Acquisition of LISP Programming Skill[1]

John R. Anderson and Albert T. Corbett

Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The acquisition of a complex skill like writing LISP code can be decomposed into the learning of an underlying set of production rules. Each production rule appears to have a simple learning trajectory and its acquisition is independent of the acquisition of other production rules. These basic results do not appear to be dependent on the instructional modality under which one learns. Learning under a more directive modality only reduces the amount of time students spend debugging their errors.

INTRODUCTION

This chapter provides a review of our work on the acquisition of LISP programming skill in the context of various intelligent tutoring systems we have developed. This work can be seen as a general test of the ACT* (Anderson, 1983) theory of skill acquisition. That theory of skill acquisition is actually quite simple. It asserts that

(1) The knowledge underlying a skill begins in declarative form. Knowledge in this form must be interpreted to produce performance. The initial declarative knowledge commonly takes the form of an encoding of illustrative examples of the skill. These examples are used by analogy to guide the problem solving. For example, when first learning how to perform arithmetic in LISP, the student may encode that the sequence:

(* 27 32)

[1]This research was supported by contracts N00014-87-K-0103 and N00014-91-J-1597 from the Office of Naval Research

calculates the product of 27 and 32. The student may proceed to use this as an analog for determining how to compute the sum of 384 and 492.

(2) As a function of its interpretive use, this knowledge becomes compiled into a production rule form. In the example above, the following production might be formed:

```
IF the goal is to apply an arithmetic operation to X and Y
   and OP is the symbol denoting that operation
THEN enter (OP X Y).
```

(3) Individual production rules, once formed, acquire strength as a function of practice.

(4) Ultimately, a skill such as programming consists of hundreds of independent production rules. Performance consists of the sequential application of these rules.

The simplicity of the ACT* theory of skill acquisition is its most important theoretical claim. It certainly is a counter-intuitive claim about the nature of complex skill acquisition which appears anything but simple.

Given the importance of the claim that skill acquisition is simple under a production rule analysis, it becomes important to provide empirical evidence for this simplicity. This can be done directly by confirming regularities in the data predicted by the theory and indirectly by failing to find effects of plausible complicating factors. In the following sections, we will describe our efforts to find regularities in superficially complex data and our efforts to find complicating factors in the structure of exercises and in individual differences. The analysis of individual differences is particularly interesting. One might believe that different subjects would display differential success in mastering clusters of subskills. But while the ACT* theory allows for different rates of acquisition across subjects, it does not allow for different styles of learning. Before describing the results, it is necessary to describe the LISP Tutor, the data it generates and our methods of analyzing the data.

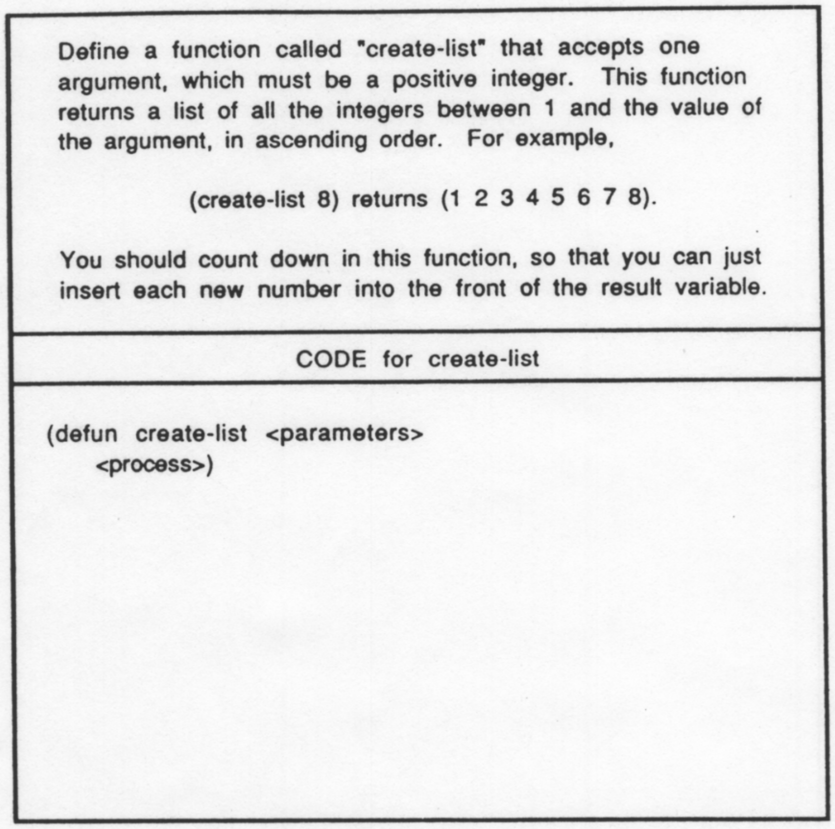
We have been engaged in the study of programming in LISP since 1980. Around 1983 it seemed that we had a good enough understanding of programming in LISP (reported in Anderson, Farrell, & Sauers, 1984) that we might actually use it as a basis for instruction. The essential idea was to organize instruction around the individual production rules that we feel the student should acquire. This set of production rules defines what we call the ideal student model. We try to match up in real time the student's problem-solving steps with some solution path that can be generated by this ideal production system model of the skill. This is called a model-tracing methodology. In the version of tutoring we will first describe, called immediate-feedback tutoring, whenever the student makes a problem-solving move that does not match any move allowed in the student model we immediately correct the student and force the student to make a move that matches the student model. Later in this chapter, we will describe what happens with less restrictive modes of tutoring.

