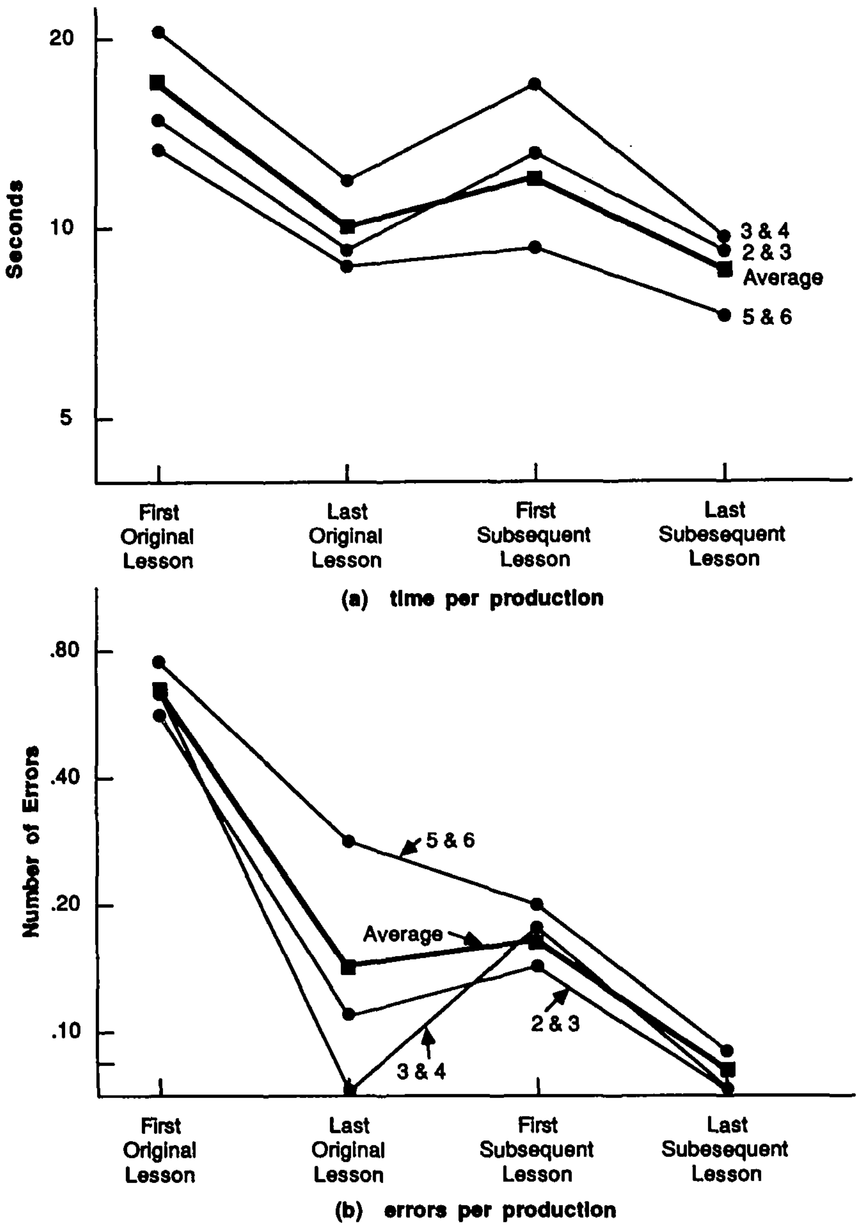


Figure 5. Transfer of productions from one lesson to the next

some cases there is a many-to-one relationship between production rules and surface LISP code. For instance, the rule p-first codes **car** when trying to get the first element of a list, but there is a different rule, p-second, that codes **car** when trying to get the second element of the list. It turns out that the simpler p-first is introduced earlier in the curriculum. After it occurs a number of times, the first opportunity for p-second to fire is introduced. The error rate on the last time p-first was used, before p-second was introduced, was .41. The error rate on the first opportunity for p-second was .68. This trend for increased error is opposite the general trend of fewer errors with more practice.

A number of other situations were considered in the early lessons in which two productions generated the same surface code. For instance, students are introduced to **cond** in Lesson 3 to code a main function body while in Lesson 6 **cond** is used for the first time to terminate iteration. The last time **cond** is used for a function body in Lesson 4 the error rate was .32 per opportunity, while it is 1.05 the first time **cond** is used for terminating iteration. The last time subjects have to code a number as a function argument, their error rate is .24. The first time they have to code a number to initialize a variable their error rate is 1.64. Subjects first use **setq** to initialize global variables and then to initialize local variables. Their error rate on the global variable **setq** production is .24 the last time before the local variable production. Their error rate on the local variable **setq** production is .84 the first time. Variables are coded by three separate productions in these early lessons: first for global variables, then for parameters of a function, and then for local variables. Their error rates drop to .38 for the global variable, jump to 2.18 for the first parameter, then decrease to .03 for parameters (which receive a lot of practice), and then jump to .84 for the first local variables. Thus, it is seen that there are sharp discontinuities in the overall learning if surface code is considered, but not if production rules are considered.

Regression Analysis

A more exhaustive analysis was performed on the data from the fall 1985 and spring 1986 semesters; while there were more students in these classes, only 42 had complete data sets and these were used for this analysis. There were 12 lessons in the tutor at that time. Regression analyses were performed in which an attempt was made to find best predictor equations for log coding times and errors separately for productions new to the particular lesson, and productions introduced in an earlier lesson. We collapsed into a single number all observations where a subject applied the same production (according to the tutor's analysis) in the same serial position on the same LISP function in a particular lesson (in order to fit the size constraints of our regression program). This meant there were 6409 observations of coding old productions, and 3350 observations of coding new productions.

The following regression equations were determined as the best fitting function for new productions:

$$\begin{aligned}\log(\text{time}) = 1.35 - .03(\text{lesson number}) - .31 \log(\text{within lesson opportunity}) \\ - .15 \log(\text{absolute position in code})\end{aligned}$$

$$\text{mean errors} = .23 - .11 \log(\text{within lesson opportunity}) - .03 \log(\text{absolute position in code})$$

where "lesson number" is just the number 1 through 12, "within lesson opportunity" is the dependent measure plotted in Figures 3 and 4, and "absolute position in code" is the serial position of the code in the function definition. A rather similar best fitting production was obtained for the old productions:

$$\log(\text{time}) = 1.31 - .01(\text{lesson number}) - .25\log(\text{within lesson opportunity}) - .26 \log(\text{absolute position in code})$$

$$\text{mean errors} = .16 - .09 \log(\text{within lesson opportunity}) - .02 \log(\text{absolute position in code})$$

Each of these predictor variables is statistically significant. Their appearance in these equations is particularly interesting when one considers the predictor variables which did not prove significant when placed in competition with these variables. They included: depth of embedding of the code which was being written, number of pending goals (or unexpanded symbols to the right), left-to-right position in the pretty-printing of the code, familiarity of the concept behind the production (as rated by a panel of four judges), and number of keystrokes in typing the symbol. It is also the case that the logarithm of lesson opportunity and the logarithm of absolute position are better predictors than are untransformed scores.

