Analysis of Student Performance with the LISP Tutor

John R. Anderson
Carnegie-Mellon University

The goal of this chapter is to present our first detailed analysis of student performance with the LISP tutor. First, we describe a little of our general theoretical orientation to the issues of intelligent tutoring. Second, we provide a description of the essential features of the operation of the LISP tutor. Third, we give some general description of characteristics of the data that are obtained with the LISP tutor.

INTELLIGENT TUTORING AND ITS RELATION TO COGNITIVE THEORY

Research on intelligent tutoring serves two goals. The obvious goal is to develop systems for automating education. Private human tutors are very effective (Bloom, 1984), and it would be nice to be able to deliver this effectiveness without incurring the high cost of human tutors. However, a second and equally important goal is to explore epistemological issues concerning the nature of the knowledge that is being tutored and how that knowledge can be learned. We take it as an axiom that a tutor is effective to the extent that it embodies correct decisions on these epistemological issues.

We chose intelligent tutoring as a domain for testing out the ACT theory of cognition (Anderson, 1983). It was a theory that made claims about the organization and acquisition of complex cognitive skills. The only way to adequately test the sufficiency of the theory was to interface it with the acquisition of realistically complex skills by large populations of students. When we read Intelligent Tutoring, edited by Sleeman and Brown (1982), it became apparent that the book’s
authors were explicitly or implicitly performing such tests of theories of cognition and that it was an appropriate methodology for testing the ACT\textsuperscript{T} theory. Fundamentally, the tutoring methodology is predicated on the assumption that one understands a skill and its acquisition. The success of the tutor constitutes a direct test of the sufficiency of the underlying theory.

The ACT\textsuperscript{T} theory has been used to construct performance models of how students actually execute the skills that are to be tutored and learning models of how these skills are acquired. The performance model is used in a paradigm we call model tracing in which we try to follow in real time the cognitive states the student goes through in solving a problem. The power of our tutoring approach depends critically on the success of our model-tracing apparatus to correctly interpret the cognitive states of students. When we interrupt students to provide instruction, that instruction is given with respect to an assumed mental state. If this model’s assumptions are wrong, the instruction will be off the mark.

The LISP tutor (Anderson & Reiser, 1985) was developed as an instantiation of this model-tracing methodology and serves to test our theory of skill acquisition (Anderson, 1982; Anderson, Farrell, & Sauer, 1984) in two ways. First, it is a sufficiency test of the theory. The fact that a system of this variety does serve to teach LISP programming skills stands as a general confirmation of the theory. Second, it is also a tool to test predictions of the theory.

Although our research is in LISP programming and its tutoring, we are using this as a vehicle to test some fundamental issues about the nature of problem-solving skills and its acquisition. Among these issues are the following:

1. **Skill Representation.** How should a skill be presented? ACT\textsuperscript{T} assumes a representation as a set of production rules.

2. **Procedural Versus Declarative Knowledge.** What is the relationship between the declarative knowledge (which is the original instruction) and the highly proceduralized form that it finally achieves?

3. **Performance Limitations.** How do fundamental performance limitations like working-memory limitations impact on skill performance and skill acquisition?

4. **Organization and Control.** How is the knowledge underlying problem-solving skill organized and controlled to permit coherent problem solving?

5. **Skill Modification.** How is one’s knowledge modified to effectively reflect experience? This issue is closely tied up with the issue of feedback.

6. **Mechanisms of Skill Acquisition.** Last but hardly least, what are the fundamental mechanisms of skill acquisition?

LISP programming is an excellent domain for studying these issues because it offers a complex but relatively well-understood domain. The tutor is an excellent tool because it brings control and experimental rigor to what would otherwise be a rather free-form learning experience.

## THE LISP TUTOR

The LISP tutor currently teaches a full-semester, self-paced course at Carnegie-Mellon University. It covers all the basic concepts in LISP. It is the first instance of a practical piece of intelligent tutoring being widely used, and it has been shown to lead to improvement in performance. Roughly, students working on problems with the LISP tutor get one letter grade higher on final exams of general competence than students not working with the LISP tutor (Anderson & Reiser, 1985). It should also be noted that students working with private human tutors have been shown to outperform students with the LISP tutor. So it is by no means a utopian system, but it may claim some pedagogical effectiveness.