AN EXAMPLE INTERACTION WITH THE LISP TUTOR

In this section we will describe an interaction with the original, immediate-feedback tutor we created. 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. The example 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, this 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,[2] or if the student requests an explanation.

[2]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.



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.

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.

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 create-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-INITIALIZATIONS>)
       <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?

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)
           (numlist (list num)))
        (loop
           (cond ((equal count 1) (return numlist)))
           (setq count (1- count))
           (cons)))
```

(continued)

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
```

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

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.

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 percent while it is approximately 30 percent 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).

GROUP DATA ANALYSES

Analysis of Computer Records

It is worth identifying how we segment a sequence of interactions, such as those in Table 1, and assign these segments to various 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 a 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.

Suppose the student is coding a function called **insert-second**, and imagine that the student has just typed **cons** as the beginning of the body of the function. 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 the code for that element. 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 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 firing? It is clearly not just a single correct production rule firing. There must be an encoding of the screen, 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 mind 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 thing that the student must learn. Also, since typing and interacting with the tutor presumably represent skills at relatively high asymptotic levels of proficiency, learning the coding rules accounts for much of the variation in performance across segments.

Learning Curves

We will present a description of the most systematically and exhaustively analyzed data collected with the LISP tutor. This data was collected in the fall of 1985 and the spring of 1986. While there were more students in these classes, we collected complete data sets from 42 students and these were used for this analysis. There were 12 lessons in the tutor at that time. Each lesson involved students solving a sequence of problems where a problem involves writing a LISP function to do a specified task. Table 2 displays exercise solutions, for a small illustrative portion of the curriculum, the first six exercises in lesson 3. For the sake of illustration, consider how performance varied across these six problems. Figure 2 presents coding time and error rate averaged across productions for each of the six exercises. There is a notable lack of any learning trend when the data is collapsed in this fashion.

Figure 3 presents the same data aggregated according to production identity rather than problem. Note that production rules appear in different patterns across the problems. For instance, the rule that codes an inequality operator ("<" or ">") applies once each in the first and third problems and twice in the fifth, while the rule for coding **cond** appears once each in the fourth through sixth. Thus, there is a poor correlation between problem number and how many times specific productions have been practiced. Figure 3 shows the data organized according to the number of practice opportunities per production. Now we see very systematic learning trends.

Thus, we see that a production analysis can convert an apparently complex pattern of data into a very simple pattern of data. The data in Figure 3 illustrate that amount of practice of specific productions is a strong determinant of performance as expected.

```
3.1     (defun compare (num1 num2)
                (> (+ num1 10) (* num2 2)))

3.2     (defun palp (lis)
                (equal lis (reverse lis)))

3.3     (defun numline (item)
                (list (zerop item) (< item 0)))

3.4     (defun carlis (object)
                (cond ((null object) nil)
                        ((atom object) object)
                        (t (car object))))

3.5     (defun checktemp (temp)
                (cond ((> temp hightemp) 'hot)
                        ((< temp lowtemp) 'cold)
                        (t 'medium)))

3.6     (defun make-list (item)
                (cond ((null item) nil)
                        ((listp item) item)
                        (t (list item))))
```

Table 2. Initial Problems in Lesson 3.