The effect of lesson number is quite significant for reaction times. It may just reflect an increased familiarity with the tutor interface. The fact that the same variable shows up for old productions as for new productions suggests that at least part of the phenomenon is a matter of general interface learning. It is also the case that lesson number is not significantly related to error rate. This is further evidence that the effect may be an interface effect and not reflect any real proficiency in coding.

The within-lesson opportunity effect is just the learning factor illustrated in Figures 3 and 4 for the spring 1985 data. Figure 6 shows the average coding times over the first six opportunities in a lesson separately for old and new productions for this data. Again, in the new productions, a marked improvement is seen from the first opportunity to later opportunities, and much slower improvement thereafter.

The effect of absolute serial position in the code is interesting because it has been established that the effect is logarithmic, not linear, and not a result of potentially confounded variables such as depth of embedding, number of pending goals, or left-to-right position in a pretty-printing. It is also the case that absolute serial position is a better predictor than relative serial position or a total length of the function. Figure 7 illustrates the average serial position effect over the first 33 positions. The initial long pause is

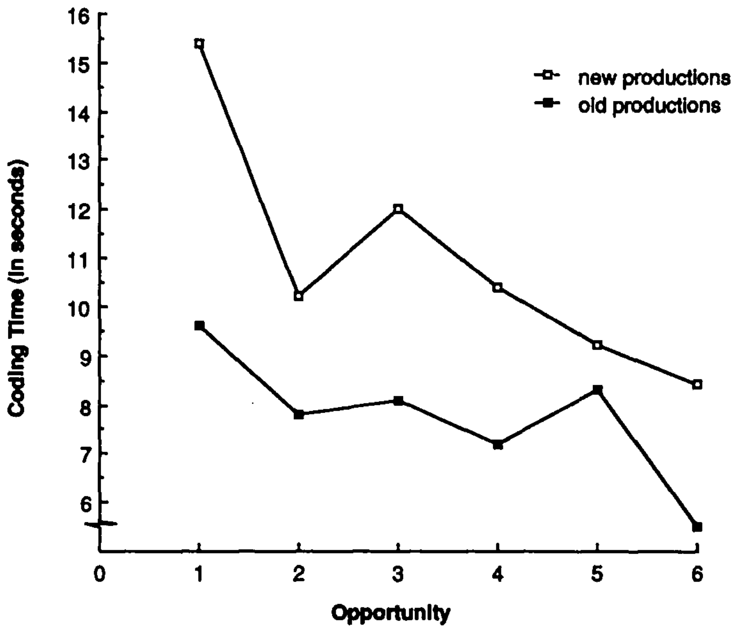


Figure 6. Mean coding time for old and new productions as a function of coding opportunity

undoubtedly due to reading the problem specifications and planning.[4] The subsequent speed up is thought to be attributable to subjects using, and hence practicing, their problem understanding as they go through the problem. This would strengthen their declarative representation and so speed their access to it. Anderson (1982), in providing an analysis of the power law relationship between practice and performance, noted that according to the ACT* theory there should be an effect both of procedural practice and declarative practice, and the overall performance improvement should be the product of two power practice functions.

It should be noted that such within-problem practice effects have also been documented in the data collected from the geometry tutor (Anderson, Boyle, & Yost, 1985). The time to fire a production in ACT* is basically the time to match the condition of the production to a declarative representation of the problem. The time to do this is both a function of the strength of the production and the level of activation of the declarative information. The level of activation of the declarative information is in turn a function of its

[4] There is a minor peak at Position 4 which occurs exclusively in Lessons 2 and 3 where some subjects will pause after typing "(defun <name>(<parameters>)..." before the function body. In later lessons "(defun <name>" is automatically presented and they do not have to type this stereotypic beginning.

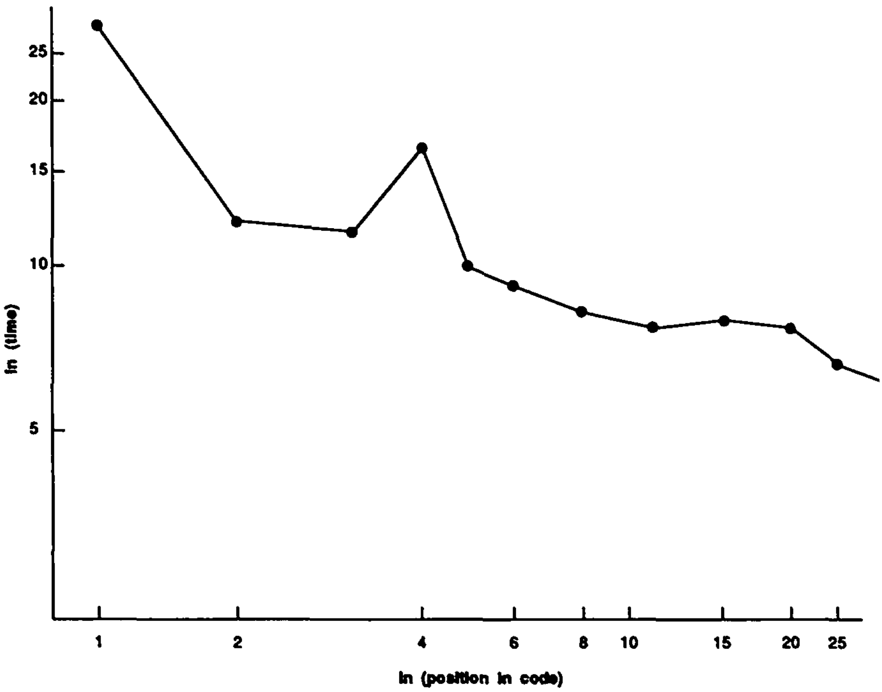


Figure 7. The effect of serial position in code on production time in Lessons 2 and 3

strength. It is also noteworthy that the ACT* theory (see Equation 4.1 in Anderson, 1983) implies that the effect of declarative activation and production strength should be superadditive. That is, there should be a greater effect of the same difference in production strength at lower declarative levels. The relationship is multiplicative. The additive relationships found among log serial position, log practice, and log time are consistent with the prediction of a multiplicative relationship among the untransformed variables.