Table 2.1 contains a dialogue with a student coding a recursive function to calculate factorial. This does not present the tutor as it really appears. Instead, it shows a “teletype” version of the tutor where the interaction is linearized. In the actual tutor the interaction involves updates to various windows. In the teletype version, the tutor’s output is given in normal type whereas the student’s input is shown in bold characters. These listings present “snapsots” of the interactions; each time the student produces a response, his input is listed along with the tutor’s response (numbered for convenience). The total code as it appears on the screen is shown, although the student has added only what is different from the previous code (shown in boldface type). For instance, in Line 2 he has added “zerop” as an extension of “(defun fact (n) (cond (..., )...”

In the first line, when the subject typed “(defun”, the template (defun <name> <parameters> <body>) appeared. The terms in <--> angle brackets denote pieces of code he will supply. The subject then filled in the <name> slot and the <parameters> slot and had started to fill in the <body> slot. Note that at all points parentheses are balanced and syntax is checked.

Although the student has some difficulty with the syntax of the conditional tests in Lines 1 and 2, he basically codes the terminating case for the factorial function correctly. Typically, we find students have little difficulty with terminating cases but have great difficulty with recursive cases. The dialogue after Line 3 illustrates how the tutor guides the student through a design of the recursive function. Basically, it leads the student to construct a couple of examples of the relationship between fact (n) and fact (n−1) and then get the student to identify the general relationship. Figure 2.1 shows the screen image at a critical point in the design of this function.

The dialogue after this point shows two errors students make in defining recursive functions. The first, in Line 4, is to call the function directly without
TABLE 2.1

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

1. (defun fact (n)
   (cond ((equal n 0) 1)
          (cond (zero n) 1)
          (let ((n-1 (1- n)))
            (fact n-1)))))

   I assume that you are trying to test if the number argument is equal to the number zero. There is a special function for testing equivalence to zero. Try to use that function instead of EQUAL.

   What code can you write to test if the condition for the terminating case is true? Try writing a predicate.

2. (defun fact (n)
   (cond ((zero n) 1)
          (cond (1 n) (fact (1- n)))
          (let ((n-1 (1- n)))
            (fact n-1)))))

   Did you mean zero? (y or n)?

3. (defun fact (n)
   (cond ((zerop n) 1)
          (cond (1 n) (fact (1- n)))
          (let ((n-1 (1- n)))
            (fact n-1)))))

   Is the value of fact in the recursive cases calculated by:

   PRESS:  IF YOU WANT TO:
   1. Calling fact recursively with n.
   2. Multiplying n by fact of n minus 1.
   3. Getting 1 2 \ldots \cdot n and then multiplying the results.
   4. have the tutor choose

   Menu choice: 3

   That is a reasonable way to think of doing factorials but it is not a plan for a recursive function. Since you seem to be having trouble with the recursive cases let us work through some examples and figure out the conditions and actions for each of these cases.

   The recursive relationship is the relationship between fact called with n and fact called with:

   PRESS:  IF YOU WANT TO:
   1. n minus 1
   2. n equal to 0
   3. n plus 1
   4. have the tutor choose

   Menu choice: 1

   We will consider some concrete examples. We will characterize the recursive relationship between fact called with 1 and fact called with 0

   What is the value of (fact 1)?
   Answer: 1

   What is the value of (fact 0)?
   Answer: 1

   EXAMPLES
   fact (n)    fact (n - 1)
   A. (fact 1) = 1    (fact 0) = 1

---

TABLE 2.1 (continued)

We will consider some more concrete examples. We will characterize the recursive relationship between fact called with 3 and fact called with 2

What is the value of (fact 3)?
Answer: 6

What is the value of (fact 2)?
Answer: 2

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

Since there is only one recursive case characterized by the examples you have just worked out the condition in this case can be just coded at T. We will just focus on characterizing the action in this case.

In examples A and B what do you have to do to get the result of fact called with n?

PRESS  IF YOU WANT TO:
1. Multiply n by one less than n.
2. Multiply n by fact of one less than n
3. Add n to the result of fact called with one less than n.
4. have the tutor choose

Menu choice: 2

We have now worked through a series of examples that illustrate the recursive relation. We will now return to the goal of trying to produce LISP code that specifies the recursive relation.

4. (defun fact (n)
   (cond ((zerop n) 1)
          (t (fact))))

   You will have to use fact at some point but right now you want to multiply.

5. (defun fact (n)
   (cond ((zerop n) 1)
          (t (times n (fact n))))))

   You cannot just code n at this point. What you want to do is subtract one from n.

6. (defun fact (n)
   (cond ((zerop n) 1)
          (t (times n (fact (difference))))))

   Difference would work in this case but it is more appropriate to use the function SUB1 when you are subtracting 1 from a number.

