

Production Systems, Learning, and Tutoring¹

John R. Anderson

Department of Psychology

Carnegie-Mellon University

Pittsburgh, PA 15213

¹The research reported in this paper was supported by Contract No. N00014-84-K-0064 from the Office of Naval Research and Grant No. IST-83-18629 from the National Science Foundation

The chapters in this book all document the role of production systems in modelling human cognition and learning. If they are in fact good models, then we might expect that they could form the bases for pedagogical applications. This paper will focus on the role of production system models in the design of a certain type of computer-based tutor. We have developed two instances of this type of tutor--one to tutor introductory LISP programming (Reiser, Anderson, & Farrell, 1985) and one to tutor doing proofs in geometry (Anderson, Boyle, & Yost, 1985). There is evidence that these tutors are effective relative to standard classroom instruction. However, we will see, the purpose of this research effort is as much to test the psychological reality of the production-system architecture as it is to yield practical application^s. The specific architecture being tested is that of the ACT* production system (Anderson, 1983), although many of the issues are not specific to that system.

A basic premise is that one needs to have a cognitive process model of the student if one is going to be able to effectively tutor the student. Instruction can only be truly principled if one has (a) an accurate interpretation of the current cognitive state of the student; (b) a precise definition of the desired cognitive state of the student; and (c) a learning theory that describes how experience changes the cognitive state of the student. If we are to believe the premise of the book, both of the student models (the actual student and the ideal student) should be cast as production systems--one capable of simulating the current behavior of the student and one capable of simulating the desired state. These are called the current and ideal student models, respectively. The learning theory should be one concerned with changes in the production set.

The Model Tracing Methodology

The instructional paradigm I work in is called model-tracing. Its basic feature is that it tries to dynamically simulate a student's problem-solving and use that simulation to

interpret the student's behavior. Our simulations take the form of production systems. Our model of the student is a non-deterministic production system in which any one of a number of productions can fire at each point. We try to find the one allowable sequence of productions which will uniquely reproduce the surface behavior of the student. This sequence of productions becomes our interpretation of that student. Instruction focuses on the points where the sequence deviates from the prescriptions of the ideal model.

This enterprise is predicated on the assumption that we can correctly model the student at two levels. First, and transparently, we have to be able to model the input-output behavior of the student. Second, and not so transparently, the productions we use must correspond to the steps in which the student actually solves the problem. This second requirement is necessary because part of our instructional strategy is to interrupt the student after a production step and engage in instruction on the previous activity. This only makes sense if we have caught the student between steps of problem solution and the previous step corresponds to what our theory claim. Thus, tutoring forces us to address seriously the grain size with which we model a student in a production system.

Ideally, we would like to have a situation in which we could monitor this student after each production firing. By hypothesis, each production firing produces a change in working memory, and we potentially could monitor for that change. There are two major advantages to monitoring a problem solution production by production. First, this minimizes the ambiguity in interpreting the student in terms of the non-deterministic model. (If we only sample the student's performance every n firings, and the average non-determinism per firing is m , we have to use the student's response to select from among m^n interpretations.) Second, as we will see, our learning theory places a premium on immediacy of feedback. Therefore, it is important to be able to respond to an error as soon as it occurs.

The success at monitoring student performance turns on our success in defining a suitable interface. We would like to design an interface where a subject can easily deposit a record of every step of problem solution. For instance, in tutoring addition, it is easy to get record of the writing of the final answer digits, but what about the calculation of running totals and carrying? In a non-computer-based system we might encourage the student to verbalize out loud the running total and put marks on the paper to indicate carrying. In a computer tutor, natural language cannot be understood, and one cannot literally write. Nonetheless, one wants to fashion an interface where activities giving the intermediate information will be at least as easy and natural as verbalization. It is not always the case that the computer is at a disadvantage to standard methodology. For instance, standard methodology has no means for a student to plan the proof for a geometry problem, whereas we created a graphical formation for this purpose in our geometry tutor (Anderson, Boyle, & Yost, 1985).

One of the realities of tutoring is that the student can generate behavior that cannot be interpreted in terms of our student model. As a simple example of this, the student's hands may wander off base on the keyboard and a sequence of garbage might be typed. The student may enter a term from another programming language. The student might refer to "adjacent supplementary segments" when doing a geometry proof, which just makes no sense. In principle, any of these perturbations could be modelled, but in practice they cannot. In our experience about 20 percent of all errors fall in this category. Once a student has made such an error, one's ability to interpret his future problem-solving can be severely limited. This is because if one cannot find a production to simulate that behavior, one cannot define the next state in the simulation. Therefore, in order to continue the model tracing, we are forced to stop the student when we detect such an error and ask him to reconsider it. Frequently, such errors are slips, and subjects will spontaneously

correct themselves. However, if they do not, we force them to take a step which is along a correct path of solution.

