4/16

5 Learning to Program Recursive Functions

John R. Anderson
Carnegie-Mellon University

Peter Pirolli
University of California

Robert Farrell
Yale University

INTRODUCTION

To iterate is human. To recurse, divine.
-Logout message on the Carnegie-Mellon CMUA computer.

Learning to write recursive programs is notoriously difficult. It is likely that students learning to program LISP would almost unanimously agree that writing recursive functions is the biggest hurdle they face. This chapter discusses (a) why learning recursive programming is so difficult, and (b) how it is successfully mastered.

To provide a framework for later discussion, we first describe how the recursive programming behavior of an expert is modeled in GRAPES (Goal-Restricted Production System), a production system developed (see Anderson, Farrell, & Sauers, 1982, 1984) to model programming in LISP. Second, we will discuss why recursive programming is so difficult to learn. To foreshadow this, our conclusion will be that it is difficult both because it is a highly unfamiliar mental activity and because it depends on developing a great deal of knowledge about specific patterns of recursive programs. Third, we will offer a general proposal as to how recursive programming is typically learned. In line with the learning of other aspects of LISP, recursive programming seems to be learned by analogy to example programs and by generalization from these examples. Fourth and last, we will discuss a series of protocols used by one subject trying to learn recursive programming. We discuss these protocols in the light of GRAPES simulations of her behavior. This last exercise is intended to provide evidence both for our

proposals about recursive programming and for our GRAPES model of LISP programming behavior.

SIMULATION OF LISP PROGRAMMING

We developed GRAPES to model how subjects write functions (i.e., programs) in the LISP language, and how subjects learn from their problem-solving episodes. GRAPES is a production system architecture which emulates certain aspects of the ACT* theory. Each production in GRAPES has a condition which specifies a particular programming goal and various problem specifications. The action of the production can be to embellish the problem specification, to write or change LISP code, or to set new subgoals. The details of the GRAPES production system are described in Sauers and Farrell (1982). The architecture of GRAPES differs from that of other production systems (e.g., Anderson, 1976; Newell, 1973), primarily in the way it treats goals. At any point in time there is a single goal being focused upon, and only productions relevant to that goal may apply. In this feature, GRAPES is like ACT* (Anderson, 1983) and other recent theories (Brown and Van Lehn, 1980; Card, Moran, & Newell, 1983; Rosenbloom and Newell, 1983).

To give a sense of what a GRAPES production system is like, let us consider some examples of productions that have been used in our simulations. A representative example of a production[1] that a pre-novice might have is:

R1: IF the total is to write a structure
and there is a template for writing the structure
THEN set a goal to map that template to the current case.

R1 might be invoked in a nonprogramming context such as when one uses another person's income tax form as a template to guide how to fill out his own. Productions like R1 serve as a basis for subjects' initial performance in LISP. A production that a novice might have after a few hours of learning is:

R2: IF the goal is to add List1 and List2
THEN write (APPEND List1 List2)

This production recognizes the applicability of the basic LISP function APPEND. With experience, subjects become more and more discriminating about how and when to use LISP functions. A rule that an expert might have is:

[1]Here and throughout the paper we will give English-like renditions of the production rules. A technical specification of these rules (i.e., a computer listing) can be obtained by writing to us. Also available is a user's manual (Sauers & Farrell, 1982) that describes the system.

R3: IF the goal is to check that a recursive call to a function will terminate and the recursive call is in the context of a MAP function
THEN set as a subgoal to establish that the list provided to the MAP function will always become NIL after some number of recursive calls

All programs in LISP take the form of functions that calculate various input–output relations. These functions can call other functions or call themselves recursively. A programming problem is solved in GRAPES by decomposing an initial goal of writing a function into subgoals, and dividing these subgoals into others, and so on, until goals are reached which correspond to things that can be directly written. The decomposition of goals into subgoals constitutes the AND-level of a goal tree—each subgoal must be successful for the goal to be successful. Alternative ways of decomposing a goal constitute the OR-level of the goal tree—any decomposition can be successful for the goal to be successful.

One of the basic observations we have made, of learning to program in LISP, is that subjects do not seem to learn much from the abstract instruction they encounter in textbooks. Rather they learn in the process of trying to solve problems. Our GRAPES simulations have therefore focussed on modeling problem-solving and the resultant learning (see Anderson, Farrell, & Sauers, 1982, 1984). We have developed in GRAPES a set of *knowledge compilation* learning mechanisms which create new production operators from the course of problem solutions (Anderson, 1983; Anderson, Farrell, & Sauers, 1984). Knowledge compilation summarizes extensive problem-solving attempts into compact production rules. These knowledge compilation mechanisms have successfully simulated a number of the learning transitions we have observed in our subjects. We will discuss in this paper some other simulations of learning transitions.

Simulation of a Recursive Solution

Here we would like to describe the GRAPES simulation of an expert's solution of a particularly interesting recursive problem called POWERSET. GRAPES' solution to POWERSET is arguably the prescriptively "ideal" solution. Having this ideal solution as a reference point, we will be in a position to make a number of important points about the nature of writing recursive programs.

Figure 5.1 illustrates the POWERSET problem as we present it to subjects. The subject is told that a list of atoms encodes a set of elements, and that he or she is to calculate the powerset of that set—that is, the list of all sublists of the list, including the original list and the empty list NIL.

```
(POWERSET '(A B C))
    = ((A B C) (A B) (A C) (B C) (A) (B) (C) ( ))
```

FIGURE 5.1. The POWERSET problem requires the student to write a recursive program that produces all possible subsets of an input set.

Each subject is given an example of the POWERSET of a three-element list. All subjects come up with basically the same solution. This solution is given in Table 5.1. The definition in Table 5.1 involves a secondary function ADDTO which takes as arguments a list-of-lists and an atom. ADDTO returns a list of lists composed by adding the atom to each list in the original list-of-lists argument. We have not provided a definition for ADDTO because the definition varies with the level of expertise of the programmer[2]. The basic structure of the POWERSET definition, however, does not change with expertise although there is easily a greater than a 10:1 ratio in the time taken by novices versus experts to generate the code.

Figure 5.2 presents the goal tree for the solution to the POWERSET problem produced by GRAPES. Each node in this tree (e.g., "try CDR-recursion") is a specific programming goal. Arrows show the decomposition of a goal into subgoals. For example, the goal "try CDR-recursion" decomposes to the subgoals "do terminating condition" and "do recursive step." The goals in this tree are set in a left-to-right, depth-first manner. The code presented in Table 5.1 is the product of carrying out the plan specified in Figure 5.2.

With the first goal set to code the function POWERSET (the topmost goal in Figure 5.2), the first GRAPES production to apply is:

P1: IF the goal is to code a function
and it has a single level list as an argument
THEN try to use CDR-recursion and set as subgoals to:

1. Do the terminating step for CDR-recursion.
2. Do the recursive step for CDR-recursion.

TABLE 5.1
Powerset solution

```
(Defun powerset (l)
    (cond ((null l)(list nil))
          (t (append (powerset (cdr l))
                     (addto (car l) (powerset (cdr l)))))))
```

[2]In fact some programmers insert a MAPCAR call without naming an auxiliary function.

