2 Knowledge Compilation: Mechanisms for the Automatization of Cognitive Skills

David M. Neves and John R. Anderson
Carnegie-Mellon University

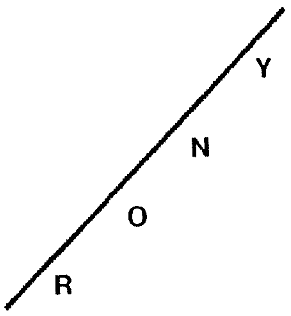
INTRODUCTION

People get better on a task with practice. In this chapter we take this noncontroversial statement, elaborate what it means to "get better," and propose two mechanisms that account for some of the ways people get better. We trace the development of a skill from the point when it is initially being memorized and applied in a slow and halting fashion to the point where it has become fast and automatic through practice.

We are interested in how students learn to use postulates and theorems in geometry tasks like that illustrated in Fig. 2.1. A scenario of how a student (based on two students we have looked at in detail in geometry and three subjects working on an artificial proof system) learns postulates is as follows: The student reads each of several postulates in a section of a textbook. After a brief inspection of the postulates the student goes on to the problems at the end of the section that require the student to use the postulates. In the student's initial attempts with the postulates there is much looking back to them in the textbook because they have not yet been committed to memory. These applications are slow and there are mutterings that show low-level matching of the postulates like "If *A* is *RO* and *B* is *NY,* then I can assert that. . . ." After some practice the student has committed the postulates to memory. After much practice their selection and application is very fast.

In this chapter we consider how postulates are initially encoded, how procedures are created out of these encoded postulates, and lastly how procedures speed up with practice. The processes we describe have been implemented and tested as mechanisms on the computer.

Given: $\overline{RONY}$; $\overline{RO} \cong \overline{NY}$
Prove: RN = OY



Proof:

Statements	Reasons
1. $\overline{RO} \cong \overline{NY}$	1. ?
2. RO = NY	2. ?
3. ON = ON	3. ?
4. RO + ON = ON + NY	4. ?
5. $\overline{RONY}$	5. ?
6. RO + ON = RN	6. ?
7. ON + NY = OY	7. ?
8. RN = OY	8. ?

FIG. 2.1. A proof to be completed by writing reasons.

In our work we have focused on a particular task within geometry. That task is providing reasons to an already worked-out proof. We have been working mainly with the geometry textbook of Jurgensen, Donnelly, Maier, and Rising (1975). In this textbook, before being asked to generate proofs the student is shown an example or two of a proof. Then the student is shown a proof lacking reasons, or justifications, for each of the lines (see Fig. 2.1). The student's task with these nearly whole proofs is to provide the justifications for each of the lines. A justification can be that the line was a given or that it was the result of the application of a definition, theorem, or postulate. Besides showing how a proof works, these problems give practice with the postulates.

Figure 2.2 shows a simple flow chart of our production system program that provides justifications for lines in a proof. When it comes to a line, it first checks to see whether the line is a given. If so, it puts down *given* and goes to the next line. If it is not a given, the program matches the consequent of a postulate to the current line. If there is a match, it then tries to match the antecedent of the postulate to previous lines. If failure occurs, it tries another postulate; and so on. When it verifies that a postulate can apply, it writes the name of the postulate and goes to justify the next line in the proof.

At this high level of analysis, the flow chart in Fig. 2.2 provided a good model for the behavior of all subjects at all levels of skill. However, there is a lot of information-processing detail being hidden in Fig. 2.2 in the two boxes (e and f) where the consequent and antecedents are being matched. Much of the learning we observed, particularly concerning the postulates, took place with respect to the matching processes. Our discussion focuses on how these matching skills evolve. Our analysis of this matching skill identifies three stages similar to the general analysis of skill acquisition by Fitts (1964). The first is encoding, where a set of facts required by the skill are committed to memory. The second is proceduralization, where facts are turned into procedures. The third is composition by which the procedures are made faster with practice. We talk about each stage in turn in subsequent sections. First we describe the encoding stage.

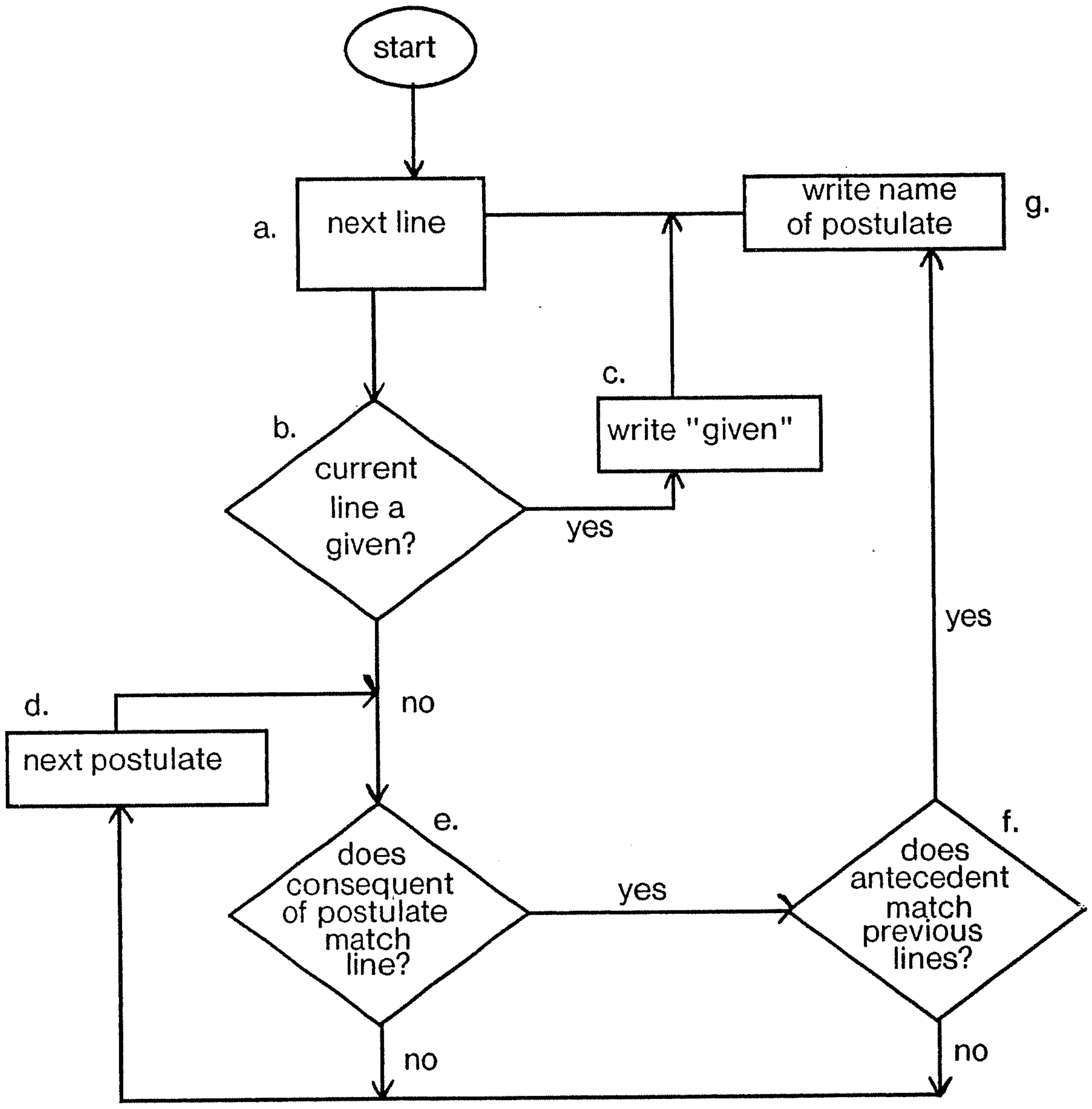


FIG. 2.2. Flow diagram of the reason giving task.

