DOCUMENT RESUME

ED 265 177 TM 860 024

AUTHOR Pirolli, Peter L.; Anderson, John R.
TITLE The Role of Mental Models in Learning to Program.
SPONS AGENCY Office of Naval Research, Arlington, Va. Personnel and Training Research Programs Office.
PUB DATE Nov 84
CONTRACT N00014-84-K-0064
NOTE 21p.; Paper presented at the Annual Meeting of the Psychonomic Society (25th, San Antonio, TX, November 1-3, 1984).
PUB TYPE Speeches/Conference Papers (150) -- Reports - Research/Technical (143)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Cognitive Structures; *Computer Assisted Instruction; *Computer Science Education; Computer Simulation; Higher Education; Hypothesis Testing; Learning Theories; *Models; *Problem Solving; *Programing
IDENTIFIERS *GRAPES Production System Model; LISP Programing Language; *Recursive Programing; SIMPLE Programing Language

ABSTRACT

This study reports two experiments which indicate that the processes of providing subjects with insightful representations of example programs and guiding subjects through an "ideal" problem solving strategy facilitate learning. A production system model (GRAPES) has been developed that simulates problem-solving and learning in the domain of writing recursive functions. In the first experiment, a mental model of recursion that employed the given representation (structure model) was contrasted with a model of recursion that emphasized how recursive functions are evaluated (evaluation model). Two groups of subjects were tested using these models, and, in the training phase, both groups reached the same level of proficiency. However, data suggest that the structure model group reached this level in a more efficient manner, having learned a general strategy for structuring their code very early on in the training phase. In the second experiment, members of the GRAPES research group implemented and tested a computer-based system for tutoring LISP. Students of a LISP programming course were divided into two groups, one that interacted with the LISP tutor and another that worked in a standard LISP environment. Overall performance for students interacting with the LISP tutor was superior to those who did not interact with the tutor. (Author/LMO)

The Role of Mental Models in Learning to Program

Peter L. Pirolli
and
John R. Anderson

Department of Psychology
Carnegie-Mellon University
Pittsburgh. PA 15213

Presented at the Twenty-fifth Annual Meeting of Psychonomic Society. San Antonio. TX November 1984. 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 and by an IBM Research Fellowship to Peter Pirolli

Abstract

A production system model (GRAPES) has been developed that simulates problem-solving and learning in the domain of writing recursive functions. Protocol analyses and simulations by the model suggest that students typically use representations of example program solutions to guide their problem-solving on initial recursion problems This process of problem-solving by analogy to examples leads to acquisition of new production rules that generalize across example and target problem features Two experiments are reported which indicate that providing subjects with insightful representations of example programs and guiding subject through an "ideal" problem-solving strategy facilitates learning

The Role of Mental Models in Learning to ı ŕogram

Over the past few years the GRAPES research project at Carnegie-Mellon has concerned itself with specifying a detailed process model of the development of problem-solving skill in programming (Anderson. Farrell. & Sauers. 1984: Anderson. Pirolli. & Farrell. in press: Pirolli & Anderson. in press) Our theory of problem-solving and learning of programming was developed in the context of the GRAPES production system (see Sauers & Farrell. 1982. for details) which was designed to emulate certain aspects of the ACT* theory of cognitive architecture (Anderson. 1983). In this paper. we present some of our findings for a subset of programming. namely learning to program recursive functions. We will focus on four issues: (a) the process of writing programs by analogy to examples. (b) the formation of generalizations from analogy processes. (c) which representations (i.e., mental models) of program examples facilitate learning. and (d) how guided use of such mental models in problem-solving facilitates learning.

Recursive functions are ones that are defined in terms of themselves A standard example of recursion in mathematics is the factorial function. $f(n) = n \times f(n-1)$. or $n > 0$ (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 factorial 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 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 et al 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

So how do students learn the unfamılıar procedure of generating recursıve programs? Our hypothesis ıs that the prımary means avaılable to students is learning from examples. By thıs we mean two things. Fırst. students solve initial problems by modifying the solutions of examples they are gıven. Second. learning mechanisms summarize solutions to these initial problems ınto new problem-solving operators which can apply to future problems.