WRITE POWERSET (L)
↓
WRITE POWERSET (L)

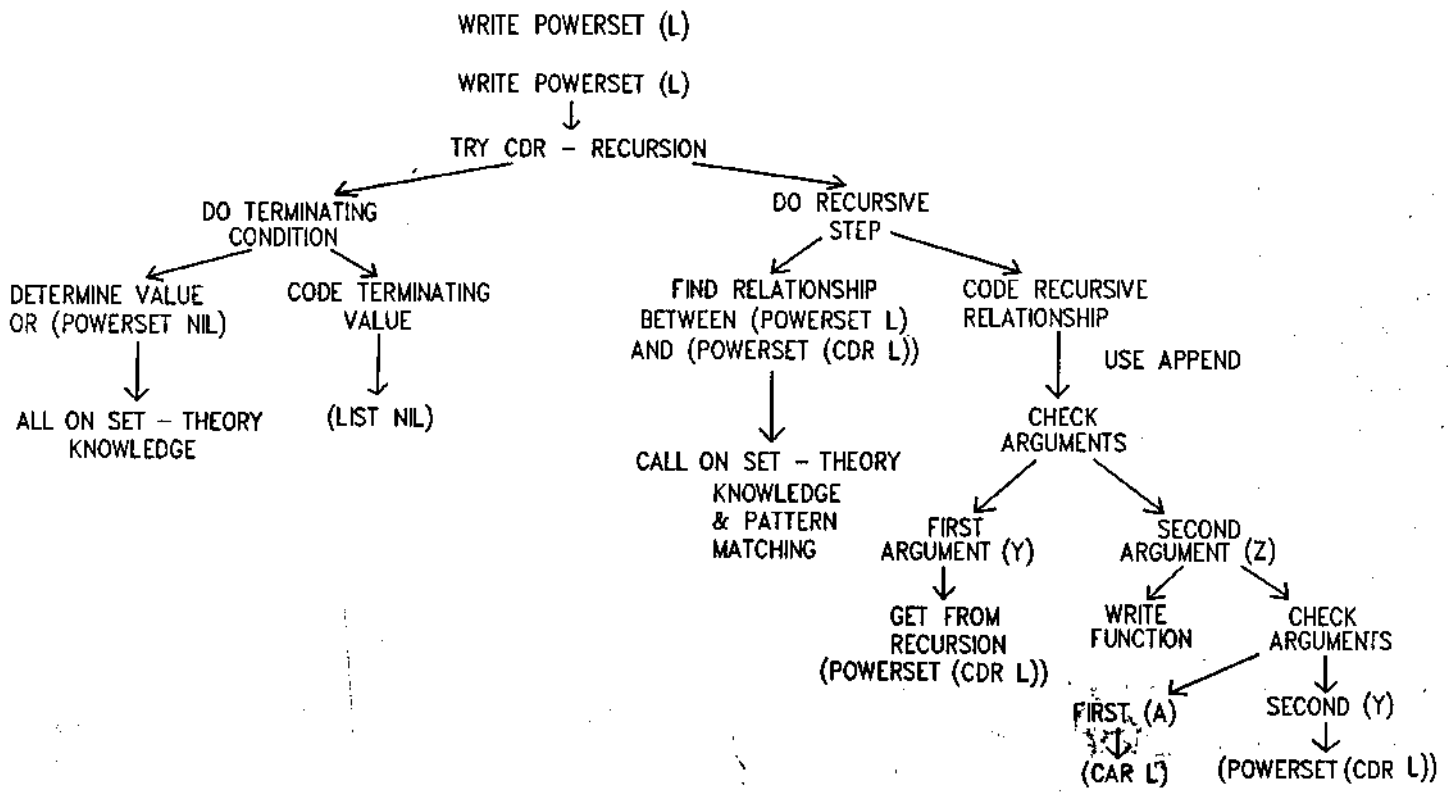

FIGURE 5.2. The goal tree for the "ideal" POWERSET solution. Arrows point from goals to their subgoals. In this tree, all of a goal's subgoals must be successful for the goal to be successful. Goals are activated in a depth-first, left-to-right manner.

CDR-recursion is a type of recursion that can apply when one of the arguments of the function is a list. It involves calling a function recursively with successively smaller lists as arguments. It is called CDR-recursion because it utilizes the LISP function CDR which removes the first element of a list. Thus, each recursive call is passed the CDR of a current argument list. Production P1 sets up the plan to call (POWERSET (CDR LIST) within the definition of (POWERSET LIST). The standard terminating condition for CDR-recursion involves the case in which the list argument becomes NIL. In this case a special answer has to be returned. So note that this "expert" production is relatively specialized—it is only concerned with a special case of recursion and only applies in the special condition that the argument list is a one-level list. Production rules are selected for application by *conflict resolution principles* in GRAPES, and one of these principles involves specificity: Productions with more specific conditions (i.e., more conditions and/or less variables) tend to be selected over productions with less specific productions. Because of this specificity principle, P1 would not apply in many situations where there was a one-level list argument. For instance, if the goal was to write a function that returned a list of the first and second elements in a list argument, other more special case productions would apply.

Activating goals in a left-to-right, depth-first manner, GRAPES turns to coding the terminating condition. In the case of CDR-recursion this amounts to deciding what the correct answer is in the case of an empty list—that is, when the list becomes NIL. The answer to this question requires examining the definition of POWERSET and noting that the POWERSET of the empty set is a set that contains the empty set. This can be coded as (LIST NIL). Each goal decomposition under "do terminating condition" is achieved by a production. We have just summarized their application here. The important feature to note is that coding the terminating condition is extremely straightforward in a case like this, where the answer can be derived from a semantically correct definition of the function. In particular, writing such code does not require an analysis of the recursive behavior of the function.

After coding the terminating condition, GRAPES turns to coding the recursive step. This is decomposed into two subgoals. One is to characterize the recursive relationship between POWERSET called on the full list and POWERSET called on the CDR or tail of the list. The other goal is to convert this characterization into LISP code. The only nonroutine aspect of applying the CDR-recursion technique is discovering the recursive relationship. Figure 5.3 illustrates what is involved. In that figure the symbol *X* denotes the result of POWERSET on a typical list and *Y* denotes the result of POWERSET on the CDR of that list. The critical insight involves noticing that *Y* is half of *X* and the other half of *X* is a list, denoted *Z*, which

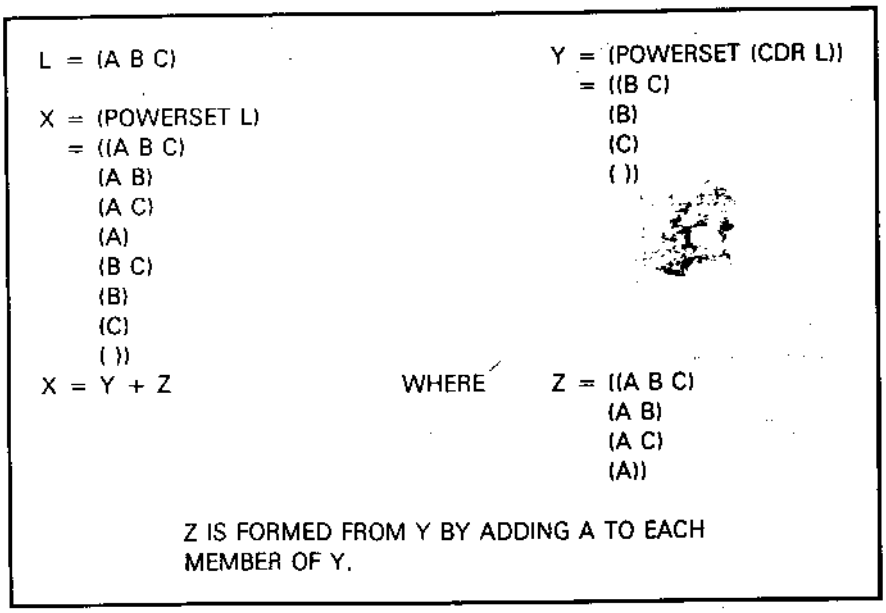

FIGURE 5.3. The POWERSET insight involves determining what must be done with the result of the recursive call, (POWERSET (CDR SETT), in order to get the result for the current function call (POWERSET SETT).

can be gotten from Y by adding the first element of the list (A in Figure 5.3) to each member of Y. Thus, $X = Y + Z$. We have developed GRAPES simulations which will produce this "correct" POWERSET characterization if given the goal to compare concrete examples of X, Y, and Z. Generally, novice LISP programmers consider this comparison only after performing many other types of comparisons of concrete examples. In contrast, experts often do not need to consider any concrete example; when they do, they typically choose almost immediately to compare a concrete example of X, the POWERSET of a whole list, and Y, the POWERSET of the CDR of the list, and search for an appropriate characterization of this relationship.