If one believed that such correction was harmful to the learning, one would be in a situation where the learning theory and methodology were in opposition. However, fortunately our learning theory implies that immediate feedback is important. Even in the 80 percent of the cases where we can interpret the error in the model, we point the error out to the student and encourage the student to return to a correct path.

An Example of Tutorial Interaction

To this point, I have outlined the basics of the model-tracing methodology but have only briefly touched on the issues of how we use the model trace to interpret the student. Before doing that it would be useful to have an example of our model-tracing methodology in action. We will use an example from the LISP tutor and then discuss some of its important features.

Our LISP tutor is based on our earlier efforts to model learning to program in LISP (Anderson, Farrell, & Sauer, 1984). Table 1 contains a dialog with a student coding recursive functions 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 while the student's input is shown in bold characters. These listings present "snapshots" of the interaction; each time the student produces a response, we have listed his input along with the tutor's response (numbered for convenience). The total code as it appears on the screen is shown, although the student has added only what is different from the previous code (shown in boldface type). For instance, in line 2 he has added "zero" as an extension of "(defun fact (n)

(cond ("

 Insert Table 1 about here

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 apply. 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 gets the
 student to identify the general relationship. Figure 1 shows the screen image at a critical
 point in the design of this function.

 Insert Figure 1 about here

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 combining the recursive
 call with other elements. The second, in line 6, is to call the function recursively with the
 same argument rather than a simpler one.

After the student finishes coding the function he goes to the LISP window and
 experiments. He is required to trace the function, and the recursive calls embed and then

unravel. Figure 2 shows the screen image at this point with the code on top and the trace below it.

Features of the Model-Training Methodology

This example illustrates a number of features of our tutoring methodology.

1. The tutor constantly monitors the student's problem-solving and provides direction whenever the student wanders off path.

2. The tutor tries to provide help with both the overt parts of the problem solution 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 tries to eliminate aspects 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 this tutor is production system consisting of hundreds of ideal and buggy rules. Below 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 combine 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 combine 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 parens around its arguments.

```

IF the goal is to combine 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.

The tutor determines which rule fired in the student's head by seeing which rule action matches the behavior of the student. Attached to the rules are templates for dialogues such as the two we have given above.

Altogether we have over a 1000 productions (correct and buggy) to model student performance in our lessons, which cover all the basic syntax of LISP, design of iteration and recursive function, use of data structures, and means-ends planning of code. The majority of these productions are like the APPEND production above in that they are directly connected to the generation of code. Another example of such a production is:

```

IF the goal is to test if arg1 is zero
THEN use the LISP function ZEROP
     and set a subgoal to code arg1

```

Because they are connected to the generation of code, it is very easy to monitor for their execution. However, a sizable minority of the productions are concerned with planning such as that involved in the design of a recursive call. Normally, students are very quiet at such

times. We were faced with the task of finding some way of exposing the planning process. The example illustrates the solution that we adopted in the LISP tutor, which is to step the student through the planning process by a sequence of menus. This is not a totally satisfactory solution because students find the overhead associated with processing the menus to be high relative to the actual planning. Therefore, we do not work students through these planning menus unless they give evidence of having problems with the planning process.

Approximately 40 percent of the 1000 productions model correct generation of code, while the residual model various bugs. The bugs are largely gathered by empirical observation of student errors. A current research project concerns the origin of such bugs. Obviously, a theory of bug origin relates strongly to a production system learning theory.

Production Learning

To this point we have discussed how we use our production-system model to organize the tutorial interactions. Clearly, the success of the tutorial interaction depends on the accuracy of the model. However, it also depends on the actual instructional activities in which the tutor engages. This is predicated on our theory of production-system learning. Since there is considerable variation in such learning theories and since my theory has had some recent revision, I will review the basic premises of the theory before considering the pedagogical application.

Learning begins with formal instruction. Consider this small fragment of instruction from the second edition of Winston and Horn (1984, p. 24):

The value returned by CAR is the first element of the list given as its argument:

```
(CAR '(FAST COMPUTERS ARE NICE))
```

```
FAST
```

This is fundamentally declarative information. It does not directly tell one how to get the first element of a list or how to apply CAR and so contrasts with productions like

```
P1      IF the goal is to get the first element of a list
        THEN write CAR
           and set a subgoal to code the list
```

or

```
P2      IF the goal is to evaluate (CAR '(elem1 ...))
        THEN the answer is elem1
```

The first claim of the learning theory is that the student has, coded as productions, various domain-general weak methods which enable the student to interpret declarative information such as in the Winston and Horn text. So, the student might have a means-end production like