An "Ideal" Strategy for Coding Recursion

Before dıscussıng how novıces program recursion. we present what is arguably the ıdeal strategy for codıng recursıve functıons. This strategy ıs based on protocol analyses of expert programmers (see Anderson et al. ın oress) Figure 1 presents a ***hierarchical goal tree*** representıng the problem-solvıng goals our GRAPES simulation will step through in executing thıs general strategy Each box ın Figure 1 represents a programming goal. Arrows show the decomposıtıon of goals ınto subgoals. The strategy depicted in Figure 1 ınvolves (a) refınıng the semantıcs and coding the terminating cases of the function and (b) refining the semantics and coding the recursıve cases. This latter step ınvolves a set of subgoals for (a) characterizıng the result of a recursive call (e.g.. $f(n\text{-}1)$ for the factorıal function). (b) characterızıng the result of the functıon (e g.. $f(n)$) and (c) determinıng the relationship between (a) and (b) (e.g.. $f(n) = n \times f(n\text{-}1)$)

It ıs noteworthy that most standard texts on programmıng do not gıve any ınstructıon that that suggests thıs general strategy for codıng recursıve functıons. Typıcally. texts descrıbe how recursive functıons work. gıve lots of examples and mav offer general consideratıons (e g.. 'start with the easıest cases') A lack of ınstruction ın codıng strategy ıs one of the many hurdles that students face ın learnıng about the unfamılıar procedure of recursıon (Anderson et al ın press)

Writing Recursive Functions by Analogy to Examples

Our GRAPES model of learning to program recursion is largely based on protocol analyses of six novices. A ubiquitous phenomenon among these subjects is the use of recursive program examples to guide solutions to the first recursion problems encountered Our GRAPES model of problem-solving by analogy to examples consists of a set of production rules which. (a) establish a partial match of the current problem features to those of the example and (b) map features of the example solution to solution steps in the 'current problem. This approach to analogy shares many common features with other recent theories of problem-solving by analogy (Carbonell. 1983: Gick & Holyoak. 1980. 1983.. Holyoak in press)

To illustrate this process more concretely. we will briefly consider portions of a GRAPES simulation of a subject (SS) solving her first recursion problem. SS was learning from Siklossy's (1976) text "Let's Talk LISP". Her first problem was to write SETDIFF. a function that takes two lists and returns all the members of the first list that are not in the second list Her solution was heavily guided by a recursion example that immediately preceded the SETDIFF problem in the text. This example was INTERSECTION1 a function that takes two lists and returns all elements that occur in both lists The INTERSECTION1 example consisted of four conditional clauses The logic of the function s presented in Figure 2: 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 to INTERSECTION1 otherwise just return the result of the recursive call

SS clearly stated that she was using the INTERSECTION1 conditional clauses as a guide for the SETDIFF solution SS's protocol suggested that she used a hierarchical represention of the INTERSECTION1 conditional clauses (see Figure 3). Our GRAPES model when presented with the goal to write SETDIFF and the reoresentation of INTERSECTION1 illustrated in Figure 3 performs the same basic problem-solving steps as subject SS. A portion of the goal tree developed by GRAPES in solving SETDIFF is presented in Figure 4. GRAPES first performed a partial match of INTERSECTION1 and SETDIFF: both are recursive functions and take two sets. Next, GRAPES mapped the conditional clauses of INTERSECTION1 onto code for SETDIFF.

The code for INTERSECTION1 and SETDIFF differs in many respects and several solution mappings made by SS failed on first attempt Problem-solving by analogy is not a straightforward copying of code and often requires search for the right example representations to mao from example to target problem.

Generalization from Problem-Solving by Analogy

The GRAPES model learns by creating new production rules based on problem-solving experience using the mechanisms of ***knowledge compilation*** (Anderson, 1982; Anderson, Farrell, and Sauers, 1984; Neves & Anderson, 1981). Essentially, knowledge compilation creates production rules that summarize several problem-solving steps and that no longer make reference to example information. The interaction between these learning mechanisms and problem-solving by analogy is illustrated in Figure 5. If example features f_1, f_2, . . f_n are matched to features f_1', f_2', . . . f_n' of the target problem and example solution components s_1, s_2, .. s_n are mapped to target solution steps s_1', s_2' . . . s_n', then GRAPES will learn a new production rule of the form IF features f_1' & f_2' & ... & f_n' THEN perform s_1' & s_2' & ... & s_n'.

One of the productions learned by our GRAPES simulation of SS for example is

C1 IF the goal is to code a recursive function on two sets SET1 and SET2
THEN code a conditional structure

and set subgoals to code four conditional clauses
1. when SET1 is empty
2. when SET2 is empty
3. when the first element
   of SET1 is a member of SET2
4. the else case.

Production rule C1 is a compilation of the problem-solving steps outlined in Figure 4. It essentially states that a recursive function taking two sets should be coded by four conditional clauses Our protocol analyses and simulations of subsequent recursion problems coded by SS (see Anderson et al.. in press: Pirolli & Anderson. in press) suggest that SS had learned C1. Production C1 can be used successfully to code some. but by no means all. recursive functions. This production sets up a plan that has little in common with the strategy outlined in Figure 1. To a large extent. SS's difficulties with later recursion problems can be traced to her mapping of a poor representation of INTERSECTION1 onto her SETDIFF solution (see Anderson et al.. in press: Pirolli & Anderson. in press). In the next section we outline how altering subjects representions of program examples can facilitate learning to program recursion

The Effects of Mental Models of Programs on Learning Recursion

Our protocol analysis and simulations of novices indicate that the particular example representations used by students in problem-solving by analogy have a large impact on the early learning of programming recursion If students would only use the "right" representations in analogy then we would expect to see rapid learning What are the "right" representations? Our hypothesis is that the right representation encodes the problem in terms of the general concepts needed to define the general strategy for coding recursion (see Figure 1) Such a representation would encode recursive functions as consisting of terminating cases and recursive cases The representation would also have to include the notion that the results of recursive cases (e.g. *f(n)*) are obtained by assuming that the results ot recursıve calls (e g.. *f*(*n*-1)) can be found.