A more direct test of the proposed superadditivity was attempted. Productions were broken into two categories which were above or below the median use. Serial positions were similarly broken into above and below the median. Data were then classified into a 2×2 matrix according to whether the production involved was above or below the median frequency, and that serial position above or below the median frequency. This analysis was done separately for each subject. Then an analysis of variance (ANOVA) was done where the factors were subject, (43 values), production frequency, (2 values), and serial position, (2 values). Separate ANOVAs were performed for old and new productions on mean coding time. These data, as well as mean frequency and mean serial position, are reported in Table 3.

There are main effects for both factors, but critically there is an interaction between the two of them ($F_{1,42}=9.09$; $p<.01$ for new; $F_{1,42}=21.75$;

TABLE 3
Results of Superadditivity Analyses

Reported in Each Cell are Mean Time, Mean Frequency, and Mean Serial Position

New Productions

		Serial Position	
		Low	High
Opportunity	Low	17.3 sec 1.0 6.2	10.1 sec 2.1 16.1
	High	13.8 sec 7.9 6.3	8.5 sec 8.9 20.4

Old Productions

		Serial Position	
		Low	High
Opportunity	Low	13.5 sec 2.0 6.8	8.9 sec 2.1 18.2
	High	9.6 sec 10.1 7.8	6.1 sec 13.3 20.6

$p < .001$ for old). In both cases, the effect of production frequency is greater at lower values of serial position. The same interaction between declarative activation and procedural practice has been noted by Carlson, Sullivan, and Schneider (1989); (see also Anderson, 1989). This is a relatively unique prediction of ACT*, and is not true of other production systems including the earlier versions of ACT, SOAR, (Rosenbloom & Newell, 1987) and its predecessors.

Use-Specificity of Production Rules

One of the peculiar features of a production rule analysis is that it predicts that the knowledge encoded in a particular production rule will be specific to that use. For instance, consider the following two production rules, the first, which encodes knowledge about the LISP function **car** for purposes of coding, and the second, which encodes the same knowledge for purposes of evaluation:

```
p-code-car
      IF the goal is to get an element
            and the element is the first member of =list
   THEN code car and set a subgoal
            1. to code =list
```

TABLE 4
Kessler (1988) Dissertation Results—Mean Time to Complete a Function and Mean Number of Errors*

	Target Task	
Prior Experience	Coding	Evaluation
Coding	163 (2.0)	358 (8.6)
Evaluation	304 (4.3)	231 (5.1)
Nothing	412 (4.9)	376 (7.2)

* Time per problem and errors n parentheses

```
p-evaluate-car
        IF the goal is to evaluate (car =lis)
           and =term is the first element of =lis
     THEN the value is =term
```

The important observation is that correcting or strengthening the procedural knowledge embodied in one rule should have no effect on its encoding in the other rule. Both sorts of knowledge might have arisen from the same declarative representation of **car** but once production rules have been formed, the future course of the knowledge becomes specific to the individual rules. This leads to the prediction that practice in one use of the knowledge should not transfer to another use of the knowledge.

There has been some indication that this prediction might in fact be true for LISP programming skills. For instance, Pirolli & Anderson (1984) showed that teaching subjects about how LISP evaluates recursive code did not seem to help them in writing recursive code. Again, McKendree & Anderson (1987) found some evidence that training subjects on the evaluation of simple LISP functions had no positive transfer to their ability to generate code.

Kessler (1988) performed a fairly thorough analysis of the transfer between coding and evaluation for simple LISP functions (such as insert-second earlier). Table 4 shows the results from one of these experiments. Kessler gave subjects either prior practice coding functions, or evaluating them, or no prior practice, and then transfered them to coding or evaluating. The coding experience occurred within the LISP tutor while the evaluation practice occurred through an evaluation tutor that required students to simulate the flow of control of the LISP evaluator. The statistics reported are mean time to code or evaluate a function and total number of errors over three test functions.

It is apparent in Table 4 that subjects show considerable learning. Their performance is much better the second time they code, and the same pattern holds for evaluation. However, the startling result is how little transfer there is from one activity to the other. While evaluation appears to produce some advantage for coding, the difference between the evaluation condition and the baseline control in coding performance is not statistically significant for

TABLE 5
Class Results

	Evaluation Tutor	No Evaluation Tutor
Time per coding production	12.49	12.03
Number of errors per coding production	.229	.232
	(%)	(%)
Percent correct on evaluation problems	66	56
Percent correct on debugging problem	61	65
Percent correct on coding problems	60	62

either dependent measure. There is no difference at all, apparent or statistically significant, between the performance of the coding-practice condition and the baseline control in evaluation performance.

This issue of transfer between coding and evaluation was pursued more extensively with the student population who used the LISP tutor in the fall of 1985. Half of the students practiced evaluating code with an evaluation tutor prior to each section of coding exercises, while the other half only did coding exercises. This continued through the 12 lessons of the course. Table 5 presents some comparisons between the two populations. First, we looked at time to code correct productions and mean number of coding errors per production in the LISP tutor. There was no statistically significant difference although subjects who did not have exposure to the evaluation tour did show a slight time advantage.

More interesting were the results obtained in the final paper-and-pencil exam. The questions on this exam could be divided into those that involved evaluation, those that involved debugging, and those that involved coding. As can be seen there is a substantial advantage for subjects exposed to the evaluation tutor on evaluation problems, but if anything, a disadvantage on the other problems. A statistical analysis was performed on the percent correct data to test for an interaction of evaluation experience and evaluation versus nonevaluation test problems. The test reached marginal significance ($F_{1,27} = 4.12$; $p \sim .05$) although the pairwise comparisons did not. It should be recognized that students know the makeup of the final exam and are motivated to study for it outside of the tutor. Still there is enough of an effect left of their tutor experiences to produce this interaction.

Individual Differences

The analyses to date serve to indicate that the improvements seen with the LISP tutor are quite regular and fit quite nicely with the ACT* production analysis. Next consider the interesting question of the individual differences noticed in the LISP tutor. One possible source of individual differences is past experience. The two courses taught over the fall of 1985 and the spring of 1986 permitted an opportunity to assess this effect. There were 38 students in the fall of 1985 who had no prior college-level programming course. There were 14 students in the spring of 1986 and all had taken the introductory Pascal course taught by the computer science department at Carnegie-Mellon University. All students were in the college of humanities and social sciences, and so had similar educational backgrounds, although as it turned out, the spring students had somewhat higher verbal and math SAT scores (fall had verbal SAT of 547 and math SAT of 566; spring had verbal SAT of 571 and math SAT of 613). These SAT differences were covaried out in a statistical test that confirmed the spring students did better than the fall students. Finally, both groups of students had the same experiences in the LISP course itself.