ENCODING

We propose that all incoming knowledge is encoded declaratively; specifically, the information is encoded as a set of facts in a semantic network. To explain what this assumption implies we should describe the consequences of storing knowledge declaratively versus procedurally. (See also Winograd, 1975, for a discussion of procedural versus declarative encodings for knowledge).

Declarative Encodings

In a declarative encoding, the knowledge required to perform a skill is represented as a set of facts. In our scheme these facts are represented in a semantic network. These facts are used by general interpretive procedures to guide behavior. To take a nongeometry example, suppose we learn in a French class the fact that *chien* means *dog*. Then this factual knowledge can be used to generate a French phrase by a general interpretive procedure of the form:

```
P1:   If   the goal is to describe an object
           and a word means the object
           and the word is masculine
    Then   say "le" followed by the word.
```

This rule can apply to any piece of vocabulary knowledge we have—for instance, that *garcon* means *boy*—to generate the correct phrase (e.g., le garcon). (By the way, the previous general procedure P1 is represented in production form; we shall be using production representations extensively.)

Perhaps the greatest benefit in representing the knowledge underlying procedures declaratively comes from the flexibility in using that knowledge. For example, suppose the system has learned the transitive rule of equality, "If $A = B$ and $B = C$, then $A = C$." This rule could then be used by one procedure to make a forward deduction. That is, if statements $RO = NY$ and $NY = WZ$ were stated, then $RO = WZ$ could be asserted. The same knowledge could also be used by a different general interpretive procedure to reason back from the consequent, "$A = C$", to test whether "$A = B$ and $B = C$" are true.

Two other benefits of declarative representations are ease of analysis and change. With the knowledge underlying procedures represented as data one can reason about the procedures and use that reasoning to plan action or to change the procedures to make them more efficient. Changing the procedure is simply a matter of adding or deleting links in the semantic network.

The major drawback of a declarative representation is that its interpretative application is slow. Each fact must be separately retrieved from memory and interpreted. The interpretive procedures, to achieve their generality, are unable to take any shortcuts available in applying the knowledge in a particular situation. Many unnecessary or redundant tests and actions may be performed.

Procedural Encodings

A second way of representing knowledge about procedures is as something that can be directly executed and so needs no costly interpretation phase. One way of representing procedures is production systems (Anderson, 1976; Newell & Simon, 1972). The ACT production system (Anderson, 1976) is a collection of If–Then rules that operate on the active part of a semantic network. When an *If* part matches some part of active memory, then the *Then* part of the rule is executed. The *Then* part either activates existing memory structure or builds new memory structure.

One of the advantages of a procedual representation such as a production system is that it handles variables in a natural and easy manner. For instance, in reason giving, variable handling can become quite difficult when matching a postulate to a line to see if it applies. When the consequent is matched to a line, the variables in that consequent are given values. Those values or bindings of the variables are used when the antecedent is matched to previous lines. This variable handling is accomplished much more felicitously if the postulate is encoded as a production than if it is encoded as a set of semantic network structures. An example of a postulate is the addition property of equality that is stated in the text as:

If $a = b$ and $c = d$, then $a + c = b + d$.

To build a production to encode this postulate for reason giving, one builds into the *If* side of the production the consequent of the postulate as a test of the current goal and the antecedent part of the postulate as a test of previous lines. So, the postulate would become:

```
P2:  If    the goal is to give reasons for $a + c = b + d$
           and a previous line contains $a = b$
           and a previous line contains $c = d$
     Then  write ``Addition property of equality''.
```

This production combines boxes e, f, and g from Fig. 2.2. In the first line (the goal is to prove $a + c = b + d$) the production binds the variables a,b,c, and d to elements in the current to-be-justified line. In the other two lines of the *If* side it checks that these bindings are consistent with elements in previous lines.

Procedural knowledge represented in this way is fast because the *If* side of the production is matched as one step. Although time to execute a procedure is largely independent of the number of productions in memory, execution time does depend on the number of productions that are applied in performing the procedure. So one can achieve even more speedup of a procedure by making its component productions bigger so that there are fewer selections of productions. We talk about this possibility later when we talk about the composition mechanism.

On the other hand, there are several disadvantages to such a procedural representation. In this form the knowledge cannot be inspected. However, some understanding of a production's content is available by noting what it does. Changes cannot be made to the productions, although there are schemes for creating new productions that will effectively delete or restrict the range of applicability of "bad" productions (Anderson, Kline, & Beasley, 1980). Also, when a rule gets put into production form, it can only be used in that form. For instance, the production P2 does not capture the use of the postulate for forward inferencing. For that, we would need another quite different production:

```
P3:  If    it is known that $a = b$
           and it is known that $c = d$
     Then  assert that it is known that $a + c = b + d$.
```

Getting the Best of Both Encodings

We clearly would like to have the advantages of both representations. One way of achieving this, of course, is to keep knowledge in both representations. When speed is needed, the procedural encoding is used. When analysis or change is needed, the declarative encoding is used.

There is little problem in creating a declarative description of a rule because creating new structure in semantic nets from external input is relatively simple. On the other hand it is much more difficult to encode that knowledge procedurally. We have seen that a procedural encoding is a highly specific interpretation of a rule. If there are many procedural encodings possible, which one should we pick? In present production systems the answer to this question changes with every task and is built in by the programmer.

From a psychological point of view a system that can directly encode rules as productions has a number of undesirable features. First, learning is faster than what we observe in people. People become proficient at a skill gradually rather than in an all-or-none manner. Second, by encoding the rule as a production you obtain an exact and specific match every time. There are no errors, and no other ways to make the match. One might want more flexibility in the match. For example, suppose P2 were to apply to the following situation (which occurs in Fig. 2.1):

Current statement:	$RO + ON = ON + NY$
Previous statements:	$RO = NY$
	$ON = ON$

The first line of production P2 would apply with the following binding of variables: $a = RO$, $c = ON$, $b = ON$, and $d = NY$. This means that it must match for its second and third lines:

$a = b$	(i.e., $RO = ON$)
$c = d$	(i.e., $ON = NY$)

Of course these attempts will fail. The problem is that the production does not appreciate the commutative nature of the plus operator and this increases the number of possible rules that should be built for the postulate. Rather than try to anticipate all the possible procedural interpretations of a postulate, we suggest that the postulate be represented declaratively and it be interpreted by a general evaluator that could deal with commutativity.

Therefore, when such rules initially enter the ACT system, they are stored as facts in a semantic net. Through mechanisms to be discussed later the rule will automatically turn into one or more productions through its use. So with the initial encoding we have the benefits of a declarative representation. That knowledge can be turned into faster production form as a consequence of being used interpretively. To illustrate how declarative knowledge can be used interpretively, we next describe the general matching productions that interpret postulates stored in semantic net form.

Interpretive Matching Productions

To match a declarative representation of a rule to a statement, we use a general interpretive matcher written as productions. These matcher productions do more deliberately and in a piecemeal fashion what is done automatically by a production system when postulates are represented as productions. They try to put two structures into correspondence and bind variables along the way.

Table 2.1 lists the seven productions that do the match. They are stated in a more understandable English-like syntax than their actual implementation. The matcher goes through both the rule and the line comparing corresponding nodes. If a constant, like =, is pointed to in the rule, then a test for equality is made to see that the rule and the line contain the same constants. If a variable in the rule, as noted by the "isa variable" tag on the element, is pointed to, then a binding of that variable is checked or made.