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

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

(continued)
TABLE 2.1 (continued)

THE LISP WINDOW

```
= > (trace fact)
(fact)
= > (fact 3)
1 <Enter> fact (3)
2 <Enter> fact (2)
3 <Enter> fact (1)
4 <Enter> fact (0)
4 <Exit> fact 1
3 <Exit> fact 1
2 <Exit> fact 2
1 <Exit> fact 6
```

In examples A and B what do you have to do to get the result of fact called with n?

PRESS: IF YOU WANT TO:
1. Multiply n by one less than n.
2. Multiply n by fact of one less than n.
3. Add n to the result of fact called with one less than n.
4. Have the tutor choose.

Menu Choice: 2

CODE FOR fact

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

EXAMPLES

<table>
<thead>
<tr>
<th>Fact (n)</th>
<th>Fact (n-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (fact 1) = 1</td>
<td>(fact 0) = 1</td>
</tr>
<tr>
<td>B (fact 3) = 6</td>
<td>(fact 2) = 2</td>
</tr>
</tbody>
</table>

FIG. 2.1. A representation of the screen image after line 3 in Table 2.1

combining the recursive call with other elements. The second, in Line 5, is to call the function recursively with the same argument rather than a simpler one.

After the student finishes coding the function he goes to the LISP window and experiments. He is required to trace the function, and the recursive calls embed and then unravel. Figure 2.2 shows the screen image at this point with the code on top and the trace below it.

Features of the Model-Tracing Methodology

The example just shown illustrates a number of features of the model-tracing methodology:

1. The tutor constantly monitors the student’s problem solving and provides direction whenever the student wanders off a solution path.
2. The tutor tries to provide help with both the overt parts of the problem solving and the planning. However, to address the planning, a mechanism had to be introduced in the interface (in this case menus) to allow the student to communicate the steps of planning.
3. The interface handles details like syntax checking, which are irrelevant to the problem-solving skill being tutored.
4. The interface is highly reactive in that it does make some response to every symbol the student enters.

The Mechanics of Model Tracing

Sitting within the tutor is a production system consisting of hundreds of ideal and buggy rules. The following are examples of a production rule that codes APPEND and two bugs. Associated with each bug is an example of the feedback we would present to the student should the student display that bug:
Production Rule in Ideal Model:

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

Related Bugs:

IF the goal is to merge LIST1 and LIST2 into a single list
THEN use the function LIST and set subgoals to code LIST1 and LIST2

You should combine the first list and the second list, but LIST is not the right function. If you LIST together (a b c) and (x y z), for example, you will get ((a b c) (x y z)) instead of (a b c x y z). LIST just wraps paren around its arguments.

IF the goal is to merge LIST1 and LIST2 into a single list
and LIST1 = LIST2
THEN use the function TIMES and set subgoals to code LIST1 and the number 2

You want to put together two copies of the same list, but you can’t make two copies of a list by using the function TIMES. TIMES only works on numbers. You should use a function that combines two lists together.

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

THEORETICAL PREMISES UNDERLYING THE LISP TUTOR

The LISP tutor is predicated on a number of assumptions about the cognitive architecture, about the nature of a complex skill like programming, and the nature of its acquisition. The actual student models are implemented in the GRAPES production system, which is a partial simulation of the ACT’ theory, and the tutoring interactions are based on assumptions about what the critical factors are underlying skill performance and execution. Without claiming to be exhaustive,
working-memory failures slow down learning and cause incorrect things to be learned.

8. Minor Relations Between Strength and Learning. As declarative knowledge or procedural knowledge is practiced it is strengthened. Stronger declarative knowledge is the more active and hence is more likely to be in working memory. Stronger productions are the more rapidly matched to what is in working memory. This means that it is more likely that well-encoded knowledge would overcome the limitations of working memory and would successfully apply. The first-order effect of strength would be on speed of performance, but it would have a second-order effect on working memory (and basically, accuracy). Although these strength factors do affect speed and accuracy performance measures, they should have little direct effect on production learning. Even when strength affects maintenance of information in working memory, these strength effects would occur after productions are learned (at least in the LISP tutor).

In total, these eight assumption constitute some profound claims about the course of skill acquisition. They also paint a rather simple picture of the process. The critical question concerns what the actual data have to say about the theory.

DATA ANALYSIS

As students interact with the LISP tutor we collect a total record of all of their responses and the time at which they complete these responses. A student’s response is defined as something the tutor reacts to. Usually this amounts to a LISP symbol, when typing code, or a menu selection. We do not collect data at the level of inter-keystroke times; that level of data collection would be just too voluminous. We also record the times at which the tutor prints prompts and the identities of these prompts. Finally, in the data files we have records of the correct or buggy productions that the LISP tutor ascribed to students responses.

