Learning to Program Recursion

Peter L. Pirolli, John R. Anderson, and Robert Farrell

Carnegie-Mellon University

Learning to program recursive functions in languages like LISP is notoriously difficult. Indeed, a primary mark of expertise in such languages is the ability to plan and code recursive functions. Recently, we have performed protocol studies of students learning to program recursion in LISP and LOGO as well as controlled experiments on learning recursion in a simple programming language. We have used the GRAPES production system model (Anderson, Farrell & Sauers, 1984) to address these results. GRAPES not only models programming performance but also learning by doing by the mechanism of **knowledge compilation**. Knowledge compilation summarizes extensive problem-solving operations into new compact production rules (see Anderson, Farrell, & Sauers, 1984; Neves & Anderson, 1981 for details)

Characteristics of Learning to Program Recursion

In Anderson, Pirolli and Farrell (1983), we hypothesized that initial performance and learning in recursive programming is primarily driven by learning from examples, since students have little relevant prior knowledge. We further hypothesized that such learning consists of two components. First, students can use the solution to a given example as an outline which must be modified in order to solve a current problem (**probelm-solving by analogy**). Second, learning (knowledge compilation) mechanisms can summarize these analogy operations into new problem-solving operators which can apply to future problems (**learning from analogy**).

To illustrate problem-solving by analogy and learning from analogy we present a GRAPES simulation of a subject, SS, in her initial encounters with coding recursive LISP functions. At the time, SS had about 15 hours of tutoring and programming experience in LISP. The first function that SS wrote was SETDIFF, which returned all the elements of the one list not contained in a second list. In coding SETDIFF, SS analogized from a textbook example function, INTERSECTION1, which returned all elements that were common to two input lists. The primary structure of SETDIFF and INTERSECTION1 consists of a series of if-then conditional statements (see Figure 1).

The GRAPES simulation (Figure 2) of SS when provided with a representation of the INTERSECTION1 code at multiple levels of abstraction, a specification of the SETDIFF relation and a goal to code SETDIFF used following analogy production:

```
P1: 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 sets goals to first check the similarity of the specifications of INTERSECTION1 and SETDIFF and then map the code structure of INTERSECTION1 onto the SETDIFF code. A set of comparison productions then found that both INTERSECTION1 and SETDIFF take two input sets and can be code recursively.

Next, **structure-mapping** productions map conditional clauses from INTERSECTION1 to SETDIFF. SS gave clear evidence in her protocol of performing the same mapping. For each INTERSECTION1 conditional clause. SS (and GRAPES) mapped the condition of the clause onto SETDIFF and determined what action should take place. Like SS, GRAPES fluctuated the level of abstraction at which these conditional clauses were mapped. For instance, initially GRAPES attempted to map the condition "test if an element should be added to the result" When this

failed, it mapped a more literal translation: "test if the first element of the first list is a member of the second list".

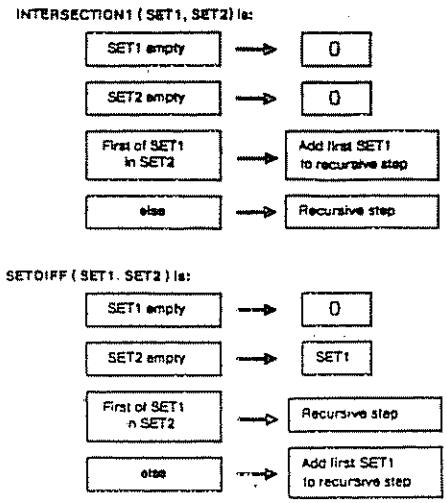

Figure 1: Abstract code specifications

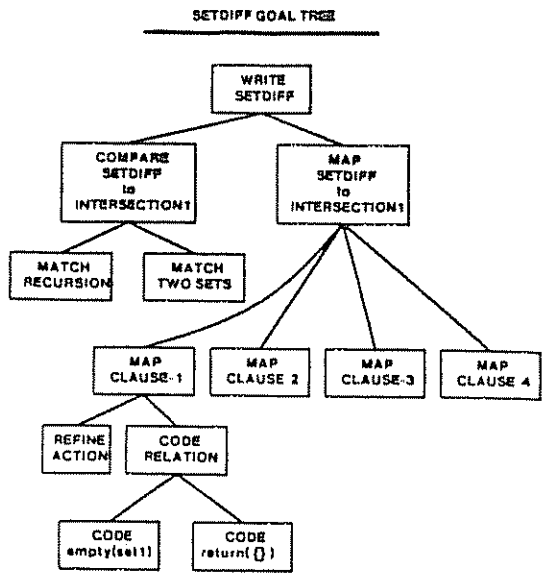

Figure 2: Part of the GRAPES solution for SETDIFF

A number of new GRAPES productions were produced by knowledge compilation in coding SETDIFF. However of primary interest is a rule which was produced by compiling the analogy processes:

```
C1:  IF the goal is to code
          a recursive relation on two sets SET1 and SET2
     THEN code a conditional and set as subgoals to
        1) Refine & code a clause
           to deal with the case when SET1 is empty
        2) Refine & code a clause
           to deal with the case when SET2 is empty
        3) Refine & code a clause
           to deal with the case when the first
           element of SET1 is a member of SET2
        4) Refine & code a clause
           to deal with the else case
```

In the SETDIFF example we see both problem-solving by analogy and learning by analogy. First, the solution of SETDIFF was heavily guided by the INTERSECTION1 example and second we see the acquisition of a new production, C1, summarizing this analogy process. This operator can only be successfully applied to a small domain of recursive functions. Both GRAPES and SS successfully used this production on the next function attempted, SUBSET, which determines whether one list is a subset of another by recursion. However, both SS and GRAPES had great difficulty in solving the next recursive function in the instructional series, POWERSET, which computes the set of all subsets of a set. This difficulty results from the inapplicability of production C1 in the POWERSET case. The analogy processes executed in SETDIFF were at a sufficiently abstract level that they generalized to SUBSET when compiled into C1. However

they were not abstract enough to generalize to POWERSET. This leads to the conjecture that transfer of learning from analogy is limited by the level of abstraction at which the analogy is initially carried out.

