Psychological Review
1987, Vol. 94, No. 2, 192–210

Copyright 1987 by the American Psychological Association, Inc.
0033-295X/87/$00.75

Skill Acquisition: Compilation of Weak-Method Problem Solutions

John R. Anderson
Carnegie-Mellon University

Cognitive skills are encoded by a set of productions, which are organized according to a hierarchical goal structure. People solve problems in new domains by applying weak problem-solving procedures to declarative knowledge they have about this domain. From these initial problem solutions, production rules are compiled that are specific to that domain and that use of the knowledge. Numerous experimental results may be predicted from this conception of skill organization and skill acquisition. These include predictions about transfer among skills, differential improvement on problem types, effects of working memory limitations, and applications to instruction. The theory implies that all varieties of skill acquisition, including those typically regarded as inductive, conform to this characterization.

Research on the acquisition of cognitive skills has received a great deal of recent attention, within both the psychological and the artificial intelligence (AI) literature (J. R. Anderson, 1981, 1982, 1983; Brown & Van Lehn, 1980; Carbonell, 1983; Chi, Glaser, & Farr, in press; Kieras & Bovair, 1986; Laird, Rosenbloom, & Newell, 1984; Langley, 1982; Larkin, McDermott, Simon, & Simon, 1980; Lesgold, 1984; Newell & Rosenbloom, 1981; Van Lehn, 1983). There are a number of factors motivating this surge of attention. First, there is increasing evidence that the structure of cognition changes from domain to domain and that behavior changes qualitatively as experience increases in a domain. This shows up both in comparisons between novices and experts within a domain (Adelson, 1981; Chase & Simon, 1973; Chi et al., in press; Jeffries, Turner, Polson, & Atwood, 1981; Larkin et al., 1980; Lesgold, 1984) and in the failure of descriptions of cognition to be maintained across domains (J. R. Anderson, 1985, chap. 9; Cheng & Holyoak, 1985). For instance, expert problem solving in physics involves reasoning forward to the goal, whereas problem solving in programming involves reasoning backward from the goal. In developing a learning theory, one is striving for the level of generality that will unify these diverse phenomena. The belief is that learning principles will show how the novice becomes the expert and how the structure of different problem domains is mapped onto different behavior. Basically, the goal is to use a learning theory to account for differences in behavior by differences in experience.

Another motivation for the recent attention given to skill acquisition is the fact that theories of cognition for a particular domain are not sufficiently constrained (J. R. Anderson, 1976, 1983). There are multiple theoretical frameworks for accounting for a particular performance, and the performance itself does not allow for an adequate basis for choosing among the accounts. A learning theory places an enormous constraint on these theoretical accounts because the theoretical structure proposed to encode the domain knowledge underlying the performance must be capable of being acquired. This is like the longstanding claim that a learning theory would provide important constraints on a linguistic theory (for recent efforts, see MacWhinney, in press; Pinker, 1984; Wexler & Culicover, 1980).

Although there may be serious questions about the uniqueness of the theoretical accounts of cognition advanced thus far, there is little question that the accounts have become increasingly more sophisticated and cover much more complex phenomena. This success at creating accounts of various complex domain behaviors has emboldened the effort to strive for a learning account. Now that we have the theoretical machinery to account for complex cognition, we have frameworks in which the learning question can be addressed. This optimism has also been fueled by researchers, mainly in artificial intelligence, who are studying mechanisms for knowledge acquisition (Hayes-Roth & McDermott, 1976; C. W. Lewis, 1978; Michalski, 1983; Michalski, Carbonell, & Mitchell, 1983; Mitchell, 1982; Vere, 1977). The psychological research on skill acquisition (Klahr, Langley, & Neches, in press) often is an attempt to transform the abstract concepts from AI so that they can be incorporated into the theoretical mechanisms which have evolved to account for the realities of human cognition.

Another motivation for interest in skill acquisition is that it seems many of the fundamental issues in the study of human cognition turn on accounts of how new human competencies are acquired. Consider the faculty position, which claims that language and possibly other higher cognitive functions are based on special principles (Chomsky, 1980; Fodor, 1983) and which seems to rest on claims that different cognitive competencies are acquired in different manners. For instance, it is claimed that language acquisition depends on using universals of language to restrict the induction problem. For another example, compare symbolic (Newell & Rosenbloom, 1981; Van Lehn, 1983) versus neural models of cognition (Ackley, Hinton,

This research 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. I would like to thank Al Corbett, David Klahr, Allen Newell, and Lynne Reder for their comments on this article.

Correspondence concerning this article should be addressed to John R. Anderson, Department of Psychology, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.

& Sejnowski, 1985; McClelland, 1985; Rumelhart & Zipser, 1985). One of the few places where they make fundamentally different (as opposed to complementary) claims is with respect to the mechanisms underlying knowledge acquisition. Many researchers in various theoretical camps are working on learning because they realize that understanding and substantiating the details of how knowledge is acquired in their framework is a prerequisite to advancing their grander claims about the nature of cognition.

ACT* Theory of Skill Acquisition

The ACT* theory of skill acquisition (J. R. Anderson, 1982, 1983) was developed to meet three constraints. First, the theory had to function within an existing theory of cognitive performance, the ACT* production system, which had its own considerable independent motivation. Second, the theory had to meet sufficiency conditions, that is, it had to be capable of acquiring the full range of cognitive skills that humans acquire under the same circumstances that they acquire it. Third, it had to meet necessity conditions, which means that it had to be consistent with what was known about learning from various empirical studies.

At the time of development of the ACT* theory, the existing data base fell seriously short of testing the full range of complex predictions that could be derived from the learning theory. The theory applies to acquisition of elaborate skills over long-time courses. For obvious reasons, there is a dearth of such research. Moreover, much of the existing research reports only gross descriptive statistics (e.g., whether discovery learning or guided learning is better) rather than analyses at the level of the detail addressed by the ACT* theory. We (e.g., J. R. Anderson, Farrell, & Sauers, 1984) had a few detailed protocol studies of skill acquisition over the course of many hours, but these studies can be criticized for the small sample size (often 1) and the subjectiveness of the data interpretation.

In response to this, in our research we have looked at the acquisition of complex mathematical and technical skills taught in various sectors of our society, such as geometry, programming, and text editing. The most ambitious line of research is the development of computer-based tutors for large-scale and objective data collection (Anderson, Boyle, Farrell, & Reiser, 1984; J. R. Anderson, Boyle, & Reiser, 1985). Most of the results from this research line are still pending. On a less ambitious scale, we have performed laboratory analysis of these skills and their acquisition. Most of the research to be reviewed in this article comes from this second line of research.

This article has three goals: The first is to set forth some general claims about the course of skill acquisition, the second is to discuss a series of relatively counterintuitive predictions to be derived from the ACT* theory and to review the state of empirical evidence relevant to these predictions, and the third is to report some revisions to that theory in response to both the empirical data and further theoretical considerations. Before turning to these goals, it is necessary to review the ACT* theory of skill acquisition and its empirical connection. I will do this with respect to the domain of writing LISP programs, in which the theory has had its most extensive application (J. R. Anderson et al., 1984). The examples are deliberately chosen to use only the most basic concepts of LISP so that lack of prior knowledge of LISP will not be a barrier to understanding the points.

Productions in ACT*

Cognitive processing in the ACT* theory occurs as the result of the firing of productions. Productions are condition–action pairs that specify that if a certain state occurs in working memory, then particular mental (and possibly physical) actions should take place. Below are two "Englishified" versions of productions that are used in our simulation of LISP programming:

P1: IF the goal is to write a solution to a problem
and there is an example of a solution to a similar problem
THEN set a goal to map that template to the current case.

P2: IF the goal is to get the first element of List1
THEN write (CAR List1)

The first production, P1, is one of a number of productions for achieving structural analogy. It looks for a similar problem with a worked-out solution. If such an example problem exists, this production will fire and propose using this example as an analogue for solving the current problem. This is a domain-general production and is executed, for instance, when we use last year's income tax forms as models for this year's income tax forms. In the domain of LISP, it helps implement the very common strategy of using one LISP program as a model for creating another.

The second production rule, P2, is one that is specific to LISP and recognizes the applicability of CAR, one of the most basic of LISP functions, which gets the first element of a list. For instance, (CAR '(a b c)) = a. One important question about the ACT* learning theory is how a system, starting out with only domain-general productions like the first, acquires domain-specific productions like the second.[1] An answer to this question will be outlined shortly.

These production rules reflect cognitive contingencies: If a certain type of situation is encoded in working memory, then a certain action should take place. The exact notation with which these contingencies are stated is just notation and carries no psychological import, despite the fact one has to be very precise and consistent about the notation in a computer simulation. The psychological claims are (a) that the human mind operates according to such rules and (b) that these rules are interpreted as specified in the ACT* system.

It is important to recognize that the ability of a production-system model to account for performance in a specific task is going to be a joint function of the rules assumed and the way they are interpreted. In this, production systems are not unique. In more traditional cognitive analyses (e.g., Clark, 1974), prediction is a joint function of the knowledge representation assumed and how it is processed. Although this practice of jointly assuming a knowledge representation and its rules of interpretation is widespread, the number of degrees of freedom

[1] The assumption, which I will elaborate later, is that the domain-general productions are innate.

in such enterprises should make one uneasy. This is one of the reasons why it is deemed so important to develop a learning theory for production systems. Such a learning theory would provide severe constraints on the production rules that can be used to encode a domain. These rules would have to be learnable by the learning mechanisms of the theory.

The motivations for the production system architecture in general and for ACT* in particular have been developed extensively elsewhere (J. R. Anderson, 1983). Although all these motivations cannot be detailed here, it is worthwhile briefly reviewing why production systems are considered a good framework for modeling skill acquisition (see also Klahr, Langley, & Neches, in press). Production rules like the P1 and P2 examples earlier are relatively well structured, simple and homogeneous, and independent of one another. In being well structured, they contrast with theoretical formalisms such as neural models (McClelland & Rumelhart, 1986; Rumelhart & McClelland, 1986). This good structuring helps guarantee that the behavior produced by learning production rules will be coherent. In being simple and homogeneous, they contrast with many of the proposals for schema systems (Rumelhart & Ortony, 1976; Schank & Abelson, 1977). Their simplicity and homogeneity make it possible to define relatively simple learning mechanisms capable of constructing new rules. In being independent, they contrast with a typical programming language in which operations are sequentially ordered and depend on one another. In contrast, it is possible to add or delete production rules individually without greatly perturbing the system. This independence of production rules makes it possible to define an incremental learning system that grows one production rule at a time and does not involve wholesale changes to the cognitive procedures.

