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, les novices utilisent pour résoudre les exemples ; (b) différentes représentations des exemples sont utilisées pour diriger la solution de problème par analogie et (c) la performance sur les 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.
The ORPES Production System

The ORPES Production System was developed in the context of the field of recursive programming.

Recursive Programming

Recursive programming is a method of programming where a function calls itself repeatedly to solve a problem. This approach is particularly useful for solving problems that can be broken down into smaller subproblems, each of which can be solved by the same function.

Section 1.2. Recursive Programming

The concept of recursive programming is based on the idea that a function can refer to itself within its definition. This self-referential nature allows for the elegant and concise solution of problems that are naturally divided into smaller, similar subproblems.

The examples provided illustrate the use of recursive functions to solve problems such as calculating factorials and traversing tree structures.

Learning from examples: By using these examples, we can understand the pattern and logic behind recursive programming.
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:

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 return 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.

GRAPE5 operates by repeatedly selecting productions whose conditions are satisfied (matched) and executing the actions of those selected productions. GRAPE5 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, GRAPE5 is like some other recent cognitive theories (Anderson, 1983; Brown & VanLehn, 1980; Card, Moran, & Newell, 1983; Rosenbloom & Newell, 1983).

Another distinguishing feature of GRAPE5 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 GRAPE5 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 proceduralization in GRAPE5 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.

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 LOGO3 language, four 6-hour protocols of novices learning the language designed for experimentation (SIMPLE, Shrager & Piroli, 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 system 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

---

1Throughout this paper we will present productions in an English-like manner. The specifications of these rules in GRAPE5 and the GRAPE5 user manual (Sauers & Farrell, 1982) are available on written request.

2LISP is a LISP Processing Language. Programs written in LISP take the form of functions that compute a particular input-output relation.

3LOGO is a language designed primarily to manipulate simple screen graphics, although it does have capabilities for performing numeric and list manipulations.
The results of the experimental design are as follows:

- **Condition A**: A control group that received no additional training.
- **Condition B**: A group that received a brief introduction to the topic and then engaged in practice exercises.
- **Condition C**: A group that received a detailed explanation of the concepts followed by practice exercises.
- **Condition D**: A group that received both the detailed explanation and practice exercises, with the addition of a reflective discussion session.

Overall, Condition D showed the most significant improvement in performance, followed by Conditions C and B, with Condition A showing the least improvement. The results suggest that a thorough combination of explanation and practice, along with reflection, leads to the best learning outcomes.
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 GRAPE 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 GRAPE 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.

(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 GRAPE, 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.

Activating goals in a left-to-right, depth-first manner, GRAPE 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
In Powerset, we define the powerset of a set X as the set of all possible subsets of X, including the empty set and X itself. In other words, the powerset of X is the set \( \mathcal{P}(X) \) defined as:

\[
\mathcal{P}(X) = \{ \emptyset \} \cup \{ Y \subseteq X \mid Y \neq \emptyset \}.
\]

To illustrate, consider the set X = \{1, 2, 3\}. The powerset of X is:

\[
\mathcal{P}(X) = \{ \emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \}.
\]