Variable bindings are kept as temporary network structure. For a variable to be bound means that a proposition describing what it is bound to is resident in memory. If a variable has no binding, as seen by the failure to retrieve a binding from memory, then the binding is made with the node being pointed to in the statement as the value. If there is already a binding, then the equality of that binding to the statement node is checked. Because the bindings are held in memory, they are reportable unlike variable bindings in a production.

We now show how the matcher productions would match the consequent of the reflexive postulate, $A = A$, to $ON = ON$. First the A on the left is compared to ON on the left.

$$\begin{array}{ccc} \underset{\uparrow}{A} = A & \qquad & \underset{\uparrow}{ON} = ON \end{array}$$

Production P6 fires in the matcher. It notes that A is an unbound variable so it

TABLE 2.1
Some General Interpretive Productions for Matching Postulates to Lines of a Proof

P4:	*If*	the current element of the rule pattern is a constant and the same element occurs in the line
	Then	move to the next elements in the rule and the line.
P5:	*If*	the current element of the rule pattern is a constant and the same element does not occur in the line
	Then	the attempt to match has failed.
P6:	*If*	the current element of the rule is a variable and this variable does not have a binding stored with it
	Then	store the current element of the line as the binding of the variable and move to the next elements in the rule and the line.
P7:	*If*	the current element of the rule is a variable and the current element of the line is stored as the value of the variable
	Then	move to the next elements in the rule and the line.
P8:	*If*	the current element of the rule is a variable and the value stored with the variable is different than the current element of the line
	Then	the attempt to match has failed.
P9:	*If*	there are no more elements in either the rule or the line
	Then	the attempt to match has succeeded.
P10:	*If*	there are no more elements in the rule but there are in the line (or vice versa)
	Then	the attempt to match has failed.

deposits "*A* is bound to *ON*" into memory. It also switches attention to the next elements in the expressions. Next the two equals signs are compared.

$$\begin{array}{ccccccc} A & = & A & \quad & ON & = & ON \\ & \uparrow & & & & \uparrow & \end{array}$$

Production P4 fires because it finds equal constants and increments the two pointers to *A* on the right and *ON* on the right.

$$\begin{array}{ccccccc} A & = & A & \quad & ON & = & ON \\ & & \uparrow & & & & \uparrow \end{array}$$

Production P7 fires here because it finds an already bound variable, *A*. It compares the binding for *A*, which is *ON*, with the element currently pointed to in the line, which is *ON*, and because they are equal, it succeeds. Now there are no more elements in either the rule or the line and production P9 fires and states that the match has succeeded. At this point the matcher has just completed a successful match and as a by-product has created a list of variables and their bindings.

The subject at this point is applying his or her knowledge in a slow interpretive fashion processing symbol by symbol. The procedural representations of the knowledge, for example, production P2, are much more efficient and apply the knowledge directly. Based on an analogy to a process in computer science, we use the term *knowledge compilation* to refer to the process by which subjects go from interpretive application of knowledge to direct application. This process translates declarative facts into productions. There are two subprocesses: proceduralization, which translates parts of declarative rules into small productions, and composition, which combines productions into larger productions.

PROCEDURALIZATION

With both a semantic net and productions to represent knowledge there is the problem of how to transfer knowledge smoothly from one store to the other. Production system models (Anderson, Kline, & Beasley, 1980; Waterman, 1975) have accomplished this transfer in the past by building productions with a special ''build'' operator in the *Then* side of a production. This special operator in ACT takes a network description of the knowledge underlying a production as an argument and adds the production to production memory. This implies that people can transfer their factual knowledge into procedures by abstractly ruminating upon the knowledge. However, this conflicts with our observation of students and with general wisdom about skill acquisition that skills only develop by exercising them.

Therefore, we have developed a different means of converting declarative knowledge into procedural knowledge—a mechanism that constitutes a major augmentation to existing production system architecture. Every time a production matches some long-term memory network structure that has to be retrieved into working memory, the proceduralization mechanism creates a new production that has that network structure incorporated into it and that avoids the need for the long-term memory retrieval. This simple mechanism merges semantic net knowledge into the production that uses it. In order for this mechanism to be selective in what memory it merges, we make a distinction between two kinds of declarative memory: a permanent memory and a transient, temporary memory. In ACT this permanent memory is the activated part of its semantic network (long-term memory) and the temporary memory is network structure that has just been created. Postulates, once committed to memory, are part of permanent memory. The representation of the current problem is part of temporary memory.

When the *If* side of a production matches memory, it matches some activated permanent memory and some temporary memory that has just been created by information entering from the environment. At this point a new production is created that has incorporated the permanent memory matched. As an example, we show how this process of proceduralization applies to production P6 of the matcher productions in Table 2.1. This production matches an unbound variable of a postulate to a corresponding part of a statement. To explain the operation of proceduralization we have to display more of the actual structure of P6 than was given in our informal statement of P6 in Table 2.1.

```
P6′:  If    the goal is to match
            and the rule pointer is pointing to an element in the rule
            and the element is a variable
            and the element is not bound
            and the line pointer is pointing to an element in the line
            and the rule pointer is before an element
            and the line pointer is before an element
      Then  bind the variable in the rule to the current element in the line
            and move the rule pointer to its next element
            and move the line pointer to its next element.
```

Now suppose the symmetric postulate "If $A = B$ then $B = A$" is a semantic network. Also, the statement "$RO = NY$" is in temporary memory because it is what ACT is currently attending to. Suppose P6′ applied to this situation when an attempt was being made to create a match between B, of the consequent ($B = A$) of the rule, and RO, of the statement. The third and fifth lines of P6′ would match to long-term memory elements as follows:

the element is a variable	<- -> B is a variable
the rule pointer is before an element	<- -> B is before =

What is accomplished by these long-term memory retrievals is that the rule elements are identified as B (currently pointed to) and = (next in the rule). These are permanent facts that are known about the postulate. We can create a production that is equivalent to P6′ in this situation by deleting the reference to these two clauses and, wherever else the rule elements are mentioned, replace them by B and =. This specialized version of the general production is given here:

```
P11:  If    the goal is to match
            and the rule pointer is pointing to B in the rule
            and B is not bound
            and the line pointer is pointing to an element in the line
            and the line pointer is before an element
      Then  bind B to the current element in the line
            and move the rule pointer to =
            and move the line pointer to its next element.
```

In creating this production ACT has effectively proceduralized a bit of the knowledge that is contained in the symmetric postulate—namely, that the consequent consists of a variable B before =. It is now no longer necessary to retrieve this knowledge from memory. Rather, the knowledge is now implicit in the production. Note that, in this step of proceduralization, we have reduced the demand on working memory in terms of the amount of information that needs to be kept active from long-term memory.

Of course, production P11 by itself is far from being a proceduralized version of the postulate. To accomplish this, every step in the application of the postulate needs to be proceduralized. When this is accomplished, we shall have a procedural version of the postulate that embodies one possible use of the knowledge. Note that the knowledge at this state is still being applied in a piecemeal fashion. To get a unitary procedural representation of the postulate, we need the process of composition that is occurring concurrently with proceduralization.

COMPOSITION

In our development of composition we have been strongly influenced by the work of Lewis (1978). Composition occurs concurrently with proceduralization but we think it continues after proceduralization is complete. It is for this reason that we designate it the third stage in skill development. The basic idea of composition is that pairs of productions that are executed in sequence are combined into single productions. It is assumed that the time to execute a production system task is roughly proportional to the number of productions that were fired. Therefore one effect of combining productions into larger productions is to decrease the amount of time for a procedure to apply.

The easiest way of combining two productions into a third is to add together their *If* sides and their *Then* sides. So suppose the following two productions fired consecutively.

P12: *If* you see a red light
Then assert danger.