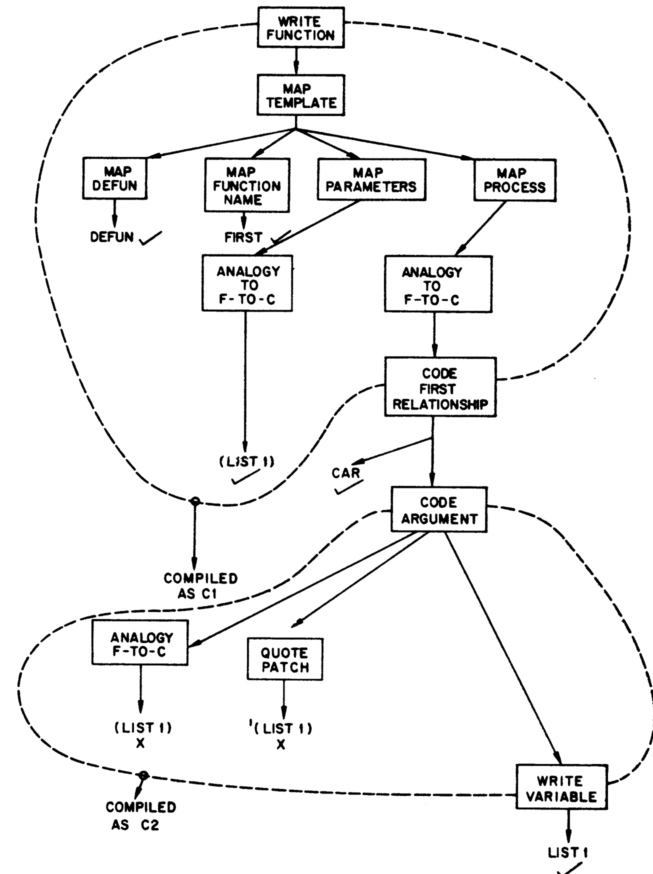

Figure 1. A representation of the goal structure in B.R.'s solution to the problem of writing the function FIRST. (The boxes represent goals, and each arrow indicates a production that tried to achieve the goal. The arrows can lead to one or more subgoals [boxes] or direct actions [LISP code]. Checks indicate successful actions and Xs indicate unsuccessful actions.)

Simulation of LISP Programming Skills

To illustrate both how productions are combined to solve a problem and how domain-specific productions are acquired from domain-general productions, it is useful to examine one subject's protocol as she first learned to define new functions in LISP. Defining functions is the principle means for creating LISP programs. The subject (B.R.) had spent approximately 5 hours before this point practicing using existing functions in LISP and becoming familiar with variables and list structures.

B.R. had just finished reading the instruction in Winston and Horn (1981) on how to define functions. The only things she made reference to in trying to solve the problem were a template and some examples in text of what a function definition looked like. The template and an example of one of the functions she looked at are given below. The function is called F-TO-C, and it converts a temperature in Fahrenheit to centigrade:

```
(DEFUN ⟨function name⟩
  (⟨parameter 1⟩⟨parameter 2⟩ . . . ⟨parameter n⟩)
  ⟨process description⟩)
(DEFUN F-TO-C (TEMP)
  (QUOTIENT (DIFFERENCE TEMP 32) 1.8))
```

The problem she was given was to write a LISP definition that would create a new function called FIRST, which would return the first element of a list. As already noted, there is a function that does this called CAR that comes with LISP. Thus, she was creating a redundant function, and this problem was really just an exercise in the syntax of function definitions.

Anderson, Farrell, and Sauers (1984) reported a simulation of B.R.'s protocol that reproduced her major steps. The production set behind this simulation produced the goal structure as shown in Figure 1, which is useful for explaining B.R.'s behavior. She started out selecting the template as an analogue for building a LISP function. A set of domain-general productions for doing analogy then tried to use this template. Subgoals were set to map each of the major components of the template. Knowing "DEFUN" was a special LISP function, she wrote this first and then wrote "FIRST," which was the name of the function.

She had trouble deciding how to map the structure "(⟨parameter 1⟩⟨parameter 2⟩ . . . ⟨parameter n⟩)" because she had no idea what parameters were. However, she looked at the concrete example of F-TO-C, saw that there was a list containing the argument to the function, and correctly inferred she should create a list of the variable that would be the argument to FIRST. She chose to call this variable LIST1.

Then she turned to trying to map "⟨process-description⟩," which she again could not understand. She saw in an example that this was just the LISP code that calculated what the function was supposed to do, and so she wrote CAR, which got the first element of a list. In the GRAPES simulation of B.R. a LISP production generated CAR. This was the only place in the original coding of the function at which a LISP-specific production fired. Presumably it was acquired from her earlier experience with LISP. To review, her code at this time was:

```
(DEFUN FIRST (LIST1) (CAR
```

The major hurdle still remained in this problem, to write the argument to the function. She again looked to the example for guidance about how to code the argument to a function within a function definition. She noted that the first argument in an example like F-TO-C is contained in parentheses: "(DIFFERENCE TEMP 32)." This is because the argument was itself a function call, and function calls must be placed in parentheses. She did not need parentheses in defining FIRST, where she simply wanted to provide the variable LIST1 as the argument. However, she did not recognize the distinction between her situation and the example. She placed her argument in parentheses, producing a complete definition:

```
(DEFUN FIRST (LIST1) (CAR (LIST1)))
```

When the argument to a function is a list, LISP attempts to treat the first thing inside the list as a call to a function. Therefore when she tested FIRST she got the error message "Undefined function object: LIST1." In the past she had corrected the error by quoting the offending list. So she produced the patch

```
(DEFUN FIRST (LIST1) (CAR '(LIST1)))
```

When she tested this function out on a list like (A B C), she got the answer LIST1 rather than A, because LISP now returned the first item of the literal list (LIST1); the single quote in front of (LIST1) causes LISP to treat this as literally a list with the element LIST1. After some work she finally produced the correct code:

```
(DEFUN FIRST (LIST1) (CAR LIST1))
```

We have observed 38 subjects solve this problem in the LISP tutor (Anderson & Reiser, 1985) and a number of further subjects in informal experiments. B.R.'s solution is typical of novice problem solving in many ways. The two places where she has difficulty, specifying the function parameters and specifying the argument to CAR, are the two major points of difficulty for our LISP tutor subjects. The first error she made in specifying the argument to CAR was made by over half of the tutor subjects. An interesting informal observation that we have made is that people with no background at all in LISP, given information about what CAR does, given the function definition template, and given the F-TO-C example, also tend to solve the problem in the same way as B.R. and tend to make the same first error. Thus, it seems that much of the problem solving is controlled by analogy and not by detailed understanding of LISP. The success of our simulation in reproducing B.R.'s behavior also suggests that this problem solving by analogy can be well modeled by a production system with a hierarchical goal structure.

Knowledge Compilation

Our simulation of B.R. after producing the solution to this problem created two new productions that summarized much of the solution process. It does this by a knowledge compilation process (described in J. R. Anderson, 1982, 1983) that collapses sequences of productions into single productions that have the same effect as the sequence. Typically, as in this case, it converts problem solving by domain-general productions into domain-specific productions. The following two productions are acquired by ACT*:

C1: IF the goal is to write a function of one variable
THEN write (DEFUN function (variable)
and set as a subgoal to code the relation calculated
by this function and then write)

C2: IF the goal is to code an argument
and that argument corresponds to a variable of the function
THEN write the variable name

Figure 1 indicates the set of productions that were collapsed to produce each of these productions. The first production summarizes the template matching and analogy process that went into figuring out the syntax of a function definition; thus this production directly produces that syntax without reference to the analogue. The second production summarizes the search that went into finding the correct argument to the function and directly produces that argument.

We gave our simulation another LISP problem to solve, armed with these two additional productions (J. R. Anderson et al., 1984). The problem was to write a function called SECOND, which retrieved the second element of a list. Although SECOND is a more complex function definition, both the simulation and our subject produced much more rapid and successful solutions to this problem. The speedup in the simulation was a result of the fact that fewer productions were involved, thanks to the compiled productions.

One feature of this knowledge compilation process is that it predicts a marked improvement from a first to a second problem of the same kind. Elsewhere (Anderson, 1982) I have commented on this marked speedup in the domain of geometry-proof generation. Studies with the LISP tutor in which students first coded FIRST and then SECOND have produced the same result. Errors in the function definition syntax (compiled into the C1 production given earlier) drop from a median of two in FIRST to a median of zero errors in SECOND, and the time to type in the function template drops from a median of 237 s to a median of 96 s. Success at specifying the variable argument (compiled into C2) changes from a median of three errors to a median of zero errors; time to enter this argument successfully drops from a median of 96 s to a median of 26 s. By any measure these are extremely impressive one-trial learning statistics. They are typical of data obtained with the LISP tutor. On an average of more than 100 rules used by the LISP tutor, the second time a rule is used the number of errors and time both drop to less than half.

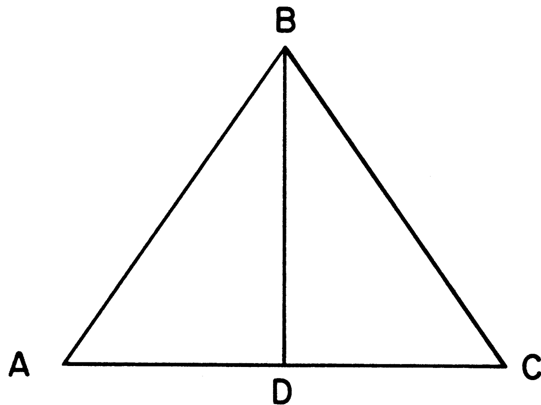

Figure 2. An example geometry proof problem that might appear in a high school geometry text.

Important Features of the ACT Theory*

This simulation of B.R. illustrates a number of important features of the ACT* theory that will serve as the basis for the predictions and empirical tests to be reported in the next section of the article and the theoretical analyses offered in the last section of the article. These features are as follows:

1. *Productions as the units of procedural knowledge.* The major presupposition of the entire ACT* theory and certainly key to the theory of skill acquisition is the idea that productions form the units of knowledge. Productions define the steps in which a problem is solved and are the units in which knowledge is acquired.

2. *Hierarchical goal structure.* The ACT* production system specifies a hierarchical goal structure that organizes the problem solving. Such a hierarchical goal structure is illustrated in Figure 1. This goal structure is a control construct that was not found in many of the original production system models (J. R. Anderson, 1976; Newell, 1973) but now is becoming popular (e.g., Brown & Van Lehn, 1980; Laird et al., 1984). It has proven impossible to develop satisfactory cognitive models that do not have some overall sense of direction in their behavior. As can be seen with respect to this example, the hierarchical goal structure closely reflects the hierarchical structure of the problem. More important than their role in controlling behavior, goals are important to structuring the learning by knowledge compilation. They serve to indicate which parts of the problem solution belong together and can be compiled into a new production.

3. *Initial use of weak methods.* This simulation nicely illustrates the critical role that analogy to examples plays in getting initial performance off the ground. There is a serious question about how the student can do the task before he or she has any productions specific to performing that task. As in this example, the student can in fact imitate the structure of a previous solution. However, this simulation also shows that in contradiction to frequent characterizations of imitation as mindless (e.g., Fodor, Bever, & Garrett, 1974), the analogy mechanisms that implement this process of imitation can be quite sophisticated. For other discussions of the use of analogy in problem solving, see Kling (1971), Carbonell (1983), and Winston (1979).

