Winston, P. H., "Learning Structural Descriptions from Examples," in *The Psychology of Computer Vision*, P. H. Winston (Ed.), McGraw-Hill, New York, 1975.

Woodworth, R. S., and Schlosberg, H., *Experimental Psychology*, rev. ed., Holt, Rinehart and Winston, New York, 1954.

11

KNOWLEDGE COMPILATION:

The General Learning Mechanism

John R. Anderson
Carnegie-Mellon University

Abstract

The ACT learning mechanisms of knowledge compilation, consisting of composition and proceduralization, are discussed. Composition operates by collapsing multiple productions into a single production that has the effect of the set. Proceduralization operates by building into productions information that previously had to be retrieved from long-term memory. It is shown how these two mechanisms can simulate the initial stages of skill acquisition in the domain of learning how to program. It is also shown that these mechanisms can reproduce the effects that have been attributed to inductive learning mechanisms involving generalization and discrimination. Generalizations and discriminations emerge as consequences of compiling the processes of analogy formation and error correction.

11.1 INTRODUCTION

One of the oldest intellectual issues is whether all forms of learning can be accounted for by associative learning—that is, learning by associating co-occurring elements (see Anderson and Bower, 1973, for a historical review). One consequence of our new technical age has been to refine this question practically out of existence. However, the issue has received a new embodiment in the world of machine learning in which there is a set of proposals about learning by induction and discovery. These learning mechanisms assume implicitly that it is not possible for adequate learning to be achieved by means of association by contiguity. Approximately half the papers at the 1983 Machine Learning Conference had this character.

The basic assumption is that any adequate learning system has to have as part of its basic architecture the ability to compare noncontiguous examples, look for commonalities and differences, formulate hypotheses, and act on these hypotheses. Two papers, however, seemed to be the intellectual heirs to the association-by-contiguity position; one was by Rosenbloom and Newell, and the other was by this author. The two proposals were quite different in detail but similar in spirit. The common spirit consisted of two assumptions about the basic architecture. The first was that behavior was controlled according to hierarchical goal structure—that the basic category of behavior was problem solving, not induction. In both papers the problem solving was encoded in a production system. The second assumption was that the only form of learning consisted of creating one-step operators that had the same effect as the multiple steps of information processing in the original problem solution. The goal structure is used in deciding which steps belong together and should be collapsed into a single operator.

The purpose of this chapter is to argue that these two architectural assumptions are adequate to account for inductive learning. This argument will be made within the context of the ACT* theory of learning (Anderson, 1983a). The ACT* theory includes both inductive learning mechanisms and operator-collapsing mechanisms. It will be shown here that operator collapsing can account for the phenomena that were attributed to the inductive mechanisms in ACT*. Of course, this leaves open the question of whether there are other inductive processes, not in ACT*, that are beyond the scope of operator collapsing. However, it will be left to others to show that such phenomena exist.

Knowledge compilation is the name given to the principles in ACT that govern operator collapsing (Anderson, 1982, 1983a; Anderson, Sauers, and Farrell, 1982; Neves and Anderson, 1981). These principles are concerned with how a new skill is acquired, such as generating a proof in geometry (Anderson, 1983b). In the general framework a learner is viewed as beginning with declarative information relevant to the execution of a skill. For instance, in the case of geometry, the student might learn about the properties of two-column proofs and various theorems and postulates. This information is stored in declarative form—that is, as facts about the domain. For this knowledge to be used, general *interpretive* procedures must be applied to it. Two types of interpretive procedures commonly observed in human subjects are general problem-solving procedures and general analogy procedures. Knowledge compilation operates on the traces of such procedures, creating more efficient procedures that are specific to the task domain.

This chapter will discuss knowledge compilation and give an example of its use to simulate the learning of initial programming skills. The rest of the chapter will be devoted to discussing how knowledge compilation can produce inductive processes of discrimination and generalization (Anderson, Kline, and Beasley, 1979; Hayes-Roth and McDermott, 1976; Langley, 1985; Michalski and Stepp, 1983; Mitchell, 1978; Vere, 1975). It will be argued further that there is evidence in the human

case that knowledge compilation is the process that underlies generalization and discrimination.

11.2 INTRODUCTION TO KNOWLEDGE COMPILATION

Knowledge compilation mechanisms are defined with respect to a production system like ACT (Anderson, 1983a), which has a separate long-term declarative memory to represent facts and a production memory to represent procedures. The knowledge compilation mechanisms operate on the traces of production applications to create new productions. Before the details of a full example or of the implementation are presented, a brief overview will be given.

The knowledge compilation processes in ACT can be divided into two subprocesses. The first, called *composition*, takes a sequence of productions that follow each other in solving a particular problem and collapses them into a single production that has the same effect as the sequence. The idea of composition was first developed by Lewis (1978). Composition speeds up the processing by creating new productions that embody the sequence of steps used in a particular problem domain. The second process, *proceduralization*, builds versions of the productions that no longer require the domain-specific declarative information to be retrieved into working memory so the information can be matched by the general interpretive productions. Thus it creates new productions that collapse the formerly separate processes of information retrieval and production matching.

The basic processes of compilation can be illustrated with the task of dialing telephone numbers. It has been noted (Anderson, 1976) that one develops a special procedure for dialing a frequently dialed telephone number. Sometimes declarative access to the number is lost, and the only access one has to the number is through a procedure for dialing it.

Consider the following two productions that might serve to dial a telephone number:

```
P1      IF the goal is to dial ?telephone-number
           and ?digit is the first digit of ?telephone-number
        THEN dial ?digit.
P2      IF the goal is to dial ?telephone-number
           and ?digit1 has just been dialed
           and ?digit2 is after ?digit1 in ?telephone-number
        THEN dial ?digit2.
```

Composition creates "macroproductions" that perform the operation of a pair of productions that occurred in sequence. Applied to the sequence of P1 followed by P2, composition would create

P1&P2 IF the goal is to dial ?telephone-number
and ?digit1 is the first digit of ?telephone-number
and ?digit2 is after ?digit1 in ?telephone-number
THEN dial ?digit1 and then ?digit2.

Compositions like this will reduce the number of production applications to perform the task.