In a recent experiment. we tested our hypothesıs that provıding students wıth the above representations would facilitate learning. We contrasted a mental model of recursıon that employed the above representatıon (***structure model***) with a model of recursıon :hat emphasızed how recursive functions are evaluated (***evaluation model***). As noted before. thıs evaluation model corresponds to the standard model taught in programming texts.

Two groups of subjects learned the basic functions. predicates. conditional structures and definitional syntax of a LISP-like language called SIMPLE (Shrager & Pirolli. 1983). All programmıng tasks centered on manipuiating a stored database of 18 entries ın a book library. The entries ın thıs database could be identified by a number (id number). a key word (title). and could be categorızed as scıence. religion. or fiction books. All recursıon problems came from a space of 16 functıons characterized by four dimensıons wıth two values on each dimensıon Each functıon could: (a) take a list of titles or an id number as ınput. (b) return a list of scıence or non-scıence items. (c) return the output list with ıtems ın the same or the reverse of the order they are encountered in recursıon. (d) skıp ıtems that are the opposıte of what ıs beıng collected or return the current accumulated result when first encountering an opposıte

One group of subjects (structure group. N = 10) was presented with ınstructıon emphasızing the structure model of recursıon. Thıs ınstructıon ıncluded the followıng descrıptıon:

> A recursive function definition consists of two components: (1) A definition of one or more terminating conditional statements in which a sımple 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 non-programming 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 subjects did not use the actual code of this example to analogize from, we removed the SORT code from view (leaving the general description of recursion and the mathematical example at subjects' disposal).

The second group (evaluation group, N=9) received a set of instructions paraphrased from a LISP text that emphasized the evaluation model of recursion. These instructions included the following description.

> **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 evaluation 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 had to first write four recursive functions correctly with feedback for errors (training phase) 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 correctly write their first four functions in the training phase (*M* = 57 4 min) than evaluation 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 reached the same level of proficiency. However. our data suggests 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

Further Facilitation of Learning Recursion: Stepping Students through the "Ideal" Model

The SIMPLE experiment illustrates the advantages of having an insightful mental model of recursion to guide problem-solving by analogy. However. as we mentioned with reference to subject SS's performance mapping a representation of an example program is not a straightforward process. Further verbal specification of how to think about programs is usually open to misunderstanding on a student's part. Our GRAPES learning theory predicts that a more direct approach to teaching programming involves guiding the student's problem-solving steps along correct solution paths during the act of program writing itself (Anderson. Boyle. Farrell. & Reiser. 1984). Not only must students have an insightful mental model they must be stepped through appropriate use of the model in problem-solving. In the context of learning recursion. this involves stepping the student through the general strategy outlined in Figure 1.

Recently members of the GRAPES research group[*] implemented and tested a computer-based system for tutoring LISP At the heart of this LISP tutor is a GRAPES production system model of "ideal" strategies for solving programming problems The LISP

[*] [illegible] Robert Farrell, Brian Reiser, [illegible] the authors

tutor uses this model to determine if a student's programming behavior is on a correct solution path. and to generate tutorial interventions (see Anderson. Boyle. Farrell & Reiser 1984 for details of this system).

Figure 6 presents a view of the LISP tutor on a terminal screen. The student types code directly into the middle window. Queries and explanations from the LISP tutor appear in the top window. As the student types code to solve a programming problem. the tutor compares the code to it's internal GRAPES model. For the most part. if the student is on a correct solution path, the tutor remains silent. At critical design points (for example. designing the recursive cases of a recursive program) the tutor will intervene. presenting examples. queries and explanations to guide the student through a program design. In addition to the "ideal" models for program solutions. the LISP tutor also knows about common mistakes made by students and the underlying causes of those mistakes. When such student "bugs" are recognized. the tutor intervenes by asking questions or giving explanations that lead back to the correct solution path

In a recent test of the LISP tutor. students of a LISP programming course were divided into two groups. one that interacted with the LISP tutor (N = 10) another that worked in a standard LISP environment (N = 10) Members of these groups were matched on prior programming experience grades on a prerequisite PASCAL course and SAT scores Both groups received the same texts lectures. and solved the same problems In a test of programming skill (coding. debugging. and evaluating LISP functions) presented immediately prior to learning recursion there was no significant difference in test scores