Although analogy is a frequently used method for getting started in problem solving, it is not the only way. Analogy is one type of a weak problem-solving method (Newell, 1969). Weak problem-solving methods can apply in a wide range of domains. They are called weak because they do not take advantage of domain characteristics. Other examples of weak problem-solving methods include means–ends analysis, working backward, hill climbing, and pure forward search.

For instance, consider the following encoding of hill climbing:

IF the goal is to transform the current state into a goal state
THEN set as subgoals
1. To find the largest difference between the current state and the goal state
2. To find an operator to eliminate that difference
3. To convert the state that results from the application of this operator to the goal state

This production might apply if the goal were to convert the initial state of a geometry proof in Figure 2 into a completed proof. This production sets three goals, which might be satisfied as follows: (a) Since there are no corresponding sides that have been proven congruent, proving corresponding sides congruent might be set as the most important difference to reduce; (b) among the operators of geometry relevant to proving triangles congruent is the reflexive property, which states that segments are congruent to themselves—a fact that might be used to prove that segment $\overline{BD}$ is congruent to itself; and (c) by similarly setting subgoals to get congruences of more corresponding parts, the student will get to the point at which the leg–leg or side–angle–side postulate can be applied.

The important feature of weak-method production rules, such as the hill-climbing production, is that they are cast in a way that makes no specific reference to any particular domain. As long as the knowledge about that domain is appropriately encoded in declarative memory, these productions can apply. Thus, it is critical that the student encode the idea that proving triangles congruent requires proving corresponding sides congruent, or this difference would not have been detected. It is also critical that the student encode the idea that the reflexive rule is relevant to proving sides congruent, or the operator never would have been applied. Which weak methods can apply and how they apply are determined by what declarative knowledge is encoded about the problem domain. The declarative knowledge encoded about the problem domain is again determined by the experiences of the learner: instruction, reading of text, examples studied, and so on.

4. *Knowledge compilation.* All knowledge in the ACT* theory starts out in declarative form and must be converted to pro-

cedural (production) form. This declarative knowledge can be encodings of examples of instructions, encodings of general properties of objects, and so on. The weak problem-solving methods can apply to the knowledge while it is in declarative form and interpret its implications for performance. The actual form of the declarative knowledge determines the weak method adopted. For instance, in the simulation of B.R., analogy was used because the declarative knowledge was almost exclusively knowledge about the template, example programs, and the symbols used therein. In a geometry simulation discussed in J. R. Anderson (1982), the declarative knowledge largely came in the form of rules (e.g., "A reason for a statement can either be that it is given or that it can be derived by means of a definition, postulate, or theorem"). This tended to evoke a working-backwards problem-solving method.

When ACT* solves a problem, it produces a hierarchical problem solution generated by productions. Knowledge compilation is the process that creates efficient domain-specific productions from this trace of the problem-solving episode. The goal structure is critical to the knowledge compilation process in that it indicates which steps of the original solution belong together. This article's Appendix contains a more technical specification of knowledge compilation and the role of goals in this process.

As detailed in the Appendix, there are two distinct processes involved in knowledge compilation. One is proceduralization: Declarative knowledge to which the weak-method production matched is built into the new domain-specific production. Thus, it is no longer necessary to hold this declarative knowledge in working memory. The second is composition: A sequence of productions is collapsed into a single production, which does the work of the sequence. This can produce a considerable speedup. It needs to be emphasized that proceduralizing weak methods does not eliminate the weak methods, nor does composing smaller productions into larger productions eliminate the smaller productions. The original, less specific productions remain around to apply in situations in which the compiled productions cannot.

There are obvious similarities between the composition process and the chunking process described by Miller (1956). In both cases knowledge units that occur together are recoded into higher units. However, the Miller chunking hypothesis applies to how stimuli are encoded and decoded, whereas the current notion is more general and applies to problem solving as well as stimulus encoding. For an attempt to take the more restrictive notion of chunking and apply it in a production system framework, see Newell and Rosenbloom (1981). Extending this chunking mechanism to problem solving is difficult because it is so tied to the process of encoding. This is probably one of the reasons why Laird et al. (1984) revised the chunking mechanism to a form closer to the current composition mechanisms.

As discussed in J. R. Anderson (1982), a number of empirical phenomena are predicted by the knowledge compilation process. In addition to those that are described later in this article, these phenomena include (a) the dramatic one-trial speedup in performance of a new rule, evidence for which was discussed earlier with respect to the LISP tutor; (b) the concomitant dropout of verbalization: since the knowledge is now encoded procedurally rather than declaratively, there is no need to rehearse it in working memory; and (c) the disappearance of set size effects with practice (Schneider & Shiffrin, 1977): productions can be built to directly recognize members of a set and circumvent the need to maintain the set in working memory.

Summary

I have outlined here a complete theory of how new skills are acquired: Knowledge comes in declarative form and is used by weak methods to generate solutions, and the knowledge compilation process forms new productions. The key step is the knowledge compilation process, which produces the domain-specific skill.

Although the four points described earlier are the key to understanding the origin of new skills, they do not address the question of how well the knowledge underlying the skill will be translated into performance. Two factors determine the success of execution of productions. These might be viewed as performance factors that modulate the manifestation of learning except, as is shown later, these factors can also be improved with learning.

5. *Strength.* The strength of a production determines how rapidly it applies, and production rules accumulate strength as they successfully apply. Although accumulation of strength is a very simple learning mechanism, there is good reason to believe that strengthening is often what determines the rate of skill development in the limit. The ACT* strengthening mechanism predicts the typical power-function shape of speedup in skill performance (J. R. Anderson, 1982).

6. *Working memory limitations.* In the ACT* theory there are two reasons why errors are made: The productions are wrong, or the information in working memory on which they operate is wrong. However, the important observation in terms of performance limitations is that perfect production sets can display errors due to working-memory failures. In fact, slips can occur in the ACT* theory only when critical information is lost from working memory and consequently the wrong production fires or the right production fires but produces the wrong result. Just as learning has an impact on production strength, it also seems to have an impact on working-memory errors. Working-memory capacity for a domain can increase (Chase & Ericsson, 1982), reducing the number of such errors with expertise.

The six features reviewed here lead to a number of interesting consequences as we see how they interact within the framework of the ACT* theory. The remainder of this article is devoted to exploring these consequences.

Transfer

The commitment to productions as the units of procedural knowledge has some interesting empirical consequences. In particular it leads to some strong predictions about the nature of transfer between two skills. Specifically, the prediction is that there will be positive transfer between skills to the extent that

the two skills involve the same productions. This is a variation of Thorndike's (1903) identical elements theory of transfer. Thorndike argued that there would be transfer between two skills to the extent that they involved the same content. Thorndike was a little vague on exactly what was meant by content, but he has been interpreted to mean something like stimulus–response pairs (Orata, 1928). The ACT* theory offers a more abstract concept, the production, to replace the more concrete concept, the stimulus–response pair. In ACT* the abstraction comes in two ways. First, the productions are general through their use of variables and hence do not refer to specific elements. Second, there is a hierarchical goal structure controlling behavior, and many of the productions are concerned with generating this goal structure rather than executing specific actions.

Unfortunately, there is a serious problem in putting the prediction about production-based transfer to test because it depends on agreement about which productions underlie two tasks. There is always the danger of fashioning the production system models to fit the observed degree of transfer. Therefore, it is important to select tasks in which there is already independent evidence as to the appropriate production-system analysis. Singley and Anderson (1985) chose text editing because there already existed production-system models or production-system-like models of this task (Card, Moran, & Newell, 1983; Kieras & Polson, 1985). Although these are not ACT* production systems, they nearly completely constrain how one would produce ACT* production-system models for these tasks.

On the basis of the Card et al. model (1983), the first production in performing a text edit would be

E0: IF the goal is to edit a manuscript
THEN set as subgoals
1. To characterize the next edit to perform
2. To perform the edit

The first subgoal sets up the process of scanning for the next thing to be changed and encoding that change. The assumption is that this process should be common to all editors, but the actual performance of the edit can vary from editor to editor. For instance, in the line editor ED, the following productions would fire among others to replace one word by another:

E1: IF the goal is to perform an edit
THEN set as subgoals
1. To locate the line
2. To type the edit

E2: IF the goal is to locate the target line
and the current line is not the target line
THEN increment the line

E3: IF the goal is to locate the target line
and the current line is the target line
THEN POP success

E4: IF the goal is to type an edit
THEN set as subgoals
1. To choose the arguments
2. To type the command

E5: IF the goal is to type the command
THEN set as subgoals
1. To type the command name
2. To type the arguments

E6: IF the goal is to type the command name for substitution
THEN type S

The goal structure of these productions are shown in Figure 3. The triangles indicate goals that would be achieved by productions other than those specified here.

Singley and I (Singley & Anderson, 1986) studied transfer among ED and two other editors: EDT and EMACS. EDT had different command names than ED, and in the subset of EDT used, subjects had to achieve the goal of locating a line by specifying a search string rather than by incrementing the line number. Otherwise, the goal structure was identical for ED and EDT. EMACS, on the other hand, was identical only in the top-level production and the process of characterizing what needed to be done. As a screen editor rather than a line editor, EMACS had very different means of making edits; where the two editors were similar and where they were different is illustrated in Figure 3. From this analysis we made the following two predictions about transfer among editors based on the production overlap:

1. There would be large positive transfer between ED and EDT. The failure of transfer would be localized mainly to the components associated with locating lines, where the two differed substantially in terms of productions. This prediction of high positive transfer held despite the fact that the actual stream of characters typed was quite different in the two editors. What is important is that the two editors had nearly identical goal structures, so it violated Thorndike's surface interpretation of the identical elements principle.

2. There would be much less positive transfer from either ED or EDT to EMACS, despite the fact that they were both text editors. Moreover, what positive transfer there was would be largely confined to the process of characterizing the edit rather than executing it.

Five groups of subjects were recruited from a local secretarial school. They had no prior experience with text editing or computers. The groups were balanced on measures of typing speed and measures of spatial reasoning, which have been shown to predict text-editing performance (Gomez, Egan, & Bowers, in press). One group practiced EMACS for 6 days and is referred to as the one-editor group. A second group practiced ED for 4 days and then learned and practiced EMACS for the last 2 days. A third group practiced EDT for the first 4 days and then transferred to EMACS for the last 2 days. Groups 2 and 3 are referred to as the two-editor groups. A fourth group practiced ED for 2 days, EDT for 2 more, and EMACS for the last 2. A fifth group practiced EDT for 2 days, ED for 2 more, and EMACS for the last 2. These groups are called the three-editor groups. All groups worked in EMACS for the last 2 days of the experiment.