P13: *If* there is danger and another person is near you
Then warn that person.

The simple composite is:

P14: *If* you see a red light
and there is danger
and another person is near you
Then assert danger
and warn that person.

There is a problem with this composite. There is a test for "danger" in the *If* side that will not be satisfied. That is because the danger that is being tested for in P14 was added by P12 in the original sentence. Because the *If* side of the composite, P14, is made of P12 and P13, the parts of P13 that are dependent on action taken by P12 must be changed in P14. The change needed in the algorithm that gave rise to P14 is that clauses in the *If* side of the composite are deleted if they were asserted in the *Then* side of the first production and tested in the *If* side of the second production. Lewis' algorithm for composition is as follows: Suppose two productions fire, one right after the other. The first *If A Then B* has *If* side clauses represented by A and *Then* side clauses represented by B. The second, *If C Then D*, has *If* side C and *Then* side D. Then, a new production can be constructed that will do the work of both.

If A Then B
If C Then D
- - - - - - - - - - - - - - - - - -
If A & $(C - B)$ *Then B* & *D*

This new production will have as an *If* side all the *If* clauses in A and all the *If* clauses in C, except the clauses that were asserted in B. As a *Then* side this new production has the union of the *Then* sides in B and D.

In this case the composite of productions P12 and P13 would be:

P15: *If* you see a red light
and another person is near you
Then assert danger
and warn that person.

The *If* side of the new production contains all the *If* parts in the first production and the test of the presence of another person in the second production. The test for "danger" in the second production is not in the new production because it is asserted in the *Then* part of the first production. The *Then* part of the composite production contains all the assertions from the first two.

We made several modifications to the Lewis algorithm. The major change came because of the use of variables in the production system. A *join* process is applied that checks to see if the same element occurs in the two productions even if it is referred to differently. A variable may be bound to an element in one production but in another production the element may be bound to a different variable or be directly referenced (i.e., is a constant). If the element is directly referenced in one production, it is directly referenced in the composite rather than being referred to by a variable.

Suppose the following two productions fired.

P16: *If* the length of segment A is equal to the length of segment B
Then the result is that segment A is congruent to segment B.

P17: *If* the result is something
Then write the result.

As the composite production is being created, the *Then* side of P16 is *join*ed with the *If* side of P17. The reference to "the result" in P17 is replaced by the more specific "segment A is congruent to segment B". The composite production is:

P18: *If* the length of segment A is equal to the length of segment B
Then the result is that segment A is congruent to segment B
and write segment A is congruent to segment B.

This join process also takes place when the same clause occurs in the *If* sides of both productions. One of the clauses must be deleted so that the actual composition rule looks more like

If A Then B
If C Then D
- -
If $(A - C)$ & $(C - B)$ *Then B* & *D*

where $(A - C)$ means all clauses in A excepting ones also in C. Other parts of the composition process are more specific to the production system language we are using and we do not discuss them here.

We have described two processes. The first process is proceduralization: the instantiation of variables bound to long-term memory structure. This process deletes *If* clauses in the specific production that match long-term memory propositions and instantiates variables that are bound to long-term memory. Proceduralization makes small and specific productions. The second process is composition. It combines two productions that fire after each other into a single production.

These two processes automatically bring about the transformation of declarative knowledge into procedural knowledge discussed earlier. When a rule is in semantic net form, they will, after much practice with it, create a production that embodies one procedural interpretation of that rule. To illustrate this process, let us look at how the consequent of the symmetric rule, "If $(A = B)$ then $(B = A)$", was turned gradually into a single production.

We used general matcher productions (that were somewhat more elaborate than the general productions showed in Table 2.1) to match the consequent, $B = A$, to several lines, such as $RO = NY$. It took 16 firings of these general matcher productions to accomplish the match the first time. During this pass the proceduralization process created 16 productions that were specific versions (specific to the consequent of the symmetric rule) of the matcher productions. There are 15 pairs of successive productions in the sequence of 16 general productions that fired. Composition of the general matcher productions was attempted and created 15 general composites with each composite able to do the work of two of the original matcher productions. However these general composites never applied later because the system favored the more specific productions created by the proceduralization process.

The consequent was then matched a second time and the 16 specific production created by the proceduralization process during the first pass fired. Because these productions were already specific, the proceduralization process could not produce any new productions. The composition process did produce 15 specific composities (i.e., 1 & 2, 2 & 3, . . . 15 & 16) from those 16 specific productions.

In the third pass the composites created during the second pass fired. Matching the consequent only took 16/2, or 8, production firings. After five sessions where the consequent of the symmetric rule was matched, a production was created that was able to match the consequent of the symmetric rule to a line in a single step. This production is:

```
P19:  If    the goal is to match the line to the consequent
            of the symmetric rule
            and there are three elements in the line
            and the second element in the line is =
      Then  bind B to the first element in the line
            and bind A to the third element in the line.
```

Although this production exists, all previously created productions are also around because proceduralization and composition do not destroy the productions they build from. At this point of practice there exists the original semantic net representation of the consequent, the general matcher productions, and the 15 general composites of those matcher productions that never executed. Also there are 16 specific productions created by proceduralization plus (15 + 7 + 3 + 1) 26 composites. Thus, in total, 57 new productions were created.

In this part of the chapter we described the mechanisms that enable one to become faster at a skill. In the second half we talk about some psychological phenomena that are accounted for by these mechanisms. But first we travel off the path of the discussion a bit and consider whether proceduralization might not be seen as a special case of composition.

Creating Specific Productions without Proceduralization

We have shown how knowledge is smoothly transferred from a declarative semantic net to production form. The proceduralization process that does this has the effect of deleting from the *If* side of composite productions clauses that match long-term memory. Proceduralization also makes specific productions. What would happen if there was no semantic net and declarative knowledge was stored as productions instead? This is the case with the OPS production system language (Forgy, 1979; Forgy & McDermott, 1977) whose only permanent memory is production memory. We show here that the effects of proceduralization can be duplicated with the composition process.

First, we must state how a semantic net can be functionally represented as productions. In a semantic net there are nodes, links leading from those nodes, and values attached to those links. One way of mapping this to productions is to have a node as the *If* part of a production and all facts known about it as the *Then* part. To retrieve facts about the node the node is asserted into memory. The production for that node then fires and deposits all facts about that node into working memory.

We now show how a production specific to matching part of the symmetric postulate can be created by composing memory retrieval productions with general matching productions. P20 is a production that deposits all links out of B (of the consequent $B = A$) when it executes. It mimics the retrieval from a semantic net of all facts about B.

```
P20:  If   B
      Then B is a variable
           and B is before =.
```

P21 is a production in the matcher that is identical to production P6′ given earlier. It matches a variable in a postulate to a piece of a statement.

```
P21:  If   the goal is to match
           and the rule pointer is pointing to an element in the rule
           and the element is a variable
           and the element is not bound
           and the line pointer is pointing to an element in the line
           and the rule pointer is before an element
           and the line pointer is before an element
      Then bind the variable in the rule to the current element in the line
           and move the rule pointer to its next element
           and move the line pointer to its next element.
```

When the two productions P20 and P21 fire, then their composite is created as the production P22. This new production has fewer clauses in the *If* side because the composition algorithm causes "B is a variable" and "B is before =" to be deleted. These two clauses were asserted in P20 and matched to in P21. Also the reference to the current element in the rule has been replaced by its instantiation, B, because of the application of composition's join process.

```
P22:  If   B
           and the goal is to match
           and the rule pointer is pointing to B
           and B is not bound
           and the line pointer is pointing to an element in the line
           and the line pointer is before an element
      Then bind B to the current element in the line
           and move the rule pointer to =
           and move the line pointer to its next element.
```

With P22 we have a production that is identical, except for the additional B, with the production P11 that was created by the proceduralization process from the application of a production to a piece of semantic network. Thus by representing all knowledge as productions the single mechanism of composition can also accomplish proceduralization.