The actual coding of the recursive relationship is extremely straightforward. One production recognizes that the LISP function APPEND is appropriate for putting the two sublists Y and Z together to form the answer X. This leaves the subgoals of coding the first and second arguments, Y and Z, for the function. Another production recognizes that Y can be calculated simply as a recursive call – (POWERSET (CDR L)). There is no LISP function that will directly calculate Z, and this evokes a default production which sets a subgoal to write an auxiliary function, ADDTO, which will calculate Z given the first element of the list L and given Y.

POWERSET is one of a large class of recursive functions that lend themselves to straightforward solution in this manner. There are a number of standard recursive paradigms in LISP in addition to CDR-recursion but they all have the same straightforward character. It is the case that not all LISP recursive functions are so straightforward. One reason for complexity is that the function may not have as precise a semantics as POWERSET, and part of the problem-solving is to settle on that semantics. Part of the trick is to settle the semantics in a way that makes the coding easy. Another reason for complexity concerns recursion in nonstandard paradigms. For instance, production R3 described earlier deals with one such nonstandard recursion. However, the important observation is that standard recursion as in POWERSET causes novices great difficulty. The students we have looked at spent from just under two to over four hours arriving at the solution to POWERSET that an expert can produce in under ten minutes.

WHY IS RECURSIVE PROGRAMMING DIFFICULT?

A starting point for understanding the difficulty of recursive programming is to note that recursive mental procedures are very difficult—perhaps impossible—for humans to execute. For instance, center-embedded structures in language, while perhaps grammatical, are impossible to understand. Interestingly, the same degree of difficulty does not arise when different types of constructions are embedded within each other—only when the same construction is embedded within itself (Anderson, 1976; Dresher & Hornstein, 1976). For instance, sentence 1 below involves the embedding of two relative clauses; sentence 2, the embedding of two complement clauses; sentence 3, the embedding of a relative clause within a complement clause; sentence 4, the embedding of a complement clause within a relative clause. Sentences 1 and 2, which involve self-embedding, are much more difficult to understand:

1. The boy whom the girl whom the sailor liked hit ran away.
2. The fact that the shepherd said that the farmer had given the book to the child to the police was to be expected.
3. The fact that the shepherd reported the girl whom the sailor liked to the police was to be expected.
4. The boy who told the girl that the farmer had read the book ran away.

This pattern makes sense if we assume that distinct procedures are responsible for understanding distinct expressions. The human mind seems incapable of doing what LISP does—creating a copy of a procedure and embedding

it within itself, and keeping track of what is happening in both copies. It is also interesting that right embedding of one linguistic construction within itself does not create the same difficulty. It seems that the mind can treat such tail recursive procedures with an iterative control structure.

Indeed, it seems that it is a typical "programming trick" in the mind to change what is naturally a recursive procedure into an iterative one. A good example is the evaluation of arithmetic expressions like 4 * [(3 − 2) * (5 + 7)]. The "logical" procedure for evaluating such expressions would be a top-down recursive evaluation as one would perform in LISP. The actual procedures that people use are iterative. For instance, a frequent procedure is to scan for an embedded expression that has no embeddings (e.g., 3 − 2), evaluate it, replace the expression by its evaluation, and reiterate.

In observing how students mentally simulate recursive functions in LISP, we see some of the clearest evidence for the difficulty of recursion. The evidence is particularly clear because the evaluation process is sufficiently slow that it can be traced as it progresses in time. Students frequently show no difficulty in simulating a function making recursive calls to itself and passing control down. However, when they have to simulate the return of results and combine the partial results they get completely lost. It seems that the human mind, unlike the LISP evaluator, cannot suspend one process, make a new copy of the process, restart the process to perform a recursive call, and return to the original suspended process. In particular, it seems impossible to hold a suspended record in our mind of where we were in a series of embedded processes.

If it is the case that minds can iterate and not recurse, then it might seem obvious why recursive programming is difficult—the mind is not capable of it. We have frequently heard various forms of this argument, but they all have a serious fallacy: Writing a recursive procedure is not itself a recursive procedure! Consider that the construction of the definition of POWERSET in Figure 5.2 was not performed by a recursive mental procedure. We would like to claim that the human inability to *execute* recursive procedures is not the direct source of difficulty in *programming* recursive functions.

The Unfamiliarity of Recursion

We feel that the fundamental reason for the difficulty of recursive programming is the unfamiliarity of the activity. People have had prior preprogramming experience with following everyday procedures (e.g., recipes) and, more importantly, with specifying such procedures to others. However, because these preprogramming procedures typically had to run in human heads, they were never recursive. Therefore, recursive programming is most people's first experience with specifying a recursive procedure. Interesting-

ly, we have only observed one student who had no difficulty with learning recursive programming. Significantly, this was a graduate student in mathematics who had done a fair amount of work in recursive function theory.

A major source of difficulty in learning recursion is an instructional one. Every textbook we have examined gives students no direct help in how to generate a recursive function. Textbooks explain what recursion is, explain how it works, give examples of recursive functions, give traces of recursive functions, and explain how to evaluate recursive functions; but they never explain how to go from a problem specification to a recursive function. Thus, students have a major induction problem: How to go from the information they are given to a procedure for creating recursive functions. Textbooks are no more lucid about how to create iterative procedures, but here the student has prior experience in structuring such an induction problem.

Another difficulty is that students often think of iterative procedures for solving recursive problems. For example, many novices coding POWERSET solve the problem at hand according to the following procedure: Place the null set in the result list, then all subsets of length one, followed by all subsets of length two and so on, until the whole set is reached[3]. This procedure of successively gathering all subsets of length N is radically different from the ideal POWERSET procedure in LISP. Such a plan is difficult to achieve in code and tends to interfere with seeing the easy-to-code solution. Thus, having nonrecursive solutions to problems tends to blind students to recursive solution.

A further exacerbating factor is that there are really many different types of recursive functions in addition to CDR-recursion. Integer recursion requires recursively calling the function with a progressively smaller integer argument. CAR and CDR recursion requires calling a function recursively on the CAR (first element) and the CDR of the list. Soloway and Woolf (1980) have argued that each of these major types of recursion has many subtypes. The student is not going to be an effective recursive programmer until he learns to deal with each type. Again, typical textbooks offer the student no help; they encourage the belief that there is just one type of recursion—a function calling itself.

The Duality of a Recursive Call

Another source of difficulty (especially in LISP) is the duality of meaning in a recursive procedure call. On the one hand the call produces some resultant data; on the other hand it specifies that an operation be carried out repeatedly. Thus, the written form of a recursive call is the symbolic ana-

[3]The ordering varies somewhat from student to student.

log to a Necker cube: It can be data or complex operations, depending on your view.

Because students often perseverate on one view of recursion, they are often blinded to solutions that could be easily attained from the other view. For example, it is often useful to determine what has to be done to the result produced by a recursive call in order to get a result for the current function call. This is a key component in the POWERSET insight (see Figure 5.3) where one must determine what has to be done with the list produced by the recursive call, (POWERSET (CDR SETT)), in order to get the current function result, (POWERSET SETT). Students often miss such insights because they perseverate on the view of the recursive call as a complex flow of control. They will often attempt to trace out the flow of control rather than consider what result will be produced by a recursive call.