A composed production like P1&P2 still requires that the information (in this case, the telephone number) be retrieved from long-term memory, held in working memory, and matched to the second and third clauses in P1&P2. Proceduralization eliminates clauses in the condition of a production that require information to be retrieved from long-term memory and held in working memory. In P1&P2, the second and third condition clauses would be eliminated. The variables that would have been bound in matching these clauses are replaced by the values to which they are bound in the special case. If this production is applied in dialing Mary's telephone number, which is 432-2815, the variables in P1&P2 would be bound as follows:

?telephone-number → Mary's number
?digit1 → 4
?digit2 → 3

Substituting these values for the variables and eliminating the second and third condition clauses transform the production into

P1&P2* IF the goal is to dial Mary's number
THEN dial 4 and then 3.

By further composition and proceduralization, a production can be built that dials the full number:

P* IF the goal is to dial Mary's number
THEN dial 4-3-2-2-8-1-5.

It should be emphasized that forming this production does not necessarily imply the loss of the declarative representation nor of the ability to use it interpretively. In the few reported cases where people can dial a number but not report it, the declarative knowledge probably has ceased to be used and has simply been forgotten.

Elsewhere (Anderson, 1982, 1983a; Neves and Anderson, 1981) the evidence for knowledge compilation from the literature of experimental psychology has been discussed. The major issue to be explained here is how these mechanisms are implemented and used in relatively complex problem-solving domains. The author's work on this topic has been done in the context of the GRAPES simulation of the ACT theory (Sauers and Farrell, 1982). GRAPES is a system devoted to simulating ACT in the context of novice LISP programming. It simulates the problem solving and programming of novices writing LISP functions, as well as the way novices learn from their problem-solving episodes. A typical GRAPES simulation of a subject will be

described here first, and then the mechanisms of knowledge compilation underlying that simulation will be examined.

11.3 A SIMULATION OF LISP PROGRAMMING

One of our consistent observations about novices is that they are not able to read instructions of even modest complexity and then generate the instructed behavior without error. This is not surprising given the ACT theory. According to that theory, instructions are initially stored in a declarative form, but behavior requires procedures that are represented as productions. Instructions cannot directly set up procedures to perform the skill. General interpretive productions must convert this knowledge into behavior. Many of the problems arise because of the indirection through these interpretive productions.

The problem-solving episode described here is one of the early ones simulated by the author. (For simulations of more complex programming, see Anderson, Sauers, and Farrell, 1982.) The difficulties experienced in this episode are typical of the difficulties people have in making the transition from instruction to experience. The subject, BR, had read the instruction on pages 33 to 37 of Winston and Horn (1981) on function definition, but she extracted virtually nothing from the text instruction. What she did extract was a template for how to write a function definition:

```
(DEFUN<function name>
  (<parameter 1><parameter 2> . . . <parameter n>)
  <process description>)
```

Winston and Horn assert that "angle brackets delineate descriptions of things." She also studied some examples of function definitions to which she referred. The most important of these converted Fahrenheit to Celsius:

```
(DEFUN F-TO-C (TEMP)
  (QUOTIENT (DIFFERENCE TEMP 32) 1.8))
```

BR's first problem was to define the function FIRST, which returns the first element of a list. She knew the function CAR and how to use it when interacting with the monitor in LISP. CAR returns the first element of the list that is its argument. She knew, for instance, that if she typed (CAR '(A B C)), the monitor would return the answer A. Thus this problem is really an exercise in using the syntax of function definition rather than one in defining a novel function.

BR's method of solving these problems relied heavily on trying to use the structure of the template and the examples to guide her function writing. This process is referred to as *structural analogy*. A production system was created in GRAPES that would simulate this protocol. The only productions required for this simulation were productions that could do structural analogy and productions that could use the LISP functions CAR and CDR at the top level. The first type of production represents a

general ability that can be used in many contexts (for instance, in filling out income tax forms). The second type was acquired from work with earlier chapters in Winston and Horn.

Figure 11-1 illustrates the goal tree generated in simulating this example. Each box in figure 11-1 represents a goal, and each arrow emanating from a box represents a GRAPES production trying to achieve the goal. If the production generates subgoals, it is connected to goal boxes below. The simulation starts with the goal of writing the function and chooses to use the template for function definition as a guide. This is referred to as *mapping* the template. The first subgoals that GRAPES processed in figure 11-1 involved mapping DEFUN and <function name> in the template. Productions responding to these goals wrote out ''(DEFUN FIRST'' without difficulty.

Like our subject, GRAPES was not able to write out the parameter part of the function template directly because GRAPES did not know what a "parameter" was. In cases like this, GRAPES' analogy productions will resort to a concrete example. The concrete example retrieved by GRAPES is the definition of F-TO-C given earlier. GRAPES solved the analogy as follows: *X* is to F-TO-C as the parameter list—that is, (<parameter 1> <parameter 2> . . . <parameter n>)—is to the abstract template, and it retrieved (TEMP) as the value for *X*. Thus it decided (TEMP) was serving the parameter role in F-TO-C. Then it solved the analogy *X* is to FIRST as (TEMP) is to F-TO-C and came up with the answer (LIST1), which it put into the function definition; that is GRAPES decided (LIST1) served the comparable role in the function it was defining as (TEMP) was serving in F-TO-C.