These data can easily be transformed into a form wherein we organize the data by the sequence of productions that the tutor assumed fired and associate times with the firing of each production. This amounts to ascribing a theoretical interpretation to the data in terms of the simulation program used by the tutor. This is a level of analysis that graduate students used to spend dissertations to achieve for a few subjects solving a few problems. We can achieve it automatically for a class full of students doing a semester’s worth of work.

A key feature of the LISP tutor is that it keeps students on a correct path of problem solution. The tutor may be prepared to follow the student on many hundreds of ways of solving a problem, but this is much less than the thousands of ways, mostly incorrect, that students have been observed trying to solve a problem. At any point in time there is a set of possible next correct productions that the tutor is prepared to have the student execute. One of a possible set of things can happen:

1. The student generates an action that matches the action produced by one of the correct productions. The tutor assumes the production that generated this action is the one that fired in the student’s head and continues to monitor for the production that follows that.

2. The student makes an error, the tutor responds to that error with feedback, and then the student generates an action that corresponds to a correct production. The tutor assumes that the feedback enabled the student to figure out the correct answer, and the student is back on track.

3. The student asks for the next step either immediately or after an error. The tutor provides the student with an explanation of the correct step and then provides the piece of code that corresponds to that step. The assumption is again that this explanation was sufficient to get the student back on track and the student is in the same mental state as the tutor.

4. The student generates three errors. In this case the tutor offers the same explanation as it would have had the student requested it and provides the next correct action.

The major complication hidden in this description concerns a dichotomy in the types of errors emitted. About 80% of the errors match buggy productions in the LISP tutor, and it is able to generate feedback specific to that error. The other 20% of productions are not matched, and the feedback is a default ("I don’t understand that"). The majority of the undiagnosed responses are clearly erroneous on the student’s part. Only on rare occasions do we observe a student with a solution that the tutor has not thought of.

The underlying assumption in these interactions is that before doing the next piece of the problem the student and the tutor are in the same mental state. From informal observations we know there are occasions when this is not true, for instance, when the student either misunderstands the problem statement or the feedback given by the tutor. This means that there is a certain noise built into our error attribution. We attribute an error to production applying in state X whereas a different production might be applying in state Y. It is difficult to know the magnitude of this "noise" in the data and how it compares with noise in other data. One test is the reliability and interpretability of the data obtained with the LISP tutor. Any experiment has noise in the data, and as in any experiment, we use the ratio of variance between conditions to variance within conditions to decide what effects could not be due to experimental noise.

Given this data base, there are two basic categories of data to collect from the LISP tutor: error measures and time measures. Both of these categories break down into two basic subtypes. For errors, we can calculate the probability of
judged weak on certain productions, they are required to do remedial problems, which offers additional practice on these productions. Students need, on average, about 15% extra remedial problems, although there is large individual variation. We ignore remedial problems in the analyses that we report.

Thus our data in its finest grain can be broken down according to the dimensions of lesson, production, and opportunity for that production within the lesson. Crossed with these are the four dependent measures listed earlier—namely, time per production, time for correct production firings, probability of a correct production firing, and mean number of errors for a production firing.

2. STUDENT PERFORMANCE WITH LISP

RESULTS

I would like to present the majority of the data organized by lesson and aggregated over production. However, to explain this aggregation process and to get a better feel for the data, I believe it would be worthwhile to look at one lesson in more detail. Lesson 2 is appropriate for this purpose. Table 2.2 lists the productions we monitored. The first 9 were first introduced in Lesson 2 and the last 9 had been introduced in the previous lesson.

Fig. 2.3 and 2.4 plot the performance on the new productions for this lesson. Fig. 2.3 plots the times for correct use of the production (where the maximum value was set at 200 seconds), and Fig. 2.4 plots the mean number of errors, where this statistic has a maximum of three. We plotted just times for correct productions in order to get a measure that is independent of errors. As noted earlier, we also analyzed time aggregated over corrects and errors. This measure does not seem to reveal any additional insights. We choose to analyze total number of errors per opportunity in Fig. 2.4 rather than probability of error because we believe it is a better measure of student difficulty. Students often make single errors and correct them as slips. Every time an error cannot be corrected, it is further evidence that a student has a fundamental difficulty.