Complexities Which Exacerbate the Difficulty of Recursive Programming

There are other factors that really have nothing to do with recursion per se but which nonetheless complicate recursive programming. For instance, textbook problems for recursive programs are typically more difficult than for iterative ones. The POWERSET example is an instance of this phenomenon. Students frequently have problems in fully understanding the input-output relations in the first place, and then face the difficulty of maintaining this complex relation in memory. Another, presumably independent difficulty is that the data structures being operated upon are often unfamiliar. For instance, students' prior experience with list structures (critical to CDR-recursion and some other forms of recursion) is weak. It has been shown (Anderson & Jeffries, 1985) that making one part of the problem difficult impacts on the difficulty of a logically separate part.

So, in summary, recursive programming is difficult principally because it is an unfamiliar activity, with hidden complexities, that must be induced in an unfamiliar and difficult domain. The unfamiliarity of creating recursive procedures can be traced to the mental difficulty of executing recursive procedures, but the mental difficulty is not the primary reason.

Having said all this, we should point out that there is one secondary problem in recursive programming that is directly related to the mental difficulty of creating recursive procedures. This concerns checking and debugging recursive programs. This requires evaluating the programs with sample arguments, and evaluation is a recursive procedure—in contrast to program generation. Of course, students learn procedures that convert recursive evaluation into an iterative procedure, such as writing down intermediate results and states in linear stack-like structures. However, dealing with logically recursive evaluation does make it harder for students to detect er-

rors in recursive programs. So, though initial program generation does not involve the mental difficulties of recursion, program debugging does. However, it needs to be stressed that the major problems of novices are with initial program generation, and not with debugging.

PROPOSAL FOR THE LEARNING OF RECURSIVE PROGRAMMING

So how do students learn the unfamiliar procedure of generating recursive programs? Explicit procedures are not given to the student. In the absence of explicit procedures, our hypothesis is that the primary means available to students is learning from examples. By this, we mean two things. First, students can try to look at worked-out examples, and map by analogy the solution for these problems to a solution for the current problem. This is learning by analogy. Second, they can try to summarize their solution to one problem by new problem-solving operators (GRAPES productions), and apply these operators to another problem. This is learning by knowledge compilation. We believe that these two learning mechanisms are logically ordered—that the first problems are solved by analogy and that solutions to these early problems give rise to the operators that can apply to later problems. The following protocol analyses provide support for our application of this analysis to recursion. For successful application to other domains of learning LISP, see Anderson, Farrell, and Sauers (1984).

PROTOCOLS AND SIMULATIONS

We will discuss the behavior of one subject, SS, as she solved her first three recursive functions. The first recursive function was SETDIFF which took two list arguments and returned all the members in the first list that were not in the second list. The second was SUBSET, a function of two list arguments which tested if all the elements of the first list were members of the second. The third function was POWERSET. All three functions may be solved by the CDR-recursion technique. (The first two are easily and more efficiently solved by iterative techniques, but SS's textbook, in the manner typical of LISP pedagogy, does not introduce iteration until after recursion.)

SS's textbook was Siklossy's (1976) *Let's Talk LISP*, which is a somewhat singular book in regard to the amount of discussion it contains of programming technique issues. It is also designed for the programming novice and attempts a very careful introduction to all relevant concepts. It does not, however, instruct directly on how to write recursive functions, but rather it instructs on "considerations" relevant to good recursive functions and gives many examples—many of which involve set theory. SS had spent over 15

hours studying LISP at the time of these protocols. In this time she had studied basic LISP functions and predicates, conditionals, and function definitions.

Solving these three problems took SS a total of five hours. In following SS through this protocol, we can see her improving from one function to the next. We cannot say that by the end she had induced all the components required to do CDR-recursion – although she had some of them. SS continued to do LISP problems long after we finished studying her, and she eventually became quite effective at writing a wide variety of recursive functions. We can only guess that she reached her proficiency by use of more learning steps of the variety we were able to document in these protocols. It is clear that it takes a great deal of time to learn recursive programming in the traditional learning situation.

SETDIFF

The first function SS tried to write was SETDIFF. She took a little over an hour to solve the problem. Table 5.2 gives a schematic protocol of her

TABLE 5.2
SS's SETDIFF Protocol

1. SS reviews code for INTERSECTION1 function (previous problem).
2. SS reads SETDIFF problem and forms the analogy SETDIFF:INTESECTION::CDR:CAR. SS also proposes the following relation:

```
SETDIFF (SET1, SET2)
      =MINUS (SET1, INTERSECTION(SET1, SET2))
```

3. Writes (DEFUN SETDIFF (SET1 SET2).
4. Decides to code SETDIFF by rearranging INTERSECTION1 code.
5. Decides to code simple cases found in INTERSECTION1.
6. Considers case (NULL SET1), decides the action will be NIL. Code is now

```
(DEFUN SETDIFF (SET1 SET2)
      (COND ((NULL SET 1) NIL)
```

7. Considers case (NULL SET2), decides action will be SET1. Code is now

```
(DEFUN SETDIFF (SET1 SET2)
      (COND ((NULL SET1) NIL)
            ((NULL SET2) SET1)
```

8. SS formulates plan to check each element of SET1 to see if it is NOT a member of SET2. Gives up on this plan.

(Continued)

TABLE 5.2
(Continued)

9. Decides to code the relation MINUS (SET1, INTERSECTION(SET1, SET2)). Realizes that MINUS is equivalent to SETDIFF and gives up on this plan.

10. Returns to using INTERSECTION1 code as an analogy. Considers case (MEMBER (CAR SET1) SET2), decides action should be "something with nothing added to it."

11. Refines action to the code (SETDIFF (CDR SET1) SET2). Code is now:

```
(DEFUN SETDIFF (SET1 SET2)
      (COND ((NULL SET1) NIL)
            ((NULL SET2) SET1)
            ((MEMBER (CAR SET1) SET2)
             (SETDIFF (CDR SET1) SET2))
```

12. Considers case in which (CAR SET1) is not a member of SET2. Formulates plan to add (CAR SET1) to the answer for SETDIFF.

13. Decides to look at INTERSECTION1 code again. Notes that 4th action of INTERSECTION maps onto 3rd action of SETDIFF, ponders whether 3rd action of INTERSECTION1 will map onto 4th action of SETDIFF. Decides that the code will work. Final code is:

```
(DEFUN SETDIFF (SET1 SET2)
      (COND ((NULL SET1) NIL)
            ((NULL SET2) SET1)
            ((MEMBER (CAR SET1) SET2)
             (SETDIFF (CDR SET1) SET2))
            (T (CONS (CAR SET1)
                     (SETDIFF (CDR SET1) SET2)))))
```

14. Checks code visually and on the computer.

solution to the problem. This is an attempt to identify the critical steps in that problem-solution episode. Very important to her solution is the example which just precedes this problem in Siklossy's book. It is a definition for set intersection and is given as:

```
(INTERSECTION1 (LAMBDA (SET1 SET2)
      (COND ((NULL SET1) ())
            ((NULL SET2) ())
            ((MEMSET (CAR SET1) SET2)
             (CONS (CAR SET1) (INTERSECTION1 (CDR SET1)
             SET2)))
            (T (INTERSECTION1 (CDR SET1) SET2)))))
```

The basic LISP control construct in this function is the conditional

COND. It evaluates a set of conditional clauses. Each conditional clause consists of a condition test and an action. The COND function executes the action of the first conditional clause it encounters whose condition part is true. There are four clauses here with the condition of the last T which stands for true. So if none of the preceding three evaluate to true, then the last will. The logic of the function is presented in Figure 5.4: If the first set is empty return the empty set; if the second set is empty return the empty set; if the first member of the first set is a member of the second set, return a set consisting of the first member added to the result of a recursive call with the CDR of the first set; otherwise just return the result of the recursive call. Note that INTERSECTION1 is a bit unusual in that there is an unnecessary test for SET2 being empty. Significantly, SS carries this unusual test into her definition of SETDIFF.