Then GRAPES turned to the process definition. Being unable to interpret directly what is meant by <process description>, it looked to its concrete example F-TO-C and saw that LISP code filled this slot, which performed the function operations. In analogy, GRAPES set its goal to write code that would perform the operations required by FIRST. A production for using CAR at the top level applied next in GRAPES, but there is no production to specify how to write the argument to the function CAR in this context. GRAPES and the subject know that CAR will operate on LIST1, but they do not know the syntax for specifying the argument LIST1. GRAPES again turns to its concrete example F-TO-C. It solves the analogy (CAR ARG) is to (QUOTIENT X) and retrieves (DIFFERENCE TEMP 32), which is the first argument to QUOTIENT. It then solves the analogy problem of what it must do to LIST1 to make it like (DIFFERENCE TEMP 32) and decides it should embed LIST1 in parentheses. The subject made the same error. The function definition at this point as written by both subject and GRAPES is

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

There are two things to note at this point. First, the subject has read information in the text that would have enabled her to know that she should not embed LIST1

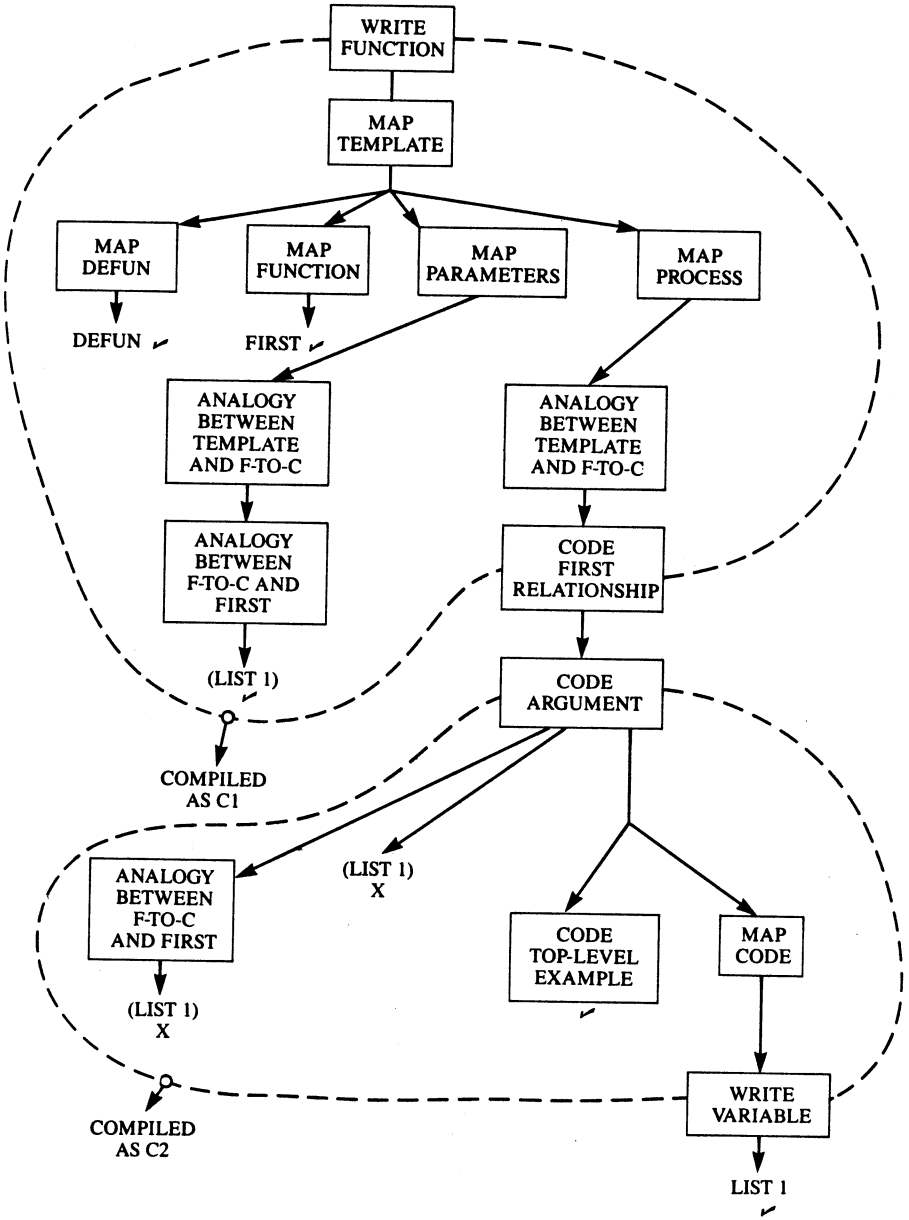

Figure 11-1: A representation of the goal structure in subject BR's solution to the problem of writing the function FIRST. The boxes represent goals, and the arrows indicate that a production has decomposed the goal above into the subgoals below. Checks indicated successful goals, and X's indicate failed goals. The dotted lines indicate parts of the goal tree combined in composition.

in parentheses, but this has no impact on her behavior. Second, on previous occasions she had correctly specified variable arguments when evaluating functions at the top level. Eventually, the experimenter used this second fact—that the subject could do it correctly at the top level—to guide her to a correct solution. Both of these observations illustrate the relative isolation of knowledge; that is, knowledge studied or used in one context is not available in another context.

When the subject tries her function definition, an error message is generated: `''LIST1: undefined function object.''` GRAPES received the same error message when it tried out the same function definition. The error occurred because LISP treats the first thing inside a parenthesized expression as a function. GRAPES associated this error with the failure to specify the argument to `CAR` correctly. On previous occasions BR had encountered the same error at the top level while typing in commands like `(CAR (A))`, where the argument to `CAR` is to be taken literally rather than evaluated. In the past she had always repaired these errors by *quoting* the argument; that is the argument is preceded by a single quotation mark: `(CAR'(A))`. It is assumed that both the subject and GRAPES have compiled from previous experience a rule and that the way to repress this error is by quoting the argument, which stops LISP from evaluating. The function definition at this point is

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

When this new function is tried on an example, LISP returns the `CAR` of `'(LIST1)`, which is `LIST1`, rather than the first element of the value of `LIST1`. This is the point at which the experimenter intervened and reminded the subject of how she would solve the problem at the top level. If the subject were writing code at the top level she would have used `(CAR LIST1)` rather than `(CAR (LIST1))` or `(CAR '(LIST1))`. This intervention was simulated in GRAPES by refocusing it on the code-argument goal in figure 11-1 and putting `(CAR LIST1)` as a top-level example in working memory. Then GRAPES and the subject both mapped this code to its current function definition and came up with the correct code.