Regression Analysis

We (Anderson, Conrad, & Corbett, 1989) have performed a fairly exhaustive analysis of the data from all 12 lessons both to better analyze the practice factor and to identify other complexity factors. Regression analyses were performed on these data in an attempt to find best predicator equations for log coding times and errors. These analyses were performed separately on "new" productions in each of the lessons (production rules introduced in that lesson) and "old" productions in each of the lessons (production rules introduced in previous lessons). There were about 100,000 observations in total, far too many for our regression program, so we collapsed across subjects, generating a mean for all observations in which different subjects applied the same production (according to the tutor's analysis) in the same serial position in the same exercise in a particular lesson (in order to fit the size constraints of our regression program). This produced about a 10 to 1 collapsing in average, yielding 6409 observations for old productions and 3350 observations for new productions.

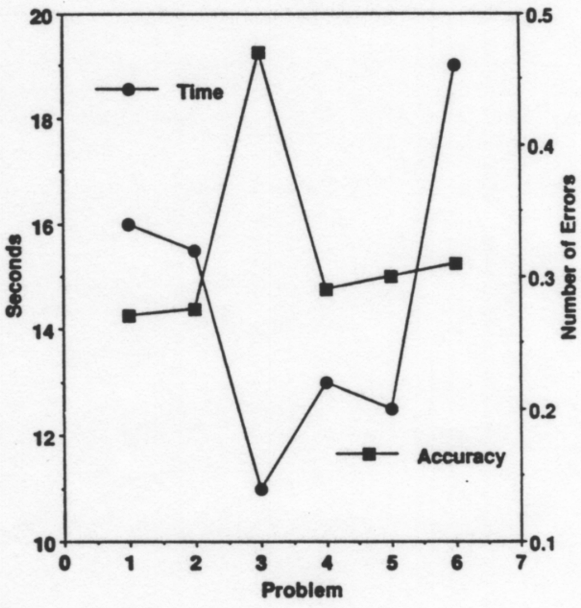


Figure 2. Mean coding time and error rate per production as a function of problem number in Lesson 3.

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

log(time) =1.35 - .03(lesson number) - .31 log(within lesson opportunity)
- .15 log(absolute position in code)

mean errors = .23 - .11 log(within lesson opportunity)
- .03 log(absolute position in code)

where "lesson number" is just the number 1 through 12, "within lesson opportunity" is the number of times the production had been used in the lesson up to and including the current opportunity, and "absolute position in code" is the serial position of the code in the function definition. A rather similar best fitting function was obtained for the old productions:

log(time) = 1.31 - .01(lesson number) - .25 log(within lesson opportunity)
- .26 log(absolute position in code)

mean errors = .16 - .09 log(within lesson opportunity)
- .02 log(absolute position in code)

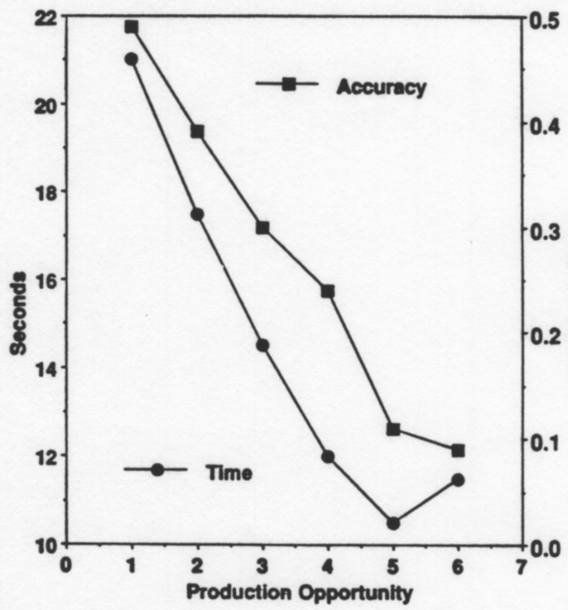


Figure 3. Mean coding time and error rate per production as a function of production opportunity in Lesson 3.

The within-lesson opportunity effect reflects the production specific practice effect and is illustrated in Figure 4. The difference between the intercepts for the old and new productions reflects the advantages subjects have when using the productions during a second or later session. It is interesting to compare the shape of the learning curves for old and new productions in Figure 4. The new productions show much larger speed up from first to second use, but after that they show similar rates of improvement. We attribute the special advantage from first to second opportunity for new productions to knowledge compilation.