This concept is fundamental in various areas of mathematics and computer science, particularly in set theory and combinatorics.
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:

```lisp
(INTERSECTION1 (LAMBDA (SET1 SET2))
  (COND ((NULL SET1) NIL)
         ((NULL SET2) NIL)
         ((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 the 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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>SS reviews code for INTERSECTION1 function (preceeding example).</td>
</tr>
<tr>
<td>2.</td>
<td>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)).</td>
</tr>
<tr>
<td>3.</td>
<td>SS writes (DEFUN SETDIFF (SET1 SET2)).</td>
</tr>
<tr>
<td>4.</td>
<td>SS decides to code SETDIFF by rearranging INTERSECTION1 code.</td>
</tr>
<tr>
<td>5.</td>
<td>SS decides to code simple cases found in INTERSECTION1.</td>
</tr>
<tr>
<td>6.</td>
<td>SS considers case (NULL SET1), decides action will be NIL. Code is now (DEFUN SETDIFF (SET1 SET2) (COND ((NULL SET1) NIL)).</td>
</tr>
<tr>
<td>7.</td>
<td>SS considers case (NULL SET2), decides action will be SET1. Code is now (DEFUN SETDIFF (SET1 SET2) (COND ((NULL SET1) NIL) ((NULL SET2) SET1)).</td>
</tr>
<tr>
<td>8.</td>
<td>SS formulates plan to check each element of SET1 to see if it is NOT a member of SET2. Gives up on this plan.</td>
</tr>
<tr>
<td>9.</td>
<td>SS decides to code the relation MINUS (SET1, INTERSECTION (SET1 SET2)). Realizes that MINUS is equivalent to SETDIFF and gives up on this plan.</td>
</tr>
<tr>
<td>10.</td>
<td>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.”</td>
</tr>
<tr>
<td>11.</td>
<td>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))).</td>
</tr>
<tr>
<td>12.</td>
<td>SS consider case in which (CAR SET1) is not a member of SET2. Formulates plan to add (CAR SET1) to the answer for SETDIFF.</td>
</tr>
<tr>
<td>13.</td>
<td>SS decides to look at INTERSECTION1 code again. SS notes that 4th action of INTERSECTION maps onto 3rd action of SETDIFF; pondered whether 3rd action of INTERACTION 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))))).</td>
</tr>
<tr>
<td>14.</td>
<td>SS checks code visually and on the computer.</td>
</tr>
</tbody>
</table>
Section of their text:

SPECIAL CASES:

1. Set difference and intersection of S₁ and S₂ is S₁ \ S₂ if S₁ is a proper subset of S₂. In this case, if an element x is in S₁, it is not in S₂, so x is in S₁ \ S₂.

2. Set difference and intersection of S₁ and S₂ is S₂ \ S₁ if S₁ is a proper subset of S₂. In this case, if an element x is in S₂, it is not in S₁, so x is in S₂ \ S₁.

3. Set difference and intersection of S₁ and S₂ is S₁ ∩ S₂ if S₁ and S₂ are equal. In this case, if an element x is in S₁, it is also in S₂, so x is in S₁ ∩ S₂.

SPECIAL CASES:

1. If the given solution contains the empty set, then the solution contains the empty set as the only element.

2. If the given solution contains the universal set, then the solution contains the universal set as the only element.

SPECIAL CASES:

1. If the given solution contains the empty set and the universal set, then the solution contains the empty set and the universal set as the only elements.

SPECIAL CASES:

1. If the given solution contains only the empty set, then the solution contains only the empty set.

2. If the given solution contains only the universal set, then the solution contains only the universal set.

PROCEDURE:

1. If the given solution contains both the empty set and the universal set, then the solution contains both the empty set and the universal set.

2. If the given solution contains neither the empty set nor the universal set, then the solution contains a single element.

PROCEDURE:

1. If the given solution contains the empty set, then the solution contains only the empty set.

2. If the given solution contains the universal set, then the solution contains only the universal set.

PROCEDURE:

1. If the given solution contains only the empty set, then the solution contains only the empty set.

2. If the given solution contains only the universal set, then the solution contains only the universal set.

PROCEDURE:

1. If the given solution contains both the empty set and the universal set, then the solution contains both the empty set and the universal set.

2. If the given solution contains neither the empty set nor the universal set, then the solution contains a single element.

PROCEDURE:

1. If the given solution contains the empty set, then the solution contains only the empty set.

2. If the given solution contains the universal set, then the solution contains only the universal set.

PROCEDURE:

1. If the given solution contains only the empty set, then the solution contains only the empty set.

2. If the given solution contains only the universal set, then the solution contains only the universal set.
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.

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 structural
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

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
mismatch of recursive functions as performing iteration is common
among novices (see also Kahn, 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 1/2 hour, and POWERSET 3/2 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
FUNCTIONS AND PROBLEM SOLVING

2.1 Functions in Problem Solving

This section introduces a programming paradigm that is particularly effective for solving problems that involve a large number of functions. The paradigm is called functional programming, and it is based on the use of pure functions. A pure function is a function that takes inputs and returns outputs without changing any state or modifying any external variables. This makes it easier to reason about the function's behavior and to test it for correctness.

A key aspect of functional programming is the use of higher-order functions, which are functions that take other functions as arguments or return functions as results. This allows for more expressive and flexible programming, as well as easier composition of tasks.

In this section, we will explore the use of functions in problem solving, and we will see how functional programming can be used to solve a variety of problems. We will also discuss some of the key principles of functional programming, such as immutability and referential transparency.

2.2 Recursion in Problem Solving

Recursion is a fundamental concept in functional programming, and it is used to solve problems that involve a large number of steps or operations. In a recursive function, the function calls itself repeatedly, with a smaller problem size on each call, until a base case is reached.

Recursion is useful for solving problems that involve a tree-like structure, such as traversing a file system or manipulating a list. It can also be used to solve problems that involve a loop-like structure, such as calculating the factorial of a number or generating all permutations of a list.

In this section, we will explore the use of recursion in problem solving, and we will see how recursion can be used to solve a variety of problems. We will also discuss some of the key principles of recursion, such as termination and avoidance of infinite recursion.

2.3 Higher-Order Functions

Higher-order functions are a key feature of functional programming, and they allow for more expressive and flexible programming. A higher-order function is a function that takes one or more functions as arguments or returns a function as a result.

Higher-order functions can be used to compose tasks, to apply a function to a list of values, or to create a new function from existing functions. They are particularly useful in functional programming, where the focus is on the manipulation of functions and data rather than on the manipulation of state.

In this section, we will explore the use of higher-order functions in problem solving, and we will see how they can be used to solve a variety of problems. We will also discuss some of the key principles of higher-order functions, such as closure and currying.

2.4 Functional Data Structures

Functional data structures are a key component of functional programming, and they are used to represent and manipulate data in a functional way. A functional data structure is a data structure that can be transformed without changing the underlying data.

Functional data structures are particularly useful in functional programming, as they allow for more efficient and expressive manipulation of data. They are also easier to reason about and to test for correctness.

In this section, we will explore the use of functional data structures in problem solving, and we will see how they can be used to solve a variety of problems. We will also discuss some of the key principles of functional data structures, such as immutability and referential transparency.
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 subroutine 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 slightly 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 INTERSECTION-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

<table>
<thead>
<tr>
<th>Schematic Protocol of Subject JP's TUNNLE Solution and Attempt to Code LINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. JP starts with an (incorrect) nonrecursive solution.</td>
</tr>
</tbody>
</table>
|   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: X = 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. |
The problem in the previous equation is that the equation returns the same value for different inputs. One solution is to use a logarithmic function in the equation to reduce the sensitivity of the results. This can be achieved by using a logarithmic scale for the variables in the equation. For example, if the equation is $y = ax^2 + bx + c$, we can transform it to $y = a\log(x^2) + b\log(x) + c$. This transformation will reduce the sensitivity of the results, as the logarithmic function will return small values for small inputs and larger values for larger inputs.

In the next section, we present an analysis of a simple example where we have used logarithmic transformations to reduce the sensitivity of the results.
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 were on manipulating a stored database of 18 entries in a 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 ensure 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

---

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 SUBI will be useful.

4. AD formulates an iterative plan to test each number less than and including the input variable until a non-science 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. AD decides to write out the conditional structure and return 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 SUBI) 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 SUBI 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 (SUBI ID) can be found.

11. Using her assumption, AD writes the recursive call:

    IF (ID ISA? SCIENCE) THEN (ID PRE (XYZ (SUBI 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 (SUBI ID))),
    ELSE []

13. AD inserts the first line of the function:

    XYZ ID IS
    IF (ID ISA? SCIENCE) THEN (ID PRE (XYZ (SUBI ID))),
    ELSE []

*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 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 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 analoguous solution. “We made an assumption earlier that it had SORTed the previous [items of a list].”

Mapping the analoguous 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, that would mean that I would have the highest number [at the end of the list].”

* 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:

```plaintext
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 analysis productions which structure-map the following representation of one of the SORT recursive cases onto the XYZ function:

```
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).

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

```
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.
```
Learning to Compute

The function, which we call the transfer function, is defined as:

\[ f(x) = \frac{1}{1 + e^{-x}} \]

This function, known as the logistic function or sigmoid function, maps any real-valued number to a value between 0 and 1. It is often used in artificial neural networks as an activation function.

The process of training a neural network involves adjusting the weights of the connections between neurons to minimize the difference between the network's output and the desired output. This is done through an algorithm called backpropagation, which computes the gradient of the loss function with respect to the weights and updates the weights in the direction that minimizes the loss.

In practice, the transfer function is applied to the weighted sum of the inputs to each neuron. The output of this function, along with the outputs of the other neurons in the same layer, is then passed as input to the next layer. This process is repeated until the output layer is reached, and the final output of the network is compared to the desired output to calculate the loss.

The gradient of the loss function with respect to the weights is used to update the weights using a method called gradient descent. The learning rate determines the size of the steps taken in the weight update process.

This iterative process of forward propagation, loss calculation, gradient computation, and weight update is repeated many times, gradually improving the network's performance on the given task.
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 suggest that the structure group got to this state in a more efficient manner because they had learned a general strategy 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