Our GRAPES simulation of SS was provided with a representation of INTERSECTION1 at multiple levels of abstraction, a specification of the SETDIFF relation and a somewhat quirky relationship that our subject recognized as she read the problem. This latter relation, which later caused some difficulty for SS, was stated as: The SETDIFF of SET1 and SET2 is SET1 minus the intersection of SET1 and SET2. Our simulation was then

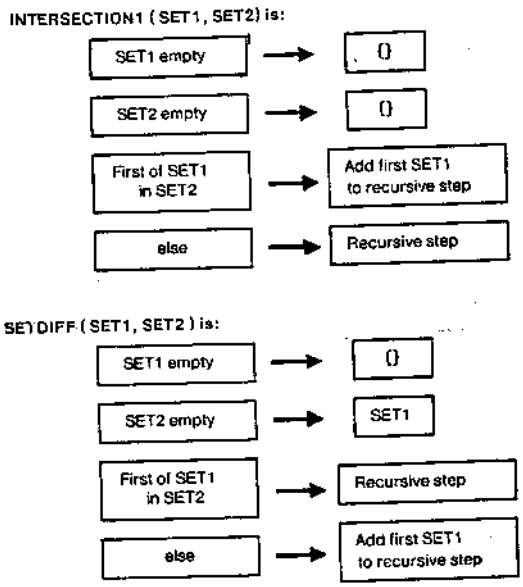

FIGURE 5.4. A schematic view of the logic of the INTERSECTION1 and SETDIFF functions. Arrows point from conditions to actions. Conditions are examined from top to bottom. The first condition that is true triggers its corresponding action to be evaluated.

given the goal of writing SETDIFF. Since it did not have any means of directly solving the problem, the following default rule applied:

IF the goal is to to write a structure
and there is a previous example
THEN set as subgoals:

1. Check the similarity of the example to the current problem.
2. Map the example structure onto the current problem.

This production sets goals to first check the similarity of the specifications of INTERSECTION1 and SETDIFF, and then map the code structure of INTERSECTION1 onto the SETDIFF code. This is an instance of setting AND subgoals. If the first subgoal fails the second subgoal is not attempted.

A set of comparison productions matched features from INTERSECTION1 and SETDIFF. These productions found that both functions take two sets, both perform some membership test, and both are recursive[4]. These productions found a sufficient similarity (the criterion is arbitrarily set in GRAPES) between INTERSECTION1 and SETDIFF, so the first subgoal was satisfied. Next, the following structure-mapping production matched the goal to map INTERSECTION1's conditional structure and the fact that such structures generally have several cases:

IF the goal is to map an example structure
onto the current problem
and that structure has known components
THEN map those components from the example
to the current solution

This production sets up subgoals to map each INTERSECTION1 conditional clause. Our subject gave clear evidence in her protocol of also intending to map the cases of the conditional. The basic rule we assume for mapping each conditional clause is:

IF the goal is to map a conditional clause
THEN map the condition of that clause
and set as subgoals:

[4]Subjects frequently state that they know a solution will be recursive because the problem is in the recursion section of the text.

1. To determine the action in the current case given the mapped condition.
2. To code the new condition-action clause.

There is some evidence that SS had acquired this rule from earlier problems involving non-recursive conditional functions.

Her mapping of the first two clauses of INTERSECTION1 were relatively straightforward. For the first clause she decided the condition would be (NULL SET1) and the action NIL. This is a verbatim copy of the first clause of INTERSECTION1, but from her treatment of later clauses we do not think she was simply copying symbols. The second clause had (NULL SET2) as its condition, just as in INTERSECTION1, but for the action she put SET1 which does not match INTERSECTION1. Her protocol at this point included ". . . if there are no elements in SET2 then all the elements in SET1 will not be in SET2, so if SET2 is the null set the the value of SETDIFF is SET1." So, it seems pretty clear that she was reasoning through the semantics of the condition of the clause and what the implications were for the action of the condition. Our simulation, working with mappings of meaningful abstractions of the conditional clauses of INTERSECTION1, produced the same problem-solving behavior as did SS.

Our GRAPES representation of the meaning of the third clause was "test if the first element should be added to the answer" – which is a rather liberal interpretation of the third condition. Our principal justification for this representational assumption is that it allows us to simulate the behavior of the subject. Both the simulation and SS refined this condition further to the condition "test if the first element is NOT a member of the second set." The problem with this representation of the condition is that neither the simulation nor the subject can directly code this in LISP, and thus the attempt fails. This led both to try to refine the alternate "quirky" definition in working memory: The SETDIFF of SET1 and SET2 is SET1 minus the INTERSECTION of SET1 and SET2. This led both the subject and simulation to set a subgoal of trying to refine "minus." In trying to refine the semantics of minus, both simulation and subject realized that it was equivalent to SETDIFF, the function they were trying to define. Thus the subject had refined the goal of defining SETDIFF into the goal of defining SETDIFF. This is another failure condition, and so simulation and subject attempted to map another representation of the third clause of SETDIFF.

At this point both simulation and subject mapped a very literal translation of the INTERSECTION1 clause: "Test if the first element of SET1 is in SET 2." Thus, the third clause of INTERSECTION1, (MEMSET (CAR SET1) SET2), was used nearly literally as the clause for SETDIFF. Using the specification of SETDIFF, both simulation and subject decided that

the output result should not contain the currently tested first element of SET1, and that SETDIFF should repeat over all elements of SET1. SS decided to simply call SETDIFF on the rest of SET1 in this case. Thus, her action became (SETDIFF (CDR SET1) SET2). The coding of the action was produced in GRAPES by another structure-mapping production:

IF the goal is to code a relation
and a code template exists for relation
THEN map the code template.

This production matched to a template which states: "To repeat a function over the elements of a set, call the function again with (CDR set)." The resulting code matched that of SS.

SS and the simulation then turned to coding the last conditional clause of SETDIFF. Both were still mapping a relatively literal copy of INTERSECTION1 and consequently both copied the T as the condition for the fourth clause of SETDIFF. The semantics of this condition were refined by both SS and the simulation to the case "the first element of SET1 is NOT is SET2." Again working from the semantics of the SETDIFF specification, both GRAPES and SS decided that this condition implies that "the tested element should be added to the result." Our subject floundered at this point because, once again, she did not know how to code the relation she had refined. She inspected the superficial structure of the relationship between SETDIFF as she had written it and INTERSECTION1. She noticed that, while the conditions of clauses 3 and 4 of INTERSECTION1 could be mapped onto the conditions of clauses 3 and 4 of SETDIFF, the action of INTERSECTION1 clause 4 had been mapped onto the action of SETDIFF clause 3. She solved the structural analogy and wrote the action from the third clause in the position of the fourth clause. We gave GRAPES the goal of solving the structural analogy between the last two clauses of the production. Having this goal given, it then set about solving the analogy just as had our subject.

After solving the problem, GRAPES goes into a knowledge compilation phase during which it compiles into single production segments of the problem-solving episode. The details of compilation are discussed in Anderson, Farrell, & Sauers (1984). For present purposes, we are interested in the products of the compilation process. A number of production rules were formed but two important ones that were invoked in the later problem-solving are the following:

C1: IF the goal is to code a relation on two sets SET1 and SET2
and the relation is recursive
THEN code a conditional and set as subgoals to:

1. Refine & code a clause
   to deal with the case when SET1 is NIL.
2. Refine & code a clause
   to deal with the case when SET2 is NIL.
3. Refine & code a clause
   to deal with the case when the first
   element of SET1 is a member of SET2.
4. Refine & code a clause
   to deal with the else case.