The other two variables reflect "complications" to a degree, although they are not inconsistent with the theory. Indeed, the effect of absolute position in the code is a confirmation of a subtle prediction of the theory. Before discussing these two variables, however, it is worth noting the additional 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.

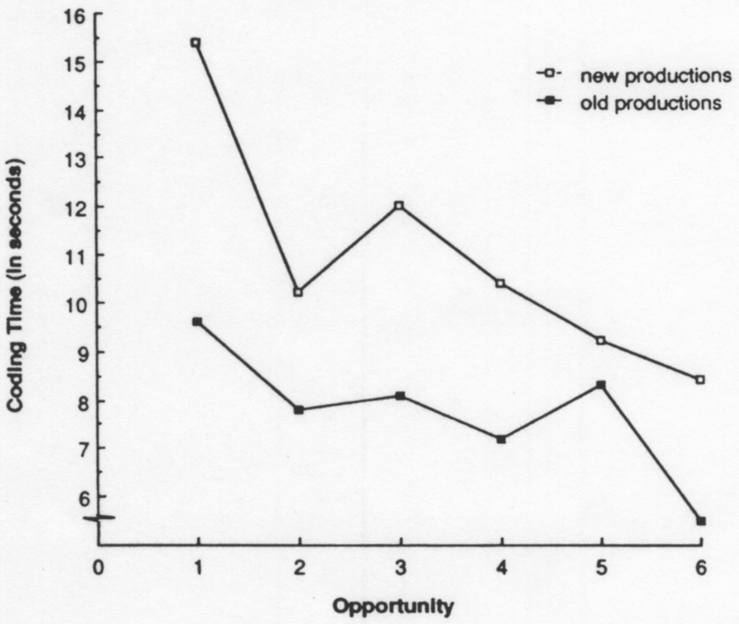


Figure 4. Mean coding time for old and new productions as a function of coding opportunity.

The effect of lesson number, while small, is quite significant for reaction times. This variable may just reflect an increased familiarity with the tutor interface. The fact that the same variable shows up for new productions (that do not appear in earlier lessons) as well as old productions suggests that this variable does not reflect production-specific practice. 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.

Problem Practice and Its Interaction with Production Practice

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 total length of the function. Figure 5 illustrates the average serial position effect over the first 33 positions. The initial long pause is at least in part due to reading the problem specifications and planning. 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 of the problem and so speed their access to it. 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. Thus, the effects of procedural practice and declarative practice should be superadditive.

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 2x2 matrix according to whether the production involved was above or below the median frequency, and the serial position above or below the median frequency. This analysis was done separately for each subject. Then an analysis of variance (ANOVA) was done with three factors: subject, (42 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 production 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 ($F_{1, 41}=9.09$; $p<.01$ for new; $F_{1, 41}=21.75$; $p<.001$ for old). In both cases, as predicted, the effect of production frequency is greater at lower values of serial position.

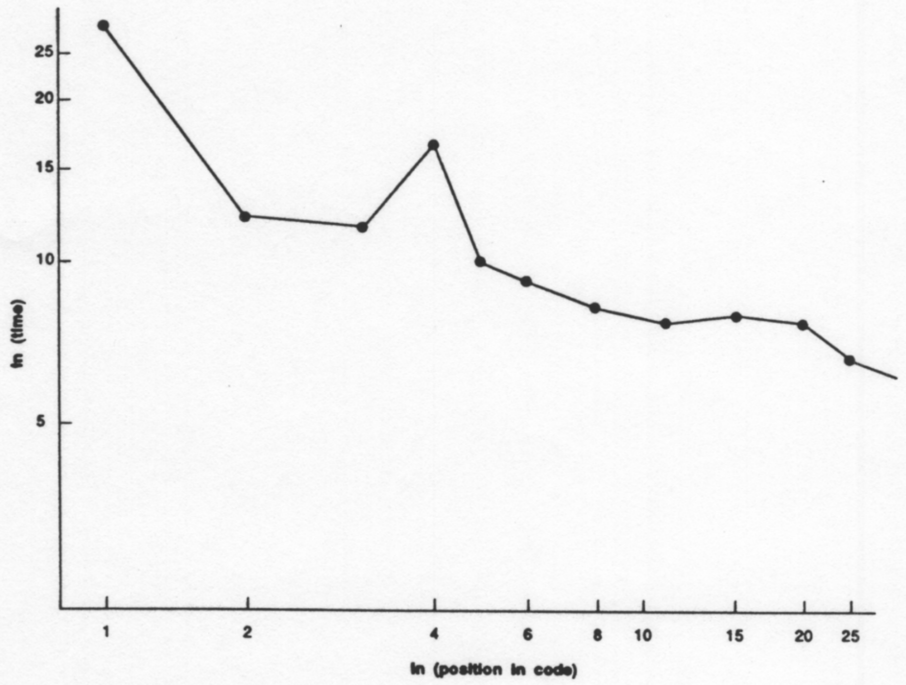


Figure 5. The effect of serial position in code on production time in Lessons 2 and 3.