We have plotted these measures as a function of the times a production has been tested in the session. Both scales are logarithmic. Different productions occurred a different number of times, but we calculated a weighted average to provide our best estimate of how the mean changed with practice. As can be seen, this mean shows a marked drop-off from first to second test and a very modest decline after that. The average improvement from first to second trial is almost 50% for time and over 50% for accuracy. We plot this on a log–log scale to make the point that this drop-off is not just part of the power-law improvement normally seen for a skill. Figures 2.5 and 2.6 plot the lesson averages for lessons

\footnote{The average for the first opportunity is the average of the logarithms. The nth average, \( a_n \), is calculated from \( n-1 \)st average \( a_{n-1} \) as \( a_n = a_{n-1} + i \), where \( i \) is the average change in the logarithm values of those productions for which there is both a \( n-1 \)st and \( n \)th observation.}
TABLE 2.2
Productions Monitored in Lesson 2

1. specify-function-name: Codes the symbol corresponding to the production name.
2. specify-function-params: Codes the parameters of the function.
3. code-nil: a production for coding the special symbol nil.
4. code-append: a production to generate the LISP combining function append.
5. code-reverse: a production to generate the LISP function reverse which codes the reverse of a list.
6. code-cosine: a production to code the LISP function cosine.
7. code-sine: a production to code the LISP function sine.
8. code-square: a production that codes the square of a number by taking the product of two numbers.
9. check-arg: a production that codes an argument to a function called a parameter in the function definition.

The following productions were introduced in lesson 1 but reappear in lesson 2

10. code-car: a production to code the LISP function car that gets the first element of a list.
11. code-cdr: a production to code the LISP function cdr that gets the rest of a list.
12. code-second: a production that gets the second element of a list. This is coded as a car-cdr combination. This is treated separately because students have difficulty with precedence of unary operators.
13. code-cons: a production to code the LISP combiner cons that inserts its first argument in front of the list that is its second argument.
14. code-list: a production to code the LISP combiner list that wraps parentheses around its arguments.
15. code-divide: a production that codes the LISP function quotient that takes the quotient of two numbers.
16. code-difference: a production that codes the LISP function difference that subtracts its second argument from its first argument.
17. code-times: a production that codes the LISP function times that multiplies its arguments.
18. code-number: a production that codes a number argument to a function.

2, 3, and 5 and the average of these averages (lesson 1 measures are peculiar on the first trials because typically the teacher is coaching, and there are very few new productions introduced on lessons 4 and 6). It makes even clearer the point that the drop-off from the first to second trial is discontinuous.

Note that the rate of improvement after the first trial is basically linear in these two logarithmic measures. This implies a power function relating either time or errors to amount of practice. This is what is typically found in studies of practice. However, the clear discontinuity from trial 1 to trial 2 is something that has not been examined in detail until now. It is consistent with the knowledge compilation mechanism in ACT, basically a one-trial learning mechanism. One might attribute the drop-off in errors to students just debugging their misconceptions from reading. Thus, it is significant that this discontinuity also shows up in times for errorless trials as well as number of errors.

Another question concerns how performance changes on productions across lessons. Figure 2.7 is an attempt to analyze this. We have plotted performance on

FIG. 2.3. Mean times for correct coding of the actions corresponding to 9 productions introduced in lesson 2. See text for explanation of the productions.

FIG. 2.4. Mean number of errors in coding the actions corresponding to 9 productions introduced in lesson 2. See text for an explanation of the productions.
last occurrence in lesson $n-1$, first performance in lesson $n$, and last performance in lesson $n$. Figure 2.7 plots these patterns for the lesson pairs 1 and 2, 2 and 3, 3 and 4, and 5 and 6 (there are few shared productions between 4 and 5). The average of these lesson averages makes the pattern even more apparent. There is perhaps a little forgetting between lessons but considerable improvement within lessons. This is more apparent with the accuracy measure than the time measure. Although they are not exactly the same productions plotted in each curve, note that the curves tend to get lower across lessons, also consistent with a gradual improvement with practice.