EFFECTS OF PRACTICE

So far we have presented a proposal to explain how an information-processing system can have both the flexibility of a declarative semantic network representation and the speed of a procedural representation for rules. Our explanation of how a system automatically develops fast productions by practice rested on the combination of proceduralization and composition. In the following subsection we show how these mechanisms account for practice effects.

Einstellung

The speedup from composition seems too good to be true. Seemingly the only price is the storage space required to store all the new rules, and in the human organism this is a plentiful resource. However, there is another price to be paid. That price is a growing inflexibility and a lack of ability to adapt to change. This phenomenon associated with composition seems to correspond closely in the problem-solving literature to a phenomenon called *Einstellung*.

The Einstellung effect was extensively studied by Luchins (1945). He observed the effect using several tasks, but the most well-known task is the water jugs task. In this task the subject sees three different sized jugs with stated capacities. The problem is to measure out an amount of water using some or all of the jugs but using no measuring instrument other than the jugs. Subjects are shown two methods, a long and general method that takes four steps and uses three jugs, and a shorter method that takes two steps and uses two jugs.

After being shown both methods, one group is given practice problems in which they can use only the long method. Then they are given a problem that can be solved by either the short or long method. Most subjects solve it using the long method. In contrast, another group of subjects is shown both methods but does not receive practice on the long method. When they are presented with the problem that can be solved with either method, most use the short method. Luchins showed this effect with other tasks, like maze tracing, proofs in geometry, and extracting words from letter strings.

Lewis (1978) demonstrated Einstellung using a symbol replacement task. He gave subjects practice with rules in a symbol replacement task. At some later point he presented a new rule that would offer a shortcut in some cases. He showed that the more practice subjects had with the original set of rules, the less chance they would use the shortcut rule.

Lewis then showed how composition would predict the same results. His explanation relies on an interpretation of how the *If* part and *Then* part of a production are processed. The *If* part is processed at the beginning of a cycle with the selection of a production to fire. Every production in the system has an opportunity to match at this time. However, once a production is selected, each clause in the *Then* side is unconditionally executed. No other production can apply at this time.

The result of composition is to turn an open, conditional procedure into a closed, unconditional one. For example, when a production is created from the composition of five other productions, it does the job of those five productions as one uninterruptible step. If there was another production that might have applied in the middle of that five-step sequence, it cannot apply now because it is bypassed.

So we see composition can cause problems. In its early life, a procedure consists mostly of propositions in a semantic net. This gives the organism great flexibility in interpreting and even changing rules. However, once composition starts, there is less of a chance that a small (new) addition will be able to compete with the larger composite rules. This is because in most versions of production systems larger rules are given a preference in application over smaller rules. There are ways to get around the Einstellung effect, but some of them involve going back to the original semantic net representation of the procedure and reinitializing the slow interpretation of that procedure.

Speedup

People speed up at a task as they practice it, often without trying to speed up. Composition predicts this kind of automatic speedup. In our implementation of composition, on every cycle of the production system the current production is composed with the previously executed production. If 120 productions fire in the performance of a task, then the next time the exact same task is performed 60 productions will fire, then 30 productions, 15, 8, and so on. This is an exponential speedup, $t = a * (\frac{1}{2})^p$, where a is initial time to perform the task and p is the trial number. By assuming that pairs of productions are composed only with a certain probability we can obtain functions that are slower than halving. Even with such assumptions, composition by itself, with no further considerations, would predict an exponential practice function (i.e., $t = ab^p$).

It has consistently been shown that human speedup is not exponential but, rather, follows a power-law function (Crossman, 1959; Lewis, 1979; Snoddy, 1926), $t = a * p^b$, where a is initial time, p is trial number, and b is rate of speedup. When time versus trials (of practice) is plotted on log–log paper, the result approximates a straight line [$t = a * p^b \Rightarrow \log(t) = a' + b * \log(p)$] where the intercept is the parameter a' and the slope is the parameter b. One explanation of this discrepancy between predicted and actual speedup that Lewis (1979) offers is based on the premise that power-law speedup is not produced by one mechanism but instead comes from the combination of speedup of many subprocesses. He notes that a power law can be approximated by the sum of many exponentially decreasing functions. Thus, there could be processes of a task that would be at various stages of exponential speedup (as composition would predict) but the measurable outcome of the task would be their sum. This summed result would follow the slower power-law function.

There are a number of other complicating factors that predict a slower-than-exponential speedup. Composed productions tend to have larger *If* sides and *Then* sides than the productions that gave rise to them. The analysis shown assumes that time to apply a production is independent of its size. However, this is not true in any known implementation and may not be true in the human head. For instance, it is reasonable to suppose that time to match the *If* side of a production, even assuming parallel machinery, would be a logarithmic function of complexity (Meyer & Shamos, 1977). Thus, part of the benefit of composition may be lost to increasing production size.

Another complexity has to do with strategy shifts that do occur in many complex tasks such as reason giving. Strategy shifts involve changing to an algorithm that requires fewer steps. It is unclear whether strategy shifts would produce immediate speedup. Although there are fewer logical steps in a better algorithm, the time to perform it may be longer because the processes underlying the steps have not been composed.

Still another complexity is pointed out by Newell and Rosenbloom (this volume) in their analyses of the power law. There is a variability in the problem situation and many production sequences will be needed to deal with different situations. This contrasts with the previous implicit assumption that there is just one sequence to be learned. Although short subsequences of productions may occur frequently, longer subsequences will occur less frequently. This means that few compositions of short sequences will be needed to cover all possible situations, but many compositions of long sequences may be needed. Thus it will take longer before enough compositions occur to cover all longer subsequences. Although we do not agree with the Newell and Rosenbloom assumptions that the number of sequences increases as a power or exponential function of length, any increase in the number of possible sequences with length will slow down the speedup.

So, to summarize: A pure application of composition leads to the prediction of exponential speedup. However, complications true of a realistic situation would tend to slow the speedup to less than exponential. This point has also been argued by Lewis (1979). True, it would be remarkable if these complications served to yield a pure power function. However, it is not possible to discriminate a pure power function from something that is close. Moreover, data do exist with systematic deviations from a power law. We discuss some of our own.

An Experiment Looking at Effects of Practice

We were interested in seeing how these various complications combined in the task we were most interested in (i.e., reason giving in a proof system). Three subjects were run for 10 one-hour sessions. These sessions were held on consecutive days except for weekends. All subjects were graduate students. We developed an artificial postulate set to minimize the effects of previous knowledge. A set of 8 postulates was used to construct 150 proofs. Each proof was 10 statements long and was made up of 4 givens and the application of 2 single antecedent rules and 4 double antecedent rules. No proof duplicated any other.

Proofs were displayed by computer on a terminal screen. Because we were interested in what subjects were looking at, only part of the proof was displayed at any time. The subject had to call explicitly for a part of the proof that was to be viewed next. In this way we could record both what the subject looked at and how much time was spent at that location. The screen had labeled columns for givens, statements, reasons, and antecedents and consequents of the postulates. At all times the statement to be justified was displayed on the screen. The subject pressed keys to view anything else.

Various commands were developed to facilitate movement through the proof. The basic command consisted of two keys, the first denoting the column to move to (i.e. *g* for given) and the second a digit for the row number. Shortcuts were also allowed. <Digit> would send the subject to row <digit> in the same column. <Period> would go to the previous line in the column, and <space> would go to the next line. Hitting the column key twice displayed an element in the same row but in the selected column. This was useful for the back and forth scanning of antecedents and consequents. Data collection consisted of the computer recording the location moved to and the time spent there. A location consisted of a given, statement, antecedent, or consequent.