INDIVIDUAL DIFFERENCES IN LEARNING

There are substantial differences among students in their performance with the LISP tutor both in error rate, time to complete exercises, and in post-test performance. The interesting question is whether there is anything more involved than some unidimensional factor of skill. In one attempt to answer this, a factor analysis was performed to see whether certain groups of subjects found certain productions categories difficult. One 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. These two separate analyses were performed because the spring 1985 subjects worked through a different curriculum. The details of the factor analysis of the spring 1985 data and the details of the methodology are reported in Anderson (1990). In the spring 1985 data, and more clearly in the combined fall 1985 data and spring 1986 data, factors emerged within each lesson 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.

New Productions

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

Old Productions

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

Table 3. Results of Superadditivity Analysis. Reported in Each Cell are Mean Time, Mean Frequency, and Mean Serial Position

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 of one of the meta-factors were new to that lesson (22 out of 34), while most of the productions that loaded on factors in 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 a subject's attention waxes and wanes while reading the text, then this will produce a local correlation among thematically related productions. Another factor that would produce this thematic clustering is that many of the new productions in a lesson are thematically related. For instance, a large fraction of the productions in the third lesson on conditionals are concerned with logical operation of some sort or another. Thus, to the extent that there is an acquisition factor, within a lesson it will produce a thematic clustering of productions. Thus, the apparent themacity of productions only reflects the fact that the new productions introduced in any lesson tended to be thematically related.

There is 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 1985-1986 correlation coefficient for the acquisition factor, all coefficients are significant.

These factors or math SATs are equally good predictors of performance on a final paper and pencil test at the end of the course. The failure to find any consistent thematic individual differences in the LISP learning is further evidence for the view that learning is simple. The only stable individual differences are very general acquisition and retention factors. This means that the acquisition of a production rule is not sensitive to its content.

LEARNING UNDER DIFFERENT TUTORING MODALITIES

So far the data we have displayed has come from the immediate-feedback version of the tutor. The suggestion has been made a number of times that the simplicity of the learning results reflect the constraints imposed by the tutor, particularly the constraint that students remain on a correct solution path, and is not representative of learning in general. Over the past few years we have begun to explore other feedback schemes that relax this constraint. We have developed a number of environments that relax the tutor's control over feedback and over the student's behavior. These newer versions of the tutor are:

No Tutor — The student receives no instruction at all and must solve the problems on their own. All they are told is whether their final solution is correct. They can use the LISP environment to try out their solutions which provides a feedback of a sort. Essentially, these students are in the same kind of environment that students have who learn without a tutor. However, they enter their code into the same structured editor as the tutor students. This enables us to perform some analyses of their data.

Demand Feedback — Students are not interrupted by the tutor with feedback but can request feedback on their solution at any time. The tutor will tell them if their solution so far is correct and if not where the first error is. If an error is found, the tutor will provide the same feedback about the error as the immediate feedback tutor does. Our experience is that as much as 80 percent of the time students wait until they have finished their solution before they request feedback. So by their own choice they tend to turn this into a delayed feedback condition.

Flag Tutor — Whenever the student makes an error, the error is immediately highlighted by the tutor. However, students are not required to fix the error immediately, they can ignore the error and continue to code. Indeed if they can come up with a code that works despite "errors" flagged on the screen (i.e., if they generate a solution that the tutor does not recognize) they are credited with solving the problem. This is true in the no-tutor and demand-feedback tutors also. At any time the student can go back and request feedback on an error, in which case the same message is presented as would be in the immediate-feedback tutor. Our experience is that about 10 percent of the time students will ignore the error signal at least initially and continue coding, about 15 percent of the time they will request the tutor's explanation of the error signal, and the other 75 percent of the time they will try to correct their code without any information from the tutor as to the nature of their error or what the correct code would be. Thus, students tend to turn this into an immediate correction tutor but one in which they only receive minimal feedback from the tutor.

