CANADIAN JOURNAL OF PSYCHOLOGY, 1985, 39(2), 240-272

The Role of Learning from Examples in the Acquisition of Recursive Programming Skills*

Peter L. Pirolli and John R. Anderson
Carnegie-Mellon University

ABSTRACT We present an analysis and simulation model of verbal protocols of two college students (SS and AD) and one 8-year-old child (JP) learning to program recursive functions. The model is formalized as a production system capable of acquiring new production rules based on problem-solving experience. The model and protocols suggest: (a) that problem solving by analogy to worked-out examples is frequent in initial attempts by novices to write recursive functions; (b) different representations of examples are used to guide problem solving by analogy; and (c) performance on later problems reflects the particular representations used in problem solving by analogy on earlier problems. The protocols and simulations suggest that learning is facilitated by using abstract representations of the structure of recursion examples to guide initial coding attempts.

RÉSUMÉ Nous présentons une analyse et un modèle de simulation de protocoles verbaux de deux étudiants de niveau collégial (Sujet SS et AD) et d'un enfant de huit ans (Sujet JP) qui apprennent à programmer des fonctions récursives. Le modèle est élaboré en termes de système de production capable d'acquérir de nouvelles règles de production à partir d'expérience de solution de problèmes. Le modèle et les protocoles révèlent que (a) Lors des premiers essais d'écriture de fonctions récursives que font des novices, la solution de problème par analogie est fréquemment utilisée pour résoudre les exemples, (b) différentes représentations des exemples sont utilisées pour diriger la solution de problème par analogie, et (c) la performance aux problèmes ultérieurs reflète les représentations particulières utilisées dans la résolution par analogie des problèmes antérieurs. Les protocoles et les simulations montrent que l'apprentissage est facilité par l'usage de représentations abstraites de la structure des exemples de récursion lors des tentatives d'encodage initiales.

Previous research on programming (e.g. Kahney, 1982; Soloway, Bonar, & Ehrlich, 1983) has tended to focus on characterizing programming behaviour at various levels of expertise. These studies provide snapshots of the development of programming skill. The research reported here attempts to get a detailed trace of the early transitions in problem-solving skill by studying the verbal protocols of subjects solving sequences of programming problems involving recursion. We have developed a theory of the problem-solving behaviour and learning evident in these protocols and will report simulations based on that theory. In the next

---

* This research was supported by the Personnel and Training Research Programs, Psychological Services Division, Office of Naval Research, under Contract No. N00014-84-K-0064 to John Anderson. The simulations reported here and the preparations of this manuscript were in part supported by an IBM Research Fellowship to Peter Pirolli. Portions of this paper have been presented at the Expertise Conference, University of Pittsburg, 1983, and the Sixth Annual Conference of the Cognitive Science Society, University of Colorado, Boulder. 1984. Address reprint requests to Peter L. Pirolli, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A.

section we provide a brief introduction to recursive functions. We then present an overview of the GRAPES production system which we have used to model the problem-solving behaviour and learning observed in individual subjects (see also Anderson, Farrell, & Sauers, 1984). In subsequent sections, we first present a model of expert performance in programming recursive functions and then the simulations and protocols of three students learning to program recursive functions. In discussing the simulations and protocols of the novices, we will focus on two things: (a) the role of problem solving by analogy to examples in early stages of learning to program recursion, and (b) the acquisition of new problem-solving operators.

RECURSIVE PROGRAMMING

Recursive functions are functions that are defined in terms of themselves. A standard example of recursion in mathematics is the factorial function, $f(n) = n \times f(n - 1)$, for $n > 0$ (called the *recursive case* because it involves the *recursive call* $f(n - 1)$), and $f(0) = 1$ (called the *base* or *terminating* case). The computation of such a function is carried out by suspending the calculation of $n \times f(n - 1)$ until $f(n - 1)$ is carried out, which in turn requires that $(n - 1) \times f(n - 2)$ be suspended until $f(n - 2)$ is carried out, and so on, until $f(0)$ is reached. Despite the formal simplicity and elegance of recursive functions, we (and other teachers we have talked to) have observed that many students have great difficulty learning to code such functions. This difficulty seems to be due, in large part, to the unfamiliarity of recursion to most students (Anderson, Pirolli, & Farrell, in press). While there are many everyday conceptual analogs to other programming constructs such as iteration (e.g., cashiers processing customers), there are few, if any, simple everyday conceptual analogs to recursion.

This unfamiliarity may be caused indirectly by the general difficulty people have with executing recursive mental procedures. That is, it seems that the human mind cannot suspend one process, perform a recursive calculation, and return to the original suspended process. For example, it has been pointed out (Anderson, 1976; Dresher & Hornstein, 1976; Miller, 1967) that people find it difficult to understand recursive linguistic structures (e.g., *The fact that the shepard said that the farmer has given the book to the child to the police was to be expected*). People have had preprogramming experience with following everyday procedures (e.g., recipes) and, more importantly, specifying such procedures to others. However, these procedures were never recursive because they typically had to run in human heads. Therefore, recursive programming is most people's first experience with specifying a recursive procedure. Interestingly, we have observed only 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.

How, then, do students learn the unfamiliar procedure of generating recursive programs? Our hypothesis is that the primary means available to students is learning from examples. By this we mean two things. First, students solve initial problems by modifying the solutions of examples they are given. Second, learning mechanisms summarize solutions to these initial problems into new problem-solving operators which can apply to future problems. One of our basic assumptions is that students fall back on analogical problem solving only in novel and difficult situations (cf. Gick & Holyoak, 1980, 1983). As students acquire problem-solving operators for programming recursion, they should rely less on using previously worked out examples as analogies.

Admittedly, students could learn about recursion without problem solving by analogy to specific example solutions. For example, Anzai and Uesato (1982) report an experiment in which students induced recursive rules for computing factorials from example patterns of factorial results (e.g., $f(3) = 3 \times f(2), f(2) = 2 \times f(1)$, etc.). Soloway and Woolf (1980) describe a method for teaching recursion involving the presentation of abstract code templates. However, we hold a rather broad view of problem solving by analogy. We assume that problem solving by analogy involves using declarative representations of rules, guidelines, and abstract or concrete example solutions presented in instructions. Problem-solving operators use these representations as templates to guide novel solutions. In contrast to other proposals for problem solving by analogy (Carbonell, 1983; Gick & Holyoak, 1980, 1983), our research has focussed on the use of examples in analogy and the impact of such analogies on future performance rather than the selection of the examples in the first place (see also Ross, 1984).

Our learning model, outlined in the next section, also makes an interesting prediction concerning the relation between problem solving by analogy and the types of operators learned from such problem solving: The particular representations used in solving a target problem by analogy to examples will determine the generality of the operators learned from such problem solving. An example can be encoded in many ways, some ways richer than others and some ways more correct than others. For example, a student solving a programming problem could use an insightful abstract characterization of an example function in analogy or the student could use the literal code symbols on the page. Our model predicts that the operators learned in the former case will have a more general range of applicability to future problems than operators learned by basically copying literal code.

The GRAPES *Production System*

Our theory of learning to program recursion was developed in the context of the GRAPES production system. GRAPES (see Sauer & Farrell, 1982, for details) was designed to emulate certain aspects of the ACT* theory of cognitive architecture (Anderson, 1983). GRAPES consists of four memories: a working memory and a long term memory that store problem specifications and other relevant declarative knowledge, a goal memory containing programming goals, and a production memory containing condition-action production rules (productions). Production rules in GRAPES have *conditions* that specify conjunctions of programming goals, problem specifications and/or other declarative facts. The *actions* of productions specify refinements to programming specifications, the writing of code, or the setting of new subgoals. An example of a production that a novice may have is:[1]

```
     IF the goal is to write a function
        and there is a previous example
   THEN set as subgoals
        1. to compare the example to the function
        2. map the example's solution onto the current problem.     (P1)
```

Such a production would apply if there was an example readily available during an attempt to write a new function. Production P1 sets subgoals to determine if the example is relevant and to use the example solution as a template for the current programming problem. A production that an expert might have is:

```
     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.     (P2)
```

Such a production applies in a very specific programming context in the LISP[2] language. In general, the transition from novice to expert in programming involves the development of many productions that apply in a variety of specific situations.

GRAPES operates by repeatedly selecting productions whose conditions are satisfied (matched) and executing the actions of those selected productions. GRAPES differs from many other production system architectures (e.g., Anderson, 1976; Newell, 1973) in its special treatment of goals. At any point in time there is a single active goal, and only productions relevant to that goal may apply. In this feature, GRAPES is like some other recent cognitive theories (Anderson, 1983; Brown & VanLehn, 1980; Card, Moran, & Newell, 1983; Rosenbloom & Newell, 1983).