One might wonder how much support these analyses offer for the existence of the production rules assumed by the LISP tutor. The apparent regularity of the data is consistent with the view that the LISP tutor provides the psychologically correct decomposition of the skill. However, these production rules do tend to correspond to pieces of code in LISP. For instance, code-car corresponds to typing car and check-arg corresponds to typing a variable name. What if we simply monitored how accurately students wrote these pieces of code and ignored the production-rule analysis? Although correlated, it is not the case that a code-based analysis is identical to a production-rule analyses. This is because in some cases there is a many-to-one relationship between production rules and types of LISP code. For instance, although code-car usually is responsible for generating car, there is a special production, code-second, that corresponds to car when we
are composing a car-cdr sequence. The first time subjects typed car as code-second in lesson 1 they made .68 errors. We can go back to their error rate on the preceding code-car and find it was just .41. This increase from .41 to .68 deviates from the general improvement within a lesson. All other comparisons we looked at show the same trend. Coding cond to terminate an iteration is different than coding cond generally. Subject make 1.05 errors in the cond in iteration compared to .32 errors for the previous cond. Coding a number to initialize a variable is different than passing it as an argument to a function. Subjects make 1.64 errors on their first use of a number to initialize compared to .24 errors in the previous use of a number. Using setq to initialize a local variable is controlled by a different production than the one that uses setq to initialize a global variable. Subjects make .89 errors on their first local variable setq compared to .29 for their previous global variable setq. Coding a variable that is a function parameter is governed by a different production than the one that codes a global variable. The difference in error rate is 2.18 versus .38. Coding a parameter is different in turn from coding a local variable in a function. By the time subjects get to local variables, their error rate for parameters has dropped from 2.18 to .03. Their error rate in the first local variable is .84. The upshot of all these comparisons is that subject improvement is better defined in terms of the LISP tutor productions than surface code. The regularity of the data really is evidence for the LISP Tutor's production-rule analysis.

The overall pattern of data displayed in these graphs is consistent with the ACT learning mechanisms. There is the discontinuous point of learning from first trial to second due to knowledge compilation, a power-law growth in strength (and hence performance measure) with practice, and some forgetting (loss of strength) between lessons. It should also be noted that, with the exception of the first trial discontinuity, we find nothing in LISP learning that would be surprising from the results of verbal learning.

**INTER-PRODUCTION CORRELATIONS**

Another question of interest is how well does performance on one production correlate with performance on another production. We looked at the measures of number of errors in calculating these correlations because they prove to yield the largest correlations. In one analysis we looked at patterns of correlation between lessons 1 and 2, 2 and 3, and 3 and 4. We calculated how well a production from one lesson correlated with itself on another lesson. Basically, this involves getting mean number of errors made on a production for each lesson for each subject and looking for a correlation over subjects for each production.

The average correlation of all the productions that repeated across trials was .26. This might not seem very large, but because most productions only occur a few times in a lesson and the individual measures are inherently quite noisy, correlations will be low. Under one model the expected maximum correlation is only .33.2

We gathered two other correlation measures between lessons. First, we broke the productions on individual lessons into those that deal with list operations and those that do not. Thus, we divided the productions into disjoint sets. We looked at correlations within sets (excluding correlations between the same production reported in the preceding paragraph) and between sets. The average correlation was .17 both within and between sets. These numbers are significantly lower than the correlation between the same production on both lessons, and obviously the two correlations were not significantly different. Intuitively, this is surprising because it indicates that there is no tendency for productions of the same type to cluster. Both correlations are quite significantly different than zero, of course, indicating some systematic individual differences among subjects.

We also calculated correlations among productions within the same lesson, excluding correlation of a production with itself, which would be 1. The average within-lesson correlation is .23, which is significantly higher than the between-lesson correlation and does not significantly differ from the between-lesson, same-production correlation. This indicates some tendency for subjects to have good lessons or bad lessons, which is not surprising. Again, if we break up these within-lesson correlations into correlations within list and non-list productions and correlations between list and non-list productions, there is no difference.

The failure to find evidence for a clustering among productions involved with list operations is surprising. We had strongly suspected that some subjects would do well on all list operations and some would not. Although this is in fact the case, it appears that the interproduction correlations are not higher than found between apparently unrelated productions. Thus, there appears to be a general ability factor, but not one associated with list operations.

One possible conclusion is that productions do not break up into thematic clusters, but it is possible that we simply did not intuit properly the factors that would cause productions to cluster. To see if this was the case, we subjected the data from the six lessons to factor analyses. We took the matrix of subject-by-production error means for each lesson and submitted it to a standard factor-analysis program. We then looked at the first two factors extracted for each lesson. We have looked at the other three performance measures, but error totals

---

2We have on average about four observations per production per lesson. Assume that each observation is a binomial with a probability \( p \) of generating an error. Assume that for half the production \( p = .33 \) and for the other half \( p = .67 \). The correlation expected between number of errors for four observations on day 1 and four observations on day 2 is .33.
consistently show largest variance accounted for in the first two factors. The data from the lessons structured as follows.

1. Number of Productions involved in the factor analysis. In all cases there are 34 subjects.
2. Variance accounted for by first factor, second factor, and third factor. The variance accounted for by the second factor indicates what we gain by including it, and the variance accounted for by the third factor indicates what we lose by excluding it.
3. Which productions loaded most heavily on each factor after rotation. If a production loads heavily on both, we report the factor it loads most heavily on. We exclude productions with a factor loading of less than + .6. We report the magnitude of the loadings in parentheses.