11.3.1 Knowledge Compilation

After finally solving this problem, the knowledge compilation mechanism formed two productions that aided its solution of the second problem. These productions summarized much of the problem solving that took place:

```
C1      IF the goal is to write ?function defined on ?variable
        THEN write (DEFUN ?function ( ?variable)
             and set as a subgoal to code ?relation calculated by ?function
             and then write).
```



```
C2      IF the goal is to code ?argument
           and ?argument corresponds to ?variable of ?function
        THEN write ?variable.
```

The portions of the goal tree that are summarized by each of these productions are encircled in figure 11-1. The first production captures the top-level syntax of a function, and the second summarizes the search involved in discovering how to specify an argument to a function. With these productions, GRAPES was able to write a second function much more easily, as was the subject. This function, called `SECOND`, was to return the second element of a list.

11.3.2 Conclusions

There are a number of conclusions to be drawn from BR's protocol and the GRAPES simulation. The first concerns the importance of structural analogy in bridging the gap between current knowledge and the needed behavior. There are two sources for the structure from which the analogy is made. One is templates and worked-out problems provided in the text. The other is structures that the subject can generate; for instance, the subject generated `(CAR LIST1)` as a top-level solution and then used this in her function definition.

A second conclusion is that a problem-solving episode is organized as a hierarchical goal structure in which the goals are expanded in a depth-first and left-to-right fashion. Jeffries et al. (1981) note this hierarchical structure in the programming behavior of experts, although their subjects use breadth-first expansion in contrast to the depth-first expansion used by these novice subjects.

The third conclusion concerns the importance of knowledge compilation in extracting new production rules from an example problem. These rules streamline the solution of later problems. As the protocol shows, the learning can be accomplished on the basis of a single example. It should be stressed that the lessons of this example "stuck"; that is, BR on later days did not have the same difficulty with the basic syntax of function definition or with argument specification. It should also be stressed that compilation depends critically on the structure of goal trees; that is, the structure of the goal tree identifies what parts of the problem-solving episode belong together and what can be collapsed into a single rule.

In these three features—structural analogy, hierarchical goal trees, and knowledge compilation—we have one complete solution to the issue of how the learner is able to make the transition to a new cognitive behavior. The important question is whether there is anything in this transition that might be called induction. In the opinion of the author, there is not. The process of analogy formation is purely a problem-solving effort to make the structure of the current solution similar to the structure of the old solutions. Knowledge compilation just puts steps that had occurred in the original problem into single operators. The system does not try to form any generalization—and, indeed, how could it, working from a single example?

Still, the result is a pair of general operators for writing LISP functions. This almost has the flavor of "black magic." In later sections this black magic will be explained.

11.4 FURTHER DISCUSSION OF COMPILATION

Before a discussion of how generalizations occur is presented, some further examples of proceduralization and composition will be considered.

11.4.1 Proceduralization

Proceduralization can be illustrated in its pure form by the following example. In GRAPES there is a production that will retrieve function definitions from long-term memory and apply them as follows:

```
IF the goal is to code ?relation on ?argument
   and there is ?function that codes ?relation
THEN use ?function with ?argument
   and set as a subgoal to code ?argument.
```

In this production, `?relation`, `?function`, and `?argument` are variables that allow the production to match different data. The second line of the condition might match, for instance, `''CAR codes the first member of a list''` with `?function` bound to CAR and `?relation` bound to first member. If this rule is proceduralized to eliminate the retrieval of the CAR definition, it becomes

```
C3   IF the goal is to code the first member of ?argument
   THEN use CAR of ?argument
      and set as a subgoal to code ?argument.
```

This is achieved by deleting the second clause in the first production, which required memory retrieval, and making the rest of the production specific to the relation *first element* and the function CAR. Now a production has been created that can directly recognize the application of CAR. The amount of information that has to be maintained in working memory is thus reduced.

11.4.2 Composition

As an example of pure composition, let us suppose we wanted to add the first member of LIST1 to LIST2. Then the following two operators would apply in sequence:

```
IF the goal is to add ?element to ?list
THEN CONS ?element to ?list
   and set as subgoals to code ?element
   and to code ?list.

IF the goal is code the first member of ?argument
THEN use CAR of ?argument
   and set as a subgoal to code ?argument.
```


The first rule above would apply binding `?element` to "the first member of LIST1" and `?list` to "LIST2." The second production would apply binding `?argument` to "LIST1." A simple case of composition would involve combining these two productions together to produce

```
C4   IF the goal is to add the first member of ?argument to ?list
   THEN CONS the CAR of ?argument to the ?list
      and set as subgoals to code ?argument
      and to code ?list.
```

Such compositions collapse repeated sequences of coding operations to create macro-operators. The result is a speedup in coding. A major issue concerns deciding which productions to compose together. The above example is a fairly simple case of collapsing two levels of a goal tree into one. However, in some cases, such as that presented in figure 11-1, many productions can be collapsed. GRAPES determines which productions to collapse by inspecting the goal tree. Composition distinguishes between two types of goals: *inherent goals* and *planning goals*. Inherent goals are intrinsic parts of the programming task. For current purposes, inherent goals are all variants of writing code. The important feature of inherent goals is that in achieving them one achieves part of the original task. Planning goals produce results that are used to guide the solution of the original problem, but the results themselves are not part of the original problem. In figure 11-1 the inherent goals are "code the function," "code the relationship," "code the argument," and "write the variable"; all the rest are planning goals.

Composition collapses productions in one of two ways. One is by eliminating the planning goals that are intermediate between two inherent goals. This is what happened when production C1, given earlier, was formed for figure 11-1. Composition formed a rule that went from the goal of "code the function" to the goal of "code the relationship." In doing this, it compiled out the planning process and simply left in the products of that planning. The same process underlies the formation of C2 from the goal tree in figure 11-1.

The second way composition collapses productions is illustrated in the case of C4. Here composition starts with the goal of adding the first element of one list to another, skips over the intermediate goal of coding the first element of the list, and sets the goals of coding the two lists. In doing this it is basically creating macro-operators similar to those in STRIPS (Fikes and Nilsson, 1971). This learning scheme requires that the learner be able to identify which subgoals are essential to the problem solutions and which are only intermediate to the final solution.

It needs to be emphasized that neither proceduralization nor composition eliminates the original production rules from which they were built. Rather, the new

compiled rules just serve as supplemental rules that produce better performance in certain circumstances.