```
G1      IF the goal is to obtain the element that bears a
           relation to an argument
           and an operation returns a value that bears the relation
           to a structure if a precondition is satisfied,
        THEN plan to use the operation
           and set a subgoal to satisfy the precondition.
```

To understand how this production applies, assume the student has his goal encoded as

"the goal is to get the first element of (a b c)"

and the Winston and Horn instruction encoded as

"CAR gets the first element of a list
if the list is the argument of CAR"

Production G1 can evoke CAR to get the first of list (a b c) if we have the following binding of clauses

1. "the goal is to get the element that bears a relation to an argument"

matches to

"the goal is to get the first element of (a b c)"

2. "an operation returns a value that bears the relation to

a structure"

matches to

"CAR returns the first element of a list"

3. "if a precondition is satisfied"

matches to

"if the list is the argument to CAR"

The production will then use CAR and set as a subgoal to make (a b c) an argument to CAR. Thus, this production can interpret the declarative instruction to have the same effect as P1.

Another type of weak method that we have studied a great deal is analogy. It seems that in the domain of LISP programming, students like to solve problems by analogy to other problems. Thus, a student asked to get the first of the list (A B C) might look at the Winston and Horn example of (CAR '(FAST COMPUTERS ARE NICE)) and write

(CAR '(A B C))

by analogy.

We have also done some simulation work of other weak-methods, in particular hillclimbing. In each of our weak methods declarative knowledge is interpreted by productions to solve novel problems. The cost of interpretation of declarative knowledge by weak methods can be high. First, the requisite declarative knowledge must be held in working memory to guide the solution, posing a load on working memory. Second, the interpretive process is often very piecemeal, requiring a great many productions to apply.

Knowledge compilation is the name we give to the process that converts this interpretive use of declarative knowledge by general procedures into productions like P1 and P2 that directly perform the action. It has been discussed in detail elsewhere (Anderson, in

press; Neves & Anderson, 1981). There are two processes--one called *proceduralization* and the other, *composition*. Proceduralization involves the deletion of reference to declarative information and creation of domain-specific rules. The essential part of the declarative information is built into the resulting productions. So, rather than G1, which refers to relations in general, P1 concerns itself with first elements of a list. The second process, composition, collapses many productions into one. It is similar to ideas discussed by Lewis and Rosenbloom and Newell elsewhere in this book. Composition also occurs to create "macro operators" out of basic operators. So, if we have a production to code CONS and another to code CAR, they can be composed together to create an operator that will insert the first element of one list into another.

The current state of the ACT learning theory is as notable for what it lacks as for what it contains. In particular, it lacks generalization and discrimination learning mechanisms, which had originally formed the inductive learning component of the theory and which are discussed elsewhere in this book. The story of their demise has been told elsewhere (Anderson, in press), but basically there has been an embarrassing failure to get empirical evidence for their existence. We also decided that induction should be treated as a problem to be solved at the production level rather than the architectural level and, by compiling solutions of inductive problems, can produce productions which are generalizations and discriminations of original productions.

It would be useful to give an example of the difference between doing induction at the architectural level versus the production level. Consider, for instance, the task of learning when it is useful to make the vertical, or opposite, angle inference in geometry (Lewis & Anderson, 1985). Among the production rules we want to learn is the following

V1: IF $\angle 1$ and $\angle 2$ are opposite angles
 and they are parts of triangles
 and there is a goal to prove the triangles congruent
 THEN conclude that the vertical angles are congruent

The second and third clauses are not part of the legal definition of the vertical angles postulate but rather are problem features that predict the usefulness of the vertical angle inference. These features are not directly taught, but students must induce them. There are two ways to arrive at these heuristic features. One is compare cases of success and failure with automatic induction mechanisms such as were proposed in Anderson (1982) and Langley (1985). This would be done at the architectural level. In contrast, if this was done at the production level, the student might consciously set as a goal to determine the features that predict the usefulness of the vertical angle inference. The student can bring problem-solving procedures to bear in answering this question and try to solve this just as he might try to figure out the identity of a murderer in a mystery. Chaining backwards in the logical structure of the proof, he might see that the vertical angle inference enabled the congruent triangle inference and so form the declarative proposition

"vertical angle inferences help prove triangle congruence."

This fact could then be evoked in deciding whether to make a vertical angle inference later on. Compiling the steps of that decision that led to this later vertical angle inference would produce the desired rule V1. This production level induction predicts, in contrast to the architectural level, that students would have conscious, declarative access to their inductions. This is one of the results that Lewis and Anderson (1985) found.