Another distinguishing feature of GRAPES is its ability to model learning by the mechanisms of *knowledge compilation* which create new productions during the course of problem solving (Anderson, 1983; Anderson et al., 1984; Neves & Anderson, 1981). Knowledge compilation in GRAPES consists of two mechanisms: *composition* and *proceduralization*. Each new production produced by composition merges the conditions and actions of several productions that executed during a problem solving episode. Proceduralization takes productions that match long term memory information or information from an example and creates new productions whose conditions no longer specify such information.

The notion of deleting references to example information is an extension of previous versions of preceduralization in GRAPES and ACT*. In general, if an example solution is mapped onto a new problem solution by a sequence of productions, then knowledge compilation will form new productions that specify the new solution without referencing example information. If the representation of the example solution used in this mapping is flawed, then the new productions will also be flawed. In our later discussion of novice programmers, we will attempt to show how the representations of examples mapped onto new programming solutions determine the generality and correctness of the productions acquired in knowledge compilation.

---

[1] Throughout this paper we will present productions in an English-like manner. The specifications of these rules in GRAPES and the GRAPES user manual (Sauers & Farrell, 1982) are available on written request.

[2] LISP is a *LISt Processing* language. Programs written in LISP take the form of functions that compute a particular input-output relation.

SIMULATIONS AND PROTOCOLS

Our protocol database includes three 30-hour protocols of novices learning the LISP language from standard programming texts, two 4-hour protocols of children learning recursion in the LOGO[3] language, four 6-hour protocols of novices learning the language designed for experimentation (SIMPLE, Shrager & Pirolli, 1983), and protocols of expert LISP programmers. The basic procedure used in gathering protocols from the novices is the same in all cases (although the instructional material may differ). A novice subject receives instruction on writing recursive functions and is asked to think aloud while writing programs that meet some program specifications. An experimenter is present to tape record the subject's verbal protocol and to prompt the subject to continue thinking aloud during prolonged silences. The experimenter is instructed not to intervene with hints, suggestions, or clarifications unless the subject is clearly lost. In addition to the verbal protocols, we record subjects' terminal interactions and scratch notes.

Our protocol analysis consists of creating *schematic protocols* which omit the irrelevant digressions present in the raw protocols (for a comparison of raw and schematic protocols see Anderson et al., 1984). These are our intuitive characterizations of the features of the protocol relevant to programming. Our simulation model addresses problem-solving behaviour at least at the grain of analysis (and often at a finer grain) in the schematic protocols presented in this paper.

Our methodology involves simulating a subject's solution to a particular problem, allowing knowledge compilation to add new production rules to the model, and then attempting the next problem faced by the subject. In simulating a subject's first problem, the sytem is initialized with a set of productions which are assumed to characterize the subject's skill prior to learning about recursive functions. New productions are added only by means of the knowledge compilation process operating on the problem solution. Thus, the compilation process predicts systematic changes in the way problems are solved. The model is validated to the extent that such predictions are corroborated by the protocol data (for discussion of the prospect of using intelligent tutoring systems as a validation technology see Anderson, 1984). We will also report convergent evidence from

---

[3] LOGO is a language designed primarily to manipulate simple screen graphics, although it does have capabilities for performing numeric and list manipulations.

more traditional (i.e., group) measures whenever relevant.

There are three conditions under which we intervene with the ongoing problem solving of our simulations. First, if a student's problem solving is altered by hints or suggestions from an experimenter, then we similarly interrupt our simulation and provide it with new goals and/or information. Second, there are cases in which several production instantiations have equal applicability (actually, this occurs rarely). In these cases, GRAPES will randomly choose one applicable production for execution. In such circumstances we intervene and force GRAPES to select the applicable rule that mimics the protocol being simulated. Third, there are cases in which students forget particular goals or problem specifications. We assume that such forgetting is a probabilistic process that simply cannot be made deterministic in the simulation of a particular protocol. When protocol evidence indicates that a subject has forgotten some information or goal, we intervene with the simulation and remove the corresponding information or goal from the model.

In the next section we present a simulation of an expert to indicate target performance in the domain of writing recursive functions. We then discuss the protocol and simulation of a novice learning to program recursion from a standard LISP textbook. In discussing this protocol and simulation, we seek to highlight the general learning and performance phenomena observed in students learning from standard instruction. We then focus on the role of analogy and conceptual models of recursion in early skill acquisition by presenting the protocols and simulations of two other novices.

Simulation of an Expert

There are two basic points we wish to make about the simulation of expert performance presented in this section. First, because the simulation model has domain-specific problem-solving operators (productions), the simulation makes no use of analogies to other functions. Second, the simulation used an instance of a general strategy that has a wide range of applicability to recursive programs. Figure 1 presents a *hierarchical goal tree* for this general strategy. Each label in Figure 1 represents a programming goal. Arrows show the decomposition of goals into subgoals. The strategy depicted in Figure 1 involves (a) refining the semantics and coding the terminating cases of the function, and (b) refining the semantics and coding the recursive cases. This latter step involves a set of subgoals for (a) characterizing the result of a recursive call (e.g., $f(n-1)$ for the factorial function), (b) characterizing the result of the function (e.g., $f(n)$) and (c) determining the relationship between (a) and (b) (e.g., $f(n) = n \times f(n-1)$).

Figure 2 illustrates the POWERSET problem as we present it to subjects who must solve the problem in LISP. The subject is told that a list of atoms (e.g., (A B C)) encodes a set of elements, and 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. Each subject is given an example of the POWERSET of the three-element list. All subjects come up with basically the same solution, given in Table 1. The basic LISP control construct in this function is the conditional COND, which evaluates a set of conditional clauses. Each conditional clause (e.g., ((NULL L) (LIST NIL))) consists of a condition test (e.g., (NULL L), which tests if the argument L is an empty list) and an action (e.g. (LIST NIL), which returns a list containing the empty list). The COND function executes the action of the first conditional clause it encounters whose condition part is true. There are two clauses in the definition of POWERSET. The condition of the second clause is T, which stands for true. Because the second condition is always true, the second action will always be executed whenever the first condition is not true. The definition in Table 1 involves a secondary function ADDTO which takes as arguments an atom and a list of lists. ADDTO returns a list of lists composed by adding the atom to each list in the original list of lists argument; for example, (ADDTO 'A '((B C) (B) (C) ())) = ((A B C) (A B) (A C) (A)). We have not provided a definition for ADDTO because the definition varies with the level of expertise of the programmer.[4] The basic structure of the POWERSET definition,

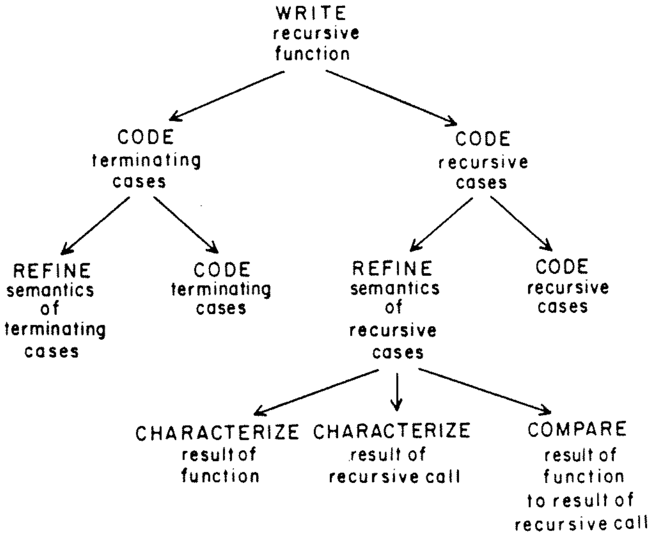


Figure 1. The hierachical goal tree for a general strategy for a large subset of recursive functions. Each box is a programming goal. Arrows point from goals to subgoals. Each subgoal of a goal must be satisfied for a goal to be satisfied.

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

Figure 2. The specification of the POWERSET problem. The POWERSET of a set represented as a list, is a list containing all possible subsets of the set.

---

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

TABLE 1

The LISP Function POWERSET

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

however, does not change with expertise, although there is easily greater than a 10:1 ratio in the time taken to generate the code by novices versus experts.

Figure 3 presents the hierarchical goal tree for the solution of the POWERSET problem produced by GRAPES which seems to capture the expert's solution. The code presented in Table 1 is the product of carrying out the plan specified in Figure 1. With the first goal set to code the function POWERSET (the top-most goal in Fig. 3), the first GRAPES production to apply is:

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. (P3)

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 P3 sets up the plan to call (POWERSET (CDR L)) within the definition of (POWERSET L). 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. Note that this expert production P3 is a relatively specialized variant of the general strategy for writing recursive functions outlined in Figure 1. It is concerned only with a special case of recursion and it applies only 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 conditions. Because of this specificity principle, P3 would not apply in many situations where there was a one-level argument. For instance, if the goal was to write a function that returned a list of the first and second element in a list argument, other more special case productions would apply.