In analyzing the data, productions for each lession were grouped into two to four categories of rules that seemed thematically similar. Table 6 lists these various groups and indicates for which groups the spring students enjoyed a statistically significant advantage. Most of these are components which they should have been able to transfer from Pascal. It should be noted that their Pascal course emphasized data structures and implementations of lists. It might seem surprising that the only aspect of iteration for which there is transfer is Do's in Lession 11. However, the major chapter on iteration, Chapter 6, was not available for analysis because students were recruited into an experiment on debugging for that material. It is noteworthy that two other types of productions never proved a source of significant differences. First there nver were any significant differences on arthmetic functions where presumably both groups had equivalent background experience. Second, there was no significant difference with respect to recursion. While the spring students had been exposed to the concept of recursion in their earlier Pascal course, they had never done any programming using it.

This result is predicted by the identical elements model of transfer (Polson, Muncher, & Kieras, in press; Singley & Anderson, 1989). According to that model, transfer between domains can be understood in terms of shared production rules. For instance, the rule for coding the function AND is:

```
p-and
        IF the goal is to determine if test1 and test2 are true
    THEN code AND and set subgoals
                    1. to code test1
                    2. to code test2
```

This is a high-level rule and is not specific to the syntax or the ordering of the arguments. Such lower level details would be taken care of by productions that embody knowledge about the syntax of LISP or Pascal. It is a high-level production rule of this sort that could transfer whole cloth from

TABLE 6
Comparison of Students with a Prior Pascal Class (Spring) and Without (Fall)—
Mean Number of Errors per Production

Production Type	Fall	Spring	Difference
Lesson 1			
arithmetic	.20	.20	.00
list operation	.66	.46	.20*
variables	.22	.09	.13
Lesson 2			
arithmetic	.15	.19	−.04
list operation	.32	.20	.12
function definition	.33	.22	.11
Lesson 3			
function definition	.13	.10	.03
list operation	.80	.65	.15
logical functions	.43	.25	.18*
Lesson 4			
helping functions and function definition	.33	.23	.10
list operation	.43	.36	.07
logical functions	.29	.13	.16*
Lesson 5			
local variables	.31	.20	.11*
function structure	.17	.20	−.03
Lessons 7 & 8			
recursion	.36	.32	.04
args & params	.19	.16	.03
Lesson 9			
prog structure	.14	.16	−.02
list operations	.24	.26	−.02
Lesson 10			
recursion	.24	.21	.03
Lesson 11			
Do's	.16	.06	.10*
mapcars	.24	.30	−.06
Lesson 12			
arrays	.31	.20	.11
property lists	.61	.43	.18*
Do's	.35	.17	.18*

* Denotes a statistically significant difference

one language to another. This high-level production is represented in a form independent of the syntax of the language. One could imagine students conflating the syntax of the language with rules of using logic, but apparently they do not use such a surface representation. Singley & Anderson (1989) found a similar abstract representation of text editor commands.

Even holding prior experience constant, there are substantial differences among students in their performance with the LISP tutor. The question is whether there is anything more involved than some raw, undifferentiated factor of ability. One attempt to answer this was to see if certain groups of subjects found certain sets of productions difficult. A factor analysis was performed on the data from the spring of 1985; a second factor analysis was then performed on the combined data for fall of 1985 and spring of 1986. Two separate analyses were performed because the spring 1985 subjects worked with a different version of the tutor. In these factor analyses it was of interest to see which patterns of individual differences might appear when prior experience was not a particularly relevant factor. Specifically, it was of interest to learn whether a subset of students would have difficulty with a subset of the productions. The details of the factor analysis of the spring 1985 data and the details of the methodology are reported in Anderson (in press). Essentially, the student performance was taken (as measured in mean number of errors) on each production for each lesson and examined for patterns that would emerge. In the spring 1985 data, and more clearly in the combined fall 1985 data and spring 1986 data, factors emerged which loaded on thematically related productions. For instance, in Lesson 1 a factor emerged which loaded on arithmetic operations, and a factor emerged in Lesson 3 that loaded on logical operations. This meant, for instance, that in Lesson 1, one group of students tended to do relatively poorly on all arithmetic productions while another group of students tended to do well.

The initially frustrating feature of these within-lesson factors is that they did not show any across-lesson consistency. Thus, productions which loaded on one factor in one lesson would split up and load on different factors in a later lesson. To help organize these within-lesson factors, a meta-factor analysis was done. That is, students' factor scores from particular lessons were taken and a factor analysis of these was performed. Two meta-factors emerged fairly strongly in the spring 1985 data and in the combined 1985–1986 data. When the spring 1985 data was examined, it was noticed that most of the productions which loaded on factors which loaded on one of the meta-factors were new to that lesson (22 out of 34), while most of the productions that loaded on the second meta-factor were old (20 out of 23). This led to a labelling of the first meta-factor as an acquisition factor and the second meta-factor as a retention factor. A similar analysis was done on the 1985–1986 data. Most of the productions associated with one meta-factor were new to that lesson (18 out of 23), while most of the productions associated with the other meta-factors were old (23 out of 31).

Thus, what seems stable across lessons are only very general learning attributes, acquisition and retention. We think we understand why thematic clusters of new productions appeared in individual lessons but disappeared thereafter. These thematically related productions were discussed in the text in close proximity. If one imagines a subject's attention waxes and wanes while reading the text, then this will produce a local correlation among thematically related productions. This also suggests that the acquisition factor may reflect how well students extracted instruction from text.

There was some external validation of these two meta-factors. Although both were defined on behavior internal to the LISP tutor, both were strong predictors of performance on paper-and-pencil midterms and final exams. These factors were also associated with math SATs but not verbal SATs. The correlation of the retention factor with math SATs was −.62 for spring 1985 and −.38 for 1985–1986. The correlation of the acquisition factor with math SATs was −.03 for spring 1985 and −.60 for 1985–1986. Except for the 1985 correlation coefficient for the acquisition factor, all coefficients are significant.

In summary, the patterns of individual differences identified to date are quite simple. Prior programming experience is beneficial to the extent it overlaps with what students have to do in LISP. In addition to this, subjects seem to differ in some general factors of acquisition and retention. The analyses here do not really shed light on what might be involved in the acquisition and retention factors.