Effects of Tutoring Modalities

More information about these various tutors can be found in Corbett and Anderson (1990) and Corbett and Anderson (1989). We find that time to complete a fixed set of exercises is inversely related to the amount of influence exercised by the tutor. Students take longest in the no tutor condition and are fastest with the immediate feedback tutor. The demand-feedback tutor, and error-flagging tutor fall in between, with the demand feedback condition taking somewhat longer of the two. The effects, comparing the extremes, can be as large as a 3 to 1 difference in times. There tends to be no difference among the three tutor conditions in final achievement measured by post-tests although the no-tutor condition sometimes performs worse. This fact is very significant to our understanding of the learning process. Students in the three tutor conditions are going through very different trajectories (and taking very different amounts of time) to reach essentially the same final solutions. Students in the no-tutor condition, however, sometimes fail to reach a correct solution at all. It seems that learning is a function of the solutions students achieve and not the process by which they achieve it—students do achieve higher levels if they solve more problems under any discipline.

This pattern of results provides an important confirmation of the ACT* theory. Recall that initial production formation takes place by analogy to an example. Thus, the ingredient for learning is a product, the solved example, from which the analogy can take place. Once the production is in place, further learning is in response to a process—further use of the production leads to further strengthening. Since learning is both in response to a product (the initial compilation) and in response to a process (subsequent strengthening) it might not be obvious why ACT* predicts that the number of problems should be the critical variable. This requires going into more detail as to what happens in the ACT* theory in particular situations:

When solving their first problems in a lesson, students are in a state in which they have adequately studied the instructional examples so that they can perform about 50 percent of the new production rules without error. This 50 percent will be learned and compiled into production rules in all conditions without hitch. The compilation will occur in the same way in all conditions. The remaining 50 percent will be learned when the student comes to an understanding of one of the problems which involves that production and uses that problem as the example for a later problem. This 50 percent of the learning will therefore be based on the products of later problem solving episodes. Since, in all tutor conditions students come essentially to the same solutions and understanding of these solutions, there would be no difference in the learning involved in this 50 percent either. Once the productions are learned they are strengthened on each subsequent trial that they apply. Since the tutoring conditions only differ in how they respond to errors, strength will accumulate with correct rule applications across conditions in the same way.

The differences among conditions can have large consequences when the student makes an error and has to correct it. In conditions that provide little guidance the student can spend a lot more time finding out how to correct errors. However, according to ACT* students do not learn from errors or error correction.[3] The only thing they learn when an error is made is the final correct code. Thus, the differences among the conditions are not relevant to learning as long as students come to the same understanding of the same correct code at the end of the correction episodes. Students do not always come to the same code and there can be other subtle differences among conditions but the learning consequences of the various conditions are substantially the same and so one would not expect to see substantial differences in learning outcome.

Learning Curves

It is also of interest to consider what the learning curves are like in the various conditions. We have collected some data on this issue. A difficulty in collecting such learning curves concerns the fact that only in the immediate feedback condition are subjects guaranteed to stay on an interpretable path of behavior to which we can apply model tracing. Our solution to this dilemma has been only to analyze that fragment of the data which is on an interpretable path. As soon as a student makes an error in an exercise, we stop trying to analyze any subsequent data for that function. In the flag tutor, 87 percent of the interactions are analyzed, in the demand-feedback tutor 70 percent are analyzed, and in the no-tutor condition 67 percent are analyzed. Also, we have only the data analyzed for three of the lessons; lessons 2, 3, and 7 from the LISP tutor curriculum, and only for 10 subjects per lesson. Therefore, we have a much smaller data base than for the analyses we reported earlier.

Figure 6 presents the change in error rate as a function of condition and Figure 7 presents the change in coding time. Clearly, we are getting learning

[3]That is, with respect to writing correct code. They may learn how to debug their code.

functions across all tutoring modalities. In the case of error rates, there appear to be no differences as a function of tutoring modality. With respect to times, there are initial speed advantages for the two delayed feedback conditions (no tutor, demand) relative to the two immediate feedback conditions (flag, immediate). There are several reasons to believe these time differences do not reflect real learning differences. First, this may reflect the selection artifact. We only look at data on a correct path and subjects may be on a roll, so to speak, in the delayed conditions whereas in the immediate conditions we are mixing in more difficult situations. Second, subjects may be more cautious in the two immediate feedback conditions. Finally, since there is no feedback given in the two delayed feedback conditions, subjects do not have the implicit positive feedback to process. It is interesting in this regard that subjects are slowest in the flag tutor condition where they have to discriminate the type case of the feedback to detect errors which are printed in bold.