Activiting goals in a left-to-right, depth-first manner, GRAPES turns to coding the terminating case. 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

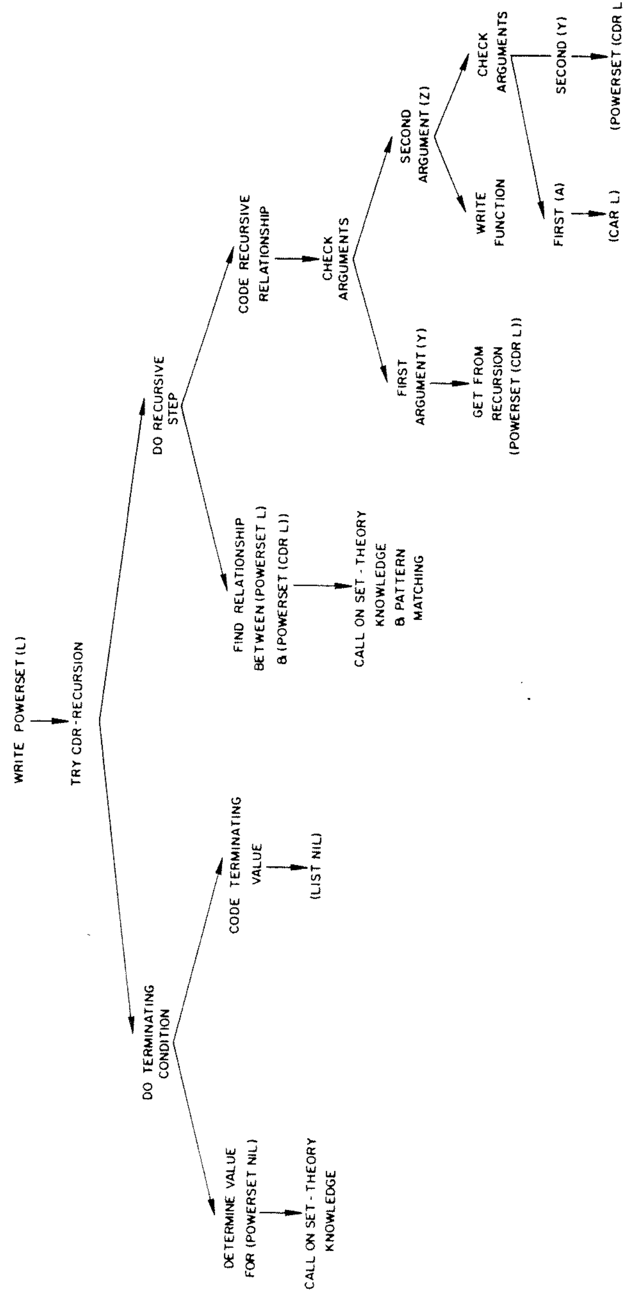


Figure 3. The hierarchical goal tree produced by a GRAPES simulation of an expert solving the POWERSET problem.

```
L= (A B C)

X= (POWERSET L)          Y= (POWERSET (CDR L)
  = ((A B C)               = ((B C)
    (A B)                    (B)
    (A C)                    (C)
    (A)                      ())
    (B C)
    (B)
    (C)
    ())
X= Y+Z          WHERE    Z= ((A B C)
                             (A B)
                             (A C)
                             (A))

Z IS FORMED FROM Y BY ADDING A TO EACH
MEMBER OF Y.
```

Figure 4. A representation of the critical insight that underlies the refinement of code for the recursive case in POWERSET.

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 the application of these problem-solving productions. The important feature to note is that coding the terminating case is extremely straightforward in a case like this where the answer can be derived from a semantically correct definition of the function.

After coding the terminating case, GRAPES turns to coding the recursive case. 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 4 illustrates what is involved in the case of POWERSET. In Figure 4 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 can be obtained from Y by adding the first element of the list (A in Fig. 4) to each member of Y. Thus, X = Y + Z.

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 list 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 calculate Z directly, 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. It is important to note that writing code for the recursive case does not require an analysis of the recursive behaviour of the function.

There are a number of standard recursive paradigms in LISP in addition to CDR-recursion (Soloway & Woolf, 1980), but most can be solved by some variant of the general strategy outlined in Figure 1. It is the case that not all LISP recursive functions are so straightforward. One source of complexity is that the function may not have as precise a semantics as POWERSET, and part of the problem solving is to settle on those semantics. Another reason for complexity concerns recursion in nonstandard paradigms. For instance, production P2 deals with one such nonstandard recursion.

One important observation is that even standard recursion, as in POWERSET, causes novices great difficulty. The students we have looked at spent from just under 2 to over 4 hours coming upon the solution to POWERSET that an expert can produce in under 10 minutes. This difficulty for novices is all the more striking when we realize how few lines of code are involved. In examining the GRAPES simulation of an expert, it is apparent that the majority of the POWERSET solution is produced by domain-specific productions. Without such domain-specific productions, novices must rely on more general problem-solving techniques such as problem solving by analogy to examples.

Protocols and Simulations of Novices

In this section we discuss the protocols and GRAPES simulations of three novices learning to code recursive functions. These three novice protocols are representative of programming students, in that novices seem to rely heavily on analogies to examples in the early stages of programming recursion. For example, we recently conducted an experiment in which we presented subjects with a recursive program example during instruction on recursion and allowed the example to be accessed on subjects' computer terminals while programming. We found that 18 out of 19 subjects used the example while programming their first recursive function and the remaining subject showed clear intrusions from having seen the example earlier. On this first recursive program attempt, subjects spent 34.3% of their time looking at the example.

The three protocols presented here provide an indication of the varying efficacy of problem solving by analogy. The first novice, SS, was a college student, and she is traced through her first three recursion problems in LISP to point out the general phenomena involved in learning to program recursion. Protocols from the second and third subjects are presented along with simulations to highlight the relationship between analogy and generalization. The second novice, JP, was an 8-year-old learning recursion in LOGO. The third novice, AD, was another college student, and she learned recursion in the SIMPLE language. Our goal in contrasting subjects SS, JP, and AD, is to show how the particular analogies performed by students on early problems affects their performance on later problems.

Subject SS: The first recursive function written by SS was SETDIFF which takes two list arguments and returns all the members in the first list not in the second list. The second was SUBSET, a function of two list arguments which tests if all the elements of the first list are 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 *Let's Talk LISP* (Siklossy, 1976), in the manner typical of LISP pedagogy, does not introduce iteration until after recursion. SS had spent over 15 hours studying LISP from Siklossy at the time of these protocols. Siklossy's book provides a great deal of discussion about how recursive functions are executed (i.e., by suspending processes and waiting for the results of recursive calls), but does not provide any indication of a general strategy for writing recursive functions. By the time SS encountered recursion, she had studied basic LISP functions, predicates, conditionals, and function definitions. Solving these three recursion problems took SS a total of 5 hours.

Our GRAPES simulation of SS was initialized with production rules for coding basic LISP functions, predicates, conditionals, and function definitions. It was also provided with a set of analogy productions and productions for simple reasoning about set theory. We assume that this initial set of rules characterized the knowledge state of SS when she started to learn about recursive functions.