The Psychological Reality of Production Rules: A Summary

These data provide strong evidence for the production rule as the right unit of analysis in understanding the acquisition of a skill. The data confirm all the features thought to be critical to the concept of a production rule. The learning curves and individual differences data support the modularity of production rules: the claim that they are separate pieces of knowledge which are acquired independently. The data on across-language and across-lesson transfer support the abstract character of production rules—that they are not tied to the surface features of the behavior. The evidence for lack of transfer from evaluation to generation supports the condition-action asymmetry of production rules. While separate production system theories may involve additional claims (such as the declarative-procedural distinction of ACT*) and all production systems involve many notational conventions, the central features associated with a production are modularity, abstractness, and conditional asymmetry.

One might wonder about circularity in this conclusion in favor of production rules. After all the tutor was designed around a production system and treated the students as if they were acquiring production rules. Is it possible that this caused their behavior to have a production-rule-like character? It is not at all obvious that the mind could even learn with such a tutor if it were not organized like a production system. Thus, the success of the tutor is regarded as further evidence for the production rule analysis. However, it is possible that a different style of instruction may have produced different regularities than those noticed here. As for this possibility, it will be necesssary to wait and see if such a type of instruction is forthcoming.

REMEDIATION AND FEEDBACK

Now, turn to the issue of what evidence can be brought to bear on how tutorial interactions will affect learning rate. As the tutor goes along, it is trying to interpret both what the student is doing in the immediate problem state, and trying to build up a general picture of what the student's knowledge state is. At the problem level, the tutor is trying to find some sequence of productions, correct and buggy, which would generate the behavior observed by the student. This is what is called the model-tracing methodology and is a form of analysis by synthesis. When the student generates an incorrect piece of code the tutor will immediately notify the student and insist that the student go back on path. If the tutor can recognize the type of error made, it will also present a remedial message explaining why the code does not satisfy the current goal. There are two features of this interaction which deserve comment. First, the student can be asked about the benefit of the feedback messages he or she has been presented with. For instance, when the student confuses the LISP function **list** for **cons** in Table 1, he got the following message:

> "If you LIST together arguments like 7 and (8) you get (7 (8)). What you want is (7 8). Try another function that combines things together."

What benefit does the student gain from an explanation of why **list** won't work, instead of simply being told that it is not correct? This will be one of the major focuses in this section.

A second question concerns the benefit or harm of providing feedback immediately upon error. This is a topic that will be analysed in some detail in a later section. For now it is worth noting that one consequence of such immediate feedback is that the student is forced to keep on a correct path, which substantially simplifies the task of interpreting the behavior.

In addition to monitoring and providing feedback to the student as he or she progresses along a particular solution, the tutor tries to monitor the student's general knowledge. It does this by a variation on what is known as an overlay model (Goldstein, 1982). We use a statistical model which is also a variation in the procedure used by Atkinson (1972). For each production, the tutor maintains an estimate of the probability that the student knows the production. In doing this, the following probabilistic model of production learning is used: Each production is assumed to be either in the state of being learned or not. Each production is assigned a probability *a* of being in the learned state initially. This reflects the probability that the appropriate declarative knowledge has been learned, and will be applied at the first opportunity. If so, the knowledge-compilation process will produce a correct production. There is also a probability *b* that, if a production is not in the learned state on one trial, it will transit to the learned state on the next trial as a function of the tutorial interaction.

Monitoring the student's knowledge state would be trivial if external behavior were a perfect reflector of internal knowledge. However, it clearly is not; sometimes students get things right by lucky guesses, and sometimes students slip and get things wrong when they know better. Therefore, for each production a probability *g* is also estimated that students will perform the right behavior even when they do not know the production, and a probability *s* is estimated that students will slip and get the production wrong even when they do know it.

Earlier student performences were used to come up with an estimate of these probabilities for each production. The average probabilities were approximately $a = .6$; $b = .4$; $g < .05$; and $s < .05$. With these parameters in place one can then use Bayesian estimation procedures to derive from the students' performance on a production, an estimate of the probability that the student knows a particular production.

These estimated probabilities drive a remediation procedure. The goal of the tutor is to achieve a state where all productions have an estimated probability greater than .95. The tutor has a set of required problems which all students must do. These problems expose students to all productions and if the students make no errors on these problems, all of the estimated probabilities would be above .95. However, if the productions for a particular lesson are below .95 for any production, the tutor selects remedial problems. It chooses remedial problems that come closest to having 10% of their productions below threshold. It continues presenting remedial problems until all the probabilities surpass .95. Typically when a subject leaves a lesson, most of the probabilities are above .99.

Predicted Effects of Remediation and Feedback

Now that the tutor's remediation and feedback policies have been described, the question of what their consequences are expected to be can be considered. Clearly the beneficial effect of remediation would be predicted since it serves to tune and practice the procedures. This is not a particularly surprising prediction and it is not thought to be particularly unique to the ACT* theory.

The predictions of the ACT* theory about the effect of feedback messages are a bit more surprising. Unfortunately, the tutor is not well designed to put them to the test. Nonetheless, it is worth going through these predictions: It proves important to make a discrimination among the information contents of the various messages. Some messages explain what is wrong with the answer the student gave. These will be called error diagnoses. Other messages provide an explanation of the correct answer. As these latter messages are necessarily redundant with the text these will be called reteaching (in line with the usage of Sleeman, Kelly, Marinak, Ward, & Moore, 1989). These two types of messages are partially correlated with what are called bug messages, and "explanations" in the tutor. Bug messages are given upon the occurrence of a recognized bug, and tend to include error explanations although many also include reteaching. Tutor explanations are given when the tutor provides the correct code—either at the request of the student or because the student has made more than three errors. These are exclusively reteaching.

There is no reason in the ACT* theory to expect an advantage of error diagnoses. This is a rather startling realization because a major part of the philosophy in design of intelligent tutors is that it is important to give individualized explanations of errors. However, in the ACT* theory there is nothing to be gained by belaboring an error state. What is needed is reteaching that will establish in the student's head the declarative information sufficient to compile the correct production. The incorrect production must simply be weakened and abandoned—there is no concept in ACT* of a repair. The only benefit of an error explanation is to convince the students they are wrong, so that they will listen to the explanation of what is correct.

This naturally raises the question of why the messages in the LISP tutor involve error diagnosis. In part, the answer is practical in that is seems socially appropriate to explain errors. In part, it reflects the fact that the LISP tutor was designed while there was a discrimination-learning process in the ACT* theory. As discussed in Anderson (1987b), this has been abandoned. A discrimination process could be guided by pointing out what was wrong about an error. There is no role for such information in the current theory with its emphasis in learning by analogy from positive examples.

Recently, Sleeman, et al. (1989) have found evidence that this prediction of ACT* may in fact be correct. They contrasted error diagnosis plus reteaching with just reteaching and found no difference, although both conditions were superior to a control that involved no reteaching. McKendree (1986) also found that focusing students on the correct answer is more effective than explaining the error in interactions with the geometry tutor.