C2: IF the goal is to code a relation causing a function
to repeat on the rest of a list
and this occurs in the context of writing a function
that codes the relation on the list
THEN insert a recursive call of a function with the argument
the CDR of the list

The first production compiles the analogy to INTERSECTION1 into a single rule. The second production was learned in the context of coding the third clause of SETDIFF. It is the first recursive rule that the subject has. Note however that its condition does not have a recursive semantics. Rather, SS conceives of the recursive call as causing the function to *repeat*.

It should be noted that C1 and C2 above constitute a fragment of CDR-recursion, and an approximation at that. C1 is the first step to setting up the case structure that is needed in CDR-recursion. C2 provides the recursive control. However, this is a long way from the control structure that we saw in the ideal model's solution to POWERSET.

Conclusions

There are three important conclusions that this example illustrates. First, we see the absolutely critical role that analogy plays in enabling the problem solution. The analogical mapping from INTERSECTION1 to SETDIFF is never a mindless symbol-for-symbol mapping. Rather, it takes advantage of the subjects' knowledge of LISP and a representation of the meaning of what is mapped. Still, we see the subject struggling with exactly what representations to map. The process of problem-solving by analogy is hardly trivial.

The second conclusion is the importance of compilation of this problem-solving episode to future performance. In looking at the solution to the next problem, SUBSET, we will find the compiled productions C1 and C2 ab-

solutely critical. The related third conclusion is about the development of recursive programming skill. It is developing piecemeal and by approximation. With successive problems, we will see the simulation developing a set of productions that handle recursive programming with increasing completeness.

SUBSET

The SUBSET problem is specified in Siklossy as:

> Define a predicate SUBSET of two sets SET1 and SET2. The value of (SUBSET SET1 SET2) is T if SET1 is a subset of SET2; otherwise it is NIL. SET1 is a subset of SET2 if all elements of SET1 are members of SET2.

The schematic protocol of SS solving this problem is given in Table 5.3. SS took under half an hour to solve this problem in contrast to more than one hour that she spent on SETDIFF.

The first production to apply in simulating this protocol was C1 formed in SETDIFF. It sets out the case structure for the conditional. Both subject and simulation consulted the semantics of the subset relation and determined that the answer for the first case, when SET1, should be NIL. SS considered the case when SET2 is NIL but decided to omit this case for reasons that are unstated in her protocol. At the analogous point in our simulation we simply deleted the goal to refine and code the second conditional clause. The third case is one in which the first element of SET1 is a member of SET2. Both subject and simulation, consulting the definition of SUBSET, decided that the program must go on to check whether the other members of SET1 are part of SET2. Setting this goal evoked C2 in the simulation and led it to code the recursive call (SUBSET (CDR SET1) SET2). Finally, the program and subject determined that in the else case, the correct value is NIL.

Thus, we see that the coding episode progresses without the search associated with the first episode. The subject already had a set of rules adequate to handle a small set of CDR-recursive functions, of which SUBSET is one. From this episode one additional relevant production was compiled to reflect the three-clause solution to this problem:

C3: IF the goal is to code a relation on two sets SET1 and SET2
and the relation is recursive
THEN code a conditional and set as subgoals to:

1. Refine & code a clause
to deal with the case when SET1 is NIL.

TABLE 5.3
SS's SUBSET Protocol

1. Reviews SETDIFF solution.
2. Reads SUBSET problem.
3. Writes

```
(DEFUN SUBSET (SET1 SET2)
       (COND
```

4. Considers case (NULL SET1). Decides action should be T. Code is:

```
(DEFUN SUBSET (SET1 SET2)
       (COND ((NULL SET1) T)
```

5. Decides not to worry about the case (NULL SET2).
6. Considers case (MEMBER (CAR SET1) SET2). Decides that the function must go on to check the rest of SET1 to see if it is a SUBSET of SET2. Code becomes:

```
(DEFUN SUBSET (SET1 SET2)
       (COND ((NULL SET1) T)
             ((MEMBER (CAR SET1) SET2)
              (SUBSET (CDR SET1) SET2))
```

7. Considers T case—(CAR SET1) is not a member of SET2. Decides value is just NIL. Code is:

```
(DEFUN SUBSET (SET1 SET2)
       (COND ((NULL SET1) T)
             ((MEMBER (CAR SET1) SET2)
              (SUBSET (CDR SET1) SET2))
             (T NIL)))
```

8. Checks code visually and on the computer.

2. Refine & code a clause to deal with the case when the first element of SET1 is a member of SET2.
3. Refine & code a clause to deal with the else clause.

POWERSET

The POWERSET problem is specified in Siklossy as:

> Define the POWERSET of a set SETT to be a function that calculates the set of all subsets of SETT.

Example: (POWERSET (QUOTE (YALL COME BACK))) has as value the set ((YALL COME BACK)(YALL COME)(YALL BACK)(COME BACK)(YALL)(COME)(BACK)()).
Hint: If a set has N elements, its POWERSET has 2-to-the-n elements.

The schematic protocol for SS's solution is given in Table 5.4. In contrast to the SUBSET problem, SS took three hours to solve the POWERSET problem. We take this as evidence that she was rather short on general recursive programming skill.

Our GRAPES simulation was presented the goal to write POWERSET, and production C3 partially matched in this situation. It was only a partial match because C1 applies to two-argument functions and POWERSET takes only one. The second goal set by C3 involved a test of whether the first element of SETT was a member of the uninstantiated second list. SS actually did momentarily consider a test of the first element when she reached this point in her coding. However, she gave up and turned to the else case. Note that the evidence is that C3, formed after SUBSET, was the one to apply here, and not C1, formed after SETDIFF. If C1 had applied we would also have expected to see mention of the (NULL SET2) test in her protocol—which had been considered and rejected in the SUBSET protocol. Instead, we only saw consideration and rejection of the (MEMBER (CAR SET1) SET2) test from C3. As with any protocol evidence, this is of course only circumstantial.

Not surprisingly, the case that caused SS difficulty was the else clause. She originally failed with efforts to enumerate all members of the answer. Eventually, the experimenter led her through an inspection of the relationship between (POWERSET (SETT) and (POWERSET (CDR SETT). She came to the insight illustrated in Figure 5.3 and clearly articulated that insight. Our simulation was similarly interrupted and given the goal to induce the POWERSET relation by performing the same comparison. It also came up with the critical POWERSET insight.

The experimenter intervened to tell SS to assume she had a magic function, CONST, which calculated Z in Figure 5.3 from Y—that is, it adds the first member of the input list to Y. The effect of this intervention in the simulation is to have GRAPES postpone the coding of CONST and complete the coding of POWERSET—which corresponds with the flow of control that we see in our subject.