The recursion section of this course consisted of 18 problems having a wide range of difficulty The text used by both groups for recursion emphasized the structure model of recursion outlined in the previous section However overall performance for students interacting with the LISP tutor was superior to those who did not interact with the tutor The LISP tutor group took significantly less time (*M* = 5.76 hours) to code the 18 recursion problems than the non-tutored group (*M* = 9.01 hours). Further. the LISP tutor group scored higher on a test of coding. debugging. and evaluating recursive functions. (*M* = 7 60 out of a possible 14 points) than the non-tutored group (*M* = 4.78). Although all students were instructed with an insightful mental model of recursion. those who were guided in using this model in problem-solving achieved a higher level of programming proficiency and got to that state in less time.

Summary

Our analysis of learning recursion suggests the following conclusions:

1. Because recursion is a novel and difficult concept. subjects typically use representations of example solutions to guide their solutions for the initial recursion problems they encounter.
2. Problem-solving by analogy leads to the learning of new production rules that generalize across the example and target solutions.
3. Learning recursion can be facilitated by instructing students in a mental model of recursion that emphasizes the key concepts necessary for a general strategy for coding recursive functions Students use this model to represent example solutions map this representation onto a target problem. and knowledge compilation summarizes this mapping into new productions that generate the general strategy for coding recursion.
4. Learning recursion can be further facilitated by guiding students directly along the correct solution path predicted by the GRAPES model of the general strategy for coding recursion.

References

Anderson. J.R. (1982). Acquisition of proof skills in geometry. In J.G. Carbonell. R. Michalski, and T. Mitchell (Eds.) *Machine learning: An artificial 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.. Boyle. C.F.. Farrell. R.. & Reiser. B.J. (1984). *Cognitive principles in the design of computer tutors* (Tech. Rep. ONR-84-1). Pittsburgh: Carnegie-Mellon University.

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 (Eds.) *The nature of expertise*

Carbonell. J.G (1982). Derivational analogy and its role in problem-solving *Proceedings of the National Conference on Artificial Intelligence.* Washington. D C. University of Maryland & George Washington University.

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

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

Holyoak. K.J. (in press). The pragmatics of analogical transfer *Advances in the psychology of learning and motivation*

Neves. D. M. and 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.

Pirolli P L & Anderson. J R (in press) The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*

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.