Figure 2.3 shows the average data for the three subjects, plotted on a log–log graph. Subjects generally took about 25 minutes to provide justifications to the first proof. After 10 hours of practice with the task they were able to do it in about 1 to 2 minutes. We have plotted total time per problem, number of steps per problem (i.e., commands executed), and time per step. We thought time per step would reflect speedup on the problem due to such automatic factors as composition. We thought number of steps per problem would reflect other factors in addition to composition, such as strategy modifications. We were interested in how these two factors combined to yield an overall power law.

It is interesting to note that two power laws appear to underlie the overall power law for total time. This is seen by the relatively good approximations to straight line functions on the log–log plot. If the number of steps is decreasing at the rate $N = A_n P^{b_n}$ where N is the number of steps, A_n is the intercept in the log–log plot, P is the number of problems (practice), and b_n is the slope of the log–log function; and if the time per step is decreasing at the rate $T = A_t P^{b_t}$ where T is the time per step, A_t is the intercept, P is the number of problems (practice), and b_t is the slope, then the total time (TT) will obey the following power law:

$$TT = (A_n A_t) P^{b_n + b_t}$$

Subjects' search steps could be classified according to whether they were scanning givens, statements, consequents of postulates, or antecedents of postulates. Figure 2.4 shows log time plotted against log practice for each component

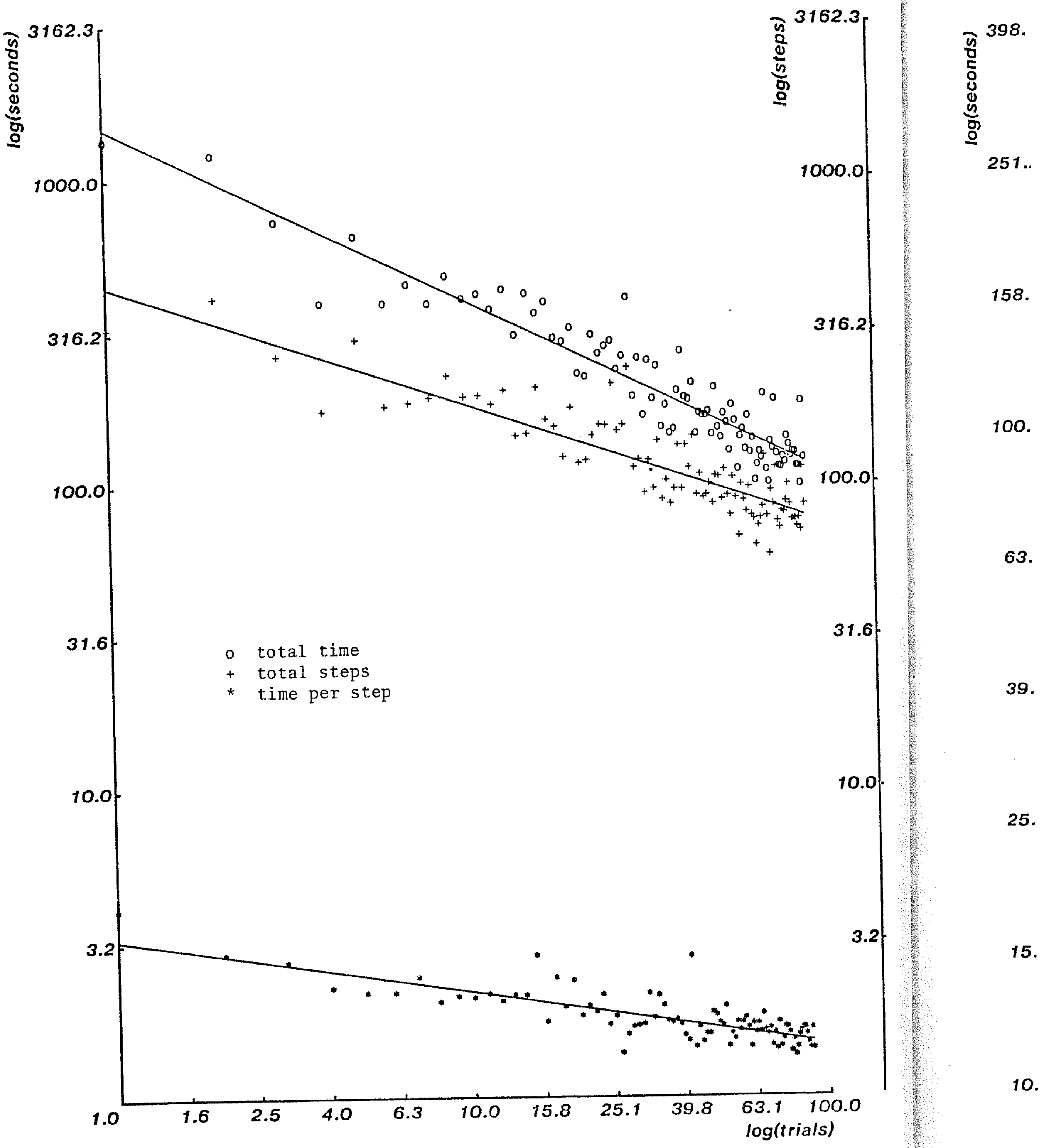


FIG. 2.3. Log time versus log practice averages for three subjects.