Subjects worked 3 hours a day for each of 6 successive days, starting on a Monday and ending on a Saturday. At the beginning of each odd day (when a new editor might be introduced), they were given a short introduction to a minimum set of commands for each editor. The rest of the time they practiced editing a series of 18-line pages, each of which contained 6 edits to

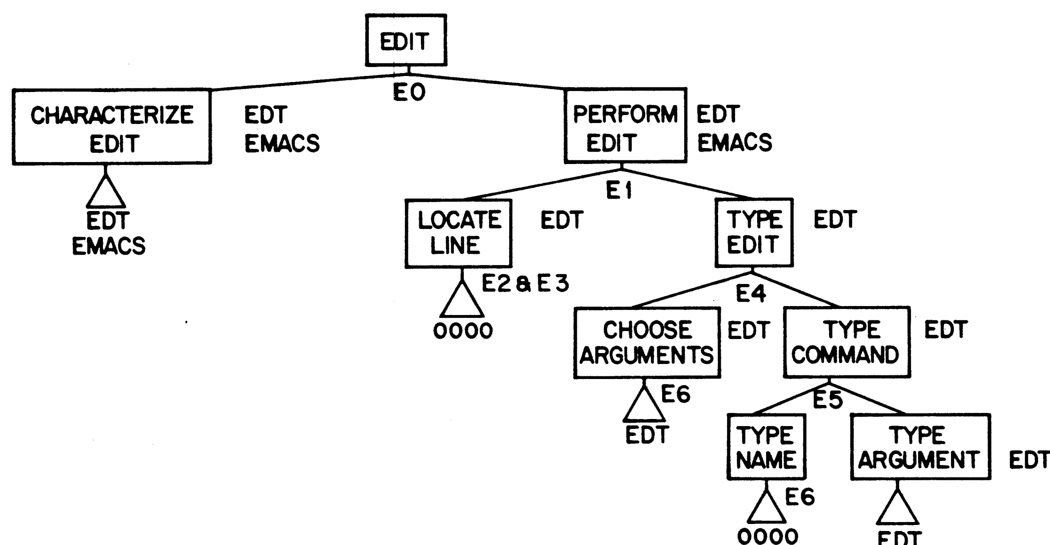

Figure 3. A representation of the goal structure set up by the production for text editing in ED. (Similar formalisms are used as in Figure 1. Goals and goal structures in common with EDT or EMACS are so labeled. Goal structures not in common with either are labeled 0000. A triangle represents a goal structure whose expansion is not represented.)

be performed. The major dependent variable was a modified time-per-edit, which combined both speed and errors. Letting T be the time to perform the six edits on a page and E be the number incorrectly performed, the derived measure T' was

$$T' = \frac{T}{6}\left(1 + \frac{E}{6}\right).$$

The results were basically the same, whether number of errors was corrected for or not, but they proved to be more stable (as determined by ratio variance of an effect to error variance) when measured by the corrected formula T'.

The results are presented in Figure 4 in terms of adjusted time to perform an edit. The first uninteresting thing to note is that EMACS was a good deal more efficient than the line editors; that is, those in the one-editor group were much faster at the beginning when they were using EMACS than those in the other groups, who were using a line editor. The second and more significant observation is that there was virtually no difference between the two- and three-editor groups. In particular, on Day 3, when the three-editor subjects were transferring to a new editor, they were almost as fast as the two-editor subjects who were using that editor for the third day. Thus, the results indicate almost total positive transfer, as predicted. Moreover, an examination of the microstructure of the times indicated that the only place the three-editor subjects were at a disadvantage to the two-editor subjects was in line location, also as predicted.

In contrast, when any of the line-editor groups transferred to EMACS on Day 5, they were at a considerable disadvantage to subjects who had been practicing EMACS all along. They were faster than the EMACS-only subjects were on Day 1, an advantage possibly due to subjects' ability to characterize the edits they have to perform. Consistent with this theory, a control group, who spent 4 days simply retyping an edited text, demonstrated nearly as much transfer to EMACS. Presumably, this group was also learning how to interpret edits marked on the page.

By looking at the timing of the sequence of keystrokes, one can identify long pauses, during which the subject was presumably planning, and bursts of typing, during which the subject was presumably executing a plan. A 2-s interkeystroke interval separated planning pauses from execution phrases. Most of the learning and transfer shown in Figure 4 was due to a decrease in the planning times. The actual time per keystroke in the execution phase did not decrease over the experiment, although there was some reduction in the number of keystrokes per edit, reflecting the acquisition of slightly more efficient procedures. This is exactly the pattern expected. What was being learned were the higher level operations that organize the text editing. The actual execution of the text-editing skill reflected existing typing procedures, which were well learned for this population.

Thus, a production-system analysis allows us to apply an identical element metric to predict transfer. Singley and Anderson (1986) should be consulted for a fine-grained analysis of the data from this experiment in terms of a production-system analysis. Indications are that production-system analyses may prove to be generally useful in predicting transfer. Quite independently, Kieras and Bovair (1986) and Polson and Kieras (1985) achieved similar success using a production-system analysis to predict transfer among a different set of skills.

Negative Transfer Among Cognitive Skills?

A peculiar consequence of the identical elements model is that it does not directly predict any negative transfer among skills in the sense of one procedure running less effectively be-

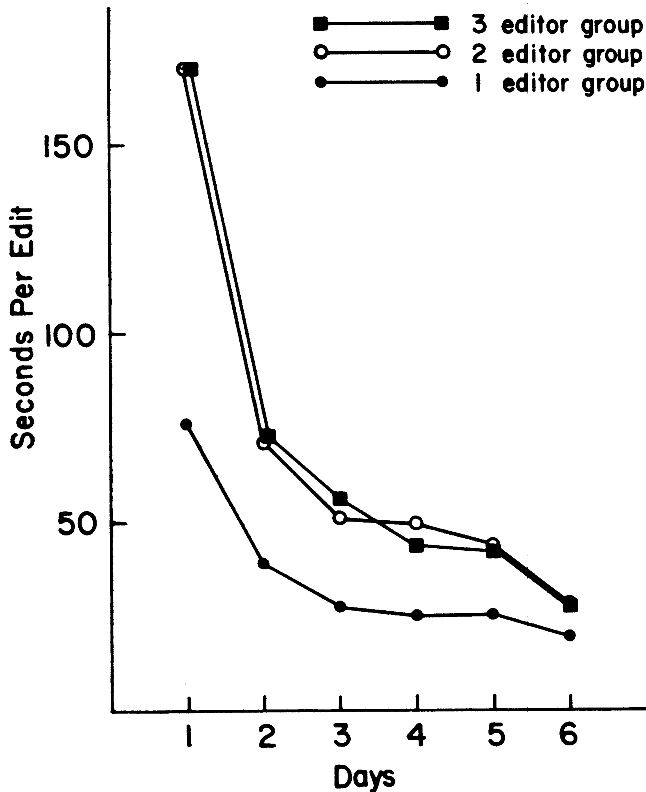

Figure 4. Transfer among text editors.

cause another has been learned. The worst possible case is when there is zero overlap among the productions that underlie a skill, in which case there will be no transfer, positive or negative. Negative transfer can be obtained in terms of an overall measure, such as total time or success, if a procedure that is optimal in one situation is transferred to another domain in which it is not optimal, as in the Einstellung phenomenon (Luchins & Luchins, 1959). However, the Act* analysis of the Einstellung effect assumes perfect transfer of the production set. It is a case of perfect transfer of productions leading to nonoptimal performance, not a case of the productions firing more slowly or incorrectly.

Interference or negative transfer is common with declarative knowledge, but it can be more difficult to get in the domain of skills (J. R. Anderson, 1985, chap. 9), suggesting a basic difference between procedural and declarative knowledge. In the text-editor domain, Singley and I (Singley & Anderson, 1985) decided to create what would be a classic interference design in the verbal learning domain. We created an editor called Perverse EMACS, which was just like EMACS except that the assignment of keys to function was essentially permuted. For instance, in EMACS, control-D erases a letter, escape-D a word, and control-N goes down a line; whereas in Perverse EMACS, control-D goes down a line, and escape-R erases a letter. If we assume that the functionalities are the stimuli and the keys are the responses, we have an A-B,A-Br interference paradigm, which produces maximal interference in paired-associate learning (Postman, 1971).

We compared two groups of subjects. One spent 6 days with EMACS while the other spent their first 2 days with EMACS, then the next 2 days with Perverse EMACS, and the final 2 days with EMACS again. The results of this experiment are shown in Figure 5. Throughout the experiment, including on the first 2 days, during which both groups of subjects were learning EMACS, the Perverse EMACS transfer subjects were slightly worse. However, the only day they were significantly worse was the third, when they transferred to Perverse EMACS. That difference largely disappeared by the fourth day, when they were still working with Perverse EMACS. Compared with performance on Day 1 on EMACS, there was large positive transfer on Day 3 to Perverse EMACS, reflecting the production overlap. The difference between the two groups on Day 3 reflects the cost of learning the specific rules of Perverse EMACS. When the transfer subjects went back to EMACS on Day 5, they picked up on the same point on the learning curve as had the subjects who had stayed with EMACS. (Although they were slower than the pure EMACS subjects in Days 5 and 6, they were no slower than they were on Day 2, when they had last used EMACS.) This is because they had been practicing in Perverse EMACS largely the same productions that they would be using in regular EMACS.

This research on text editors supports the claim that production systems should be viewed seriously as analyses of procedural knowledge. Transfer was predicted by measuring produc-

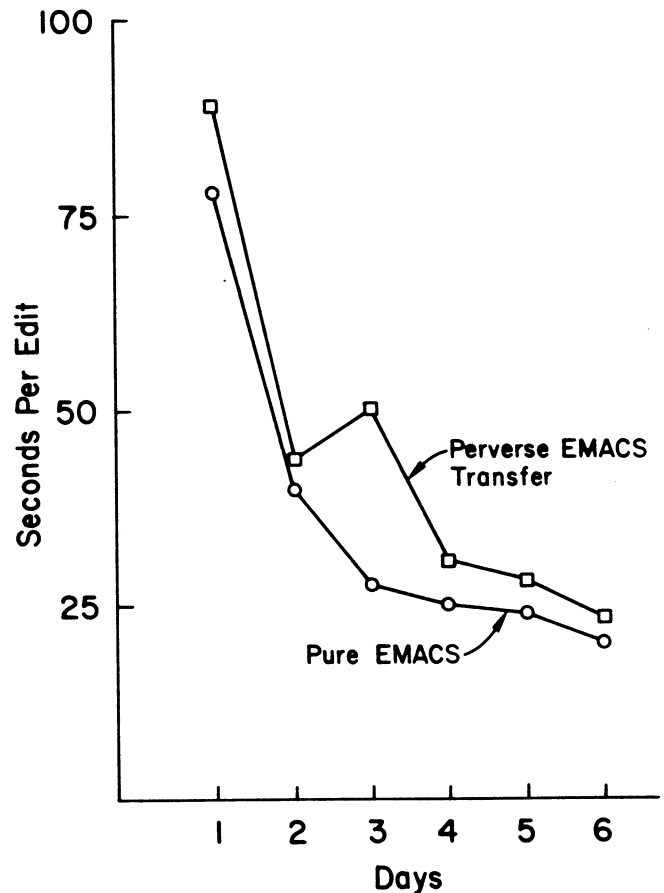

Figure 5. Transfer between EMACS and Perverse EMACS.