There is one other learning mechanism in the current ACT in addition to the mechanisms of knowledge compilation. This is the strengthening mechanism by which productions gain strength through practice. This mechanism accounts for the continued improvement of a skill long after opportunity for compilation has ceased (Anderson, 1982). While a very simple learning mechanism, it has important implications for tutoring.

Implications for Skill Acquisition

In this section I want to address the issue of what, according to the ACT theory, would impact on the growth of proficiency (speed and accuracy) with which a skill is manifested. These will be both learning factors that impact on the acquisition of the underlying skill and performance factors which impact on how well that skill is executed. As we follow the development of a skill in the ACT theory, we find that there are relatively few factors

1. Instruction The student has to get the requisite knowledge, and in most instructional situations that is not directly given. For instance, LISP instruction seldom tells students how to write recursive functions. This is something they have to figure out for themselves. It has been shown (Pirulli, 1985) that students learn faster if they are told directly--which may seem an obvious result but has not impacted much on pedagogical practice. Most students induce recursive programming by analogy to examples.

More generally, whatever instruction the student receives will be encoded declaratively and will be interpreted by weak problem-solving procedures. Instruction is effective to the degree that it provides information that can be easily interpreted by the student's weak procedures.

2. Correctness of Declarative Encoding Students may incorrectly represent the knowledge to which their weak methods apply. For instance, consider the following statement of the side-angle-side postulate.

"If two sides and an included angle of one triangle are congruent to the corresponding parts of another triangle, then the two triangles are congruent."

Students often interpret "included" to mean contained in the triangle rather than contained between the two congruent sides. Any production compiled from this declarative

understanding will itself be flawed. Similarly, Pirolli and Anderson (1985) have argued that what controls what is learned by analogy from an example is how well students represent why the example solves the problem it does.

3. Working Memory Limitations Assuming that instruction provides information in a useful declarative form and assuming it is correctly encoded, it has to be held in working memory in order to be compiled into production form. Any factor which increases the amount of information that must be held or the time for which it must be held will increase the likelihood of working memory failures. These working memory failures will result in either incorrect compilations--if certain details are missing--or aborted compilations--if the goal structure that holds the information together is lost. Thus, awkward instruction can result in failure by increasing load or comprehension time. Clearly, there is a premium in compacting the necessary instruction into its bare essentials. Also, one wants to avoid unnecessary working memory burdens in the actual problem solving.

Once productions have been compiled they can be composed further into more effective macro operators. This requires holding in working memory the goal structures that indicate how the individual productions should be composed together to form the macro operator. Thus, working memory capacity can control the composition process.

Finally, working memory failures may cause correct productions to randomly misapply and produce errors. These are what Norman (1981) would classify as slips. Anderson and Jeffries (in press), looking at a population of undergraduates just learning LISP, concluded that the majority of their errors with basic concepts derived from working memory failures rather than fundamental mistakes.

4. Production Strength The strength of a production impacts on performance of a skill in multiple ways. Weak productions may fail to apply, producing permanent lapses until

remediated by more instruction. Also, weak productions take longer in the ACT theory and so increase the period over which knowledge must be maintained in working memory for purposes of composition.

One of the surprising things to come out of our work with the LISP tutor is how very systematic the improvement of productions appears to be with practice. Figures 3-5 illustrate speed-up in times for subjects to code actions corresponding to the firing of productions in our simulations of their problem solving. Many of the productions in the LISP tutor have coding action associated with them. For instance, the production labelled constant produces the typing of a quoted constant like 'dog. What we have plotted in these figures is the time for the student to type this action (measured from the last keystroke of the previous action to the last keystroke of this action). This is plotted as a function of the number of times that production fired in the simulation of the student. The three figures represent some newly introduced productions from each lesson. While the trace of individual production times show random perturbations, the overall pattern is clear: There is a rapid speed up from first to second execution, reflecting knowledge compilation, and then a slow strengthening process thereafter. A similar pattern is found in the error rates

Tutorial Implications

Corresponding to each of these considerations are a set of implications for tutorial action.

1. Instruction The first consideration about instruction is that it should be based on a production-system model of the target skill so that we know what it is that we want to instruct. This is the basic presupposition of the paper. One of the important features of an ACT production system is its goal structure, and it is essential that the student acquire

the correct goal structure to control the problem solving. Finding a way to communicate the goal structure becomes a major issue in the design of a tutorial interface. It can be difficult because the goal structure is abstract and not easily described. In the LISP tutor we use various annotated symbols for various code structures.