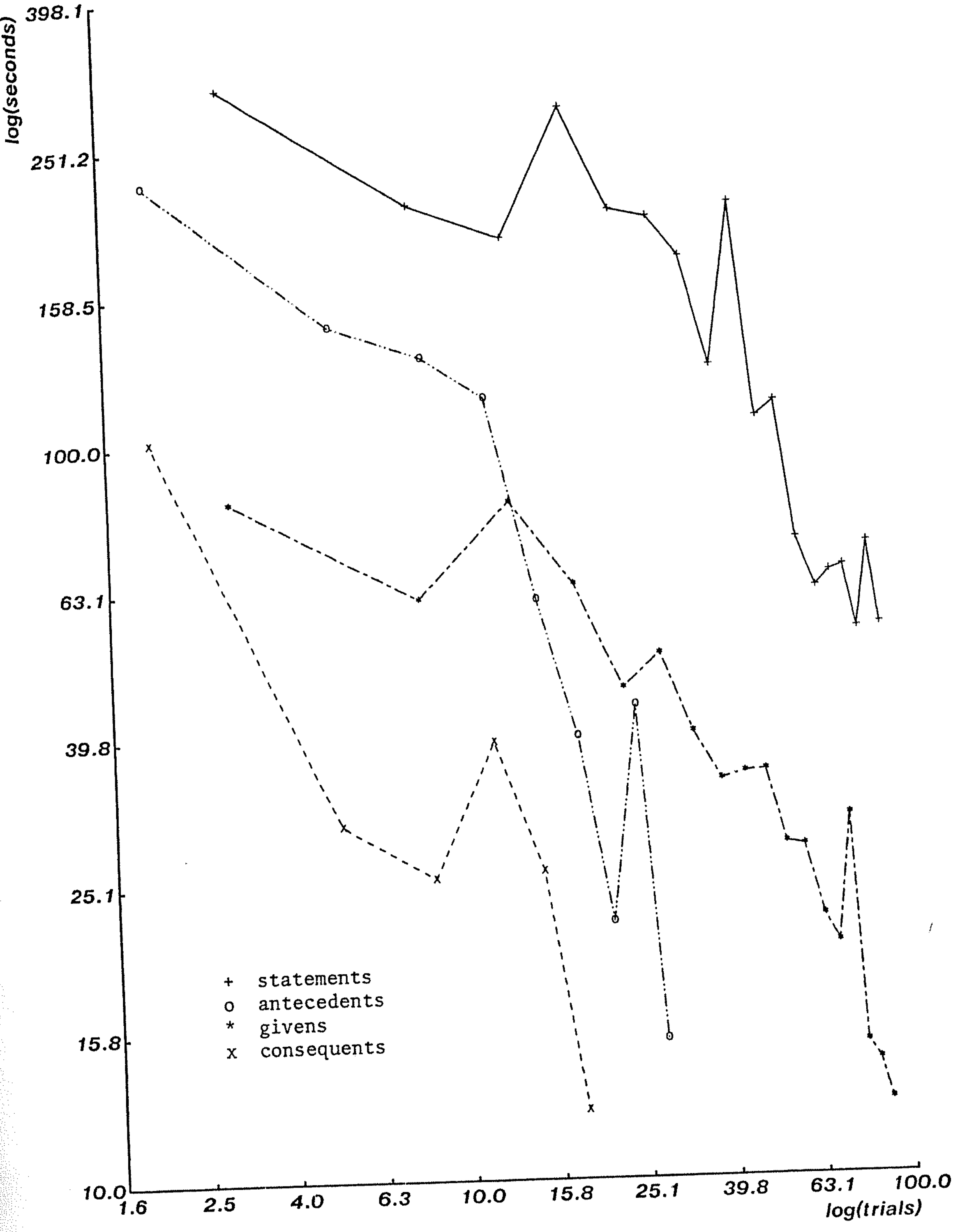


FIG. 2.4. Log time versus log practice of the components for subject R.

for subject R. The differing slopes in Fig. 2.4 reflect some major trends in subject improvement. Like the algorithms we developed earlier, subjects started out indexing the postulates by their consequents. They first learned the nine consequents and their order in the postulate list. Subjects fairly quickly stopped scanning the postulates by consequent and rather went to the correct one. This is reflected in Fig. 2.4 by the rapid drop-off on the curve for consequent scanning. Subjects' learning of the antecedents was much slower, partly no doubt, because they had less practice with these. Subject R came to the point where she completely learned the antecedents. The other two subjects never quite achieved that state. To the degree subjects were able to commit antecedents and consequents to memory, any inspection of the postulate list just dropped out of their protocols.

Although both the number of givens and the number of statements in Fig. 2.4 are fit fairly well by linear functions on the log–log scale, there is not much drop-off in these functions until after trial 20. Despite the fact that all the functions in Fig. 2.4 show a strong linear component, there is also a strong quadratic component in the deviations from linearity. Basically, all the functions appear to be speeding up faster than a power law. In fact for subject R, unlike the other two subjects, three out of the four components are better fit by an exponential function than by a power law.

To summarize our presentation of this data, it seems that a rather broad spectrum of changes underlie the systematic improvement of subjects at the reason giving task. There are strategy changes, such as searching statements backward from the to-be-justified line instead of forward from the beginning of the statements. One major development is memorization of the postulates so that they can be directly applied without search. There are other kinds of optimizations detected by subjects such as not searching the givens. The other major speedup occurs with respect to the time subjects spend in the individual steps of a problem. However our main point here is that in our task there seem to be many processes speeding up faster than a power law. When these are combined. they better approximate a power law. Thus, we do not think that the fact that overall speedup better fits a power law refutes the psychological validity of the composition process as one of the underlying mechanisms producing the speedup.

Automaticity

In their 1977 *Psychological Review* articles Shiffrin and Schneider identify two modes of human processing: one controlled and the other automatic. They describe controlled search as serial and requiring both processing and memory resources. Automatic search is parallel and does not interfere with other processes requiring resources. Although they provide a description of automatic processing, they do not say how it can develop from a controlled process. They do describe the circumstances under which it will develop. Parallel search of a list develops when a search procedure is applied over and over to the same list (their consistent mapping condition) and does not occur when a search procedure is applied over and over to varying lists (their varied mapping condition). In this section we show how composition and proceduralization turn a controlled, serial process into an automatic parallel process. But first we are more specific about some of the psychological data we want to explain.

The Neisser, Novick, and Lazar (1963) visual search task is one that showed a transition from serial to parallel search. Subjects scanned a matrix of letters for characters in either a 1-, 5-, or 10-character list. These lists were embedded, so that the 10-item list contained characters in the 5- and 1-character list. After about 20 days of practice, on the same lists, subjects were able to search a matrix for any one of 10 characters as fast as they searched for a single character.

Gibson and Yonas (1966) ran a visual search experiment with several age groups. After practice their subjects could search a matrix for two characters as fast as for one. Briggs and Blaha (1969) used a memory scanning task in which several characters are kept in memory and the subject responds whether a stimulus character is in that set. They found that subjects were as fast to respond with two characters in memory as with one character. Mowbray and Rhodes (1959) found that after 15 days of practice a subject responded as fast in a four-alternative forced-choice task as he did in a two-choice task.

In general, though, the data do not overwhelmingly point to the development of a completely parallel search. There are other data that, although they show a substantial reduction in search rate with practice, still show some residual effect of number of memory elements on search rate. It may be difficult to find evidence for complete, unlimited-capacity parallel search because any other serial process, like double-checking the answer, will hide it. Even if a serial subprocess is only occasionally added to an unlimited-capacity parallel one, the average data will look serial (i.e., show an effect of list length).

Searching memory sets that change from trial to trial does not show the same pattern of results as searching fixed memory sets. Kristofferson (1972) showed no change in the search rate per item for one- and four-item lists with practice in a memory scanning task. Nickerson (1966) showed only a slight reduction in search rate.

In summary, then, a motivated individual searching a never-changing list will, after much practice, search it in parallel (i.e., where reaction time is largely independent of the number of elements). However, if the list changes from trial to trial, the individual will have to search it serially and the rate of search will either not decrease or decrease very little with practice. We now show how composition and proceduralization predict these results.

It was shown earlier that composition can turn a several-step process into a single step. This can also be looked at as turning a serial process into a parallel one. We show how this occurs with a memory scan task with a fixed list, *A Q R T*. Two general productions follow that will compare a probe with a list in memory. These productions scan a list from left to right until an element in the list matches the probe.

P23: *If* considering an element in the list
and the probe equals that element
Then say yes.

P24: *If* considering an element in the list
and it is not equal to the probe
and the element is before another element
Then consider the next element.

After some practice searching the list for *T* the following specific productions are created by the proceduralization process. Proceduralization causes the memory set elements, which are in long-term memory, to become incorporated into productions.

P25: *If* considering *A*
and Probe is not *A*
Then consider *Q*.

P26: *If* considering *Q*
and Probe is not *Q*
Then consider *R*.

P27: *If* considering *R*
and Probe is not *R*
Then consider *T*.

P28: *If* considering *T*
and Probe is *T*
Then say yes.

Then with more practice searching for *T* composition will combine these specific productions into a production that will recognize that the probe is equal to *T* in a single step. (Note that this production along with all the others produced by composition are shown with smaller *Then* sides than they actually have.) That production is:

P29: *If* the probe is *T*
and not *A* or *Q* or *R*
Then say yes.

After practice searching for the other members of the memory set, composition also creates the following productions that will recognize when a probe is equal to a member of the list. These three productions are:

P30: *If* the probe is *A*
Then say yes.

```
P31:   If   the probe is Q
            and not A
     Then   say yes.

P32:   If   the probe is R
            and not A or Q
     Then   say yes.
```

Once these productions are created, then any of the elements in the memory set can be recognized with the application of a single production. If we assume that time to execute these productions is the same, then we have parallel search. So, with a fixed list, composition predicts parallel search. According to Schneider and Shiffrin (1977) another effect of automaticity is to make processing time independent of the number of alternatives in the display. This would be a result of composing together the productions that searched through the display. Then there would be specific productions that recognized each element in each possible position of the array.