Table 1
Design of the McKendree and Anderson Experiment

Item	Frequency	
	Group 1	Group 2
(CAR (CAR '((A B) (C D) (E F))))	6	2
(CAR (CDR '((A B) (C D) (E F))))	2	6
(CDR (CAR '((A B) (C D) (E F))))	2	6
(CDR (CDR '((A B) (C D) (E F))))	6	2

tion overlap (for actual detailed measures see Singley & Anderson, 1986). It is also clear that procedural knowledge behaves differently than declarative knowledge in that it does not seem to be subject to principles of interference.

Effects of Knowledge Compilation

A number of predictions follow from the knowledge compilation process, which is the basic production learning mechanism in the ACT* theory. This process collapses sequences of productions into single operators. If one repeatedly presents problems that require the same combination of productions, the subject should compose productions to reflect that combination and strengthen those productions. This implies that frequently co-occurring combinations of operations should gain an advantage over less frequently co-occurring combinations of the same operations.

The methodology of an experiment by McKendree and Anderson (in press) that was designed to test this prediction is illustrated in Table 1. We were looking at subjects' ability to evaluate various combinations of the LISP expressions CAR, CDR, CONS, and LIST. For instance, subjects had to evaluate (CAR (CDR '((A B)(C D)(E F)))). CDR returns all but the first element of a list, so (CDR '((A B)(C D)(E F))) is ((C D)(E F)). The function CAR will return the first element of this new list, which is (C D), the correct answer to the original problem. We did the experiment to see whether subjects would learn special operators to evaluate combinations of functions, such as CAR–CDR.

Subjects studied various combinations of functions with differential frequency. In Table 1 are the frequencies with which subjects saw combinations of each function each day. (While subjects saw the actual function combination with the given frequency, we had subjects apply that function combination to new arguments each time.) Subjects saw some function combinations 3 times as frequently as other function combinations. As indicated in Table 1, we counterbalanced over two groups which combinations were the high frequency and which were the low frequency. Although the frequency of combination varied between the two groups, the actual frequency of individual functions did not. Thus, an advantage observed for a combination was in fact for that combination, not for the individual functions.

The knowledge-compilation process predicts a speed advantage in evaluating high-frequency combinations because of two factors: (a) The compiled productions for the high-frequency combinations are likely to be learned sooner, and (b) once learned they will acquire strength more rapidly because of their greater frequency.

The experiment involved presenting students with these function combinations, combinations of other pairs of LISP functions that similarly varied in frequency, problems that required subjects to evaluate single LISP functions, and problems that required subjects to evaluate triples of functions. Subjects were given 4 days of practice in which the frequencies of the pairs were maintained as in Table 1. Each day they had to evaluate 150 such functions. We measured subjects' time to type solutions into the computer for these problems. Averaged over the 4 days, subjects took 9.35 s to evaluate the high-frequency combinations and 10.75 s for the low-frequency combinations. The difference in accuracy was 87.7% for high frequency versus 80.8% for low frequency. Both the effects for reaction time and accuracy were significant. These differences were predicted by the assumptions that subjects compile productions to reflect various function combinations, and they strengthen these compiled productions according to the frequency of their applicability.

This also shows that the development of skill is highly specific to its use. That is, even though the subject may be good at evaluating the combinations CAR–CDR and CDR–CAR, the subject may be poor at evaluating the combinations CAR–CAR and CDR–CDR, despite the fact that these evaluations involve the same primitive knowledge as the first two.

Use-Specificity of Knowledge

Declarative knowledge is flexible and not committed to how it will be used. However, knowledge compilation often derives productions from declarative knowledge that can only be used in certain ways. Often the production sets underlying different uses of the same knowledge can be quite different.[2] For instance, productions for language comprehension are different from productions for language generation.

Thus, in many situations the ACT* theory does not predict transfer between different uses of the same knowledge. Neves and Anderson (1981), for instance, compared subjects' ability to generate a proof in a logiclike system with their ability to give the reasons that justified the statements of a worked-out proof. This comparison is interesting because even though these two tasks made use of the same logical postulates, the production systems that implemented the two skills did not overlap. Generating a proof involves search for a path between givens of the problem and the statement to be proven, whereas giving a reason involves a search through the rule space looking for a rule whose consequence matches the statements to be justified.

[2] This lack of overlap among the production sets is not dependent on any particular choice of production representation. A production is specific to a condition–action pairing. Since the actions in proof generation versus proof checking are fundamentally different, productions sets specific to these two tasks need different types of actions and could not overlap.

Neves and I found evidence that 10 days of practice at giving a reason had no significant positive transfer to proof generation. However, that evidence was based on a very small sample of subjects and informal evaluation of their proof-generation skills.

McKendree and Anderson (in press) looked at this issue more thoroughly in their study of LISP evaluation skills. In that experiment, subjects evaluated the results of LISP functions applied to arguments. A typical problem was

(LIST (CAR '(A B)) '(B C)) = ?,

where the correct answer is (A (B C)). In addition to practicing these evaluation problems a great deal, on the first and fourth day subjects received a few generation problems that were isomorphic to these evaluation problems. For the previous example, the isomorphic generation problem required that subjects write a LISP expression that would operate on (A B) and (B C) and produce (A (B C)), for which this expression would be the correct answer.

Again, productions that solved the evaluation problems and productions that solved the generation problem did not overlap, although these productions were based on the same abstract declarative knowledge. However, this declarative knowledge is compiled into different production form for a different use. For instance, the LISP function LIST, which inserts its arguments in a list, is encoded in evaluation form by the following production:

```
IF    the goal is to evaluate (LIST X Y)
      and A is the value of X
      and B is the value of Y
THEN  (A B) is the value
```

In generation form, this is encoded by the following production (actually based on our LISP tutor):

```
IF    the goal is to code (A B)
THEN  use the function LIST and set as subgoals
      1. To code A
      2. To code B
```

This is basically the same knowledge in different form, but the ACT* theory holds that the difference in form is significant and that there will not be transfer between the two forms. Most people find this prediction counterintuitive. Standard educational practice (at least at the college level) involves getting students to understand general principles and how to apply them in one domain. The belief is that if they have this knowledge, they will know how to apply the principles in any domain.

The McKendree and Anderson (in press) experiment allowed us to address the issue of whether massive practice on evaluation would generalize to generation. Averaging over single-, pair-, and triple-function evaluations, subjects' evaluation time dropped from 18.2 s on Day 1 to 10.0 s on Day 4. Subjects' accuracy increased from 58% correct to 85% correct over the 4 days that they were practicing evaluation. In contrast, they only changed from 71% correct in the first-generation test at the end of the first day to 74% correct in the second-generation test at the end of the fourth day. (We did not have reaction time measures for the generation task because subjects had to work out the solutions by hand rather than typing them into a computer.) The pattern of generation errors was similar in the first and fourth days. For instance, the most common error on both days was confusion of LIST and CONS. Subjects had this difficulty despite the fact that such confusions were infrequent in evaluating LIST and CONS.

Thus, little transferred from evaluation to generation, two different skills that use the same knowledge. This reinforces the idea that production representation captures significant features of our procedural knowledge and that differences between production forms are psychologically real.

The results of the McKendree and Anderson (in press) experiment show that there are qualitative changes with practice in a skill, and it is not just a matter of everything becoming uniformly better. Subjects became especially effective at the frequently recurring problem patterns, and subjects became more proficient at particular uses of the knowledge.

In combination with the earlier mentioned results on transfer, a rather startling pattern is emerging. Nearly total transfer across tasks is possible if the same knowledge is used in the same way (e.g., transfer between text editors ED and EDT). On the other hand, almost no transfer occurs if the same knowledge is used in different ways (e.g., transfer between evaluation and generation).

Effects of Proceduralization

Part of the compilation process is the elimination of the need to hold declarative information in working memory for interpretation; rather, that information is built into the proceduralized production. A number of predictions about qualitative changes in skill execution follow from proceduralization. J. R. Anderson (1982) has discussed many of these. These changes include disappearance of verbalization of the declarative knowledge and decrease in loss of partial products in the problem solution.

An unreported part of the Singley and Anderson (1985) experiment was a specific test of the qualitative changes implied by proceduralization. Before proceduralization, information about how to use the text editor must be held in working memory. Therefore, there should be declarative interference between performing the text-editing skill and remembering similar declarative knowledge about other text editors, but this content-based interference should disappear when the knowledge is proceduralized. In a dual-task experiment Singley and Anderson required subjects to memorize facts being presented to them in the auditory modality and to perform text-editing operations involving the visual modality. We varied the content of the information they were being asked to remember to be either the description of a hypothetical text editor or something unrelated to text editing. Memory for the hypothetical text-editing information started out poorly but improved with practice. Memory for the unrelated information started out better but did not improve. This is consistent with the view that subjects were initially suffering declarative interference with the text-editor information from the execution of the text-editor skill, which disappeared when the skill became proceduralized with practice.

Working Memory Limitations

As noted earlier, there is reason to believe that in domains such as writing computer programs, the major source of errors

will be working-memory failures. The student gets good feedback on the facts about the programming language, and so misconceptions would probably be rapidly stamped out. On the other hand, the student is being asked to coordinate a great deal of knowledge in writing a program, and one would expect working-memory capacity to be frequently overwhelmed.

J. R. Anderson and Jeffries (1985) looked at the errors made by over 100 students in introductory LISP classes. Over 30% of all answers were errors. These results were consistent with the working-memory failure analysis. Increasing the complexity of one part of the problem increased errors in another part, suggesting that the capacity requirements to represent one part overflowed and had an impact on the representation of another part. Further, errors were nonsystematic; that is, subjects did not repeat errors as one would expect if there were some systematic misconception.

Interestingly, the best predictor of individual subject differences in errors on problems that involved one LISP concept was number of errors on other problems that involved different concepts. This in turn was correlated with amount of programming experience and experience with LISP. Subjects differed in their general proneness to these working-memory errors in a domain, and experience in the domain increased working-memory capacity. Jeffries et al. (1981) found that memory capacity was a major difference separating beginning programmers from experienced programmers. Chase and Ericsson (1982) showed that experience in a domain can increase capacity for that domain. Their analysis implied that storage of domain information in long-term memory became so reliable that long-term memory developed into an effective extension of short-term memory.

Consequences of Working-Memory Failures

Loss of declarative information from working memory can cause good procedures to behave badly. This can happen in a number of ways. Most simply, if one loses information that is needed in the answer, the answer will be missing this information. Interestingly, J. R. Anderson and Jeffries (1985) found that dropping parentheses was the most common error in calculating LISP answers. This happened more than twice as frequently as adding parentheses. This error would occur if parentheses were lost from the representation of the problem students were manipulating in working memory.

A more profound type of working-memory error occurs when students lose a goal they are trying to achieve. Katz and Anderson (1985) found that a major error in syntactically correct programs was the omission of parts of code that correspond to goals. Often subjects mentioned these goals in protocols taken while writing the programs, but they just never got around to achieving them.