Figure 1: The hierarchical 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

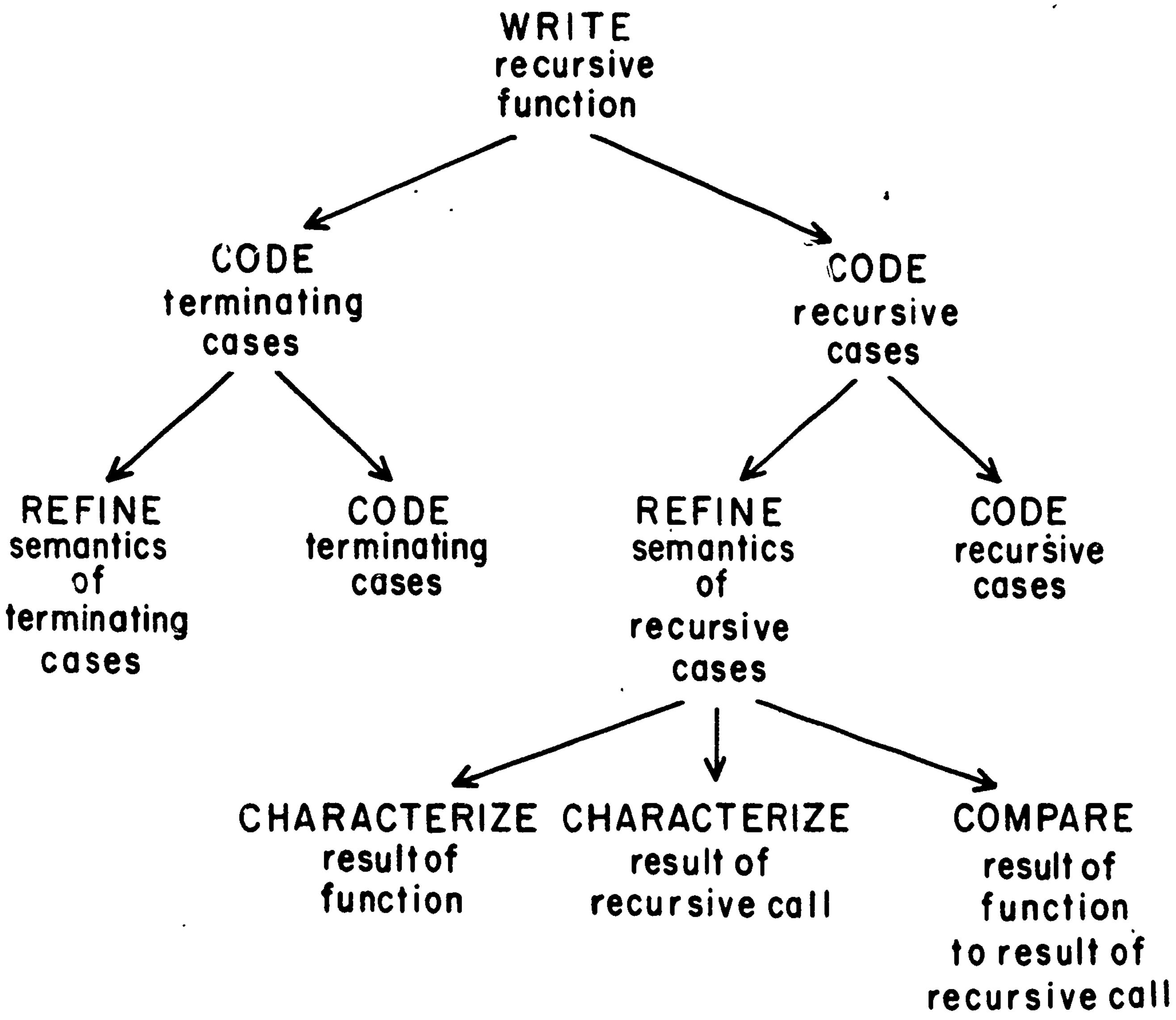

WRITE recursive function
CODE terminating cases
CODE recursive cases
REFINE semantics of terminating cases
CODE terminating cases
REFINE semantics of recursive cases
CODE recursive cases
CHARACTERIZE result of function
CHARACTERIZE result of recursive call
COMPARE result of function to result of recursive call

Figure 2: The iogic of the conditional structures of the INTERSECTION1 and SETDIFF functions. Each arrow points from a condition to an action of a conditional clause.

```
INTERSECTION1 (SET1, SET2) IS
          SET1 empty      →      ::

          SET2 empty      →      ::

          First of        →      Recursive step
          SET1 in SET2

          ELSE            →      Add first of SET1
                                 to recursive step


SETDIFF (SET1. SET2) IS
          SET1 empty      →      ::

          SET2 empty      →      SET1

          First of        →      Add first of SET1
          SET1 in SET2           to recursive step

          ELSE            →      Recursive step
```

Figure 3: A portion of the hierarchical representation of the INTERSECTION1 example used by SS in solving SETDIFF