The effect of the knowledge compilation process is to create a set of productions that mirror the structure of LISP. They may explicitly involve LISP functions like CAR or COND or LISP programming techniques like tail-recursion. These productions will preserve the inherent goals that are specific to LISP and delete the planning goals involved in domain-general processes like structural analogy. Thus representative productions become the following (see Anderson, Sauers, and Farrell, 1982):

C5 IF the goal is to code the second member of ?list
THEN use CADR and set a subgoal
to code ?list.

C6 IF the goal is to obtain all the elements which have
?relation to any member of ?list
THEN use MAPCONC and set as subgoals
1. To code ?function that will return all the elements that have
?relation to ?argument.
2. To code ?list.

11.5 GENERALIZATION

Generalization is a mechanism for learning new productions in ACT*. It is the learning mechanism in ACT* that is most transparently inductive. As it is typically formulated (see, e.g., Anderson, Kline, and Beasley, 1980), it takes a pair of productions and generates what is called the *maximal common generalization.* This is the most specific production that will apply everywhere that the original productions would and that will have the same effect as the original productions. For a simple example, consider the following pair of productions:

S1 IF Fred is rich
and Fred is ugly
and Fred is smart
and Fred is of medium height
THEN Fred is in club 1.

S2 IF Gail is rich
and Gail is ugly
and Gail is stupid
and Gail is of medium height
THEN Gail is in club 1.

The following production would be formed as the generalization of the two:

G1 IF ?person is rich
and ?person is ugly
and ?person is of medium height
THEN ?person is in club 1

where ?person is a variable. The pair of productions can be viewed as specific observations and G1 as a generalization formed from these observations. Note that G1 is formed both by deleting condition clauses and by replacing constants by variables. These are the two transformations that occur when one moves to generalizations.

The interesting observation is that compilation results in clause deletion and replacement of constants with variables. Compilation deletes clauses associated with omitted goals and with planning. Variables from planning productions can remain in the compiled productions. This is how the effect of generalization is obtained through compilation.

Specifically, it appears that generalizations can be formed through the process of compiling analogies. The basic framework is as follows: The system has instances committed to memory as declarative facts. When a new instance is encountered, the system compares it to a memorized instance and uses the structure of the memorized instance to guide response. Compiling this compare-and-respond behavior produces a production that is a generalization from the two instances. This is basically what happened in the case of forming C1 and C2 from figure 11-1. A simpler example will be presented here to make this process more transparent. Then psychological evidence will be introduced indicating that this is the correct conception of generalization in humans.

Consider a very simple production set that will classify new instances according to their similarity to studied instances:

P1 IF the goal is to classify ?object
and ?reference has been studied
THEN initialize the measure of overlap.
and set as subgoals to compare ?object to ?reference
and to determine if ?object is in the same category as ?reference.

P2 IF the goal is to compare ?object to ?reference
and ?object has ?feature
and ?reference has ?feature
THEN increment the measure of overlap.

P3 IF the goal is to compare ?object to ?reference
and there are no more matching features
THEN POP the goal.

P4 IF the goal is to determine if ?object is in the same category as
?reference
and the measure of overlap is above threshold
and ?reference is in ?category
THEN ?object is in ?category.

P5 IF the goal is to determine if ?object is in the same category as ?reference
and the measure of overlap is below threshold
THEN POP failure.

This production set will keep comparing the object to be classified to candidate references from memory until a high-similarity reference is found. Then it will place the object in the same category as the high-similarity reference.

Suppose the system has committed to memory that Fred is in category 1 and that he is rich, ugly, smart, and of medium height. Then the system is asked to categorize Gail, who is rich, ugly, stupid, and of medium height. Figure 11-2 illustrates the goal tree generated by GRAPES in applying this production set to the classification problem. It found matches on three features and a mismatch on one, which, it will be assumed, was sufficient to exceed threshold. The compilation process regarded all the similarity comparisons as planning goals and compiled all of this out, forming the following single production:

IF the goal is to classify ?object
and ?object is rich

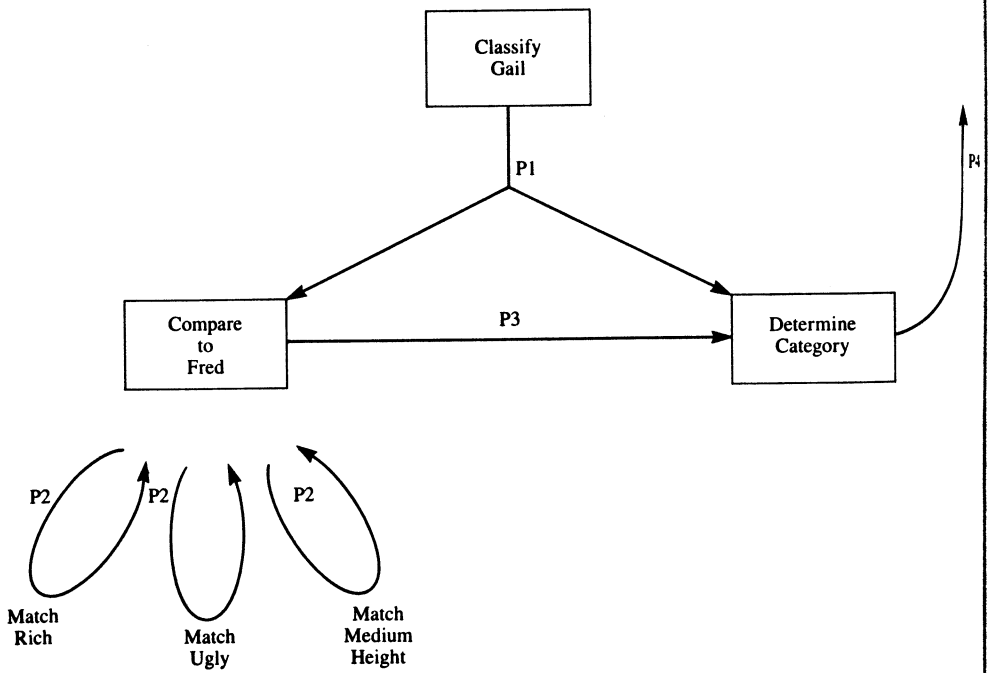

Figure 11-2: A representation of flow of control in the classification of Gail by analogy to Fred.

and ?object is ugly
and ?object is of medium height
THEN ?object is in club 1.