A third kind of error that can occur because of working-memory loss is that subjects will lose a feature that discriminates between two productions and fire the wrong production. For example, in geometry students will apply a postulate involving congruence of segments to information about equality of measure of segments. When queried, students can explain why they are wrong, but there is just a momentary intrusion of the postulate. An example in the context of LISP is the confusion of similar LISP functions in programs. Again, subjects can explain their mistake when queried, but it seems that the production generating the wrong function intruded. J. R. Anderson and Jeffries (1985) have shown that the frequency of these intrusions can be increased by increasing the concurrent memory load on subjects. This is consistent with the view that discriminating information is being crowded out of working memory.

Norman (1981) has provided a classification of what he calls action slips, which appear to be identical to the working-memory errors described here. However, he interpreted these with respect to an activation-based schema system rather than a production system. In regard to cognitive-level errors that occur in LISP programming, there has hardly been adequate research to separate the two views. However, evidence that such errors increase with working-memory load is certainly consistent with the working-memory-plus-production-system hypothesis.

Immediate Feedback

The importance of immediate feedback is an interesting consequence of the interaction between compilation and working-memory limitations. For knowledge compilation to work correctly with delayed feedback, it is necessary to preserve information about the decision point until information is obtained about the right decision. Then a production can be compiled, attaching the correct action to the critical information at the decision point. Working-memory failures will cause information about the decision point to be lost, and consequently incorrect operators will be formed. Such failures will increase with the delay of the feedback. Thus, the commonly recommended practice of having subjects discover their errors may have a negative impact on learning rate.

M. W. Lewis and Anderson (1985) performed an experiment to see whether it was better to have subjects discover errors or to inform them immediately that an error had been made. We had subjects learn to solve dungeon-quest-like games. A typical situation that a subject might face is shown in Figure 6. Certain features of a room were described, and the subject could try certain operators (like waving the wand, slaying the room-keeper) that would move the subject on to other rooms. A subject might move to a room that leads to a dead end. In the immediate-feedback condition, subjects were immediately told they had entered such a room, whereas in the delay condition, they had to discover by further exploration that they had reached a dead end.

M. W. Lewis and Anderson found that subjects performed considerably better, in terms of number of correct moves, when given immediate feedback. Other researchers (R. C. Anderson, Kulhavy, & Andre, 1972) found that the type of feedback was critical to determining its effectiveness. It was important that the feedback not give away the correct answer but only signal to the learner that an error had been made, because the student needed to go through the process of calculating the correct answer rather than copying it if an effective operator was going to be compiled. If subjects copied an answer, they would compile a procedure for such copying, which would make them more efficient at copying answers but not at producing them.

The issue of immediate feedback versus learning by discovery is controversial, and I do not mean to imply that immediate feedback is the perfect condition for all types of learning. For

You are in a room which contains

a brick fireplace
a foul-smelling troll roomkeeper
a polished brass door

You hear	wicked laughter
The room also contains	hairy spiders
The room is	musty and dusty
You also notice	burning torches

... What do you want to do?

Figure 6. An example of the problem description used in the M. W. Lewis and Anderson (1985) experiment. (From Figure 5, "Discrimination of Operator Schemata in Problem Solving: Learning From Examples" by M. Lewis and J. R. Anderson, 1985, *Cognitive Psychology, 17,* pp. 26–65. Copyright 1985 by Academic Press. Reprinted by permission.)

instance, M. W. Lewis and Anderson (1985) found, reasonably enough, that subjects allowed to learn by discovery were better able to recognize errors. Similarly, if one wants students to learn a skill such as debugging computer programs, one might let the students make errors in their programs so that there is something to debug. However, the theoretical point is that immediate feedback on an operator is important to learning the operator. Both in the error detection and the debugging examples, skill at successfully generating solutions is being sacrificed so that circumstances can be created to facilitate learning how to deal with errors.

Further Implications for Instruction

This view of skill acquisition has some important implications for instruction. One straightforward implication is that in a system in which one can only learn skills by doing them, the importance of formal instruction diminishes and the importance of practice increases. There is already reason to doubt the value of elaborate textual instruction for communicating declarative knowledge (Reder & Anderson, 1980). Carroll (1985) showed that standard text-editor manuals can be made more effective if they shortened and focus just on the information necessary to perform the skill. Carroll suggested that a shorter text gives the student more opportunity to focus on practicing the skill. Reder, Charney, and Morgan (1986) showed that the only elaborations that are effective in a personal-computer manual are those that provide examples of how to use the computer.

Much technical instruction is not focused on telling students how to solve problems in the technical domain but on explaining why solutions work in that domain. This distinction might seem subtle, but it is important. Recursive functions in the LISP language are good examples. Most textbooks focus on how recursive programs are evaluated in LISP to produce their results rather than on how to create a recursive function to solve a programming problem. Pirolli and Anderson (1984) tested whether being told how recursive programs are created would be more effective than being told how recursive functions are evaluated for the purpose of writing programs.

There are analyses in AI (Rich & Shrobe, 1978; Soloway & Woolf, 1980) about how to write recursive programs. Pirolli and Anderson developed an ACT* production-system model of recursive programming based on these analyses. In an experiment on this model, students received either the standard instruction about how recursion is evaluated or the instruction about how to generate recursive code, on the basis of the production-system model. The standard instruction was modeled on existing textbooks. The how-to instruction basically described the productions in our model.

The students tried to write a set of four simple recursive programs. Students given the how-to information took 57 min, whereas students given the how-it-works information took 85 min. Of course, how-to and how-it-works information need not be mutually exclusive as they were in this experiment, but this experiment makes the point that instruction for a skill is most effective when it directly provides information needed in a production-system model of that skill.

Although the Pirolli and Anderson (1984) experiment confirmed that how-to instruction was better than conventional instruction, there is reason to question how effective it really was. Even the best designed instruction is given out of the context of the actual problem-solving situation and is often difficult for students to integrate properly into their problem-solving efforts. There are memory problems in retrieving what has been read in one context when it is needed in another. Second, it is easy to misunderstand abstract instruction. For instance, many high school students misunderstand the following statement of the side–angle–side postulate: "If two sides and the included angle of one triangle are congruent to the corresponding parts of another triangle, the triangles are congruent." They interpret "included" to mean included within the triangle rather than included between the two sides. This occurs despite the fact that the statement of the postulate is accompanied by an appropriate diagram. Although this particular misunderstanding is so common one might imagine placing remedial instruction right in the text, one cannot write remedial instruction into the textbook for all possible confusions. This is one of the major advantages of instruction by private tutors: Such tutors can diagnose an individual's particular misconceptions and provide the appropriate remedial instruction in context for just those misconceptions.

In a number of studies of the LISP tutor, J. R. Anderson and Reiser (1985) contrasted one group solving problems on their own with another group solving problems with the LISP tutor. The important aspect of the tutor is that it remediates students' confusions by providing instruction in the context of these confusions. One study compared two groups of students who read instruction on recursion much like the instruction used in the original by Pirolli and Anderson (1984) study. However, one group had the LISP tutor, which forced them to follow this advice and corrected any misconceptions they had about what that advice meant. The other group was left to try to use the

instruction as they saw fit, as was the case in the original Pirolli and Anderson study. The tutor group took an average of 5.76 hr to go through the problems and got an average of 7.6 points on the recursion section of a paper-and-pencil final exam. The no-tutor group took 9.01 hr and got 4.8 points.

Much of the advantage of intelligent computer-based tutors is their ability to facilitate the conversion of abstract declarative instruction into procedures by providing that instruction appropriately in context. Of course, a prerequisite to this is being able to correctly interpret the student's behavior in terms of a cognitive model. Much of our actual efforts in intelligent tutoring have gone into developing such a cognitive model and developing techniques for diagnosing student behavior.

Inductive Learning

The ACT* learning mechanisms of strengthening and compilation have been the focus of this article so far. In the original ACT* theory, there were two other learning processes that were responsible for formation of new productions. These learning processes actually changed what the system did rather than simply make existing paths of behavior more effective. These were the inductive learning mechanisms of generalization and discrimination. In contrast to our success in relating strengthening and compilation to empirical data, we have had a notable lack of success in our tests for certain purported properties of generalization and discrimination.

The mechanisms of generalization and discrimination can be nicely illustrated with respect to language acquisition. Suppose a child has compiled the following two productions from experience with verb forms:

IF the goal is to generate the present tense of KICK
THEN say KICK + S
IF the goal is to generate the present tense of HUG
THEN say HUG + S

The generalization mechanism would try to extract a more general rule that would cover these cases and others:

IF the goal is to generate the present tense of X
THEN say X + S

where X is a variable.

Discrimination deals with the fact that such rules may be overly general and need to be restricted. For instance, this example rule generates the same form, whether the subject of the sentence is singular and plural. Thus, it will generate errors. By considering different features in the successful and unsuccessful situations and using the appropriate discrimination mechanisms, the child would generate the following two productions:

IF the goal is to generate the present tense of X
and the subject of the sentence is singular
THEN say X + S
IF the goal is to generate the present tense of X
and the subject of the sentence is plural
THEN say X

These learning mechanisms have proven to be quite powerful, acquiring, for instance, nontrivial subsets of natural language (J. R. Anderson, 1983).

These discrimination and generalization mechanisms are very much like knowledge-acquisition mechanisms that have been proposed in the artificial intelligence literature (e.g., Hayes-Roth & McDermott, 1976; Vere, 1977). They are called syntactic methods, in that they only look at the form of the rule and the form of the contexts in which it succeeds or fails. There is no attempt to use any semantic knowledge about the context to influence the rules that are formed. A consequence of this feature in the ACT* theory is that generalization and discrimination are regarded as automatic processes, not subject to strategic influences and not open to conscious inspection. The reports of unconscious learning by Reber (1976) are consistent with this view. He found that subjects learned from examples whether strings were consistent with the rules of finite state grammars, without ever consciously formulating the rules of these grammars.

There are now a number of reasons for questioning whether the ACT* theory is correct in its position that inductive learning is automatic. First, there is evidence that the generalizations people form from experience are subject to strategic control (Elio & Anderson, 1984; Kline, 1983). In Elio and Anderson's prototype formation experiment, subjects could adopt either memorization or hypothesis formation strategies, and the two strategies led them to differential success, depending on what the instances were. Second, M. W. Lewis and Anderson (1985) found that subjects were able to restrict the application of a problem-solving operator (i.e., discriminate a production) only if they could consciously formulate the discrimination rule. Indeed, there is now reason to believe that even in the Reber unconscious learning situation, subjects have conscious access to low-level rules that help them classify the examples (Dulany, Carlson, & Dewey, 1984). They do not form hypotheses about finite-state grammars but do notice regularities in the example sentences (e.g., grammatical strings have two X's in second and third position).

Another interesting problem with the ACT* generalization mechanism is that subjects often appear to emerge with generalizations from a single example (Elio & Anderson, 1983; Kieras & Bovair, 1986), whereas the syntactic methods of ACT* are intended to extract the common features of a number of examples. It is interesting to note that the coding of the LISP function FIRST at the beginning of this article was based on extraction of generalizations by analogy from the single F-TO-C function. By compiling that analogy process, the student simulation emerged with general production rules for the syntax of function definitions and for specifying a function argument within a function–definition context. It is hard to see how one could extract a generalization from a single example without some semantic understanding of why the example worked. Otherwise, it is not possible to know which features are critical and which features can be ignored (i.e., generalized over).