SETDIFF. SS took a little over an hour to solve the SETDIFF problem. Table 2 gives a schematic protocol of her solution to the problem. Very important to SS's solution is the example that just precedes this problem in Siklossy. 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))))).
```

There are four conditional clauses in INTERSECTION1. The logic of the function is presented in Figure 5. If the first set is empty, then return the empty set; if the second set is empty, then return the empty set; if the first member of the first set is a member of the second set, then 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) a representation of the INTERSECTION1 conditional structure at multiple levels of abstraction, (b) a specification of the SETDIFF relation, (c) set-theory facts relevant to intersection and set-difference operations, and (d) 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 given

TABLE 2

Schematic Protocol of SS's SETDIFF Solution

---

1. SS reviews code for INTERSECTION1 function (preceeding example).
2. SS reads SETDIFF problem and forms the analogy SETDIFF:INTERSECTION::CDR:CAR. SS also proposes the following relation:
   SETDIFF (SET1, SET2)
   = MINUS (SET1, INTERSECTION(SET1, SET2)).
3. SS writes (DEFUN SETDIFF (SET1 SET2).
4. SS decides to code SETDIFF by rearranging INTERSECTION1 code.
5. SS decides to code simple cases found in INTERSECTION1.
6. SS considers case (NULL SET1), decides the action will be NIL. Code is now
```
(DEFUN SETDIFF (SET1 SET2)
  (COND ((NULL SET1) NIL).
```
7. SS 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.
9. SS decides to code the relation MINUS (SET1, INTERSECTION (SET1 SET2)). Realizes that MINUS is equivalent to SETDIFF and gives up on this plan.
10. SS 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. SS refines third clause 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. SS considers case in which (CAR SET1) is not a member of SET2. Formulates plan to add (CAR SET1) to the answer for SETDIFF.
13. SS decides to look at INTERSECTION1 code again. SS notes that 4th action of INTERSECTION maps onto 3rd action of SETDIFF, ponders whether 3rd action of INTERACTION1 will map onto 4th action of SETDIFF. Ss 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. SS checks code visually and on the computer.

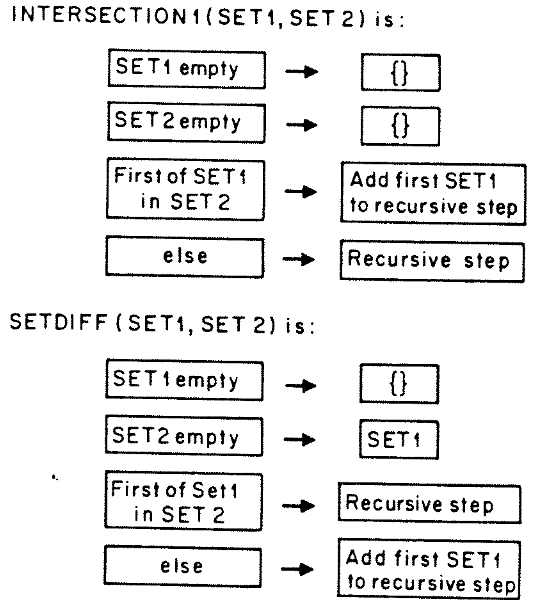


Figure 5. The logic of the conditional structures of the INTERSECTION1 and SETDIFF functions. Each arrow points from a condition to an action of a conditional clause.

the goal of writing SETDIFF. Since GRAPES did not have any means of directly solving the problem, the analogy production rule P1 applied. This production set 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.

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.[5] These productions found a sufficient similarity (the criterion is abitrarily 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. (P4)

---

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

This production set subgoals to map each INTERSECTION1 conditional clause onto the SETDIFF solution. 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 conditional of that clause
and set as subgoals
1. to determine the action in the current case given the mapped condition
2. to code the new condition-action clause (P5)

This production maps the condition of a clause of an example function onto a clause for the problem function and then sets a goal to determine what action should be associated with the condition. There is some evidence that SS had acquired this rule from earlier problems involving nonrecursive conditional functions.

SS's mappings 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 cause 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 INTERSECTION 1, but for the action she put SET1 which does not match the NIL action of 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 then the value of SETDIFF is SET1." Thus, it seems quite clear that she was reasoning through the semantics of the condition of the clause and what the implications were for the action of the clause. Our simulation, working with mappings of meaningful abstractions of the conditional clauses of INTERSECTION1, produced the same problem-solving behaviour as SS.

One GRAPES representation of the meaning of the third INTERSECTION1 condition was "test if the first element should be added to the answer," which is a rather liberal interpretation. Our principal justification for this representation is that it allows us to simulate the behaviour 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 (SS had not yet been taught the LISP function NOT). 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 thus 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 condition: "test if the first element of SET1 is in SET2." Thus, the third condition of INTERSECTION1, (MEMSET (CAR SET1) SET2), was used nearly literally as the condition for SETDIFF. Using the specification of SETDIFF, both simulation and subject decided that the output list should not contain the currently tested first element of SET1 and that SETDIFF should repeat over all elements of SET1. SS decided simply to 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. (P6)

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)." We assume that this template represents an abstraction made by SS from reading one and a half chapters on recursion from her text. The final code produced by GRAPES 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 condition "the first element of SET1 is NOT in 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 so far and INTERSECTION1. She noticed that, while the conditions of the third and fourth clauses of INTERSECTION1 were mapped onto the conditions of the third and fourth clauses of SETDIFF, the fourth action of INTERSECTION1 had been mapped onto the third action of SETDIFF (see Figure 5). She solved the stuctural analogy and concluded that the action from the third clause of INTERSECTION1 should be in the position of the fourth clause of SETDIFF. We gave GRAPES the goal of solving the structural analogy between the last two clauses of the production. Having been given this goal, it then set about solving the analogy just as our subject had.

After solving the problem, GRAPES went into a knowledge compilation phase during which it compiled segments of the problem-solving episode into single productions. A number of production rules were formed, but two important ones that were invoked in the later problem solving are the following:

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 and code a clause to deal with the case when SET1 is NIL
2. refine and code a clause to deal with the case when SET2 is NIL

3. refine and code a clause to deal with the case when the first element of SET1 is a member of SET2
4. refine and code a clause to deal with the else case. (C1)

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 (C2)

The first production, C1, compiles the analogy to INTERSECTION1 into a single rule. It is particularly important to note the relation between the analogy performed by SS and compiled rule C1. SS basically mapped the conditional structure of INTERSECTION1 onto the SETDIFF solution, and C1 reflects that particular mapping. Rule C1 will produce correct code for a subclass of CDR-recursive functions, but does not generalize to the entire class. The second production, C2, was learned in the context of coding the third clause of SETDIFF. This is the first rule for coding a recursive call learned by the subject. Note, however, that the condition of C2 has a nonrecursive semantics. This models SS's conception of the recursive call as causing the function to *repeat*. This mischaracterization of recursive functions as performing iteration is common among novices (see also Kahney, 1982; Kurland & Pea, 1983).

SUBSET and POWERSET. It turns out that these two productions, C1 and C2, were enough to enable SS and the simulation to solve the next problem in the series, SUBSET, which determines if the elements of one list are a subset of a second list (see Anderson et al., in press). However C1, and C2 are rather specialized and do not provide a basis for solving very many recursive functions. This inadequacy was exposed in the difficulty that SS and the simulation had in solving POWERSET (again, for details see Anderson et al., in press). Specifically, the iterative character of C2 provides no hint of the general recursive strategy discussed with respect to the expert's solution of POWERSET. The experimenter basically guided SS through the difficult parts of POWERSET. As a gross measure of the relative difficulty of these functions for SS, the SETDIFF problem took SS 1 hour to code, SUBSET ½ hour, and POWERSET 3½ hours.

The SUBSET and POWERSET functions coded by SS are presented in Table 3. Also included in Table 3 is a subfunction, CONST, written by SS that was needed to solve the POWERSET problem (CONST computes the same relation as ADDTO presented in our discussion of the expert simulation). In simulating SS's generation of the functions presented in Table 3, GRAPES compiled a number of productions that seem to characterize the skill development of SS. The coding of the three-clause conditional structure of SUBSET was compiled into the following production:

IF the goal is to code a relation on two SET1 and SET2
and the relation is recursive
THEN code a conditional and set as subgoals to
1. refine and code a clause to deal with the case when SET1 is NIL

TABLE 3

The SUBSET, POWERSET, and CONST Functions Coded by Subject SS

```
SUBSET
Example: (SUBSET '(a c) '(a b c)) = T  (SUBSET '(a d) '(a b c)) = NIL
Function: (defun subset (set1 set2)
             (cond ((null set1) t)
                   ((member (car set1) set2)
                    (subset (cdr set1) set2))
                    (t nil)))

POWERSET
Example: (POWERSET '(a b c)) = ((a b c) (a b) (a c) (a) (b) (c) ())
Function: (defun powerset (sett)
             (cond ((null sett) '(()))
                   (t (union (powerset (cdr sett))
                             (const (car sett)
                                    (powerset (cdr sett)))))))

CONST
Example: (CONST 'a '((b) (c) ())) = ((a b) (a c) (a))
Function: (defun const (sil lis)
             (cond ((null lis) nil)
                   (t (cons (cons sil (car lis))
                       (const sil (cdr lis))))))
```

```
      2. refine and code a clause to deal with the case when the first element
         of SET1 is a member of SET2
      3. refine and code a clause to deal with the else clause;          (C3)
```

and the coding of the two-clause conditional structure of POWERSET was compiled into a similar production:

```
   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 and code a clause to deal with the case when SET1 is NIL
      2. refine and code a clause to deal with the else case.            (C4)
```

Note that production C4 is very much like production P3 for setting up CDR-recursion in the expert model.

Another important production was learned by GRAPES in coding the action to the last conditional clause of CONST, (CONS (CONS SIL (CAR LIS)) (CONST SIL (CDR LIS))). The function CONS adds an element to a list (e.g., (CONS 'a '(b c)) = (a b c)), and in CONST it is used to accumulate elements into an output list. The production learned in CONST is:

```
   IF the goal is to code a relation adding result1 to result2
      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).                                     (C5)
```

Production C5 uses the LISP function 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 piece of recursive programming.

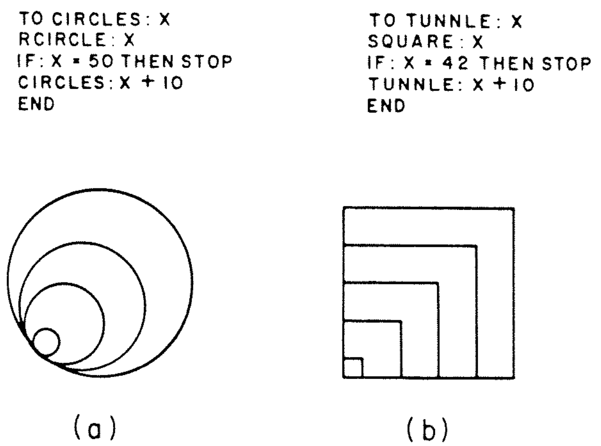
Sumary of SS's Learning and Performance. There are several general conclusions to make from our three studies of novices learning LISP from standard programming texts. Each of these conclusions is corroborated by evidence in SS's protocol (see Anderson et al., in press). First, recursion is difficult because of its unfamiliarity, and there are a number of factors that exacerbate this difficulty: (a) Programming texts typically do not provide explicit instruction on how to write recursive functions, they teach how recursive functions work. Later, in the discussion of the protocol and simulation of subject AD, we will show that providing a conceptual model for the structure of recursive function code can facilitate learning. (b) Difficulties arise due to interference from naive conceptions of recursion. SS initially characterized recursion as a form of iteration. This led to a number of SS's difficulties with POWERSET. (c) Recursion is difficult because there are many different patterns of recursive functions (see Soloway & Woolf, 1980). After the POWERSET problem, SS was still learning the numerous patterns that define the application of the CDR-recursion technique to various problems. It is also worth noting that she was learning only 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.

Second, because of the unfamiliarity and difficulty of recursion, we see analogy playing an absolutely critical role in enabling the solution of the initial recursion problems encountered. The analogical mapping from INTERSECTION1 to SETDIFF was never a mindless symbol-for-symbol mapping. Rather, it involved the subjects' knowledge of LISP and a representation of the meaning of what was mapped. Still, the subject struggled with exactly what representations to map. The process of problem solving by analogy is hardly trivial.

Finally, new domain-specific productions are acquired by the mechanisms of knowledge compilation. These new productions summarize particular sequences of operations performed in solving a problem. Thus, the particular analogies used in problem solving have an impact on the productions learned from a solution attempt.

In the next two simulations and protocols, we examine the role of analogy and conceptual models of recursion in learning to program recursion in further detail.

Subject JP: Subject JP was an 8-year-old who had participated in ten 2-hour weekly LOGO classes. Prior to learning about recursion, JP had coded functions taking variables and functions that called subfunctions. She was introduced to both recursion and conditional statements in the same lesson. Part of JP's lesson on recursion included a demonstration and discussion of a recursive function CIRCLES, which drew a set of circles of increasing diameter (see Figure 6a). This program used a subprocedure RCIRCLE (read as "right circle") which drew circles in a clockwise manner. JP was then given the problem of creating a function that would recursively draw a set of squares of increasing size. JP was



```
TO CIRCLES: X
RCIRCLE: X
IF: X = 50 THEN STOP
CIRCLES: X + 10
END
```

```
TO TUNNLE: X
SQUARE: X
IF: X = 42 THEN STOP
TUNNLE: X + 10
END
```

Figure 6. (a) The CIRCLES function presented to subject JP. The figure drawn by CIRCLES is a series of concentric circles. The function RCIRCLE draws a circle. (b) The TUNNLE [sic] function coded by subject JP. TUNNLE draws a series of concentric squares.

also informed that she could use a function SQUARE to draw a square of variable size.

JP's final solution (called TUNNLE [sic]) and the figure it drew are presented in Figure 6b. The first line of code in Figure 6b specifies that the function TUNNLE takes a single argument, :X. The second line calls on the subfunction SQUARE to draw a square of size :X. The third line is a terminating case that stops the recursive process when the variable :X is 42, and the fourth line is a recursive call to TUNNLE.

A schematic protocol of JP's attempts to solve the TUNNLE problem is presented in Table 4. Subject JP basically could not code the problem function until the experimenter re-presented and explained the code for the CIRCLES function. The experimenter then removed the CIRCLES code from JP's view, and JP coded a correct function drawing concentric squares. However, immediately after writing TUNNLE, JP was unable to code a function for a slight variant of the concentric squares figure. We present an analysis and simulation of JP's coding of her recursive TUNNLE function by analogy to the CIRCLES example. In doing so, we wish to contrast the efficacy of the CIRCLES-TUNNLE analogy and resultant learning of this problem-solving episode with that of SS's INTERSECTION1-SETDIFF analogy and resultant learning.

The GRAPES long-term memory was initialized with information which included the following, slightly degenerated, encoding of the CIRCLES code:

```
TO CIRCLES :X
right-circle :X
IF :X = large-number THEN STOP
CIRCLES :X + 10
END
```

TABLE 4

Schematic Protocol of Subject JP's TUNNLE Solution and Attempt to Code LINE

1. JP starts with an (incorrect) nonrecursive solution.

```
TO TUNNLE
SQUARE 3
RT SQUARE 3
SQUARE 13
RT SQUARE
SQUARE 22
RT SQUARE 13
```

2. The experimenter interjects to point out that JP should use recursion and prompts JP to recall how the CIRCLES function worked. JP erases code.

3. JP decides to use the SQUARE program. JP wants to draw one square (assumes that :X is set to 1).

```
TO TUNNLE :X
SQUARE :X
```

4. JP wants to make a right square of size 2.

```
TO TUNNLE :X
SQUARE :X
RT SQUARE :X + 1
```

5. JP says she now has one square drawn and wants to make a square of size 2. JP ponders whether to write SQUARE :X + 1 or SQUARE :X = 2. Codes:

```
TO TUNNLE :X
SQUARE :X
RT SQUARE :X + 1
SQUARE :X
```

JP says "now :X = 2."

6. The experimenter prompts JP to remember how CIRCLES worked. JP can't remember. The experimenter writes out the CIRCLES code and explains the function. JP wants to substitute TUNNLE for CIRCLES in the recursive line. JP deletes all the TUNNLE code and forgets the CIRCLES code. JP asks to view the CIRCLES code one more time.

7. JP tries to decide between writing RSQUARE or RT SQUARE (analogous to RCIRCLE: right circle). The experimenter points out that there are no such programs. JP writes:

```
TO TUNNLE :X
SQUARE: X
IF = 42 THEN STOP
TUNNLE :X + 10
```

8. JP tries out the function but gets an error message regarding a missing variable before = 42. Inserts the variable :X.

```
TO TUNNLE :X
SQUARE :X
IF :X = 42 THEN STOP
TUNNLE :X + 10
```

9. The experimenter prompts JP to explain what each line of TUNNLE does. When JP gets to the recursive line JP says she doesn't understand what it does.

10. The experimenter prompts JP to try the function and its works with an input of 2. The experimenter again prompts for an explanation of the code, and JP still does not understand the recursive call.

TABLE 4 (Continued)

Schematic Protocol of Subject JP's TUNNLE Solution and Attempt to Code LINE

11. The experimenter asks JP to write a function like TUNNLE that draws exactly 10 squares if given the input 2. JP writes:

```
TO LINE
LINE 2
IF :X = 42 THEN STOP
LINE :X + 10
```

12. JP tries to function and gets an infinite recursion. JP has to stop to get the next lesson from the teacher.

All items except *right-circle* and *large-number* are the actual LOGO code presented to JP. The item *right-circle* represents a verbal trace of the LOGO code RCIRCLE, and the item *large-number* represents a more abstract propositional trace corresponding to the number 50 presented in the CIRCLES code.

When GRAPES was given the goal to code a procedure called TUNNLE to produce the concentric squares figure, the analogy production P1 applied. Production P1 set subgoals to compare the CIRCLES problem to the TUNNLE problem and to structure-map the CIRCLES code onto the TUNNLE function. A set of comparison productions then found that: (a) the functions TUNNLE and CIRCLES differ in name, and (b) TUNNLE draws squares while CIRCLES draws circles. Structure-mapping productions then used these differences to alter the CIRCLES code to fit the specification for the concentric squares figure. These productions scanned each line of code, substituting RT (a function called *right*) for the verbal trace *right*, SQUARE for *circle*, and the name TUNNLE for CIRCLES. GRAPES also selected an arbitrary number (fixed to be 42) to instantiate *large-number*. The incorrect specification of RT SQUARE was corrected by interrupting the GRAPES execution and inserting the correct substitution goal (just substitute SQUARE). This corresponds to the experimenter's interruption of JP to correct the same error.

The code produced by GRAPES was correct and matches that of JP. However, it should be pointed out that this execution of problem solving by analogy used relatively nonabstract representations of the CIRCLES code and the differences between CIRCLES and TUNNLE. Knowledge compilation over this episode produced the following production:

```
    IF the goal is to code a figure
        and the function for the figure is called NAME
        and a subfigure of the figure is coded by SUBFUNCTION
 THEN write
        TO NAME :X
        SUBFUNCTION :X
        IF :X = large-number THEN STOP
        NAME :X + 10
        END.                                              (C6)
```

Production C6 differs in character from production C1 compiled in simulating the adult subject SS. It matches to relatively nonabstract representations of the figure and function to be coded, and its actions are likewise relatively nonabstract. Consequently, production C6 has an even more restricted applicability in the domain of recursive function specifications than C1.

The experimenter's prompting for explanations of the TUNNLE code indicated that JP did not understand what purpose the recursive call performed. Further, JP's lack of comprehension of recursion became especially evident when JP was asked to write a new function, identical to TUNNLE, except that the new function would draw exactly 10 squares given an input of 2. The function JP wrote was:

```
TO LINE
LINE 2
IF :X = 42 THEN STOP
LINE :X + 10
END
```

Although JP correctly inserted a recursive call in this function, she failed to code a terminating case which would permit 10 squares to be drawn. In addition, JP failed to make a call to the SQUARE function. Instead she inserted a recursive call to LINE.

Production C6 will mimic JP's erroneous coding of the LINE function to a good approximation. The major difference between JP's and GRAPES' performance would concern the first statement in LINE. JP chose to code LINE 2. GRAPES would insert the line SQUARE :X. GRAPES could simulate JP's error by assuming that it results from either a working memory failure or faulty program specification which would cause the system to choose LINE as the appropriate subfunction to call.

Over the course of the 4 hours of protocols obtained from subject JP writing recursive functions, we saw no indication that she could write such functions without a lot of assistance from examples or the experimenter. To a large extent this appears to result from an inability to encode the abstract and general semantics of recursive functions from the examples that were presented to her and to use those encodings in analogy.

In the next section we present an analysis of a subject (AD) who learned to code a subset of recursive functions quite rapidly. We attribute this to the fact that she had been presented with an abstract explanation of an example recursive function that she used in analogy. The analogy performed by AD from the example led to her learning a general strategy for coding recursive functions very similar to that presented in Figure 1 and that used in the expert simulation solving the POWERSET problem (Figure 3).

Subject AD: The previous protocols and simulations indicate that the particular example encodings used by students in analogical problem solving have a large impact on the early learning of programming recursion. If students would only use the right encodings in analogy, then we would expect to see rapid learning. What are the right encodings? Our hypothesis is that the right encoding represents the problem in terms of the general concepts needed to define the general strategy for coding recursion (outlined in our section on the expert model for programming recursion and in Fig. 1). Such an encoding would represent recursive functions as consisting of terminating cases and recursive cases. The encoding would also have to include the notion that the results of recursive cases (e.g., $f(n)$) are obtained by assuming that the results of recursive calls (e.g., $f(n-1)$) can be found. Subject AD was provided with a description of recursive functions that emphasized this conceptual model of recursion.

Prior to learning to program recursive functions, AD had spent 4 hours learning the basic functions, predicates, conditional structures, and definitional syntax in the SIMPLE language and had written four nonrecursive functions. All of these programming tasks centred on manipulating a stored database of 18 entries in a book library. The entries in this database could be identified by a number (id number), a key word (title), and could be categorized as science, religion, or fiction books.

AD's introduction to recursive functions included the following description:

> A recursive function definition consists of two components: (1) A definition of one or more terminating conditional statements in which a simple answer is returned, (2) A definition of one or more recursive cases in which the answer to the current problem is solved by assuming that the answer to a simpler version of the problem can be found.

Two examples were then discussed in the context of this description. The first was a nonprogramming example from mathematics: $X^n = X \times X^{n-1}$, for $n > 0$, and $X^0 = 1$. The second example was a SIMPLE function, SORT, which sorted an input list of book titles such that all science books were at the beginning of the list. In order to insure that AD did not use the actual code of this example to analogize her solutions, we removed the code from her view (leaving the general description of recursion and the mathematical example at her disposal). The recursive function problems given to AD came from a space of 16 functions characterized by four dimensions with two values on each dimension. Each function could (a) take a list of titles or an id number as input, (b) return a list of science or nonscience items, (c) return the output list with items in the same or reverse order they are encountered in recursion, and (d) skip items that are the opposite of what is being collected or return the current accumulated result when first encountering an opposite. AD's task was first to write four functions correctly with feedback for errors and then write all 16 functions with no feedback.

XYZ. AD called her first recursive function XYZ. The problem was specified to AD as:

> Write a function which takes a book id as variable and tests ids less than and including the variable. The function should return an answer list which contains all science ids from the variable to the first non-science id less than the variable. Answer list ids should be in ascending order.

A schematic protocol of AD's XYZ solution is presented in Table 5. AD at first came up with an iterative characterization of the function she was writing which she refined into the following components: (a) if the input variable is a science

TABLE 5

Schematic Protocol of AD's XYZ Solution

1. AD reads through her instructions which include a general description of recursion, a mathematical example (X^n), and a SIMPLE function (SORT) that sorts a list of titles such that all science books are at the beginning of its output list. The experimenter takes away the SORT example.
2. AD reads the problem specification.
3. AD thinks back to earlier functions she's encountered and says that SUB1 will be useful.
4. AD formulates an iterative plan to test each number less than and including the input variable until a nonscience book is encountered.
5. AD begins to type in the first line of the function (specifying the function name and its arguments) but accidently presses the return key.[a] AD decides to write out the conditional structure and return to the first line later.
6. AD decides to write a test to determine if the input variable ID is a science book. AD types IF (ID ISA? SCIENCE).
7. AD decides that if the ID variable is a science book, then it should be placed in a list and other science items could be added to this list. Using an example of the PRE function presented in her instruction, AD's code becomes:
   ```
   IF (ID ISA? SCIENCE) THEN ID PRE []
   ```
8. AD wants the function to test one less than ID (using SUB1) but cannot figure out how to add this to the clause she has just written. Instead, AD deletes a portion of the written code and writes:
   ```
   IF (ID ISA? SCIENCE) THEN SUB1 ID.
   ```
9. AD realizes that her code still would not work and decides to try to recall the SORT example.
10. AD recalls that the SORT example assumed that the result of a recursive call to SORT could be found. AD makes the analogous assumption for the function she is coding: that the recursive call XYZ (SUB1 ID) can be found.
11. Using her assumption, AD writes the recursive call:
    ```
    IF (ID ISA? SCIENCE) THEN (ID PRE (XYZ (SUB1 ID))).
    ```
12. AD considers the case in which the input variable ID is not a science book and decides the result should just be an empty list. The code is:
    ```
    IF (ID ISA? SCIENCE) THEN (ID PRE (XYZ (SUB1 ID))),
    ELSE [].
    ```
13. AD inserts the first line of the function:
    ```
    XYZ ID IS
    IF (ID ISA? SCIENCE) THEN (ID PRE (XYZ (SUB1 ID))),
    ELSE [].
    ```

---

[a] Subject AD used a simple computer editor to generate her functions. This editor consisted of two modes. First, lines were inserted in the function by typing code and pressing return. Second, a period was typed and code could be altered by inserting, deleting, or changing lines. The first mode always preceded the second (i.e., AD could not switch back and forth between modes). In the first mode, AD accidently pressed return and entered a partial line of code as the first line of XYZ. She then waited until she had written her last line of code before switching to the second mode and altering the first line of XYZ.

TABLE 5 (Continued)

Schematic Protocol of AD's XYZ Solution

---

14. AD realizes that she has confused the function PRE with the function post and corrects her code to be:
    XYZ ID IS
    IF (ID ISA? SCIENCE) THEN (ID POST (XYZ (SUB1 ID))),
    ELSE [].
15. AD tests her function and gets several error messages as well as the correct function. The experimenter points out that XYZ does not have a terminating case.

---

book then put it in a list, (b) repeatedly test id numbers less than the input variable, and (c) for each id that is a science book, put it at the beginning of the current output list. AD then attempted to map each of these components into code. AD first wrote the conditional statement IF (ID ISA? SCIENCE) THEN ID PRE [], which places the input variable ID in a list if it is a science book. The action is this statement (THEN ID PRE []) forms a list by placing the variable ID in an empty list using the PRE function. AD formulated this code by using an example of PRE in her instruction text as a guide. AD then decided that she wanted XYZ to determine if the id that was one less than the variable ID was a science book. However, she realized that she would be violating the syntax of the SIMPLE language if she placed more code after the statement THEN ID PRE[]. Faced with this impasse, AD mapped the remainder of her solution from what she could recall of the SORT example that had been presented to her.

Our GRAPES model of AD consisted of a set of production rules for the syntax of function definitions and conditional statements. It was also provided with some specific productions for coding conditional tests to determine if an item is a science book, and for coding the SIMPLE functions FIRST (returns the first of a list), REST (returns a list of all but the first element of a list), and SUB1 (returns one less than an input number). This production set characterizes the skill development of AD who had written functions involving only these code patterns. The model was also provided with a set of rules for refining general iterative procedures.[6] As with our other simulations, our model of AD was also provided with a set of productions for writing code by analogy.

The goal tree produced by GRAPES in solving the XYZ problem is presented in Figure 7. GRAPES was given the goal to code XYZ and the following representation of the XYZ problem specification: "Given the number ID, test all items less than or equal to ID. IF the item is a science book then it is in the result." The following production for refining the specification and coding the function applied:

---

[6] This is consistent with our hypothesis that people are familiar with iterative procedures that occur in everyday life.

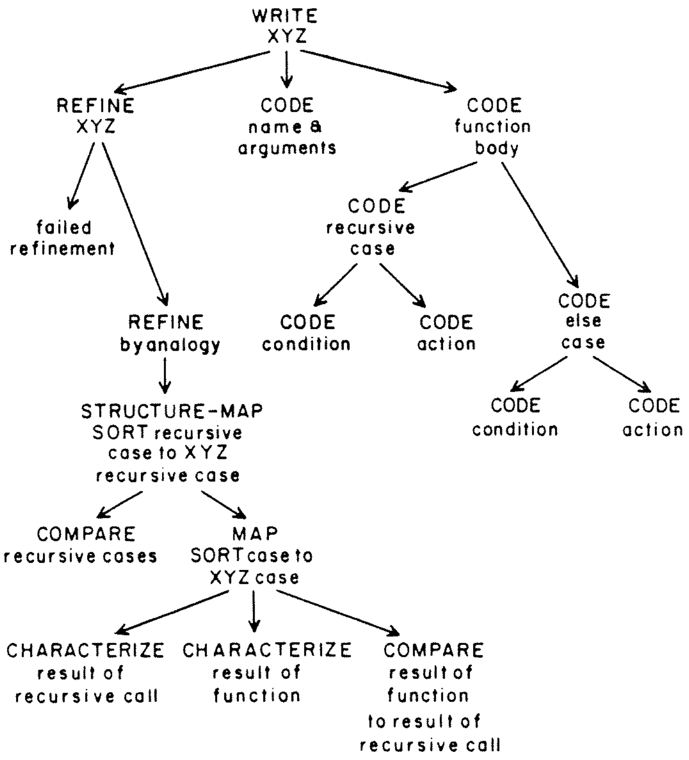


Figure 7. The final goal hierarchy produced by the GRAPES simulation of subject AD writing the XYZ function.

IF the goal is to write a function
THEN set goals to
  1. refine the specification for the function
  2. code the function name and argument
  3. code the function body. (P7)

Production P7 characterizes the general strategy for coding functions that AD appeared to have learned in coding nonrecursive functions.

In Figure 7 we have summarized the first abortive attempt by *failed refinement*. This refers to a refinement by which GRAPES transformed the problem specification into an iterative plan to add the variable ID to the result if it was a science book, and then to repeat this process for items less than ID. The GRAPES simulation then coded the function name and argument (using a production assumed to have been learned previously) and turned to the goal of coding the body of XYZ. GRAPES mimicked AD's initial code by applying productions for coding conditional statements and for the test to determine if the input variable ID is a science book. GRAPES then turned to the goal of placing the variable ID in the result list. The action, THEN ID PRE [], was coded by the structure-mapping

TABLE 6

Protocol Excerpts of AD's SORT-XYZ Mapping

Finding the analogy: "I'm thinking back to the example. It did that [conditional test for science books] and it made an assumption."

Refining the analogous solution: "We made an assumption earlier that it had SORTed the previous [items of a list]."

Mapping the analogous solution to the current problem: "So first I want it [XYZ] to test (SUB1 ID)."

Determining the recursive case: "ID is going to be the highest number and (SUB1 ID) is going to be less. So if I put PRE[a], that would mean that I would have the highest number [at the end of the list]."

---

[a] The correct function to use was POST. However, AD confused the function PRE for POST.

production P6, which mapped an example of the PRE function that was provided to GRAPES. The simulation then set a goal to write code that would repeat the conditional test on one less than the variable ID. The syntax violation that would occur by writing code after THEN ID PRE [] was caught by a *critic* (Brown & VanLehn, 1980) production rule that does not allow two function calls after a conditional test:

```
    IF the goal is to code a relation
       and the code occurs after another relation
       and both relations occur after "THEN"
 THEN pop the current goal as failed.                                    (P8)
```

Having failed to code the conditional statement for XYZ, GRAPES returned to the goal of refining the specification of that conditional.

Table 6 presents the significant statements made in AD's protocol that provide a clear indication of how she mapped the SORT solution onto her first recursive function. These statements took place over the course of about 5 minutes. First, AD was reminded of the recursive case in the SORT function. Next, she recalled that the answer in the recursive case was produced by first assuming that SORT had sorted all but the first of the items in its input list. Since AD was coding a function that took an id number as input, she decided that the appropriate assumption to make in her function was that it had processed all items less than her input variable; that is, it had recursed with (SUB1 ID). Having made that assumption, AD reasoned that since she needed an output list in ascending order, her function should place the input id number at the end of the result of the recursive call using the SIMPLE function PRE. Unfortunately, AD confused the function PRE, which puts an item at the beginning of a list, with the function POST, which puts an item at the end of the list. However, she later caught this PRE/POST confusion, and the analogy resulted in the correct coding of the first recursive case in XYZ: IF (ID ISA? SCIENCE) THEN (ID POST (XYZ (SUB1 ID))).

AD also realized that her function needed another conditional statement to deal with the case in which the input id number is not a science book. This she did quite easily (since it does not require a recursive call) by coding ELSE [], which returns an empty list. AD did, however, forget to code a terminating case for XYZ, and thus the XYZ function originally produced an error message indicating an infinite recursion had taken place. AD was then presented with the correct code for XYZ by the system.

Our GRAPES simulation of AD performs the same analogy as AD as a second attempt to refine the XYZ specification. At this point GRAPES falls back on its analogy productions which structure-map the following representation of one of the SORT recursive cases onto the XYZ function:[7]

```
Conditional test: the first of a list is a science book
Action: assume that the SORT of the remainder of the list
        has been calculated and use that result to
        get the result of SORT of the list.
```

The goal tree produced by GRAPES in mapping this representation onto XYZ is presented in Figure 7 under the goal "refine by analogy." First, the test of the SORT conditional statement for the recursive case is compared to that of the XYZ and found to be of the same type. Second, the method by which the action of the SORT recursive case is calculated was mapped. The mapping of this method produced three subgoals: (a) to characterize the result of XYZ of numbers less than ID, (b) to characterize the result of XYZ of ID, (c) to characterize a method for getting the XYZ of ID from the XYZ of items less than ID. GRAPES calls on knowledge of the function specification to arrive at each of these characterizations and then turns to coding the XYZ recursive case.

The key point to make about this goal tree is that it has the same basic structure as the general strategy for coding recursive functions in Figure 1. Because of this similarity, knowledge compilation of the GRAPES XYZ solution produced a production rule that sets up a variant of the general strategy in Figure 1:

```
    IF the goal is to write a function
       and the function repeats some operation over items
 THEN set as subgoals
       1. to characterize the result of a recursive call to the function
       2. to characterize the result of the function
       3. determine the relation between the result of the recursive call and the result
          of the function
       4. code the function name and argument
       5. code the function body.                                        (C7)
```

This rule sets up a plan that generalizes to a large subset of recursive functions (and certainly the 16 functions coded by AD).

YXZ. The second recursive coded function by subject AD was called YXZ. The problem was specified as:

---

[7] We did not provide GRAPES with a representation of the literal code for SORT based on the assumption that AD had forgotten the actual SORT code.

> Write a function which takes a list of books as variable. The function should return an answer list which contains all science titles from the first of the list to the first non-science title. The answer list should contain titles in the order they appear in the variable list.

Almost immediately after reading this specification AD verbalizes her ideas about coding YXZ:

> Experimenter: "What's the plan of action?"
>
> AD: "I haven't really got one. OK I've been thinking about what I'll want to use. I want to use FIRST because I'm working with a list. So I make the assumption again that when it takes the first of the list it's already checked all of the other ones. Then I want to form another list, with ... in the right order."

In a similar manner, GRAPES applies newly-learned production C7 which sets goals to assume the result of a recursive call and to use that result as a basis for formulating the recursive cases of the YXZ function.

Most of the time spent coding YXZ by AD and GRAPES is taken up by refinement of this general plan. Both AD and GRAPES reason that the correct ordering of items in the output list can be achieved using the SIMPLE function PRE, that YXZ should be called recursively with the tail of the input list (using the function REST), and that the terminating case for the function should deal with the case in which the input variable is an empty list.

Summary of Novice Protocols and Simulations

In contrasting subjects SS, JP, and AD, we see the critical role of problem solving by analogy to examples in initial attempts to code recursive functions. We also see the impact of these analogies on structuring solutions to subsequent problems. Production C6 was compiled in simulating subject JP's mapping of a relatively literal representation of example code onto her first recursive function TUNNLE. This production structures a very small subset of recursive functions, and JP did have a great deal of difficulty with subsequent recursion functions. Production C1 was compiled in simulating subject SS's mapping of a more abstract representation, the conditional structure, of an example onto her first recursive function SETDIFF. Production C1 structures a subset of CDR-recursive functions. SS had little difficulty with the SUBSET function which has the same general conditional structure as SETDIFF, but performs a rather different computation. On the other hand, SS could not code the POWERSET function without assistance, in part because POWERSET does not have a membership test as part of its conditional structure, as do SUBSET and SETDIFF. Production C7 was compiled in simulating subject AD's mapping of an abstract representation of the formal structure of the recursive cases of an example onto her solution for her first recursive function XYZ. This production structures a very large class of recursive functions, and AD had very little difficulty coding the remaining 16 recursive functions presented to her.

In the simulations of subjects SS and AD we also saw how a large component of learning involved the acquisition of productions that deal with particular patterns of code that occur in recursive functions. Productions C2 and C5 have the character of producing code for particular coding specifications that occur in the context of recursive functions (e.g., production C5 for accumulating results). Our GRAPES model of learning suggests that the acquisition of such specific coding productions is a matter of being exposed to problems that involve such specifications and solution code. However, subject AD had a distinct advantage over subject SS, in that AD's program solutions were correctly structured after her first recursion problem by production C7. Having this general structure essentially reduced her solutions to refinement of the general plan outlined by production C7. Having her problem search reduced by a correct general strategy allowed AD to get through recursion problems with less difficulty. Subject SS on the other hand, lacking a correct general solution structure, floundered in her attempts to code POWERSET until the structure of that problem was outlined by her experimenter. Problems that did not fit in with the conditional structure outlined by production C1 required more search effort by SS for solution.

This comparison of subjects AD and SS suggests that the acquisition of productions for coding recursive functions can be facilitated by having subjects learn, at the outset, a general strategy for coding a large class of recursive functions. Using this general strategy, students should take less time working through their initial programming problems and, consequently, should be exposed to more recursive function patterns in less time.

A recent experiment of ours corroborates this hypothesis. Two groups of subjects learned the basic functions, predicates, conditional structures, and definitional syntax of the SIMPLE language in the same manner as subject AD. One group of subjects (structure group, $N = 10$) was presented with the same instruction on recursion as subject AD. The second group (process group, $N = 9$) received a set of instructions paraphrased from a LISP text that, in a manner of standard recursion pedagogy, emphasized how recursive functions work, not how they are written:

> A recursive function is one which uses itself in its own definition. Such a function solves a complicated problem by handing a simpler version of the problem to a copy of itself. This process may be repeated. When a function copy solves a simpler problem, the answer is substituted back into a more complex copy.

The process group was presented with the same examples as the structure group (definitions of X^n and the SIMPLE program SORT); however, these were discussed in the context of how they worked by showing traces and explanations of sequences of recursive calls. Both groups of subjects first had to write four recursive functions correctly with feedback for errors (training phrase) from the space of 16 functions outlined previously. When they reached the criterion of being able to generate all four recursive functions without error, they then moved to the transfer phase. In this phase, they attempted to write all 16 functions with no feedback.

As predicted, structure group subjects took significantly less time to write their first four functions correctly in the training phase ($M = 57.4$ min) than process group subjects ($M = 85.3$ min). Interestingly, the groups did not differ in either time to write functions or number of incorrectly coded functions in the transfer phase. We take this as evidence for the notion that in the training phase both groups acquired a set of productions to deal with specific coding situations. However, our data suggests that the structure group got to this state in a more efficient manner because they had learned a general stategy for structuring their code very early on in the training phase.

CONCLUSIONS

Our GRAPES production system model and its knowledge compilation mechanisms seem to characterize a large number of performance and learning phenomena observed in students learning to program recursive functions (see also Anderson et al., 1984; Anderson et al., in press). Our protocol data indicates that people rely heavily on examples to guide their solutions to novel and difficult problems. Knowledge compilation of the problem solving involved in analogizing new solutions from examples produces new productions that generalize to other problems. The generality of these new productions is dependent on the generality of the representation of the example solution used to produce a novel solution.

The analysis we have provided is concerned with only the initial organization of a skill. Other learning mechanisms (see Anderson, 1983, Chapter 6) are probably responsible for the further development of programming as it becomes a smooth skill. However, we believe that this analysis of the origins of a skill is not restricted to just programming, but describes the beginning of any skill with a strong problem-solving component. This is because the ACT* learning theory is quite general and has already been applied to a number of domains such as geometry proof generation (Anderson, 1982) and language acquisition (Anderson, 1983, Chapter 7). Thus, we believe that this analysis is appropriate to the beginnings of a number of the domains discussed in this issue.

REFERENCES

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

Anderson, J.R. (1982). Acquisition of proof skills in geometry. In J.G. Carbonell, R. Michalski, & T. Mitchell (Eds.), *Machine learning: An artifical intelligence approach* (pp. 191-220). San Francisco: Tioga Press.

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

Anderson, J.R. (1984). Cognitive pychology and intelligent tutoring. *Proceedings of the Sixth Annual Conference of the Cognitive Science Society* (pp. 37-43). Boulder, CO: Institute of Cognitive Science.

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

Anderson, J.R., Pirolli, P.L., & Farrell, R. (in press). Learning to program recursive functions. In M. Chi, R. Glaser, & M. Farr (Ed.), *The nature of expertise.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Anzai, Y., & Uesato, Y. (1982). *Is recursive computation difficult to learn?* (Tech. Rep. CIP 439). Pittsburgh: Department of Psychology, Carnegie-Mellon University.

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.

Carbonell, J.G. (1983). Derivational analogy and its role in problem solving. *Proceedings of the National Conference on Artificial Intelligence.* Washington, DC: University of Maryland and George Washington University.

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

Gick, M.L., & Holyoak, K.J. (1980). Analogical problem solving. *Cognitive Psychology*, **12**, 306-355.

Gick, M.L., & Holyoak, K.J. (1983). Schema induction and analogical transfer. *Cognitive Psychology*, **15**, 1-38.

Kahney, H. (1982). *An in-depth study of the cognitive behaviour of novice programmers.* (Tech. Rep. No. 5). Milton Keynes, England: The Open University, Human Cognition Research Laboratory.

Kurland, D.M., & Pea, R.D. (1983). Children's mental models of recursive LOGO programs. *Proceedings of the Fifth Annual Conference of the Cognitive Science Society.* Rochester, NY: University of Rochester, The Cluster in Cognitive Science.

Miller, G.A. (1967). *The psychology of communication.* New York: Basic Books.

Neves, D.M., & Anderson, J.R. (1981). Knowledge compilation: Mechanisms for the automization of cognitive skills. In J.R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 57-84). Hillsdale, NJ: Lawrence Erlbaum Associates.

Newell, A. (1973). Production systems: Models of control structures. In W.G. Chase (Ed.), *Visual information processing.* 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). Urbana-Champaign, IL: University of Illinois.

Ross, B.H. (1984). Remindings and their effects in learning a cognitive skill. *Cognitive Psychology*, **16**, 371-416.

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

Shrager, J., & Pirolli, P.L. (1983). *SIMPLE: A simple language for research in programmer psychology* [Computer program]. Pittsburgh: Carnegie-Mellon University, Department of Psychology.

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

Soloway, E.L., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, **26**, 853-860.

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). Cambridge, MA: University of Massachussetts.