**Lesson 1—15 Productions:**

<table>
<thead>
<tr>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Factor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>41% of the variance</td>
<td>13% of the variance</td>
<td>10% of the variance</td>
</tr>
<tr>
<td>global variables (.81)*</td>
<td>quotient (.74)*</td>
<td></td>
</tr>
<tr>
<td>list function (.79)*</td>
<td>formula for square (.61)*</td>
<td></td>
</tr>
<tr>
<td>cdr function (.77)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coding a number (.76)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>first arg to eq (.75)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cons function (.73)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>setq function (.72)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quoted constant (.63)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>plan for second element in a list (.61)*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lesson 2—19 Productions:**

<table>
<thead>
<tr>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Factor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>34% of the variance</td>
<td>12% of the variance</td>
<td>8% of the variance</td>
</tr>
<tr>
<td>coding a number (.83)*</td>
<td>reverse function (.81)*</td>
<td></td>
</tr>
<tr>
<td>cdr function (.81)</td>
<td>coding parameters in function</td>
<td></td>
</tr>
<tr>
<td>coding function name (.78)*</td>
<td>definition (.72)*</td>
<td></td>
</tr>
<tr>
<td>difference function (.68)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quotient function (.65)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>formula for square (.64)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defun function (.62)*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lesson 3—31 Productions:**

<table>
<thead>
<tr>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Factor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>28% of the variance</td>
<td>11% of the variance</td>
<td>10% of the variance</td>
</tr>
<tr>
<td>coding a number (.86)*</td>
<td>coding an else (i) clause (.71)*</td>
<td></td>
</tr>
<tr>
<td>cond function (.82)*</td>
<td>coding a parameter (.68)</td>
<td></td>
</tr>
<tr>
<td>coding a constant (.78)</td>
<td>member function (.65)*</td>
<td></td>
</tr>
<tr>
<td>lessp function (.75)*</td>
<td>reverse function (.64)</td>
<td></td>
</tr>
<tr>
<td>cons function (.72)*</td>
<td>or function (.63)*</td>
<td></td>
</tr>
<tr>
<td>null function (.70)*</td>
<td>not function (.62)*</td>
<td></td>
</tr>
<tr>
<td>global variable (.62)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lesson 4—18 Productions:**

<table>
<thead>
<tr>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Factor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>28% of the variance</td>
<td>12% of the variance</td>
<td>10% of the variance</td>
</tr>
<tr>
<td>equal function (.79)</td>
<td>case within a cond (.68)</td>
<td></td>
</tr>
<tr>
<td>car function (.74)</td>
<td>coding an else (i) clause (.66)</td>
<td></td>
</tr>
<tr>
<td>last function (.74)</td>
<td>coding a parameter (.62)</td>
<td></td>
</tr>
<tr>
<td>cdr function (.74)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lesson 5—28 Productions:**

<table>
<thead>
<tr>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Factor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>45% of the variance</td>
<td>10% of the variance</td>
<td>6% of the variance</td>
</tr>
<tr>
<td>Prog function (.90)*</td>
<td>code a local variable (.83)*</td>
<td></td>
</tr>
<tr>
<td>numberp function (.80)*</td>
<td>print function (.74)*</td>
<td></td>
</tr>
<tr>
<td>case within a cond (.77)</td>
<td>coding a parameter (.72)</td>
<td></td>
</tr>
<tr>
<td>coding a loop tag (.73)</td>
<td>plus function (.69)</td>
<td></td>
</tr>
<tr>
<td>not function (.73)</td>
<td>cond function (.69)</td>
<td></td>
</tr>
<tr>
<td>go function (.72)</td>
<td>read function (.69)</td>
<td></td>
</tr>
<tr>
<td>resetting a variable (.67)*</td>
<td>coding the variable to be reset (.68)*</td>
<td></td>
</tr>
<tr>
<td>coding a constant (.67)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return function (.64)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>initializing a variable (.62)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>difference function (.61)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lesson 6—27 Productions:

Factor 1

29% of the variance
code a local variable (.88)
coding a parameter (.81)
coding a loop tag (.76)
equal function (.76)
coding repeat tag (.76)
coding a variable (.68)
plus function (.68)
coding a number (.63)
difference function (.62)
greaterp function (.61)

Factor 2

11% of the variance
code an initial value for iteration (.83)*
initialize a variable for iteration (.83)*

Factor 3

7% of the variance
prog function (.76)

WHAT DOES IT MEAN?