```
TO TUNNEL :X
SQUARE :X
IF :X = 50 THEN STOP
TUNNEL :X + 10
```

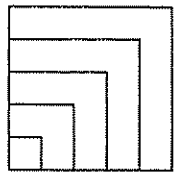

Figure 3: The TUNNEL function

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

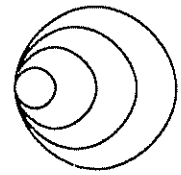

Figure 4: CIRCLES function

A further illustration of how the level of abstraction of an analogy impacts on generalization comes from a protocol of an eight year old student (J) coding her first recursive program in LOGO. The student's background consisted of a semester and a half of weekly LOGO lessons. J's first problem was a function, TUNNEL, which drew concentric squares on the computer screen (Figure 3). Her coding was guided by an example function CIRCLES which drew concentric circles (Figure 4). Unlike subject SS who basically mapped the conditional structure of INTERSECTION1 onto SETDIFF, subject J basically mapped the actual code of CIRCLES onto TUNNEL. The only portions of the CIRCLES code that J did not map at the literal level was the name of the function (CIRCLES changed to TUNNEL) and a subfunction called by the program (CIRCLES calls a circle drawing program while TUNNEL calls a square drawing program. The production rule produced by GRAPES by compilation of the CIRCLES - TUNNEL analogy is:

```
C2:  IF the goal is to code a figure
        and the function for figure is called <NAME>
        and a repeated subfigure of the figure is coded by <SUBFUNCTION>
     THEN write
          TO <NAME> :X
          <SUBFUNCTION> :X
          IF :X = 42 THEN STOP
          <NAME> :X + 10
```

In contrast to production C1, the condition of production C2 matches to relatively spurious program specifications and its actions largely specify literal code rather than an abstract plan. The fact that subject J had great difficulty writing subsequent recursive programs seems to corroborate the view that she had failed to learn any operators of even limited generality for coding recursion from this analogy episode.

The GRAPES simulations of programming recursion also suggest that learning to program recursion occurs by a process of piecemeal approximation. For example, the operator producing code for a tail-recursive call is compiled when coding SETDIFF, while an operator for combining the result of a recursive call with another result is not learned until GRAPES codes POWERSET. The protocols of subject SS support this analysis: after coding SETDIFF, SS has no problem coding a tail-recursive call in SUBSET, but does not initially know how to combine results of recursive calls with other results in POWERSET. However, after coding POWERSET, SS successfully wrote code combining recursive results with other results.

Conceptual Models of Recursion

Our protocols of LISP subjects suggest that novices typically view recursive functions with a "flow of control" mental model (see also Kahney, 1982). In attempting to plan code for a recursive function, subjects will frequently simulate or trace the flow of control of the function and its recursive calls. Using such a model generally leads to errors because such tracing is

often complicated and involves keeping track of many partial results. In addition, such a model does not readily map onto program generation since it is not specifically a model of how to write a recursive function: the flow of control model describes how already written functions are evaluated.

We have modelled in GRAPES what is arguably an ideal strategy for coding recursive functions. This strategy has the following "formal model" of code generation for recursive programs: "A recursive function consists of (a) one or more terminating cases in which a simple answer is returned and (b) 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 same problem (a recursive call) has been solved."

We hypothesized that subjects instructed with the formal model would learn to code recursive functions faster than those using the flow of control model. We conducted an experiment in which subjects initially learned the basic functions, predicates, conditionals, and definitional forms of a simple programming language modelled after LISP. One group of subjects was then introduced to recursive functions using the formal model, another group's instructions for recursion emphasized the flow of control model. All subjects were then presented with four recursive program specifications (one at a time) for which they had to write code. Subjects were re-presented with specifications until they had written a correct program for each specification. The formal model group took significantly less time (**M** = 3444 seconds) than the flow of control group (**M** = 5116 seconds) to code all functions correctly.

Implications for Intelligent Tutoring Systems

Our efforts to design intelligent computer-aided instruction (ICAI) system for programming (and especially recursion) has been influenced by the current results. First, our ICAI system avoids examples since our GRAPES model suggests that although problem-solving by analogy can facilitate initial performance it may not necessarily facilitate learning. Second, our model suggests that students learn recursion in a piecemeal fashion. The ICAI system thus presents a wide variety of recursion problems to expose students to a large range of coding patterns. Finally, our GRAPES ideal model and experimental results indicate that a formal mental model of recursive programs facilitates learning because it reduces working memory load and more directly maps onto program generation than the flow of control model typically used by novices. This conceptual model and the ideal programming model are employed by the ICAI system in teaching recursion.

References

Anderson, J.R., Farrell, R., & Sauers. (1984) Learning to program in LISP. **Cognitive Science** in press.

Anderson, J.R., Pirolli, P.L., & Farrell, R. (1983, October). **Learning to program recursive functions**. Paper presented at the Conference on Expertise, University of Pittsburgh, Pittsburgh, PA.

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

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

Acknowledgements: This research was supported by an IBM Fellowship to Peter Pirolli and Office of Naval Research grant No. N00014-81-C-0335 to John Anderson.