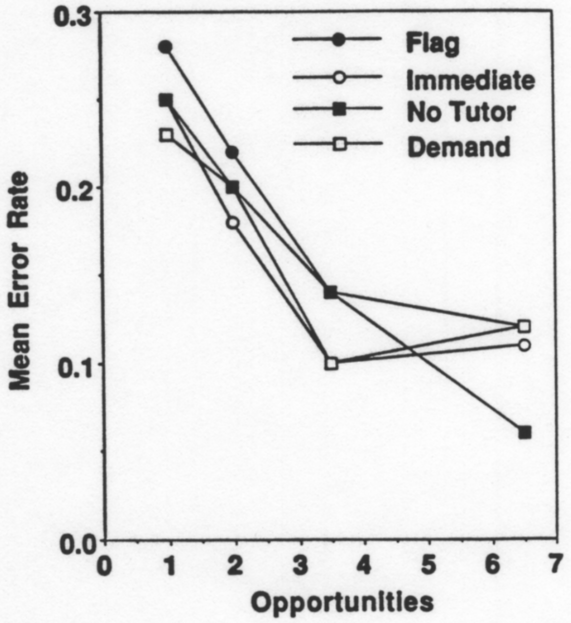


Figure 6. Learning in various tutor modalities as a function of amount of production practice: mean number of errors per coding attempt.

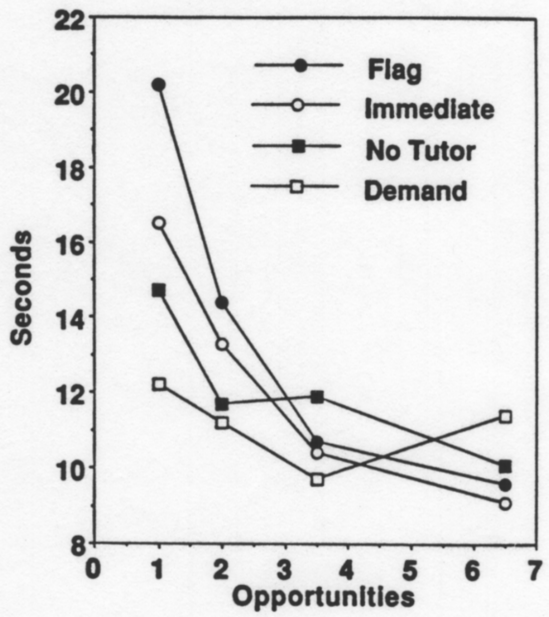


Figure 7. Learning in the various tutor modalities as a function of amount of production practice: mean time per correct coding attempt.

The basic similarity of the learning functions across tutoring conditions offers further support for the ACT* conception of the learning process. As discussed earlier, learning should be a function of the number of solutions the student has passed through and not of how the student passes through these solutions.

CONCLUSIONS

This chapter has reviewed the effects we have found in our research with the LISP tutor. Perhaps more interesting are the effects we did not find. There were no interesting effects of different content of problems, of individual differences, or of instructional style. The picture of learning a complex skill was every bit as simple as advertised at the beginning of the chapter. Learning individual production rules underlying a complex skill does not appear much different than learning simple paired associates. The resulting skill is complex because of the interrelations among the units but the learning of the units themselves is quite straight forward.

REFERENCES

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

Anderson, J. R. (1990). Analysis of student performance with the LISP tutor. In N. Fredericksen, R. Glaser, A. Lesgold, & M. Shafo (Eds.), *Diagnostic Monitoring of Skill and Knowledge Acquisition.* Hillsdale, NJ: Erlbaum, 27-50.

Anderson, J. R., Conrad, F.R. & Corbett, A.T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science, 13,* 467-506.

Anderson, J.R., Farrell, R. & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8,* 87-130.

Corbett, A. T. & Anderson, J. R. (1990). The effect of feedback control on learning to program with the Lisp Tutor. *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, 796-803.

Corbett, A. T. & Anderson, J. R., (1989). Feedback timing and student control in the LISP Intelligent Tutoring System, Artificial Intelligence and Education. *Proceedings of the Fourth International Conference on AI and Education,* 64-72.