Unfortunately, a clean test in the LISP tutor is not available because the error messages reflect a mixture of reteaching and diagnosis. To the extent they do the former, they should be beneficial. Thus, the only prediction here is the relatively weak prediction that error messages should help.

An Experiment on Remediation and Feedback

In the fall of 1987 an experiment was performed in which each student in the class was presented one of four versions of the LISP tutor defined by two dimensions: (1) whether the remediation feature was turned on or not,

TABLE 7
Fall, 1987 Experiment

	Feedback	No Feedback
(a) Performance Internal to the Tutor		
Remediation	10.4 sec	11.1 sec
	.13 errors	.18 errors
No Remediation	12.4 sec	10.6 sec
	.18 errors	.26 errors
(b) Effectiveness of Feedback		
Remediation	37% repeat errors	60% repeat errors
No Remediation	33% repeat errors	64% repeat errors
(c) Performance External to the Tutor		
Remediation	95% quiz	94% quiz
	87% exam	82% exam
No Remediation	86% quiz	88% quiz
	65% exam	79% exam

and (2) whether the tutor provided any explanatory text when notifying the student of errors or providing correct answers. In conditions with remediation turned off, all students went through the minimal required problems. In conditions with feedback turned off, students were simply told they were wrong when they made an error and, if the tutor provided a correct answer at any step (either at the student's request or because the student seemed to be floundering), it just provided the correct code without explanatory comment.

The results are shown in Table 7. Section (a) reports execution time per production and errors per production in completing the exercises. There are no significant time effects, but there are marginally significant effects of remediation on errors ($F_{1,29} = 2.99$) and of feedback on errors ($F_{1,29} = 3.08$). Section (b) of Table 7 presents a different analysis of feedback effectiveness. This is a measure of the probability of another error at the same goal after the first error. As can be seen, students make almost twice as many repeat errors when only told they have made a mistake, as when given a diagnostic message about the nature of the mistake.

Section (c) of Table 7 presents an analysis of performance on two post-test assessments of the tutor manipulations. After every second lesson students were given a computer-administered quiz that required them to write two function definitions and debug two others in an hour's time. Average scores for these six quizzes are reported, along with average scores on paper-and-pencil midterms and final exams. Both measures show a substantial effect of remediation and no effect of feedback explanation. The remediation effect on quiz scores is significant, $F_{1,29} = 6.48$ and the effect on the paper-and-pencil tests is of marginal significance, $F_{1,29} = 2.9$. In the case of the quiz scores, students with remediation are essentially perfect. This is a gratifying confirmation of the remediation algorithm which is supposed to help students achieve a near perfect command of the individual production rules.

Thus, it appears there is a remediation effect. As noted earlier, almost any theory would predict this. The situation with respect to a feedback effect is interesting. The strongest contrast is between Parts (b) and (c) of Table 7 where substantial immediate impact of feedback without any long-term consequences was seen. The fact that a remediation effect was able to be shown, indicates that the problem is not lack of sensitivity of our long-term dependent measure. Overall, the results of this experiment suggest that well-designed feedback can minimize the time and pain of learning but has no effect on final instructional outcome.

This is not what was predicted from the ACT* theory. One would expect feedback to be effective to the extent it was effective immediately. The best post-hoc explanation available is that subjects were able to generate their own explanations of answers once these answers were provided by the tutor. Thus, feedback from the tutor helped subjects self-correct as seen in Part (b) of Table 7, but once subjects saw the correct answer they were able to understand it, and so there was no long-term benefit as seen in Part (c) of Table 7. There is an interesting generalization that characterizes the results in Part (c) of Table 7. Subjects' final learning achievement is determined by what problems they solve, and not by the manner in which they solve them. The next experiment will reinforce this generalization.

Delay of Feedback

As noted earlier, some of the motivation for giving feedback immediately in the LISP tutor is technical. This makes it easy to trace where the student is at any point in time. In addition to making tutoring easier, it also makes data analysis a good bit simpler because one can easily segment the student interactions into pieces that can be attributed to individual production rules. However, it is certainly not the case that immediate feedback is uniformly popular with students. While some students claim to like it, other students complain that they could correct their errors if given a chance. This is a complaint that occurs much more often in students with prior programming experience. They remark that they often want to use the terminal as a scratch pad to try out various ideas and edit them, but the tutor will not permit them to do this.

In addition to the technical reasons there are two pedagogical motivations for immediate feedback. First, it is a simple observation that a good fraction of student time is spent floundering about trying to diagnose and recover from errors. There is no reason in this framework to suppose that such time contributes to anything but student frustration. Second, when students finally do repair their errors, they may have gone through such a confusing trajectory to reach the correct solution that they do not understand why it works. Certainly we have seen students outside of the tutor who got some code to work after 20 patches, but had the code in such a confused state they could not understand it.

The original formulation of this immediate feedback principle (Anderson, Boyle, Farrell, & Reiser, 1987) stressed another aspect to the confusion point. According to the 1983 ACT* theory, for a new rule to be compiled from an error episode, one had to hold in working memory the whole episode involved in applying the wrong production, apply a sequence of other productions predicated on the wrong production, hit an impasse, eventually find the difficulty, and correct it. Immediate feedback was thought important because it eliminated the working memory burden of holding all of this information. In the current ACT* theory (Anderson, 1987b), with its emphasis on analogy, this is not a critical issue. One can simply learn the correct production from analogy to the final correct solution and ignore all of the intermediate-state information.

The research of Lewis & Anderson (1985) has been cited in support of the original rendition of the immediate feedback principle. There it was shown, in a dungeon-and-dragons-like game, that subjects suffered substantial confusion with delayed feedback. However, that was a situation where the total correct solution was never laid out before subjects and they had to integrate a sequence of moves in memory. By contrast, in the LISP domain, students ultimately have a working LISP function in front of them to study. Provided that code is not confusing, one should be able to learn from the final product of an extended, self-correcting, delayed-feedback episode. One would learn from it in the same way one learns by analogy from other examples. Students might take longer to achieve the final state with delay of feedback but should learn as much from it.