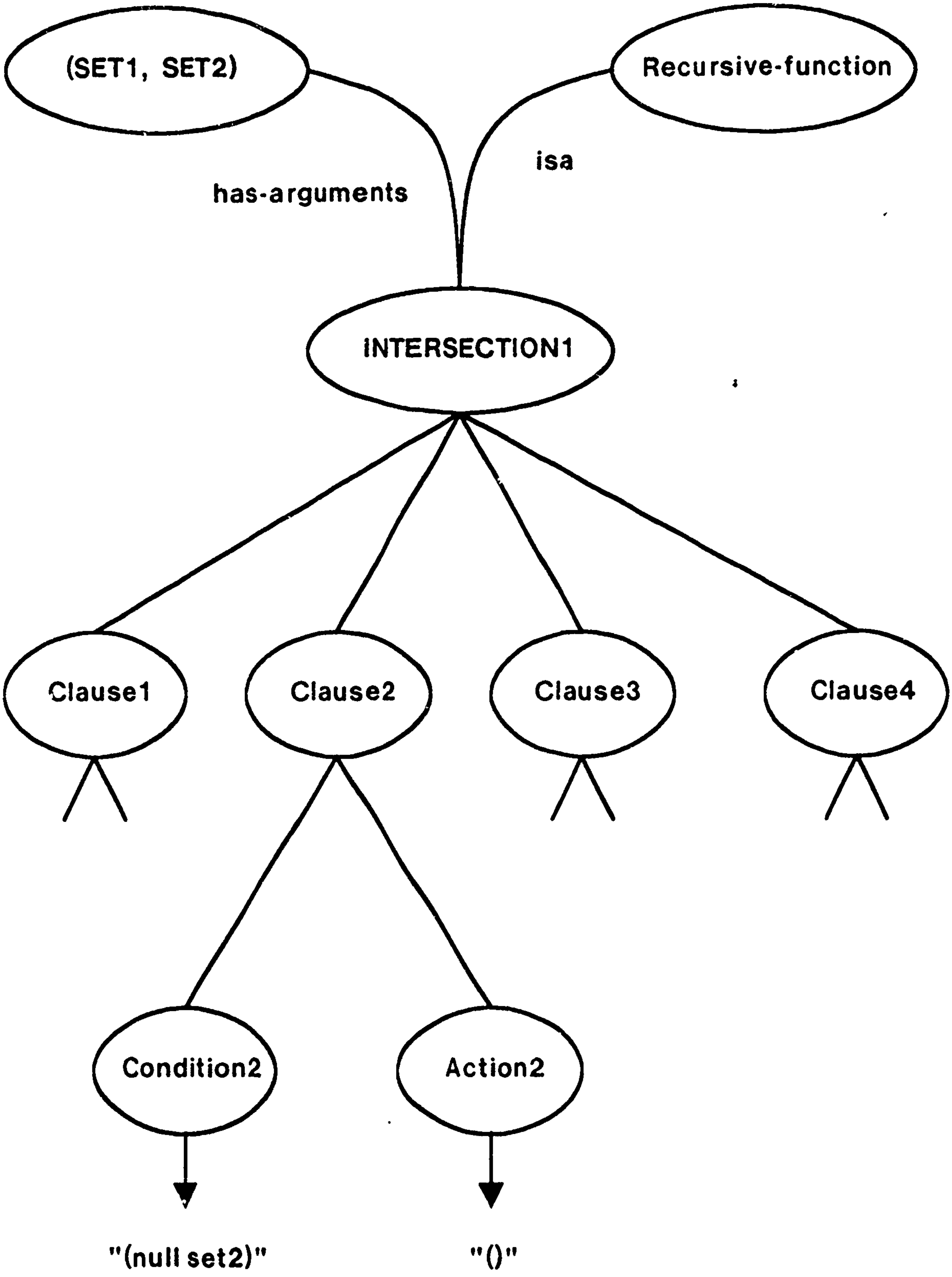


Figure 4: A portion of the hierarchical goal produced by GRAPES in simulating SS's solution of SETDIFF by analogy to INTERSECTION1