The subject, in trying to code the recursive step, at first just wrote (CONST (CAR SETT) (POWERSET (CDR SETT))). Only when the experimenter pointed out that this would not include the subsets without (CAR SETT) did she change her code to (UNION (POWERSET (CDR SETT))(CONST (CAR SETT)(POWERSET (CDR SETT)))). We were able to simulate this in GRAPES by letting the insight $X = Y + Z$ in Figure

TABLE 5.4
SS's POWERSET Protocol

1. Reads problem.
2. Writes (DEFUN POWERSET (SETT).
3. Uses the example POWERSET ((YALL COME BACK)) to define the answer as the set of all sets of one element, all combinations of two elements, the whole list and the null list.
4. Considers case (NULL SETT). Decides that result will be NIL.

```
(DEFUN POWERSET (SETT)
       (COND ((NULL SETT) ()))
```

5. Considers T case. Decides result will be the set containing SETT, the null list, the (CAR SETT) and the (POWERSET (CDR SETT)). Omits the null list since it is the action of the first clause. Code is:

```
(DEFUN POWERSET (SETT)
       (COND ((NULL SETT) ())
             (T (SETT (CAR SETT)
                      (POWERSET (CDR SETT))))))
```

6. Realizes that function will not obtain all combinations of two elements.
7. Decides that (CDR SETT) is one combination. Begins to rewrite final clause as

```
(T (SETT (CAR SETT) (CDR SETT)
```

8. Realizes that other combinations of two elements will not be easily obtained.
9. Attempts to plan a way of getting the combinations of two elements by taking the UNION of each pairwise combination of set elements. Has difficulty generalizing beyond example.
10. Decides that for N-element list, the result contains the UNION of the (CAR SETT) with all successive CDRs of the list and all CARs of the CDRs, and with all of the combinations of each. Still cannot plan a way of getting the combinations.
11. Tutor gives a hint: Assuming that POWERSET works for a set of a certain size, how would it work for the next larger size? SS doesn't use hint.
12. Tutor asks SS to write out the answer for the POWERSET of the (CDR SETT).
13. Tutor instructs SS to compare the POWERSET (CDR SETT) with the POWERSET (SETT).
14. SS realizes that POWERSET (SETT) is the POWERSET (CDR SETT) plus all the sets containing the (CAR SETT).
15. Tutor instructs SS to make an explicit comparison between the sets in the POWERSET (CDR SETT) and all the sets containing the (CAR SETT).
16. SS realizes that the sets containing the (CAR SETT) can be obtained by CONSing the (CAR SETT) into each set in the POWERSET (CDR SETT).

(Continued)

TABLE 5.4
(Continued)

17. SS wonders how she is going to CONS the (CAR SETT) into each element in a list of lists. Tutor tells SS to assume that she has a subfunction that will do this.

18. At the tutor's prompting SS names the subfunction CONST, specifies that it will take two arguments (an atom and a list of lists), and returns a list of lists in which each element list contains the atom argument.

19. Decides that the value of POWERSET in the T case will be

```
(CONST (CAR SETT) (POWERSET (CDR SETT)))
```

20. Begins to work through examples. Tutor points out that the POWERSET of () is not just () but (()).

21. Tutor points out that the current function will not return the sets in POWERSET that don't contain the (CAR SETT).

22. SS decides that the T case should result in the union of the POWERSET (CDR SETT) with the (CONST (CAR SETT) (POWERSET (CDR SETT))). Code is:

```
(DEFUN POWERSET (SETT)
       (COND ((NULL SETT) (()) )
             (T (UNION (POWERSET (CDR SETT))
                       (CONST (CAR SETT)
                              (POWERSET (CDR SET)))))))
```

23. Tries out examples of length one and two. Hand solutions work.

24. Realizes that CONST is undefined as POWERSET is being typed in. Begins definition of CONST.

25. Formulates plan to CONS the first argument to CONST into each element of the second argument. Writes:

```
(DEFUN CONST (SIL LIS)
```

26. Decides that function does not need any conditional statements.

27. Decides that the value of CONST will be the UNION of the (CONS SIL (CAR LIS)) with (CONST (CDR LIS)). Code is:

```
(DEFUN CONST (SIL LIS)
   (UNION (CONS SIL (CAR LIS)) (CONST (CDR LIS))
```

28. While typing in function, realizes that recursive call to CONST is missing an argument. Replaces (CONST (CDR LIS)) with (CONST SIL (CDR LIS)).

29. Works through examples by hand. Types in CONST.

30. Tries out example on computer. Gets an error message indicating an infinite recursion taking place.

(Continued)

TABLE 5.4
(Continued)

31. Traces CONST. Realizes that function does not stop when second argument is empty.

32. Decides to use conditional statements. Considers case of (NULL LIS) and decides that result will be (CONS (SIL ())). Then decides that this result is equal to (SIL). Code is:

```
(DEFUN CONST (SIL LIS)
       (COND ((NULL LIS) (SIL))
```

33. Considers T case and realizes that the result will be returned by the previously written code.

```
(DEFUN CONST (SIL LIS)
       (COND ((NULL LIS) (SIL))
             (T (UNION (CONS SIL (CAR LIS))
                       (CONST SIL (CDR LIS))))))
```

34. Tries out function on computer. Gets error message stating that SIL is an undefined function. Changes (SIL) back to (CONS SIL ()).

35. Tries out function again. Answer is not the right list of lists.

36. Traces UNION and CONS and tries function again. Cannot understand why the function is not returning the desired result.

37. Tutor intervenes and prods SS into reconsidering the first condition—(NULL LIS). Convinces SS that action should be ().

```
(DEFUN CONST (SIL LIS)
       (COND ((NULL LIS) ())
             (T (UNION (CONS SIL (CAR LIS))
                       (CONST SIL (CDR LIS))))))
```

38. Tries out function on the computer. Answer is still not in right form.

39. Tutor prods SS into realizing that UNION should be CONS. SS has difficulty because she believes that CONS takes an atom as its first argument. Code is:

```
(DEFUN CONST (SIL LIS)
       (COND ((NULL LIS) ())
             (T (CONS (CONS SIL (CAR LIS))
                      (CONST SIL (CDR LIS))))))
```

40. Tries out CONST on computer and it works.

41. Tries out POWERSET on computer and gets error message saying that NIL is an undefined function.

(Continued)

TABLE 5.4
(Continued)

42. Quotes the list of () in the first condition of POWERSET. Code is:

```
(DEFUN POWERSET (SETT)
       (COND ((NULL SETT) '(()) )
             (T (UNION (POWERSET (CDR SETT))
                       (CONST (CAR SETT)
                              (POWERSET (CDR SETT)))))))
```

43. Tries out POWERSET again and function works.

5.3 degrade to $X = Z$. This was produced simply by a loss of features in the working-memory representation of the insight. The experimenter's intervention was simulated by replacing the full $X = Y + Z$ into working memory.

The simulation formed a production rule to summarize the two-clause solution to this problem:

C4: IF the goal is to code a relation on one list SET1
and the relation is recursive
THEN code a conditional and set as subgoals to:
1. Refine & code a clause
to deal with the case when SET1 is NIL.
2. Refine & code a clause to deal with the else case.

This is very much like the production for setting up CDR-recursion in the ideal model, but we will see that there still is a critical issue of having it evoked in the right situations.

For purposes of our analysis of recursion, the important observation is that the recursive calls to POWERSET were not generated by the production C2 compiled in the context of doing SETDIFF. Rather the code was generated by structure-mapping over the representation built up in the formation of the POWERSET insight.

Another production learned in the POWERSET episode has an impact on future performance:

C5: IF the goal is to add result1 and result2 to form a list
THEN write (UNION result1 result2)

This production is a compilation of the problem-solving involved in coding the final action of POWERSET, and it applies when there is a goal to com-

bine the results of two function calls into a list. As we will see in the coding of CONST, this production is overly general. Often in synthesizing recursive function one must consider more specific details concerning the results of function calls and the form of the list being constructed.

CONST

After defining POWERSET, SS went to the solution of CONST. Here she immediately saw an iterative plan for performing CONST: "I've got to take successive CARs of the second argument and each time CONS the CAR of SETT into that set and the final value of CONST will just be the set of all those CONSes." SS refined this iterative conception further into a plan to perform a CONS operation with the first element of the input list and to make the function CONST repeat on the rest of the list. She then wrote two separate segments of code to instantiate this plan. First, she wrote (CONS SIL (CAR LIS)), which satisfies the first component of her plan. Next, she wrote the recursive call (CONST SIL (CDR LIS)) which satisfies the plan to make CONST repeat. Our simulation, which was provided with a representation of SS's plan for iterative solution, wrote the same code segments. The simulation first applied a production coding CONS and its arguments (learned in previous sessions). It then applied the production C2 learned in SETDIFF to code the recursive call. We should point out that the semantics of production C2 imply "something that causes repetition to happen," and this appears to be the meaning SS associated with the recursive call that she coded.