In summary, it seemed that in coding LISP function definitions, subjects who receive delayed feedback on errors should have neither an advantage nor a disadvantage in terms of final achievement—a contrast to Lewis and Anderson (1985). This was tested by providing a version of the tutor more like the one the advanced students requested—one that allowed them to play around with the code and control when the tutor evaluated it and provided help. This is called the demand-feedback tutor. The technical issues involved in creating such a tutor are described in Corbett, Anderson, and Patterson (in press); here, just its behavior will be described. As the student types code, the tutor acts as a structured editor, ensuring that the code is syntactically correct. The interface does not restrict the order in which the code is entered, so students can deviate from top-down, left-to-right coding. At any point students can press a key to request the tutor to examine whatever they have written. The tutor then progresses through the code step-by-step, skipping over any goal symbols the student has not filled in yet. The tutor stops at the first error it encounters, provides the same feedback mes-

TABLE 8
Study of Student Controlled Feedback*

	Standard Tutor	Student Control
Time to complete problem	5.7 min	8.6 min
Percent correct on quiz	83	83
Number of errors per problem caught by tutor	1.15	.83

* Corbett, Anderson, & Patterson (in press).

sage as the immediate feedback tutor, and moves the remaining unanalyzed code into a buffer from which the students can retrieve it if they wish.

In the fall of 1987, a first comparison of the demand feedback tutor and the immediate feedback tutor on the first two lessons of the curriculum was made with subjects who participated in the experiment for pay. Some relevant statistics from the study are displayed in Table 8. Students spent considerably longer per programming problem with the delayed feedback tutor, thus reflecting the extra effort in hunting down their own mistakes and the cost of having to abandon large pieces of code which proved to be erroneous. However, there was no difference in performance across the two groups on a quiz that followed. Thus, in contrast to Lewis and Anderson, who found an effect of delay of feedback on both learning time and final achievement, only an effect on learning time was found here. A similar result is reported by Lewis & Anderson (1989) in the domains of geometry proofs and formal logic proofs. In both of these domains, like LISP, the final solution is represented on the screen, so students do not have to call upon memories for solution steps.

Also examined were the number of errors the tutor caught per problem. As can be seen from Table 8, the tutor caught about a third of an error less per exercise in the demand feedback condition. This corroborated students' claims that some errors could be self-corrected.

The analysis of student self-correction is quite interesting. Students attempted to correct 34% of all the errors they made but were successful only just a little over half the time (57%), resulting in a 19% net reduction in the number of errors the tutor had to correct. These attempted error corrections accounted for 80% of the code changes students made. The remaining 20% of the students' changes involved code that was already correct. Of these modifications, 43% changed the code to another correct form while the remaining 57% changed it to an incorrect form. Thus, of all the revision students made, 46% were successful error corrections, advancing the students toward a correct solution.

Some other interesting statistics were gathered about student performance in the demand feedback tutor condition. When they did ask for evaluation of their code, 88% of the time they had written a complete answer. The problems in these first two lessons are short (not getting much longer than insert-second) but this result does indicate that students are not willing to ask for anything like constant and immediate feedback. Of the 12% of the cases where they asked for feedback on partial code it was never the case that the partial code had any "holes" in it. That is to say, in every case of partial code, what was missing was at the end of the function. Also there were very few deviations (6 out of 850 opportunities) from top-down, left-to-right coding anywhere. This suggests that students are not being restricted by the requirement in the immediate feedback tutor to write their code in the top-down, left-to-right order.

In summary, it seems that subjects do take longer with delayed feedback, reflecting the cost of pursuing wrong paths. However, there is no evidence that they suffer any learning deficit. Apparently, subjects are able to learn from their final solutions independent of whether the feedback is immediate or not. As in the previous study it seems that learning achievements are determined by what problems they solve and not by how they reach the solution.

CONCLUSIONS

This article presents a fairly complete summary of efforts made to analyze student learning with the LISP tutor, and the search for evidence about how a complex skill is learned. When the production system model is used to factor out the complexity of the skill, it is found that the actual learning process is quite simple; there are no surprising phenomena with respect to basic learning, transfer, or individual differences. One could hardly claim that the impact of the various tutorial interventions poses a complex picture. So far, the only instructional effects observed are

1. Remedial practice produces long-term learning benefits;
2. Explanation helps students immediately correct their code; and
3. Delayed feedback means students will take longer to get through problems.

These three outcomes perhaps deserve the label obvious. We have not found any evidence for complex instructional effects and we have found evidence that learning can be very successful in simple instructional situations.

The inability to find substantial effects of instructional manipulations such as feedback content raises an interesting question: What is the basis of the achievement gains reported in earlier work for the LISP tutor? We think there are two substantial benefits. One is that some sort of immediate feedback tutor serves to cut down on total time in going through the material. It may not be necessary for the tutor to take away control from the student to achieve this economy. Rather than forcing the student to correct an error immediately, one might imagine a tutor which signals the error but allows the student to continue to code and returns to correct the error when the student wants to. Such a version of the LISP tutor (Corbett & Anderson, 1989) has recently been completed. Certainly for purposes of student acceptance, it would be desirable to have a version of the tutor which combines the time-savings benefit of immediate feedback with a sense of control for the student.

The second benefit in the tutoring work here is the careful task analysis that went into developing production system models of programming skills. This led to a more rational curriculum development and instruction. Some of the major benefits involve the reanalysis of recursive programming (Pirolli, 1986). It is not clear how important the tutors are in conveying this benefit. Perhaps it is sufficient simply to have students read the textual explanation and solve the problems on their own. Solving on their own would lead to longer programming times, but it may have no impact on eventual achievement.

Evaluation of ACT*

In view of the findings, a reassessment of the theoretical issues mentioned in the introduction would be beneficial. To what extent do these data support the ACT* conception of a production, and the ACT* conception of how productions are learned? (This data does not really shed any light on the initial declarative stage of learning.) Also, reviewers have asked how this data separates ACT* from alternative theories. This is a difficult enterprise because other theories have not been applied to learning LISP. However, given the request, we believe we have some license to speculate. The data support a number of production rule features as conceived of in ACT*.

(a) A production rule is a modular unit of knowledge learned independently from other units of knowledge. This is shown in the regular learning curves defined on production rules, and their independence from similar rules in the factor analysis. This feature seems contrary to schema-like theories which would emphasize larger units of knowledge.

(b) The rules are abstract and not tied to specific content. Evidence for this was the transfer of production rules across contents and languages. This seems contrary to connectionist approaches which deny the existence of abstract rules.

(c) The application of knowledge is specific to use. This was seen in the evidence that the same concept had different learning histories depending upon its use. Similar results have been reported in Singley and Anderson (1989). This seems contrary to schema-like approaches which emphasize more flexible execution of knowledge.

(d) The interaction between practice of the problem and practice of the rules is evidence for the procedural-declarative distinction and for the ACT* conception of it. Again, there is other evidence for such declarative-procedural interaction (Carlson et al., 1989). This would seem contrary to SOAR (Rosenbloom & Newell, 1986) and other production systems, which do not make a procedural-declarative distinction.