This is an example of a more general point, toward which my current thinking about induction (J. R. Anderson, 1986) is leaning: The actual process of forming a generalization or a discrimination can be modeled by a set of problem-solving productions. The previous example is a case in which the problem-solving method is analogy but other weak problem-solving methods can apply also. The inductive processes of generalization and discrimination are things that can be imple-

mented by a set of problem-solving productions. Also, because productions are sensitive to the current contents of working memory, induction can be influenced by semantic and strategic factors, as it apparently is. Since the information used by the productions has to be in working memory, people should have conscious access to the information they are using for induction, which they apparently do. Knowledge compilation can convert these inductive problem-solving episodes into productions that generalize beyond the current example.

Thus, proposals for separate learning processes of discrimination and generalization are unnecessary. Besides this argument of parsimony, the empirical evidence does not seem consistent with such unconscious learning processes. On the other hand, the empirical evidence does seem consistent with the unconscious strengthening and compilation processes. Subjects have not reported the changing strengths of their procedures nor compilation of productions. Thus, these learning processes do not seem to involve computations that leave partial products in working memory to be reported, in contrast to discrimination and generalization, which do.

Conclusions

The ACT* theory contains within it the outline of an answer to the epistemological question: How does structured cognition emerge? The answer is that we approach a new domain with general problem-solving skills such as analogy, trial-and-error search, or means–ends analysis. Our declarative knowledge system has the capacity to store in relatively unanalyzed form our experiences in any domain, including instruction (if it is available), models of correct behavior, successes and failures of our attempts, and so on. A basic characteristic of the declarative system is that it does not require one to know how the knowledge will be used in order to store it. This means that we can easily get relevant knowledge into our system but that considerable effort may have to be expended when it comes time to convert this knowledge to behavior.

The knowledge learned about a domain feeds the weak but general problem-solving procedures that try to generate successful behavior in that domain. Knowledge compilation produces new production rules that summarize the outcome of the efforts, and strengthening enhances those production rules that repeatedly prove useful. The knowledge-compilation process critically depends on the goal structures generated in the problem solving. These goal structures indicate how to chunk the sequence of events into new productions.

Most of our research has examined learning by adults or near-adults of very novel skills like LISP programming. However, the learning theory is intended to generalize downward to child development. Young children are the universal novices: Everything they are asked to learn is a novel skill. Experimentation with LISP programming provides a good approximation to the child's situation in the experimentally more tractable adult. The claim is that children bring weak problem-solving methods to new domains and, like adults, eventually compile domain-specific procedures. Klahr (1985) has found evidence for early use of weak methods like means–ends analysis and hill climbing.

The theory of knowledge proposed here is not an extreme empiricist theory in that it requires that the learner bring some prior knowledge to the learning situation. On the other hand, the amount and kind of knowledge being brought to bear is quite different than assumed in many nativist theories. In fact, the real knowledge comes in the form of the weak but general problem-solving procedures that the learner applies to initial problems in these domains. These procedures enable the initial performance and impose a goal structure on the performance so that the knowledge compilation process can operate successfully. Once the knowledge has been organized into a goal structure, knowledge compilation chunks contiguous portions of the goal structure. Thus, knowledge compilation is really an enhanced version of learning by contiguity. The goal structures provide the "belongingness" that Thorndike (1935) came to recognize as a necessary complement to learning by contiguity.

Of course, much learning is not in completely novel domains. If the learner knows something relevant, then the course of knowledge acquisition can be fundamentally altered. We saw how productions can transfer wholesale from one domain to another. Even if this is not possible, one can use the structure of a solution in one domain as an analogue for the structure of the solution in another domain. Also, one can use one's knowledge to interpret the declarative information and store a highly sophisticated interpretation of the knowledge that is presented. We have been comparing students learning LISP as a first programming language with students learning LISP after Pascal. The Pascal students learn LISP much more effectively, one of the reasons being that they better appreciate the semantics of various programming concepts, such as what a variable is. Weak problem-solving methods like analogy can be much more effective if they operate on a rich representation of the knowledge. Indeed, Pirolli and Anderson (1985) looked at the acquisition of recursive programming skill and found that almost all students developed new programs by analogy to example recursive programs but that their success was determined by how well they understood why these examples work.

If this view of knowledge development is correct, the relevant question concerns the nature and origin of the weak methods. Here they are modeled in ACT* as production sets, and in their analysis of weak methods Laird and Newell (1983) model them in a similar way. In ACT* they are assumed as givens, whereas Laird and Newell treat them as the product of encoding the domain. Laird and Newell propose that people start out with one universal weak method that serves to interpret knowledge encoded about a domain. This universal weak method is encoded as a set of productions. When the person encodes new information about a specific domain, this information is encoded by additional productions. (The Laird and Newell theory does not have declarative, nonproduction encodings.) The domain-specific productions in the context of the universal weak method can lead to other weak methods. Currently, there is no clear empirical basis for separating these two views of the origin of weak methods.

References

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science, 9,* 147–169.
Adelson, B. (1981). Problem solving and the development of abstract

categories in programming languages. *Memory and Cognition, 9,* 422–423.

Anderson, J. R. (1976). *Language, memory, and thought.* Hillsdale, NJ: Erlbaum.

Anderson, J. R. (1981). *Cognitive skills and their acquisition.* Hillsdale, NJ: Erlbaum.

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). *Cognitive psychology and its implications* (2nd ed.). New York: Freeman.

Anderson, J. R. (1986). Knowledge compilation: The general learning mechanism. In R. Michalski, J. Carbonnell, & T. Mitchell (Eds.), *Machine learning II* (pp. 289–310). Palo Alto, CA: Tioga Press.

Anderson, J. R., Boyle, C. F., Farrell, R., & Reiser, B. (1984). Cognitive principles in the design of computer tutors. In *Sixth Annual Conference of the Cognitive Science Program* (pp. 2–16).

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

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

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

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

Anderson, R. C., Kulhavy, R. W., & Andre, T. (1972). Conditions under which feedback facilitates learning from programmed lessons. *Journal of Educational Psychology, 63,* 186–188.

Brown, J. S., & Van Lehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4,* 379–426.

Carbonell, J. G. (1983). Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning* (pp. 137–162). Palo Alto, CA: Tioga Press.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human–computer interaction.* Hillsdale, NJ: Erlbaum.

Carroll, J. (1985). *Designing minimalist training materials* (Research Rep.). Yorktown Heights, NY: IBM Watson Research Center.

Chase, W. G., & Ericsson, K. A. (1982). Skill and working memory. In G. H. Bower (Ed.), *The psychology of learning and motivation* (Vol. 16). New York: Academic Press.

Chase, W. G., & Simon, H. A. (1973). The mind's eye in chess. In W. G. Chase (Ed.), *Visual information processing* (pp. 215–281). New York: Academic Press.

Cheng, P. W., & Holyoak, K. J. (1985). Pragmatic reasoning schemas. *Cognitive Psychology, 17,* 391–416.

Chi, M. T. H., Glaser, R., & Farr, M. (Eds.). (in press). *The nature of expertise.* Hillsdale, NJ: Erlbaum.

Chomsky, N. (1980). Rules and representations. *Behavioral and Brain Sciences, 3,* 1–61.

Clark, H. H. (1974). Semantics and comprehension. In R. A. Sebeok (Ed.), *Current trends in linguistics.* The Hague: Mouton.

Dulany, D. E., Carlson, R. A., & Dewey, G. I. (1984). A case of syntactical learning and judgment: How conscious and how abstract? *Journal of Experimental Psychology: General, 113,* 541–555.

Elio, R., & Anderson, J. R. (1983). Effects of category generalizations and instance similarity on schema abstraction. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 7,* 397–417.

Elio, R., & Anderson, J. R. (1984). The effects of information order and learning mode on schema abstraction. *Memory and Cognition, 12,* 20–30.

Fodor, J. A. (1983). *The modularity of mind.* Cambridge, MA: MIT Press/Bradford Books.

Fodor, J. A., Bever, T. G., & Garrett, M. F. (1974). *The psychology of language.* New York: McGraw-Hill.

Gomez, L. M., Egan, D. E., & Bowers, C. (in press). Learning to use a text editor: Some learner characteristics that predict success. *Human Computer Interaction.*

Hayes-Roth, F., & McDermott, J. (1976). Learning structured patterns from examples. In *Proceedings of the Third International Joint Conference on Pattern Recognition* (pp. 419–423). Coronado, CA: International Joint Conference on Pattern Recognition.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 255–283). Hillsdale, NJ: Erlbaum.

Katz, I. R., & Anderson, J. R. (1985). *An exploratory study of novice programmers' bugs and debugging behavior.* Unpublished manuscript.

Kieras, D. E., & Bovair, S. (1986). The acquisition of procedures from text: A production system analysis of transfer of training. *Journal of Memory and Language, 25,* 507–524.

Kieras, D. E., & Polson, P. G. (1985). An approach to the formal analysis of user complexity. *International Journal of Man–Machine Studies, 22,* 365–394.

Klahr, D. (1985). Solving problems with ambiguous subgoal ordering: Pre-schoolers' performance. *Child Development, 56,* 940–952.

Klahr, D., Langley, P., & Neches, R. (in press). *Self-modifying production systems: Models of learning and development.* Cambridge, MA: MIT Press/Bradford Books.

Kline, P. J. (1983). *Computing the similarity of structured objects by means of a heuristic search for correspondences.* Unpublished doctoral dissertation, University of Michigan, Ann Arbor.

Kling, R. E. (1971). A paradigm for reasoning by analogy. *Artificial Intelligence, 2,* 147–178.

Laird, J. E., & Newell, A. (1983). Universal weak method: Summary of results. In *Proceedings of the Eight IJCAI.* IJCAI.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1984). Towards chunking as a general learning mechanism. In *Proceedings of AAAI84.* AAAI.

Langley, P. (1982). Language acquisition through error recovery. *Cognition and Brain Theory, 5,* 211–255.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980). Models of competence in solving physics problems. *Cognitive Science, 4,* 317–345.

Lesgold, A. M. (1984). Acquiring expertise. In J. R. Anderson & S. M. Kosslyn (Eds.), *Tutorials in learning and memory* (pp. 31–60). San Francisco: Freeman.

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

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

Luchins, A. S., & Luchins, E. H. (1959). *Rigidity of behavior: A variational approach to the effect of einstellung.* Eugene: University of Oregon Books.

MacWhinney, B. (in press). *Mechanisms of language acquisition.* Hillsdale, NJ: Erlbaum.

McClelland, J. L. (1985). Putting knowledge in its place: A scheme for programming parallel processing structures on the fly. *Cognitive Science, 9,* 113–146.

McClelland, J. L., & Rumelhart, D. E. (Eds.). (1986). *Parallel distributed processing: explorations in the microstructure of cognition* (Vol. 2). Cambridge, MA: MIT Press/Bradford Books.