In addition to basing instruction on a production-system analysis and emphasizing communication of goal structures, there are a number of other general recommendations to make about the design of instructional activities. It makes sense to provide the instruction in the context where it should be used to maximize the probability that the student will retrieve and try to use the knowledge. Also, it makes sense to provide that knowledge in a form that can be most easily used by weak methods. For instance, we try to fashion our instruction to take the form of rules for means-ends solutions or as carefully fashioned examples for use by analogy.

2. Correctness of Declarative Encodings Unfortunately, it is not the case that the ACT theory *per se* provides any guidance about how to design instruction to minimize possibilities of misunderstanding. This is largely a language comprehension (and to some degree a graph comprehension) issue for which there is no ACT model, although in principle and with great effort we could create a production-system model for comprehension. We have instead developed a set of heuristics that seem to optimize communication. One of the principal of these heuristics is to minimize what we say.

Of course, the second line of defense against misunderstandings is to be able to remediate them when they occur. So far we have simply catalogued observed confusions and associated appropriate instruction with them. Other work such as Buggy (Brown & VanLehn, 1982) has tried to come up with a generative theory of the origins of these confusions. One approach is to try to enumerate the possible syntactic variations on correct

rules, and that certainly seems possible in domains like LISP. The other approach is to predictively generate the bugs. To the extent that these bugs have their origin in language comprehension, that seems like a hopeless goal, however.

3. Working Memory Limitations The obvious pedagogical cure to working memory problems is to minimize unnecessary load. One way we try to achieve this in the LISP tutor is by doing syntactic checking. It also follows that immediate feedback on errors is important because this limits the amount of time and information over which a student must integrate to learn. The nature of the feedback is important in that it should inform the student of why there was an error but not what the correct answer is. If it gives away the correct answer, the student will compile a production to copy the answer rather than one to generate the answer when the answer is not given. The error messages generated by the LISP tutor have this quality of giving the right amount of information without giving away the answer.

Another aspect of working memory limitation that the tutor must deal with is the slips that produce errors but which do not reflect fundamental misconceptions. These can be remediated by direct correction rather than the debugging process of pointing out the error and asking the student to correct it himself. This is because there is no fundamental error that needs correction. Unfortunately, slips and fundamental misconceptions often generate the same surface behavior. Rather than trying to diagnose which it is, the LISP tutor remediates all as if they were fundamental bugs. We intend to use a student model in future versions of the tutor to decide the appropriate remedial action. That is, if the student has displayed successful use of the production with some consistency, we will assume the current error is a slip.

One might question why slips should be corrected at all, given they do not reflect a

fundamental misconception in need of remediation. The problem is that a flawed solution confounds later steps of problem solving and instruction. Students believe the slips they put down on paper or on the screen and begin to take erroneous actions based on them.

4. Production Strength The principal implication of considerations of production strength is that problems should be given to practice productions. However, this leaves open the questions of what is enough practice and how that practice should be distributed. We need to balance the following three considerations

- a. As long as the student is making errors in a production, it needs practice.
- b. Too much practice in one session is not helpful because of the massing of practice. There is consistent evidence that further within-session practice on a piece of knowledge becomes valueless.
- c. Between sessions the subject may have forgotten and may need remedial practice.

To truly optimize performance would require making complex computations about the interaction of production strength and activation in the ACT model. However, the important observation is that at this level of description of memory phenomena, ACT is nearly indiscriminable from many other memory theories, and we might as well select a computationally tractable theory. Therefore, we intend to do our problem selection on the basis of the three-state memory model (Atkinson, 1972), which has already proven quite successful in dealing with phenomena (a)-(c) above. It assumes that a production can be in a permanent state, a temporary state, or a no-knowledge state. One can go from the no-knowledge state to either the temporary or permanent state, and one can drop back from the temporary to the no-knowledge state. To use this model, one simply calculates

Bayesian estimates of the probability of a production being in each of the states.

The assumption is that probability of transiting to the permanent state from a no-knowledge state is maximum when the surrounding productions have been performed correctly. Therefore, our problem selection algorithm favors problems that present unlearned productions in familiar context, since this is what will maximize probability of learning new productions.

Production Learning and Tutoring: Concluding Remarks

What needs to be stressed is the very simple concept that this framework depicts of the nature of skill acquisition and the tutoring of a skill. Specifically, to tutor a skill optimally, we need to