There were a number of results which supported the ACT* theory of production learning:

(a) The power functions are as predicted by ACT*. Besides ACT*, there are a number of other theories that predict power law learning (Crossman, 1959; Lewis, 1978; Logan, 1988; Mackay, 1982; Newell & Rosenbloom, 1981) but power-law learning does not seem predicted by connectionist or schema-based theories. Although we are leery of the statistical dilemmas encountered in establishing a discontinuity, the apparent extra drop off from Trial 1 to Trial 2 is uniquely predicted by ACT*.
(b) The research indicated that the final solution was what was critical to learning, and not the trajectory to that final solution. This is what was predicted by the 1987 version of ACT*, which, unlike the 1983 version, emphasized learning by analogy to example solutions. Again, it is a result that does not seem to be predicted by SOAR, another production system model of learning.

More important than the implications of this data for specific theories of skill acquisition is the result that skill acquisition is simple under appropriate factoring out of the structure of the domain. Should this conclusion generalize to other skills acquired under other circumstances, it implies that the key to understanding skill acquisition is a careful analysis of the structure of the domain. Once the units of knowledge are identified, acquisition of the skill can be predicted by composing simple learning functions for these units.

■ Original Submission Date: August 15, 1988

REFERENCES

Anderson, J.R. (1982). Acquisition of proof skills in geometry. In J.G. Carbonell, R. Michalski, & T. Mitchell (Eds.), *Machine learning, An artificial inteligence approach.* Palo Alto, CA: Tioga Press, 191–220.

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

Anderson, J.R. (1987a). Production systems, learning, and tutoring. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development* (pp. 437–458). Cambridge, MA: MIT Press.

Anderson, J.R. (1987b). Skill acquisition: Compilation of weak-method problem solutins. *Psychological Review 94,* 192–210.

Anderson, J.R. (1989). Practice, working memory, and the ACT theory of skill acquisition: A comment on Carlson, Sullivan, & Schneider. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 15,* 527–530.

Anderson, J.R. (in press). Analysis of student performance with the LISP tutor. In N. Frederksen, 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 modeling 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* (pp. 93-134). New York: Wiley.

Anderson, J.R., Boyle, C.F., & Yost, G. (1985). The geometry tutor. In *Proceedings of IJCAI-85*. Los Angeles, CA: IJCAI.

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., & Reiser, B.J. (1985). The LISP tutor. *Byte, 10,* 159-175.

Atkinson, R.C. (1972). Optimizing the learning of second-language vocabulary. Journal of Experimental Psychology, 96, 124-129.

Barr, A., & Feigenbaum, E. A. (1982). Automatic programming. In *Artificial Intelligence Handbook*. Los Altos, CA: Kaufmann.

Carlson, R.A., Sullivan, M.A., & Schneider W. (1989). Practice and working memory effects in building procedural skill. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 15,* 517-526.

Corbett, A.T., & Anderson, J.R. (1989, May). The LISP intelligent tutoring system: Research in skill acquisition. In *Proceedings of the 4th International Conference in Artificial Intelligence in Education* (pp. 64-72). Amsterdam.

Corbett, A.T., Anderson, J.R., & Patterson, E.J. (in press). Student modeling and tutoring flexibility in the LISP tutor. In C. Frasson & G. Gauthier (Eds.), *Intelligent tutoring systems: At the crossroads of artificial intelligence and education*. Norwood, NJ: Ablex.

Crossman, E.R.F. (1959). A theory of the acquisition of speed-skill. *Ergonomics, 2,* 153-166.

Goldstein, I.P. (1982). The genetic graph: A representation for the evolution of procedural knowledge. In D. Sleeman & J.S. Brown (Eds.), *Intelligent Tutoring Systems* (pp. 51-77). New York: Academic.

Julesz, B. (1971). *Foundations of Cyclopean perception*. Chicago: University of Chicago Press.

Kessler, K. (1988). *Transfer of programming skills in novice LISP learners*. Unpublished doctoral dissertatioan, Carnegie-Mellon University, Pittsburgh, PA.

Lewis, C.H. (1978). *Production system models of practice effects*. Unpublished doctoral dissertation. University of Michigan, Ann Arbor.

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., & Anderson, J.R. (1989). *Effects of immediacy of feedback on learning proof skills*. Manuscript in preparation.

Logan, G.D. (1988). Toward an instance theory of automatization. *Psychological Review, 95,* 492-527.

Mackay, D.G. (1982). The problem of flexibility, fluency, and speed-accuracy trade-off in skilled behavior. *Psychological Review, 89,* 483-506.

McKendree, J.E. (1986). *Impact of feedback during complex skill acquisition*. Unpublished doctoral dissertation, Carnegie-Mellon University, Pittsburgh, PA.

McKendree, J.E., & Anderson, J.R. (1987). Frequency and practice effects on the composition on knowledge in LISP evaluation (pp. 236-259). In J.M. Carroll (Ed.), *Cognitive aspects of human-computer interaction*. Cambridge, MA: MIT Press.

Newell, A., & Rosenbloom, P. (1981). Mechanisms of skill acquisition and the law of practice. In J.R. Anderson (Ed.), *Cognitive skills and their acqusition* (pp. 1-55). Hillsdale, NJ: Erlbaum.

Pirolli, P. (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction, 2,* 319-355.

Pirolli, P.L., & Anderson, J.R. (1984). Learning to program recursion. In *Sixth Annual Cognitive Science Meetings* (pp. 277-280). Boulder, CO.

Polson, P.G., Muncher, E., & Kieras, D.E. (in press). Transfer of skills between inconsistent editors. *Human-Computer Interaction.*

Rosenbloom, P.S, & Newell, A. (1986). The chunking of goal hierarchies: A generalized model of practice. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine Learning* (Vol. II, pp. 247–288). Los Altos, CA: Morgan Kaufmann.

Rosenbloom, P., & Newell, A. (1987). Learning by chunking: A production system model of practice. In D. Klahr, P. Langley, & R. Neches, (Eds.), *Production System Models of Learning and Development.* Cambridge, MA: MIT Press.

Singley, K., & Anderson, J.R. (1989). *The transfer of cognitive skills.* Cambridge, MA: Harvard Press.

Sleeman, D., Kelly, A.E., Martinak, R., Ward, R.D., & Moore, J.L. (1989). Studies of diagnosis and remediation with high school algebra students. *Cognitive Science, 13,* 551–568.

Stevens, A., & Collins, A. (1977). The goal structure of a socratic tutor. In *Proceedings of the Association for Computing Machinery Annual Conference.* Association for Computing Machinery.