McKendree, J., & Anderson, J. R. (in press). Frequency and practice

effects on the composition of knowledge in LISP evaluation. In J. M. Carroll (Ed.), *Cognitive aspects of human-computer interaction.*

Michalski, R. S. (1983). Theory and methodology of inductive learning. In R. Michalski, J. G. Carbonnell, & T. M. Mitchell (Eds.), *Machine learning* (pp. 83-134). Palo Alto, CA: Tioga Press.

Michalski, R. S., Carbonnell, J. G., & Mitchell, T. M. (1983). *Machine learning.* Palo Alto, CA: Tioga Press.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review, 63,* 81-97.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence, 18,* 203-226.

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* (pp. 57-84). Hillsdale, NJ: Erlbaum.

Newell, A. (1969). Heuristic programming: Ill-structured problems. In J. Aronofsky (Ed.), *Progress in operations research* (Vol. 3, pp. 361-414). New York: Wiley.

Newell, A. (1973). Production systems: Models of control structures. In W. G. Chase (Ed.), *Visual information processing* (pp. 463-526). New York: Academic Press.

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

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

Orata, P. T. (1928). *The theory of identical elements.* Columbus: Ohio State University Press.

Pinker, S. (1984). *Language learnability and language development.* Cambridge, MA: Harvard University Press.

Pirolli, P. L., & Anderson, J. R. (1984, November). *The role of mental models in learning to program.* Paper presented at the Twenty-fifth Annual Meeting of the Psychonomic Society, San Antonio, TX.

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.

Polson, P. G., & Kieras, D. E. (1985). A quantitative model of the learning and performance of text editing. In *Proceedings of the Conference on Human Factors in Computer Systems* (pp. 207-212). New York: Association for Computing Machinery.

Postman, L. (1971). Transfer, interference, and forgetting. In L. W. King & L. A. Riggs (Eds.), *Experimental psychology* (pp. 1019-1132). New York: Holt, Rinehart & Winston.

Reber, A. S. (1976). Implicit learning of synthetic languages: The role of instructional set. *Journal of Experimental Psychology: Human Learning and Memory, 2,* 88-94.

Reder, L. M., & Anderson, J. R. (1980). A comparison of texts and their summaries: Memorial consequences. *Journal of Verbal Learning and Verbal Behavior, 19,* 121-134.

Reder, L. M., Charney, D. H., & Morgan, K. I. (1986). The role of elaborations in learning a skill from an instructional text. *Memory and Cognition, 14,* 64-78.

Rich, C., & Shrobe, H. (1978). Initial report of a LISP programmers' apprentice. *IEEE Transactions on Software Engineering, SE-4(6),* 456-466.

Rumelhart, D. E., & McClelland, J. L. (Eds.). (in press). *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1). Cambridge, MA: MIT Press/Bradford Books.

Rumelhart, D. E., & Ortony, A. (1976). The representation of knowledge in memory. In R. C. Anderson, J. R. Spiro, & N. E. Montague (Eds.), *Schooling and the acquisition of knowledge* (pp. 99-135). Hillsdale, NJ: Erlbaum.

Rumelhart, D. E., & Zipser, D. (1985). Feature discovery by competitive learning. *Cognitive Science, 9,* 75-112.

Sauers, R., & Farrell, R. (1982). *GRAPES user's manual* (Tech. Rep. No. 1). Carnegie-Mellon University, Department of Psychology.

Schank, R. C., & Abelson, R. P. (1977). *Scripts, plans, goals, and understanding: An inquiry into human knowledge structures.* Hillsdale, NJ: Erlbaum.

Schneider, W., & Shiffrin, R. M. (1977). Controlled and automatic human information processing: I. Detection, search, and attention. *Psychological Review, 84,* 1-66.

Singley, M. K., & Anderson, J. R. (1985). The transfer of text-editing skill. *Journal of Man-Machine Studies, 22,* 403-423.

Singley, M. K., & Anderson, J. R. (1986). *A key-stroke analysis of learning and transfer in text-editing.* Manuscript submitted for publication.

Soloway, E. L., & Woolf, B. (1980). *From problems to programs via plans: The content and structure of knowledge for introductory LISP programming* (Tech. Rep. No. 80-19). University of Massachusetts, Department of Computer and Information Science.

Thorndike, E. L. (1903). *Educational psychology.* New York: Lemke & Buechner.

Thorndike, E. L. (1935). *The psychology of wants, interests, and attitudes.* New York: Appleton-Century-Crofts.

Van Lehn, K. (1983). *Felicity conditions for human skill acquisition: Validating an AI-based theory* (Tech. Rep. No. CIS-21). Palo Alto, CA: Xerox Parc.

Vere, S. A. (1977). Induction of relational productions in the presence of background information. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 349-355). Boston, MA: Publisher.

Wexler, K., & Culicover, P. (1980). *Formal principles of language acquisition.* Cambridge, MA: MIT Press.

Winston, P. H. (1979). Learning and reasoning by analogy. *Communications of the ACM, 23,* 689-703.

Winston, P. H., & Horn, B. K. P. (1981). *LISP.* Reading, MA: Addison-Wesley.

Appendix

Knowledge Compilation: Technical Discussion

The description below describes knowledge compilation as it is implemented in the GRAPES production system (Sauers & Farrell, 1982), which is intended to embody certain aspects of the ACT* theory. Knowledge compilation consists of two subcomponents, proceduralization and composition. Proceduralization eliminates reference to certain declarative facts by building into productions the effect of that reference. Composition collapses several productions into one. Each is discussed separately.

Proceduralization

Proceduralization requires a separation between goal information and context information in the condition of a production. Consider the following production:

G1 IF the goal is to create a structure
and there is a operation that creates such a structure
and it requires a set of substeps
THEN set as subgoals to perform those steps

This is a classic working-backwards operator, an instance of a general, weak, problem-solving method. This production might apply if our goal was to insert an element into a list and we knew that there was a LISP function, CONS, that achieved this goal, for instance, (CONS 'A '(B C)) = (A B C). In this production, the first line of the condition describes the current goal and the subsequent context lines identify relevant information in declarative memory. Proceduralization eliminates the context lines but gets their effect by building a more specific goal description:

D1 IF the goal is insert an element into a list
THEN write CONS and set as subgoals to
1. Code the element
2. Code the list

The transition from the first production to the second is an example of the transition from a domain-general to a domain-specific production.

To understand in detail how domain-general productions apply and how proceduralization occurs, one needs to be more precise about the encoding of the production, the goal, and the knowledge about CONS. With respect to the production, we have to identify its variable components. Below is a production more like its GRAPES implementation, where terms prefixed by an equals symbol denote variables:

G1' IF the goal is to achieve =relation on =arg1 and =arg2
and =operation achieves =relation on =arg1* and =arg2*
and requires =list as substeps
THEN make =list subgoals

The goal is "to achieve insertion of arg1 into arg2," and our knowledge about CONS is represented as follows:

CONS achieves insertion of argument1 into argument2, and requires the substeps of writing "(," writing CONS, coding argument1, coding argument2, and writing")."

The production G1' applies to the situation with the following binding of variables:

=relation : insertion
=operation : CONS
=arg1 : arg1
=arg2 : arg2
=arg1* : argument1
=arg2* : argument2
=list : writing "(", writing CONS, coding argument1, coding argument2, and writing ")"

Thus, the production would execute and set the subgoals of doing the steps in =list.

The actual execution of the production required that the definition of CONS be held active in working memory and be matched by the production. This can be eliminated by proceduralization, which builds a new production that contains the relevant aspects of the definition within it. This is achieved by replacing the variables in the old production by what they matched to in the definition of CONS. The proceduralized production that would be built in this case is as follows:

D1' IF the goal is to achieve insertion
of =arg1 into =arg2
THEN make subgoals to
1. write (
2. write CONS
3. code =arg1
4. code =arg2
5. write)

or, as we generally write such productions for simplicity:

D1" IF the goal is to insert =arg1 into =arg2
THEN write CONS and set as subgoals to
1. code =arg1
2. code =arg2

In general, proceduralization operates by eliminating reference to the declarative knowledge that permitted the problem solution by the weak-method productions and building that knowledge into the description of the problem solution.

It would be useful to have an analysis of proceduralization of a solution by analogy, since this figured so prominently in our discussion. Usually, analogy is implemented in ACT* by the operation of a sequence of productions (e.g., see the chapter 5 appendix in Anderson, 1983), but the example here will compact the whole analogy process into a single production.

G2 IF the goal is to achieve
=relation1 on =arg3
and (=function =arg4) achieves
=relation1 on =arg4.
THEN write (=function =arg3)

Suppose the subject wanted to get the first element of the list (a b c) and saw the example from LISP that (CAR '(b r)) = b. Then this production would code (CAR '(a b c)) by analogy. Proceduralization can operate by eliminating reference to the declarative source of knowledge; in this case, the declarative source of knowledge is the example that (CAR '(b r)) = b. It would produce the new production:

D2 IF the goal is to achieve the
first-element of =arg3
THEN write CAR and set as
a subgoal to
1. code =arg3

Composition

Composition is the process of collapsing multiple productions into single productions (C. W. Lewis, 1978). Whenever a sequence of pro-

ductions apply in ACT* and achieve a goal, a single production can be formed that will achieve the effect of the set. Note that the existence of goals is absolutely critical in defining what sequences of productions to compose.

Whereas many times composition applies to sequences of more than two productions, its effect on longer sequences is just the composition of its effect on shorter sequences. Thus, if S1,S2 is a sequence of productions to be composed and C is the composition operator, C(S1,S2) = C(C(S1),C(S2)), so all we have to do is specify the pairwise compositions.

Let "IF C1 THEN A1," and "IF C2 THEN A2" be a pair of productions to be composed, where C1 and C2 are conditions and A1 and A2 are actions. Then their composition is "IF C1 & (C2 − A1) THEN (A1 − G(C2)) & A2." C2–A1 denotes the conditions of the second production not satisfied by structures created in the action of the first. All the conditional tests in C1 and (C1 − A2) must be present from the beginning if the pair of productions are to fire. A1 − G(C2) denotes the actions of the first production minus the goals created by the first production that were satisfied by the second.

As an example, consider a situation in which we want to insert the first element of one list into a second list. The first production, given earlier, would fire and write CONS, setting subgoals to code the two arguments to CONS. Then the second production would fire to code CAR and set a subgoal to code the argument to CAR. Composing these two together would produce the following production:

C3	IF	the goal is to insert the first element of =arg2 into =arg3
	THEN	code CONS and then code CAR and set as subgoals to 1. code =arg2 2. code =arg3

It is just productions of this form that we speculate subjects were forming in the McKendree and Anderson (in press) experiment described in the main part of this article.

Composition and Proceduralization

Much of the power of knowledge compilation in the actual simulations comes from the combined action of proceduralization and composition together. The composed productions C1 and C2 discussed with respect to Figure 1 earlier have such a history. A number of analogical productions had to be proceduralized and composed to form each production. Thus, in real learning situations we simultaneously drop out the reference to declarative knowledge and collapse many (often more than two) productions into a single production. This can produce the enormous qualitative changes we see in problem solving with just a single trial.

Received August 12, 1985
Revision received August 28, 1985 ■