1. Perform a task analysis to identify the productions that a student must acquire;
2. Find a form of instruction--presentation of examples, statement of abstract rules, etc.--which the student's weak methods can use to solve the problems;
3. Give the student enough practice to compile and strengthen these rules; and
4. Monitor the student's progress and only introduce new rules when the old rules have achieved a sufficient degree of strength that they no longer interfere with the acquisition of new rules.

To help recognize how simple an instructional strategy is implied by this learning theory, it is interesting to consider just two of the factors that are not important according to it but which might be according to other plausible learning theories.

1. According to this theory, the learning of separate productions is independent, and we should not see, as one might intuitively expect, clusters of productions (say productions for coding arithmetic expressions in LISP) behaving alike and so reflecting some fundamental misconceptions. In fact, statistical analyses of the data gathered with the LISP tutor suggest individual productions are learned independently

2. Many learning theories would suggest that problem sequence is important. For instance, discrimination learning mechanisms would suggest that it would be good to juxtapose problems that involve similar but different productions, such as for coding LIST and CONS. This would enable acquisition of discriminating features. In contrast, the current ACT theory assumes nothing about problem sequence is special except that it is important to minimize the number of new productions to be learned simultaneously, as we noted earlier. We have done only a little exploration of problem sequence with out tutor, but so far we have gotten only null results, consistent with the ACT theory.

As another piece of evidence for the simplicity of the learning theory, note that to do tutoring it is not necessary to run a dynamic simulation of the learning processes. Rather, the tutor has compiled into its behavior all the pedagogical implications of the learning theory. The theory implies that productions are basically acquired through practice, and the only purpose of a tutor is to increase the effectiveness of that practice.

A number of decades ago, S-R theory was rejected in psychology because it implied that knowledge was nothing but the accumulation of S-R bonds, and an accounting of the number of S-R bonds necessary to produce interesting behavior came up with an astronomical number. Production systems provide a computationally more powerful

framework than S-R theory, and production systems like ACT are more structured with goals, weak methods, and knowledge compilation. However, given that more powerful framework, it may be that knowledge (or at least problem-solving skill) is just the accumulation of production rules. A modified empiricism may still be feasible. While I would not want to claim definitive results, our tutoring work is basically putting this modified empiricism to test.

References

- Anderson, J.R. (1982). Acquisition of proof skills in geometry. In J.G. Carbonell, R. Michalski & T. Mitchell (Eds.), *Machine Learning, An Artificial Intelligence Approach*. Palo Alto, CA: Tioga Press.
- Anderson, J.R. (1982). Acquisition of Cognitive Skill. *Psychological Review*, 89, 369-406.
- Anderson, J.R. (1983). *The Architecture of Cognition*. Cambridge, MA: Harvard University Press.
- Anderson, J. R. (1985). *Skill acquisition: Compilation of weak-method problem solutions*. Office of Naval Research Technical Report Carnegie-Mellon University.
- Anderson, J. R. (In press) Knowledge compilation: The general learning mechanism. In R. Michalski, J. Carbonnell, & T. Mitchell (Eds.) *Machine Learning II*. Palo Alto, CA: Tioga Press
- Anderson, J.R. and Jeffries, R. (In press). Novice LISP Errors: Undetected Losses of Information from Working Memory. *Human-Computer Interaction*.
- Anderson, J. R., Boyle, C. F., & Yost, G. (1985) The Geometry Tutor. In *Proceedings of IJCAI-85*. Los Angeles, CA: IJCAI,
- Anderson, J.R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, 8, 87-129.
- Atkinson, R. C. (1972). Optimizing the learning of second-language vocabulary. *Journal of Experimental Psychology*, 96, 124-129.
- D. Klahr, P. W. Langley, & R. Neches (Eds.). Langley, P. W. (1985). *A general theory of discrimination learning*. Cambridge, MA: Bradford Books/MIT.
- Lewis, M. W. & Anderson, J. R. (1985). Discrimination of operator schemata in problem solving: Learning from examples. *Cognitive Psychology*, 17, 26-65.
- Neves, D. M., & Anderson, J. R. (1981) Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their*

acquisition. Hillsdale, NJ: Erlbaum.

Norman, D. A. (1981). Categorization of action slips. *Psychological Review*, 88, 1-15

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

Reiser, B. J., Anderson, J. R., & Farrell, R. G. (1985). Dynamic student modelling in an intelligent tutor for LISP programming. In *Proceedings of IJCAI-85* Los Angeles, CA: IJCAI,

Winston, P. H. & Horn, B. K. P. (1984). *LISP, Second Edition*. Reading, MA: Addison-Wesley

Figure Captions

1. The screen configuration before line 4 in Table 1.
2. The screen configuration at the end of the dialogue in Table 1
3. Times to perform actions corresponding to firings in the ideal model Constant = coding quoted constant, global = coding global variable, number = coding a number. The bold curve represents average. This data is for productions introduced in Lesson 1.
4. Lesson 2 times analogous to Figure 3. arg = coding an argument to a function that is a parameter of a function definition; params = coding parameters of a function definition; reverse = coding function reverse; defun = coding function defun; name = coding name of function definition.
5. Lesson 3 times analogous to Figure 3 else = coding t test in cond; predicate = coding predicate; cond = coding function cond; number test = coding numbers; cond clause = coding a clause in a cond.

Table 1

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