This production is essentially identical to the generalization G1 given earlier. Thus compiling the process of making an analogy will result in generalization.

There are a number of features that distinguish this path to generalization from a standard induction paradigm such as the ACT* generalization mechanism. First, it is based on a single example. Second, being the result of conscious problem solving, it is more flexible. If the system thought certain features (such as appearance) should be discounted in analogy formation, this could be accommodated and a different analogy would appear. If the system thought that what was important was number of extremely positive features (e.g., rich, smart), again this could be accommodated and a different generalization would appear.

What has been described is a procedure for producing the same effect as generalization. What reason is there to believe that this procedure is the process underlying generalization rather than the more direct generalization mechanism? First, it is more plausible within the ACT production system framework. To produce a generalization in ACT it is necessary to start out with very specific productions like S1 and S2 given earlier. This is because the generalization mechanism only works on productions. These highly specific productions are patently nonintuitive, and moreover the ACT theory would claim it is not possible to form such productions directly. Thus an ACT theory that produces generalization through compilation is more plausible than one that uses the direct generalization mechanism.

Second, independent of the ACT framework, the evidence is against an automatic generalization mechanism that has no strategic component. Both Elio and Anderson (1984) and Kline (1983) have shown that the generalizations learners emerge with depend on both their problem-solving set and the order of the examples. Thus it does not seem that humans extract always and only the maximal common generalizations. The generalizations they do extract are determined by what they are looking for. This is better modeled by a system like the preceding one that leaves similarity detection at the strategic level.

Third, notice that the system can classify on the basis of a single studied instance. A generalization scheme requires at least two instances to form a generalization before novel instances can be classified. Elio and Anderson (1981) showed that subjects can classify novel instances on the basis of similarity to a single studied instance when there is no possibility of generalization. This is consistent with the current conception. In addition, Elio and Anderson showed that subjects could better classify a novel instance when there was a relevant generalization that could be formed from two studied instances—even when overall similarity of studied instances to novel instances was kept constant. This shows that generalizations also

are made and that all classification is not a matter of analogy. Again the evidence is consistent with the current scheme.

Furthermore, experiments have been done in which subjects can classify novel instances after studying instances while unaware of their classification structure (Brooks, 1978). For instance, subjects learn to associate animals and cities to stimuli in a simple paired-associate experiment. After doing the paired-associate task, they are told that all the stimuli associated to new-world animals and cities form one category and all stimuli associated to old-world responses form another category. Subjects were unaware of the old-world/new-world dimension during study. They can now reliably classify new instances as old- or new-world stimuli. Since subjects were not aware of the classification structure at study there was no opportunity to form generalizations. The classification must be on the basis of comparing specific studied stimuli to specific test stimuli.

Carbonell provides an elaborate discussion of how analogy might be used to guide problem solving that goes beyond the discussion here (1983). He also speculates on how generalizations might emerge from the analogy process. He proposes that the learner would store analogical solutions and generalize from these in just the way one typically generalizes from example. This discussion shows that generalizations can emerge as a by-product of the analogy process without a separate generalization phase.

11.6 DISCRIMINATION

Less work has been done on discrimination in the knowledge acquisition literature, and there is less consensus about how discrimination is done. However, the general situation calling for discrimination is one in which there is an overly general rule and thus a need to restrict its range. It is doubtful that there is a single way to form discriminations, and this probably accounts for the lack of consensus in the literature. Certainly there is not a uniform discrimination process in humans. However, for purposes of discussion a kind of discrimination called *action discrimination* in the ACT theory (Anderson, 1982, 1983a) will be presented here. Again for clarity it will be illustrated with simple classifications, but Anderson (1982) can be consulted for more complex applications.

Suppose we have the following general production:

```
G2      IF  ?person is rich
            ?person is of medium height
        THEN ?person is in club 1.
```

Suppose this rule is applied to David, who is smart, good-looking, rich, and of medium height. It would classify David as being in club 1. However, suppose it is subsequently learned that David is in club 2. Thus an error has been made, and the system sets out to form a discrimination that will correct this error.

In forming a discrimination the system looks for some past instance to which the rule correctly applied. Suppose in this case Fred, who is smart, ugly, rich, and of medium height, is retrieved as someone correctly classified by the rule. The discrimination mechanism looks for some feature that was true of the successful instance but not true of the unsuccessful instance. In this case, the only difference is that David is good-looking and Fred is ugly. Therefore, a new production is formed that has this added feature:

```
D1      IF  ?person is rich
            and ?person is of medium height
            and ?person is good-looking
        THEN ?person is in club 2.
```

This is called an action discrimination because it has a different action from the production (G2) from which it was formed. D1 does not replace G2, but because of the specificity ordering of productions in ACT, D1 will take precedence over G2 whenever both match. G2 will only apply when the person is not good-looking. Note that in forming an action discrimination the system both adds to the condition of a production and changes the action.

The interesting thing to note is that compilation can have the effect of adding to the condition of a production and changing the action. When we compose two productions, P1 followed by P2, we can add to the conditions of P1 some of the conditions of P2, and we can change the action of P1 to the action of P2. Thus it appears that we might be able to get the effect of discrimination through compilation.

The basic scheme for getting discrimination through compilation is to have the system consciously and deliberately go through the steps involved in forming a discrimination. Discrimination is done by problem-solving productions rather than as an automatic process above the production system. Thus a sequence of productions will compute a discrimination. Compiling that sequence will result in a discriminate production. As will be seen, doing this is not as straightforward technically as the generalization case. This is because it would be necessary to inspect the contents of productions to mimic perfectly the automatic discrimination process, whereas this is not necessary in order to mimic generalization. In the ACT theory one production cannot inspect the contents of another. Still, we can get nearly the effect of automatic discrimination in a fairly plausible way.

As noted earlier, there are multiple ways of forming discriminations. This section will focus on the action discrimination as just sketched out. However, this general scheme will probably extend to other types of discrimination. In this scheme the productions compute the discrimination and then compile that computation. After the compilation of an action discrimination is illustrated, the evidence that this view of discrimination is psychologically correct will be considered.