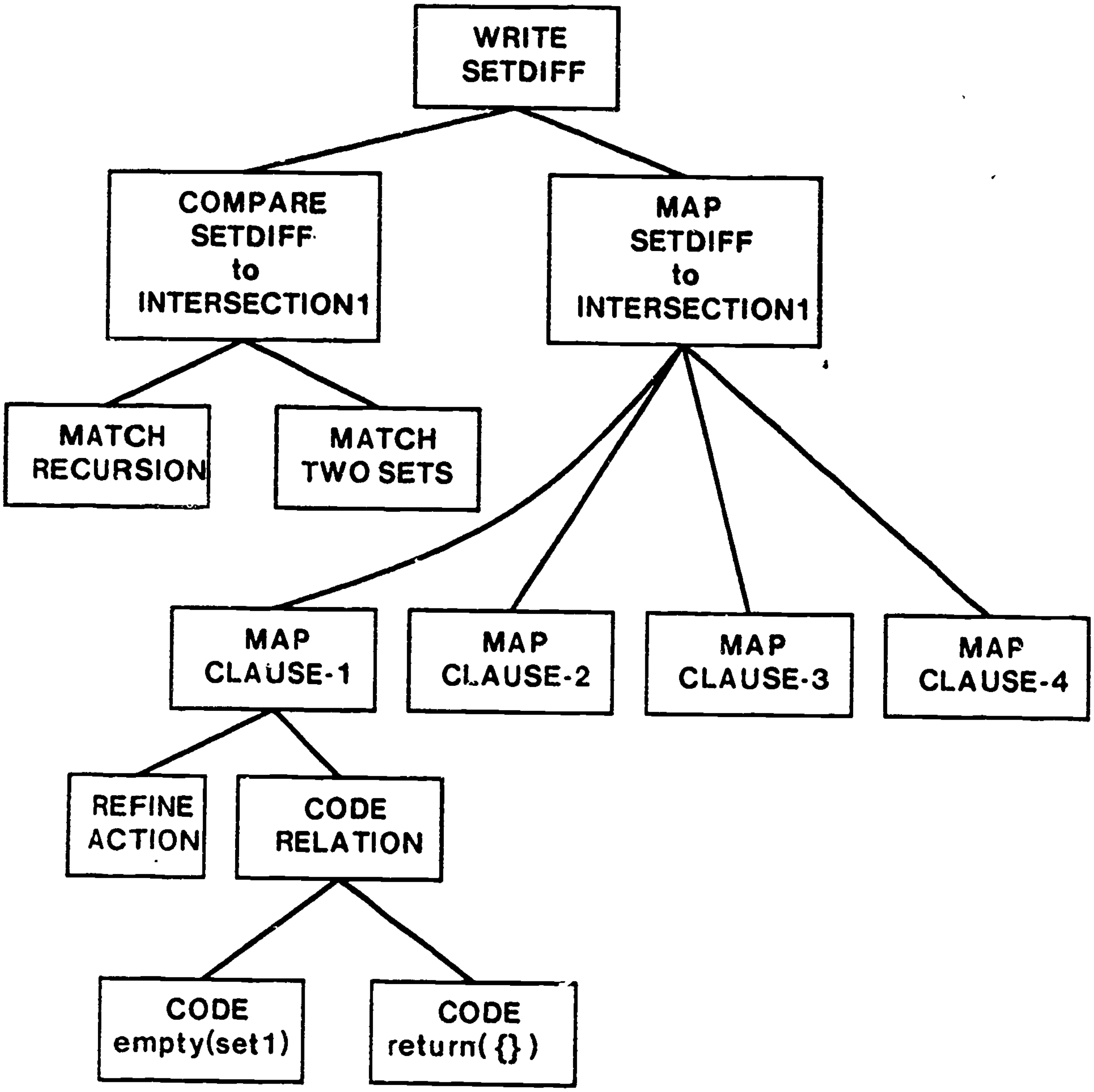


Figure 5: Problem-solving by analogy involves (a) establishing a partial match of example features f_i to target problem features f_i' and (b) mapping example solution components s_i to target problem solution steps s_i'. Learning mechanisms compile such problem-solving into new productions that generalize across example and target.