We could not make a great deal of sense out of this pattern. Productions were not apparently clustering according to any semantic feature. In an attempt to make sense of this we took each subject’s factor scores for the two factors for each lesson and thus got twelve measures for each subject. We subjected these to a factor analysis to determine which lesson factors would cluster together. The first meta-factor extracted accounted for 36% of the variance and the second meta-factor 16% of the variance. The third meta-factor (which we will ignore) accounted for 14% of the variance.

The following factors loaded on the first meta-factor—factor 2, lesson 1; factor 2, lesson 2, factor 2, lesson 3; factors 1 and 2, lesson 5; and factor 2, lesson 6. The following factors loaded on the second meta-factor—factor 1, lesson 3; factors 1 and 2, lesson 4; and factor 1, lesson 6. Factor 1 from lesson 1 and factor 1 from lesson 2 did not load on either meta-factor, suggesting that these may reflect peculiar start-up features in dealing with LISP.

It only became apparent after considerable inspection what unites the factors categorized together under each meta-factor. Twenty-two of the 34 productions organized under meta-factor 1 were introduced in that lesson, whereas only 3 of the 23 productions organized under meta-factor 2. The new productions are starred in the listings just given. Thus the first factor is basically an acquisition factor because it reflects performance in productions being acquired in that lesson, whereas the second is a retention factor because it reflects performance in productions presumably already acquired in previous lessons, and, thus, deficits must be due to forgetting. This helps explain why the actual clustering of productions seemed arbitrary semantically and why the clusters do not stay constant across lessons. The second factor correlates .62 with math SAT; the first factor only correlates .03. Neither factor correlates with verbal SAT or grade-point average at Carnegie Mellon.

It seemed worthwhile to see how these categories did at predicting performance on paper-and-pencil tests. We used the midterm and final exam tests of the 34 students. The midterm was administered right after completing lesson 6, whereas the final was administered after 6 more lessons with the tutor. We have not had the opportunity to analyze the data from these 6 lessons. We took subjects’ factor scores on these two meta-factors and classified them into above or below the median. This gave us ten subjects in both the high–high and low–low categories and seven in both the high–low and low–high categories.

Table 2.3 presents the data from the midterm exams in terms of scores out of 24, and Table 2.4 presents the final grades out of 28 similarly classified. There are significant effects of both factors on midterm grades—$F(1,30)=5.1$ for acquisition and $F(1,30)=6.4$ for retention. The interaction is not significant—$F(1,30)=2.8$. Again, only the main effects were significant on final exam—acquisition factor with $F(1,30)=6.3$ and retention factor with $F(1,30)=9.7$.

<table>
<thead>
<tr>
<th>TABLE 2.3</th>
<th>Midterm Grades</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low Acquisition Factor</strong></td>
<td><strong>High Acquisition Factor</strong></td>
</tr>
<tr>
<td>Low Retention Factor</td>
<td>3.8 ± .9</td>
</tr>
<tr>
<td>High Retention Factor</td>
<td>10.1 ± 1.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 2.4</th>
<th>Final Exam</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low Acquisition Factor</strong></td>
<td><strong>High Acquisition Factor</strong></td>
</tr>
<tr>
<td>Low Retention Factor</td>
<td>9.8 ± 1.7</td>
</tr>
<tr>
<td>High Retention Factor</td>
<td>14.2 ± .9</td>
</tr>
</tbody>
</table>
CONCLUSIONS

In my opinion this analysis of student behavior is marvelous for the lawfulness and simplicity of the picture it paints. It is particularly consistent with the ACT theory of production-system analysis of skill acquisition. The behavior is quite regular when analyzed with respect to the production rules used in the LISP tutor. It shows evidence for a one-trial learning episode consistent with knowledge compilation and a more gradual learning consistent with the power-law strengthening process.

Productions are independent and modular, as would be predicted by the ACT theory. The only production rules with which a particular production correlates especially strongly is with itself. This is consistent with the notion that each production is learned independently. In particular, we found no evidence that productions were being clustered because some subjects were having difficulty with list manipulations. Some productions are clearly more difficult than others and some subjects are clearly more capable than others, but there does not appear to be any interaction between the two factors. Subjects' overall ability impacts on how fast they learn all aspects of LISP, but except through this general ability factor the learning of one production is independent of the learning of others.

When we did an atheoretical factor analysis looking for evidence of an interaction, the only thing we found were two somewhat independent general-ability factors of acquisition and retention on which subjects could be sorted and on which old and new productions could be sorted. This result clearly does not compromise the basic cognitive assumption of independence of productions, although it raises some interesting questions about what the real nature of these two ability factors might be.

REFERENCES