This example will be developed within the framework of productions P1–P6 given earlier for classifying objects by analogy. The learning situation here is one in

which the learner makes a classification of an object, receives feedback, and if the classification is incorrect tries to find a discriminating feature. Assume the learner has compiled the following too-general production:

G2* IF the goal is to classify ?object
and ?object is rich
and ?object is of medium height
THEN ?object is in club 1.

This is a variant of G2 given earlier but now set up for the current framework. As earlier, assume it misapplies to classify David, who is smart, good-looking, rich, and of medium height, as being in club 1. An error is detected, and a goal is set to reclassify David.

The following three productions are relevant to the correction of a misclassification:

P7 IF the goal is to classify ?object
and ?object is in ?category1
but ?object was classified as in ?category2
and ?reference is in ?category2
THEN remove the classification of ?object
and set as subgoals to find a difference between ?object and ?reference
and then to reclassify ?object
and then to correct the classification of ?object.

P8 IF the goal is to find a difference between ?object and ?reference
and ?object is in ?category1
and ?object has ?feature1 on ?dimension
and ?reference has ?feature2 on ?dimension
and ?feature1 is different from ?feature2
THEN conclude ?object is in ?category1 because of ?feature1.

P9 IF the goal is to correct the classification of ?object
and ?object has been classified as in ?category2
but ?object is in ?category1 because of ?feature1
THEN change the classified category of ?object to ?category1.

Production P7 will apply to correct the situation. It selects some instance that was in the incorrect category. Suppose again it selects Fred, who is rich, ugly, smart, and of medium height. This is an instance the overgeneral production G2* would have correctly classified. There is no guarantee that the instance selected by P7 will be one that fits the overgeneral production, because P7 cannot inspect G2*. However, the spreading activation retrieval mechanisms in ACT (see Anderson, 1983a) would tend to select an instance that overlaps highly with the current instance and hence the rule

that classified the current instance. P7 sets goals to find a difference between David and Fred and then to reclassify David.

Production P8 will note that David is good-looking and Fred is ugly and so enter into memory the fact that "David is in club 1 because he is good-looking." Then the system will return to the goal of classifying David once again. Again G2* would apply to misclassify David as being in club 1, but now production P9 can apply to correct the classification of David. P7 followed by P8 followed by G2* followed by P9 constitutes a goal tree that can be composed together and proceduralized. The result will be the following production:

D1* IF the goal is to classify ?object
and ?object is rich
and ?object is of medium height
and ?object is good-looking
THEN ?object is in club 2.

This is essentially the same as D1 formed earlier by the automatic discrimination mechanism.

There are two essential features that distinguish this means to discrimination from the automatic discrimination of the current ACT theory. First, it requires that the learner initially make a classification of the object and then correct that classification if it is in error. Second, it requires that the learner make a conscious hypothesis about what distinguishes the current instance from prior instances that were in the hypothesized category. Both of these features have been confirmed in two series of experiments by Lewis and Anderson (1985). One series of experiments involved rules for proving triangles congruent in geometry, and the other series involved rules for traveling through a maze. In both series subjects were given overgeneral rules that had to be discriminated. Subjects who passively studied instances failed to learn the discriminating features. Subjects learned only when they made active hypotheses about the correct rule, which could then be disconfirmed. Furthermore, the only subjects to learn were those who had some conscious awareness of what the discriminating features were. Thus subjects discriminated only when both conditions of the current scheme of discriminating through compilation were satisfied.

The example illustrated above is somewhat unrealistic since it assumes the learner both finds the discriminating feature and uses it to reclassify the item. However, this is what is required if a discriminated production is to be formed on the same trial as the discriminating feature is identified. It seems more reasonable to assume that this is stretched out over multiple trials—on one trial the learner forms a declarative proposition about the importance of a discriminating feature like "good-looking," and on another trial the learner acts on this information. The discriminating production would be acquired on the later trial when the learner acted. There is no guarantee in this situation that the production learned would be identical to what

is formally defined as an action discrimination in ACT*, but the learning would be in the general direction of discrimination.

11.7 GENERAL CONCLUSIONS

There is ample evidence for the existence of something close to knowledge compilation as a basic process of human learning (Anderson, 1982; Anderson, Farrell, and Sauers, 1983; Neves and Anderson, 1981). The evidence had always been rather scarce for the details of the ACT mechanisms of generalization and discrimination, and more recently some rather negative evidence has been gathered. Clearly, humans can approach the task of extending experience as typical problem solving. Generalized or discriminated productions appear as the product of compiling this problem solving.

It is something of an embarrassment that the author worked so long with the ACT mechanisms before realizing how the knowledge compilation mechanisms could be recruited to provide the effect of generalization and discrimination. This is because he thought of knowledge compilation as simply making existing paths of processing more efficient rather than enabling novel paths of processing. To get novel behavior it seemed that inductive mechanisms of learning such as generalization and discrimination were needed. What was not recognized was that if the results of acting on the basis of similarity detection and difference detection were compiled, productions could be produced that enabled novel behavior.

The fundamental point then is that the induction process occurs as a conscious problem-solving effort to find a basis for dealing with a new case. In compiling the results of this problem solution, productions are formed that will extend to the new situation. The fundamental category of behavior is problem solving, not induction. This theory is one of learning not by temporal contiguity but by contiguity in the problem-solving goal structure. There is no such thing as unconscious induction of features. Recently, Dulany, Carlson, and Dewey (1984) have demonstrated that in situations in which subjects are supposedly engaged in unconscious induction they can be shown to have conscious inductive hypotheses on which they are acting.

References

Anderson, J. R. *Language, Memory, and Thought*, Erlbaum, Hillsdale, N.J., 1976.

———, "Acquisition of Cognitive Skill," *Psychological Review*, Vol. 89, pp. 369–406, 1982.

———, *The Architecture of Cognition*, Harvard University Press, Cambridge, 1983a.

———, "Acquisition of Proof Skills in Geometry," in *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, Palo Alto, Calif., 1983b.

Anderson, J. R., and Bower, G. H., *Human Associative Memory*, Winston, Washington, D.C., 1973.