The data for searching varying lists show little or no reduction of search rate with practice. In these cases the composition algorithm will only be combining general productions. Proceduralization will not occur because the memory sets are only kept in temporary memory. This is unlike the case when fixed lists are used and specific productions are also created. Here we show a general composite production that would be created after matching probes to the fourth item in lists with different elements.

```
P33:   If   pointing to the first element in the list
            and it is not equal to the probe item
            and the first element is before a second
            which is not equal to the probe item
            and the second is before a third element
            which is not equal to the probe item
            and the third is before a fourth
            which is equal to the probe item
     Then   say yes.
```

Unlike the specific production that searches for *T* this general production has more clauses in the *If* side. These extra elements are needed to match the structure of the memory set. The cost of generality is an increase in the size of the *If* side of a production. If we assume that working memory is limited in size, then we will not be able to keep all the memory set in working memory and so the large general production will never apply. Thus, in a search task with lists that vary we predict little or no improvement in search rate for subjects relatively experienced in searching lists because their general search productions are already as large as possible.

It is important to recognize that under our analysis two things are happening in the automaticity paradigm. For one thing, composition is collapsing a series of productions into one, producing a loss of the set size effect. In addition, proceduralization is relieving working memory of the need to maintain a representation of the memory set. This second factor we think is responsible for the loss of interference with concurrent tasks that Shiffrin and Dumais (this volume) report occurs with practice; that is, performance of one task does not suffer interference from a concurrent automatic task. This result is predicted because working memory no longer needs to maintain in active state long-term memory facts for performance of the automatic process.

Losing Track of Intermediate Results

There is some evidence that suggests that people lose conscious track of intermediate results with practice on a problem (Ericsson & Simon, 1980). Composition can predict this effect. We have seen that when two productions are composed, the resulting production will likely have fewer *If* side clauses than the sum of the number of *If* side clauses in each of the two productions. However, the *Then* side of the resulting production is precisely the sum of the *Then* sides of both productions. So, these *Then* sides can grow quite large. Intermediate results are deposited into working memory by means of the actions in *Then* sides. If the number of clauses in the *Then* side exceeds the capacity of working memory, the information conveyed by these clauses (i.e., intermediate results) will be lost.

CONCLUSION

We have described an automatic learning system based on proceduralization and composition. These mechanisms allow us to maintain the flexibility of representing knowledge in a semantic net and also to build production rules that will embody directly certain uses of the knowledge. The knowledge underlying procedures starts out as propositions in a network. Knowledge in this form can be changed and analyzed by the cognitive system. As one applies knowledge, the proceduralization process turns it into faster production rules automatically. Then composition forms larger units out of the individual proceduralized productions, in a gradual manner. These processes help explain some effects in the practice literature such as automatic speedup, development of parallel search, and inability to introspect on the application of well-learned procedures.

Composition speeds up procedures, but it does not change them. This is unlike other mechanisms of learning such as analogy, strategy modification, generalization, and discrimination (see Anderson, Greeno, Kline, & Neves, this volume) that may actually change procedures. Still we believe that composition does produce an effect that will change behavior. Problem solving by students is always under some time constraint, whether it be 30 minutes given for an exam or a few minutes allocated for homework. If we assume that a relatively constant amount of time will be spent on problems, then in the initial stages of practice some good solutions will not be discovered because they involve too much search time. As the search process with a problem becomes faster, more and more of the search tree can be explored and so new solutions will be discovered.

We observed this kind of development of competence in one of the students to whom we taught geometry. His early problem-solving behavior showed much linear search through the textbook for concepts he had read previously but had forgotten. Usually this search took much time and after several minutes he would give up even though eventually he would have found what he was looking for. Then we showed him a more efficient way of searching the textbook by checking the glossary. At this point he was able to find information much faster and so solved more problems successfully.

Similarly, proceduralization does not change the procedures. It only makes them specific to the knowledge used. However, it too can change the behavior the system is capable of by reducing the demand on working memory to maintain long-term memory facts. Thus, more different kinds of information can be kept in working memory and so new relationships can be seen between active information. Also proceduralization releases working memory for concurrent tasks that might facilitate the problem solving.

So we see that behavior will be changed by these simple automatic learning mechanisms. The interesting thing about them is that expertise comes about through the use of knowledge and not by analysis of knowledge. There is no intelligent homunculus deciding whether incoming knowledge should be stored declaratively or procedurally or how it should be made more efficient.

ACKNOWLEDGMENTS

We would like to thank David Klahr and especially Jola Jakimik for comments on an earlier draft. We also thank Bill Chase for helping our search through the memory scanning literature. The research described was supported by ONR Contract N00014-78-C-0725.

REFERENCES

Anderson, J. R. *Language, memory, and thought.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1976.

Anderson, J. R., Kline, P. J., & Beasley, C. M. Complex learning processes. In R. E. Snow, P. A. Federico, & W. E. Montague (Eds.), *Aptitude, learning, and instruction: Cognitive process analyses.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1980.

Briggs, G. E., & Blaha, J. Memory retrieval and central comparison times in information processing. *Journal of Experimental Psychology,* 1969, *79,* 395–402.

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

Ericsson, K. A., & Simon, H. A. Verbal reports as data. *Psychological Review,* 1980, *87,* 215-251.

Fitts, P. M. Perceptual-motor skill learning. In A. W. Melton (Ed.), *Categories of human learning.* New York: Academic, 1964.

Forgy, C. *The OPS4 reference manual.* Department of Computer Science, Carnegie-Mellon University, 1979.

Forgy, C., & McDermott, J. OPS, a domain-independent production system. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* 1977, 933-939.

Gibson, E. J., & Yonas, A. A developmental study of visual search behavior. *Perception & Psychophysics,* 1966, *1,* 169-171.

Jurgensen, R., Donnelly, A., Maier, J., & Rising, G. *Geometry.* Boston: Houghton Mifflin, 1975.

Kristofferson, M. W. Effects of practice on character-classification performance. *Canadian Journal of Psychology,* 1972, *26,* 540-560.

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

Lewis, C. H. *Speed and practice.* Unpublished manuscript, 1979.

Luchins, A. S. Mechanization in problem solving. *Psychological Monographs,* 1945, *58,* No. 270.

Meyer, A. R., & Shamos, M. I. Time and space. In A. K. Jones (Ed.), *Perspectives on Computer Science.* New York: Academic, 1977.

Mowbray, G. H., & Rhodes, M. V. On the reduction of choice reaction times with practice. *Quarterly Journal of Experimental Psychology,* 1959, *11,* 16-23.

Neisser, U., Novick, R., & Lazar, R. Searching for ten targets simultaneously. *Perceptual and Motor Skills,* 1963, *17,* 955-961.

Newell, A., & Simon, H. A. *Human problem solving.* Englewood Cliffs, N.J.: Prentice-Hall, 1972.

Nickerson, R. S. Response times with a memory-dependent task. *Journal of Experimental Psychology,* 1966, *72,* 761-769.

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

Shiffrin, R. M., & Schneider, W. Controlled and automatic human information processing: II. Perceptual learning, automatic attending, and a general theory. *Psychological Review,* 1977, *84,* 127-190.

Snoddy, G. S. Learning and stability. *Journal of Applied Psychology,* 1926, *10,* 1-36.

Waterman, D. A. Adaptive production systems. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence.* Tbilisi, USSR, 1975, 296-303.

Winograd, T. W. Frame representations and the declarative-procedural controversy. In D. G. Bobrow & A. Collins (Eds.), *Representation and understanding: Studies in cognitive science.* New York: Academic, 1975.