```
1 (defun fact (n)
  (cond ((equal) <ACTION>)
        <RECURSIVECASE>))
```

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

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

```
2 (defun fact (n)
  (cond ((zero) <ACTION>)
        <RECURSIVECASE>))
```

Did you mean zerop (y or n)? y

```
3 (defun fact (n)
  (cond ((zerop n) 1)
        <RECURSIVECASE>))
```

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

PRESS: IF YOU WANT TO:

- 1 Calling fact recursively with n
- 2 Multiplying n by fact of n minus 1
- 3 Getting 1 2 n and then multiplying the results.
- 4 have the tutor choose

Menu choice: 3

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

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

PRESS IF YOU WANT TO

- 1 n minus 1
- 2 n equal to 0
- 3 n plus 1
- 4 have the tutor choose

Menu choice: 1

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

What is the value of (fact 1)?

Answer: 1

What is the value of (fact 0)?

Answer: 1

EXAMPLES

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

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

What is the value of (fact 3)?

Answer: 6

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 as 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 <NUM> <NUM2>))))
```

What do you want to do first?

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

Menu choice: 1

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

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

```
7 (defun fact (n)
  (cond ((zerop n) 1)
        (t (times n (fact (difference ))))))
```

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

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

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

THE LISP WINDOW

= > (trace fact)

(fact)

= > (fact 3)

1 <Enter> fact (3)

2 <Enter> fact (2)

3 <Enter> fact (1)

4 <Enter> fact (0)

4 <EXIT> fact 1

3 <EXIT> fact 1

2 <EXIT> fact 2

1 <EXIT> fact 6

6

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

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

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

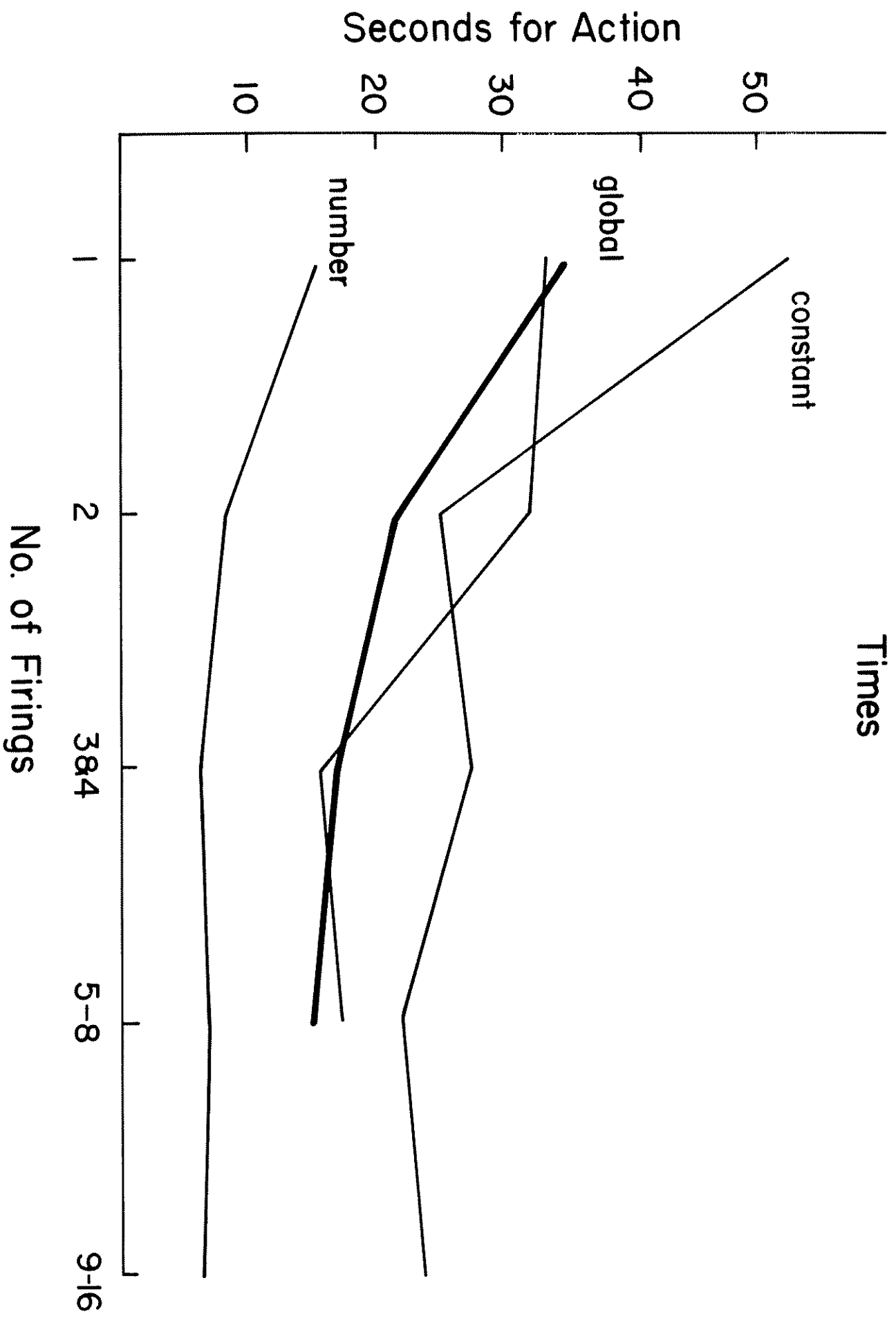
```
(defun fact (n)
  (cond ((zerop n) 1)
        (t (times n (fact (sub1 n))))))
```