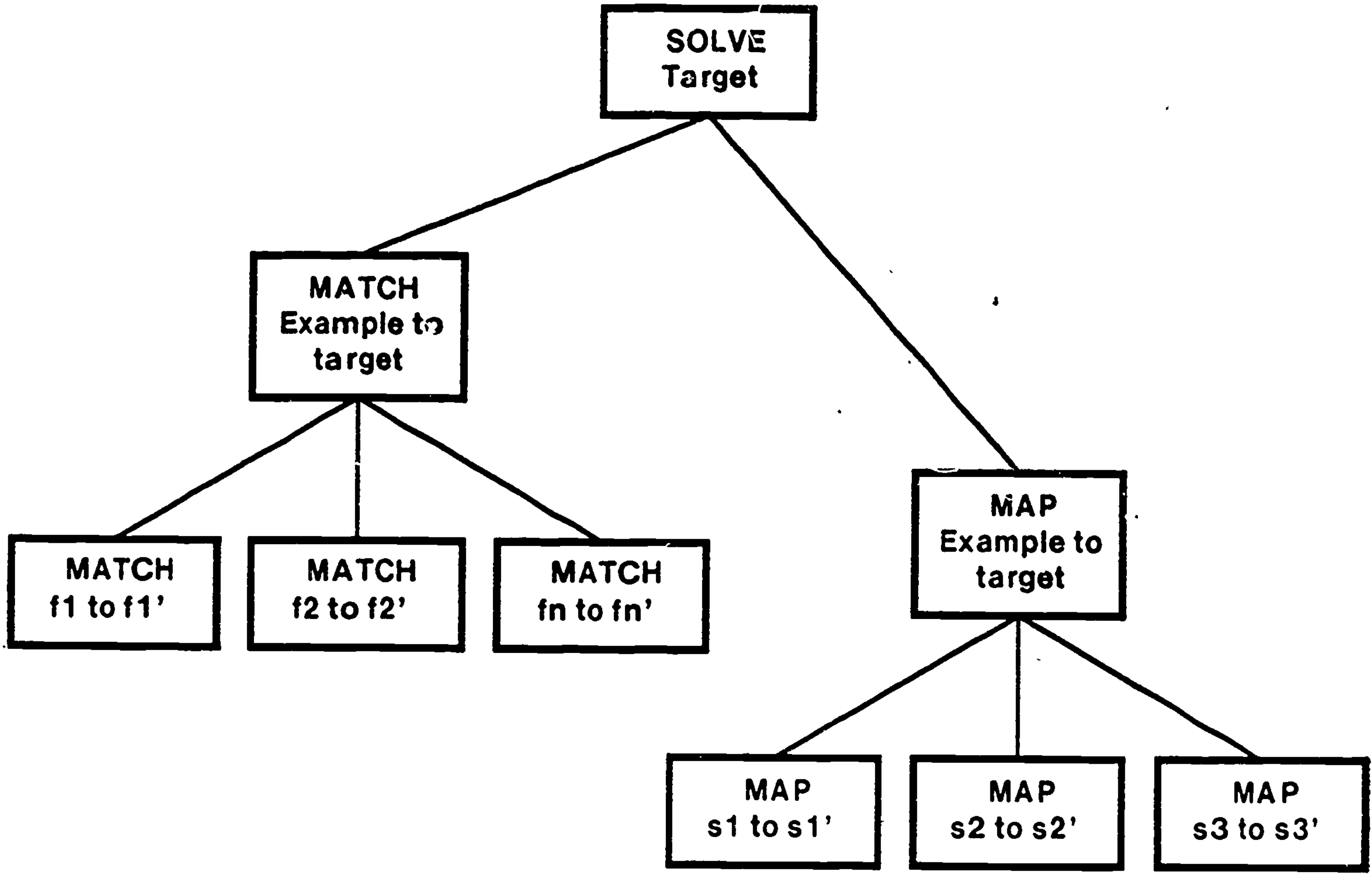
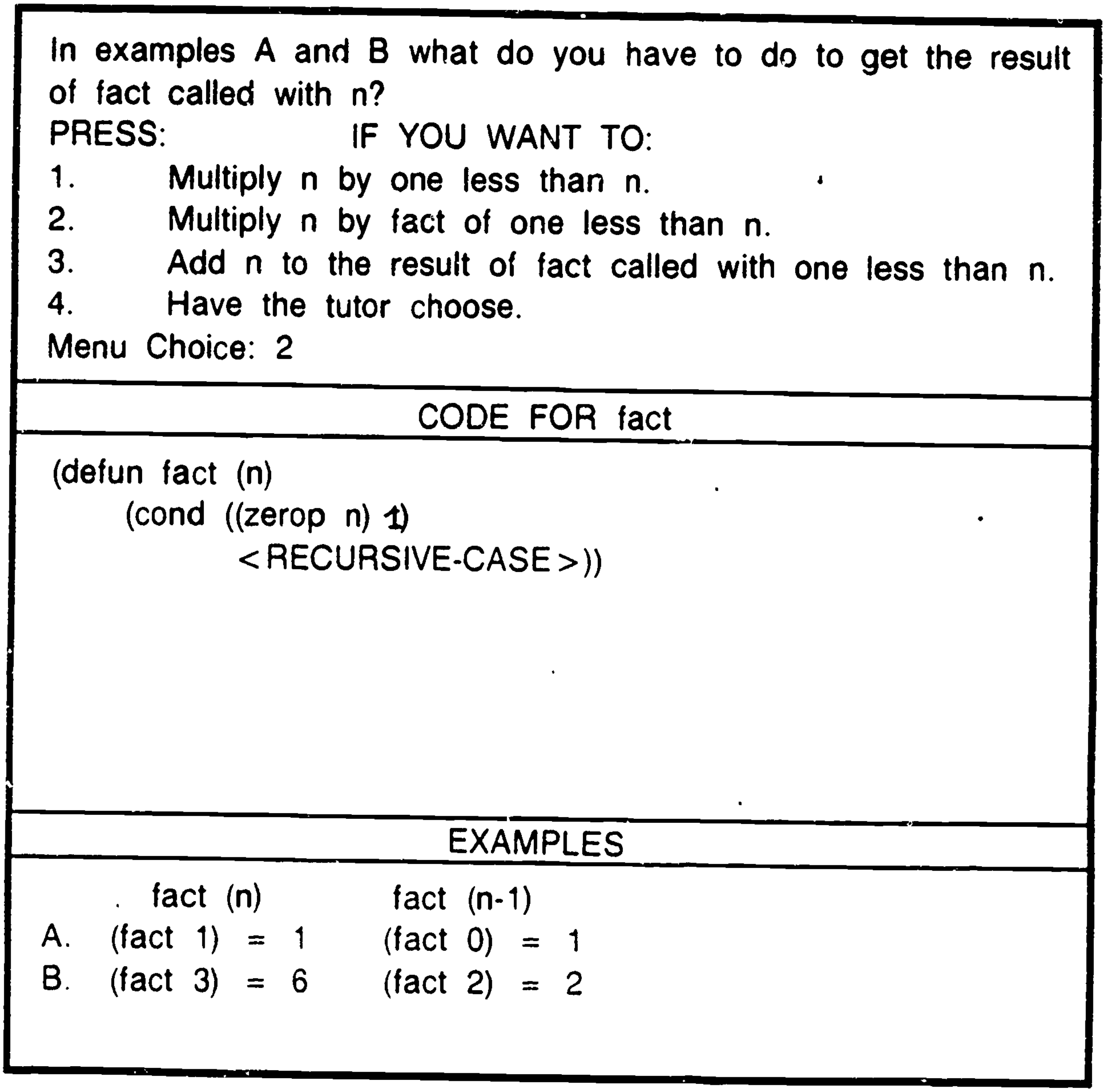


Figure 6: A typical screen from the LISP tutor



In examples A and B what do you have to do to get the result of fact called with n?

```
PRESS:              IF YOU WANT TO:
1.      Multiply n by one less than n.
2.      Multiply n by fact of one less than n.
3.      Add n to the result of fact called with one less than n.
4.      Have the tutor choose.
Menu Choice: 2
```

CODE FOR fact

```
(defun fact (n)
    (cond ((zerop n) 1)
          <RECURSIVE-CASE>))
```

EXAMPLES

```
      fact (n)        fact (n-1)
A.  (fact 1) = 1    (fact 0) = 1
B.  (fact 3) = 6    (fact 2) = 2
```