Anderson, J. R., Kline, P. J., and Beasley, C. M., "A General Learning Theory and Its Application to Schema Abstraction," in *The Psychology of Learning and Motivation*, Vol. 13, G. H. Bower (Ed.), Academic Press, New York, 1979.

———, "Complex Learning Processes," in *Aptitude, Learning, and Instruction*, Vol. 2, R. E. Snow, P. A. Frederico, and W. E. Montague (Eds.), Erlbaum, Hillsdale, N.J., 1980.

Anderson, J. R., Sauers, R., and Farrell, R., "Learning to Plan in LISP," Technical Report ONR82-1, Carnegie-Mellon University, 1982.

Brooks, L. "Nonanalytic Concept Formation and Memory for Instances," in *Cognition and Categorization*, E. Rosch and B. B. Lloyd (Eds.), Erlbaum, Hillsdale, N.J., 1978.

Carbonell, J. G., "Learning from Analogy: Formulating and Generalizing Plans from Past Experience," in *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, Palo Alto, Calif., 1983.

Dulany, D. E., Carlson, R. A., and Dewey, G. I., "A Case of Syntactic Learning and Judgment: How Conscious and How Abstract?" *Journal of Experimental Psychology: General*, Vol. 113, pp. 54–55, 1984.

Elio, R., and Anderson, J. R., "The Effects of Category Generalization and Instance Similarity on Schema Abstraction," *Journal of Experimental Psychology: Human Learning and Memory*, Vol. 7, pp. 397–417, 1981.

———, "The Effects of Information Order and Learning Mode on Schema Abstraction," *Memory and Cognition*, Vol. 12, pp. 20–30, 1984.

Fikes, R. E., and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, Vol. 2, pp. 189–208, 1971.

Hayes-Roth, F., and McDermott, J., "Learning Structured Patterns from Examples," *Proceedings of the Third International Joint Conference on Pattern Recognition*, Coronado, Calif., pp. 419–23, 1976.

Jeffries, R.; Turner, A. A.; Polson, P. G.; and Atwood, M. E., "The Processes Involved in Designing Software," in *Cognitive Skills and Their Acquisition*, J. R. Anderson (Ed.), Erlbaum, Hillsdale, N.J., 1981.

Kline, P. J., "Computing the Similarity of Structured Objects by Means of a Heuristic Search for Correspondence," Ph.D. diss., University of Michigan, 1983.

Langley, P., "A General Theory of Discrimination Learning," in *Self-modifying Production System Models of Learning and Development*, D. Klahr, P. Langley, and R. T. Neches (Eds.), Bradford Books, Cambridge, Mass., 1985, in press.

Lewis, C. H., "Production System Models of Practice Effects," Ph.D. diss., University of Michigan, 1978.

Lewis, M., and Anderson, J. R., "The Role of Feedback in Discriminating Problem-solving Operators," *Cognitive Psychology*, 1985, in press.

Michalski, R. S., and Stepp, R. E., "Learning from Observation: Conceptual Clustering," in *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), Tioga, Palo Alto, Calif., 1983.

Mitchell, T. M., "Version Spaces: An Approach to Concept Learning," Ph.D. diss., Stanford University, 1978.

Neves, D. M., and Anderson, J. R., "Knowledge Compilation: Mechanisms for the Automatization of Cognitive Skills," in *Cognitive Skills and Their Acquisition*, J. R. Anderson (Ed.), Erlbaum, Hillsdale, N.J., 1981.

Sauers, R., and Farrell, R. "GRAPES User's Manual," Technical Report ONR-82-3, Carnegie-Mellon University, 1982.

Vere, S. A., "Induction of Concepts in the Predicate Calculus," *Proceedings of the Fourth IJCAI*, Tbilisi, Georgia, USSR, pp. 281–87, 1975.

Winston, P. H., and Horn, B. K. P., *LISP*, Addison-Wesley, Reading, Mass., 1981.

12

LEARNING PHYSICAL DOMAINS:

Toward a Theoretical Framework

Kenneth D. Forbus
Dedre Gentner
University of Illinois
at Urbana-Champaign

Abstract

This chapter presents a theoretical framework that is being developed in an attempt to construct a computational account of human learning of physical domains. Qualitative Process theory is used to model portions of people's physical knowledge, and Structural Mapping theory is used to characterize the computations that move a learner from one representation to another. The chapter outlines the component theories and proposes a learning sequence for physical domains.

12.1 INTRODUCTION

People use and extend their knowledge of the physical world constantly. Understanding how this fluency is achieved would be an important milestone in understanding human learning and intelligence, as well as a useful guide for constructing machines that learn. The authors' purpose is to construct a computational account of human experiential learning in physical domains.

This work is still at the stage where questions are being refined rather than answers provided. In many cases, there is no direct evidence for the claims made here. In other instances, support for the theory is obtained by combining evidence from several different areas, including developmental psychology, studies of

MACHINE LEARNING

An Artificial Intelligence Approach
Volume II

Contributors:

Saul Amarel
John R. Anderson
Ranan B. Banerji
Robert C. Berwick
Gary L. Bradshaw
Mark H. Burstein
Jaime G. Carbonell
Gerald DeJong
Nachum Dershowitz
Thomas G. Dietterich
Kenneth D. Forbus
Jean-Gabriel Ganascia
Dedre Gentner
John H. Holland
Smadar T. Kedar-Cabelli
Yves Kodratoff
Patrick Langley
Michael Lebowitz
Douglas B. Lenat
Sridhar Mahadevan
Ryszard S. Michalski
Donald Michie
Allen Newell
J. Ross Quinlan
Paul S. Rosenbloom
Claude Sammut
Bernard Silver
Herbert A. Simon
Robert E. Stepp III
Gail E. Thornburg
Paul E. Utgoff
Patrick H. Winston
Jan M. Zytkow

Editors:

Ryszard S. Michalski
University of Illinois
at Urbana-Champaign, IL

Jaime G. Carbonell
Carnegie-Mellon University
Pittsburgh, PA

Tom M. Mitchell
Rutgers University
New Brunswick, NJ

MORGAN KAUFMANN
PUBLISHERS, INC.
95 FIRST STREET, LOS ALTOS, CALIFORNIA 94022