THE LISP WINDOW

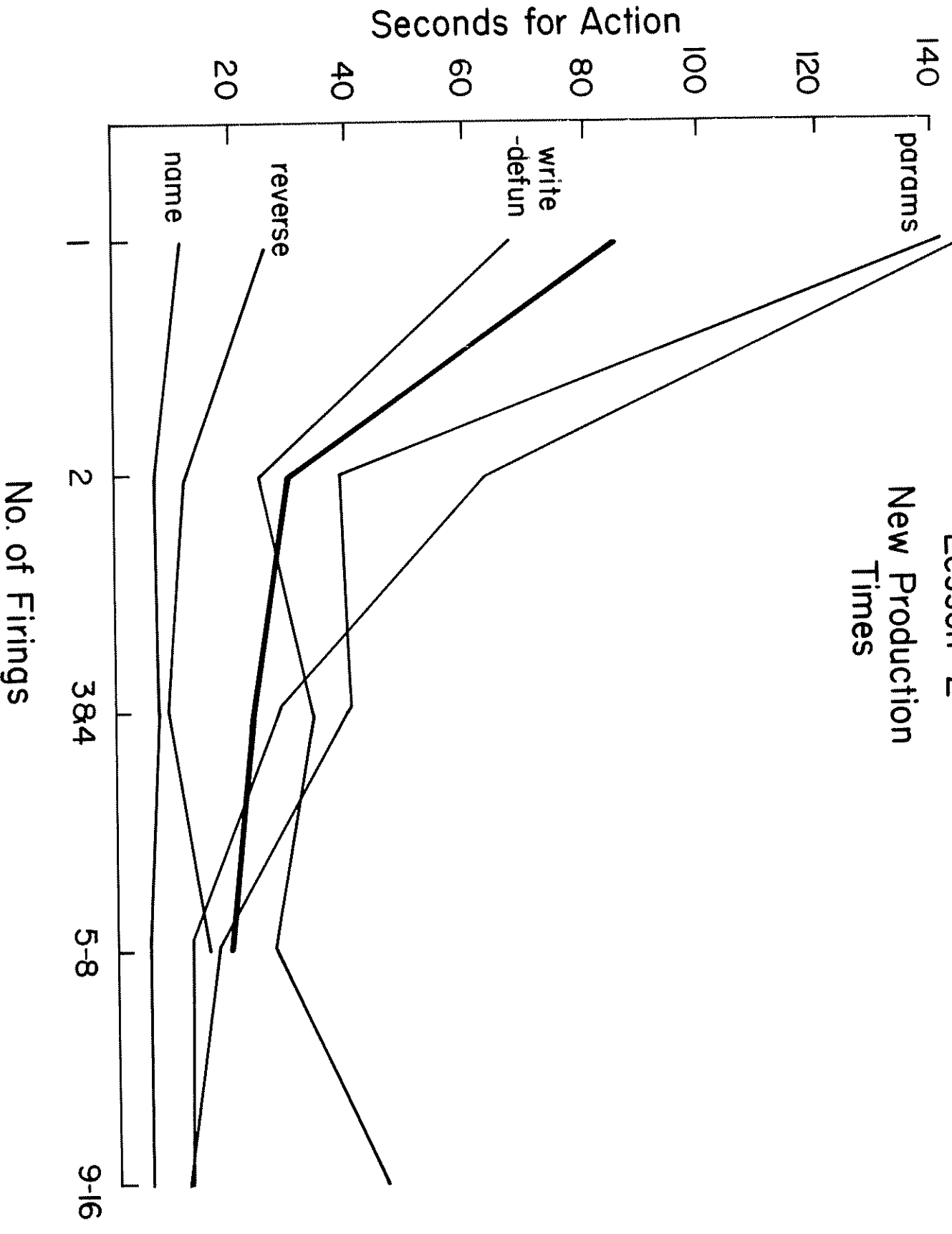
```
=> (trace fact)
(fact)
```

```
=> (fact 3)
1 <Enter> fact (3)
|2 <Enter> fact (2)
| 3 <Enter> fact (1)
| |4 <Enter> fact (0)
| |4 <EXIT> fact 1
| 3 <EXIT> fact 1
|2. <EXIT> fact 2
1 <EXIT> fact 6
6
```

Lesson 1 Times



Lesson 2
New Production
Times



Lesson 3 New Times