At this point in her protocol, SS decided to combine the results produced by her two code segements and she used UNION to perform this operation. Our simulation also selected UNION to combine function results by applying the newly learned production C5. The code produced by SS and the simulation was:

```
(DEFUN CONST (SIL LIS)
    (UNION (CONS SIL (CAR LIS))(CONST SIL (CDR LIS))))
```

This solution lacks a feature critical to all recursive programs. It does not specify a terminating condition for the recursive process. Without a terminating condition the above function leads to an infinite recursion. It is interesting that the subject notes that her definition of CONST differs from previous recent function definitions in that it does not have a COND structure, but is adamant that such a structure is not needed.

Note that the application of C4 in this case would have led to the correct control structure. We have assumed that the reason why C4 did not apply can be traced to the way SS identified recursive programs from problem

specifications. We believe that she characterized SETDIFF, SUBSET, and POWERSET as recursive functions because they were implicitly identified as such in the textbook's problem section on recursion. CONST, however, was simply identified by SS's tutor as a helper function to POWERSET. It has been noted in other studies (e.g., Larkin, McDermott, Simon, & Simon, 1980) that novices often have the skills required to solve a problem, but are inept at characterizing the problem features in an appropriate manner to evoke those skills. Thus it is not enough to just have the right operators; they have to be invoked in the right circumstances.

SS and our simulation tried out the above version of CONST, and an error message was generated indicating infinite recursion. By using LISP facilities which trace out the recursive operation of a function, SS's tutor pointed out that the function did not stop when the input list LIS was empty. This led SS and the simulation back to the goal of redefining the function. This time SS characterized the function as having two cases: when the input set is empty, and in the else case. She and the simulation then each set goals to determine and code the conditional clause for the (NULL LIS) case, and to simply copy the already written recursive code into the action for the else case.

The remainder of the protocol was devoted to working through her misconceptions about the code. One of SS's misconceptions concerned the use of UNION to combine the results of (CONS SIL (CAR LIS)) with the recursive call to CONST. The function UNION combines the elements from two lists into a single list. However the appropriate action in this case is to insert the result of (CONS SIL (CAR LIS)) into the result of the recursive call. To illustrate this more clearly, the UNION of the lists (A B) and (C) is (A B C). CONSing the same lists yields ((A B) C). This difference often confuses students (Anderson & Jeffries, 1985). When SS ran the version of CONST which contained UNION, she obtained an output list that differed from the desired result. Her tutor assisted her by pointing out that the terminating recursive case would return the empty set NIL, and that the most appropriate plan was to insert elements returned by (CONS SIL (CAR LIS)) into the result of recursive calls. Given this information, both SS and our simulation substituted CONS for UNION to produce the correct code:

```
(DEFUN CONST (SIL LIS)
       (COND ((NULL LIS) ()))
             (T (CONS (CONS SIL (CAR LIS))
                      (CONST SIL (CDR LIS))))))
```

The important production compiled from this final episode of debugging is:

C6: IF the goal is to code a relation adding result1 and result2
and the code occurs in the context of writing a function
and the result2 is produced by the function repeating
an operation on a list
and the function returns NIL when given NIL as an argument
THEN write (CONS result1 result2)

Thus, the simulation has learned a rule which uses CONS to add data elements to a list constructed by CDR-recursion. This is a fairly standard code pattern seen in CDR-recursive functions (see Soloway & Woolf, 1980). Thus we see the learning of another important component of recursive programming.

SUMMARY

This POWERSET episode gives further evidence for the gradual accrual of a set of productions that will adequately apply CDR-recursion. We see the formulation of a basic CDR-recursion procedure involving two cases – a terminating case and a recursive case. The latter production is highly similar to the expert's general recursion production P1. We should note however that the productions learned in POWERSET do not generalize to the full space of CDR-recursive functions. Although SS got noticably better at coding later recursion problems, there were still many unfamiliar patterns and instances that forced her to fall back on general problem-solving skills.

POWERSET also provides another example of the importance of analogical processes in LISP problem-solving and learning. This time, a complex interrelation of worked-out concrete examples provides the source of the mapping. However, because the goals regarding the exact comparisons to be made were generated by an external source (SS's tutor), they are not compiled into any productions which specify a more general means of characterizing recursive relations.

GENERAL CONCLUSIONS

We see a fair bit of evidence for our analysis of the difficulty of recursion in these protocols. Specifically:

1. Recursion is difficult because it is unfamiliar. The subject is forced to try to work by analogy from examples.

2. Recursion is difficult because of imprecise instruction. The subject was really forced to learn almost everything from her mistakes. An interesting issue is what would happened if the text had provided her directly with the rules that she instead had to induce. These rules would have been in English, rather than in production form, and so there would still be some learning in converting them; but we would predict more rapid and less painful and error-ridden learning.

3. Recursion is difficult because of interference from other methods of solution. SS initially characterized recursion as a form of iteration. This eventually led to her failure to correctly code CONST on her first attempt.

4. Recursion is difficult because it is complex. The subject is still learning the patterns that define the application of CDR-recursion to various problems. It is also worth noting that she is only learning about CDR-recursion. In other research we have done, we have shown that there is little transfer from CDR-recursion to other types of recursive functions.

5. Recursion is difficult because it is exacerbated by the difficulty of LISP. We saw numerous examples, particularly in POWERSET, in which the subject's difficulty with the nonrecursive aspects of LISP programming complicated the learning of recursion.

It is interesting that nowhere in these protocols from SS do we find her trying to perform a recursive mental operation. This is further evidence for our claim that the difficulty of recursive programming does not directly arise from the difficulty of performing recursive mental operations.

Finally, we would like to note that the success of GRAPES in simulating the protocols of SS is further evidence for our theory of how LISP functions are typically programmed. Specifically, the basic flow of control is top-down problem decomposition. Initial problems are solved by structural analogy to worked-out examples. Subjects summarize these solutions by new compiled operators. These operators are keys to the solution of later problems.

ACKNOWLEDGMENT

This research was supported by the Personnel and Training Research Programs, Psychological Services Division, Office of Naval Research, under Contract No.: N00014-81-C-0335.

REFERENCES

Anderson, J. R. (1983). *The Architecture of Cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. R. (1976). *Language, Memory, and Thought.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, J. R., Farrell, R., & Sauers, R. (1982) *Learning to plan in LISP* (Tech. Rep. ONR-82-2). Pittsburgh, PA.: Carnegie-Mellon University.

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

Anderson J. R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human Computer Interaction, 1,* 107–131.

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

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

Dresher, B. E., & Hornstein, N. (1976). On some supposed contributions of artificial intelligence to the scientific study of language. *Cognition, 4,* 321–398.

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

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

Rosenbloom, P. S., & Newell, A. (1983). The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop* (pp. 183–197). University of Illinois at Urbana-Champaign.

Sauers, R., & Farrell, R. (1982). *GRAPES user's manual* (Tech. Rep. ONR-82-3). Pittsburgh, PA.: Carnegie-Mellon University.

Siklossy, L. (1976). *Let's talk LISP.* Englewood Cliffs, New Jersey: Prentice-Hall.

Soloway, E. L., & Woolf, B. (1980). *From problems to programs via plans: The content and structure of knowledge for introductory LISP programming* (Tech. Rep. Coins 80–19). Amherst: University of Massachusetts.