COGNITIVE PSYCHOLOGY 20, 362–404 (1988)

# Cognitive Objectives in a LOGO Debugging Curriculum: Instruction, Learning, and Transfer

DAVID KLAHR AND SHARON MCCOY CARVER

*Carnegie-Mellon University*

In this paper we report two studies in which elementary-school children learned a complex computer-programming skill—how to debug LOGO graphics and list-processing programs—and then transferred the high-level goal structure of that skill to nonprogramming domains. Instruction, its assessment, and the transfer tasks were all derived from an explicit model of the debugging process, cast as a computer simulation. Debugging skills were acquired over a period of several months as part of a LOGO programming course; the transfer tasks involved correcting written instructions in a variety of domains, including arranging items, allocating resources, and following map routes. Students showed clear improvement in the transfer tasks following instruction in debugging programs, and in the second study, amount of transfer was correlated with the degree of debugging skill acquisition. Our results contrast with many earlier studies that found little transfer of problem-solving skills in general and of high-level programming skills in particular. We suggest that the key to the success of our procedure is the fact that we used an extremely precise computer simulation model of the skills required to debug LOGO graphics and list-processing programs as a concrete manifestation of the notion of "cognitive objectives." 

The psychology of computer programming has become a sufficiently active field of research in the past few years to warrant its own specialized meetings and publications (e.g., Mandinach, Linn, Pea, & Kurland, 1986; Mayer, 1988; Olson, Sheppard, & Soloway, 1987; Soloway & Iyengar, 1986). Programming's complexity and practical relevance make it an inherently challenging and important domain for investigation by cognitive psychologists (cf. McKeithen, Reitman, Rueter, & Hirtle, 1981; Pea &

This research was supported by a grant jointly funded by the Program for Research in Teaching and Learning and the Program for Applications of Advanced Technology at the National Science Foundation (MDR-8554464). It is based in part on a doctoral dissertation completed by the second author while she held a National Science Foundation Graduate Fellowship. Preliminary reports were presented at the Biennial Meeting of the Society for Research in Child Development and the Annual Meeting of the American Educational Research Association, April 1987, and the Second Workshop on Empirical Studies of Programmers, December 1987. We thank the students, teachers, and administrators of the Montessori Center Academy in Glenshaw, PA, and the Ellis School in Pittsburgh, PA, for their cooperation and participation in this project. Address reprint requests to David Klahr, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213, or Klahr @ psy. cmu. edu.



0010-0285/88 $7.50

Sheingold, 1987). But a more fundamental, and perhaps more tantalizing, reason for psychologists' interest in computer programming derives from the possibility that it may be the long-sought mental exercise that enables its practitioners to increase their general thinking abilities. Indeed, a common justification for teaching children how to program is the claim that they will acquire generalizable high-level skills, such as planning, problem decomposition, and debugging from this unique problem-solving domain (Linn & Fisher, 1983; Papert, 1980).

Although this is an appealing claim, there are both theoretical and empirical grounds for doubting its validity. The theoretical problem is that the mental exercise view of programming's impact is precariously close to the thoroughly discredited "faculty psychology" approach to learning, memory, and problem solving (Angell, 1908). Empirically grounded pessimism derives from transfer studies of both problem solving and programming. First, there is the notoriously difficult issue of transfer from one problem-solving domain to another. Problems with superficially dissimilar cover stories but similar deep structures are not recognized as such by most subjects (Gick & Holyoak, 1983), and transfer between problem isomorphs is scant (Simon & Hayes, 1976) and, as Gray and Orasanu (1987) note, "surprisingly specific" (e.g., Bassok & Holyoak, 1987; Reed, Ernst, & Banjeri, 1974). Second, investigations of skill transfer from programming to related nonprogramming domains have yielded mixed results. With respect to low-level tasks closely related to LOGO spatial skills, such as those involving knowledge about turns and angles or about novel extensions of figural components, demonstration of acquisition and transfer has been relatively successful (Clements & Gullo, 1984; Dalbey & Linn, 1984; Geva & Cohen, 1987; Gorman & Bourne, 1983). However, the results of most studies of mastery and transfer of high-level skills do not support assertions about the general cognitive benefits of learning to program (Dalbey & Linn, 1984; Garlick, 1984; Gorman & Bourne, 1983; McGilly, Poulin-Dubois, & Shultz, 1984; Pea, 1983).

In this paper, we describe two studies in which students *did* successfully acquire and transfer a high-level skill—debugging—from a programming to a nonprogramming context. We focused on debugging because it plays a central role within the programming domain and because it is one of the "powerful ideas" (Papert, 1980) that, on the face of it, might be expected to generalize beyond programming and become a broadly applicable cognitive skill. Our goal was to establish a set of sufficient conditions for acquisition and transfer of this class of complex skills. We assumed that transfer from programming to nonprogramming domains, if it could be achieved at all, would require that both instruction and assessment of transfer be grounded in well-specified cognitive objectives. More specifically, our thesis is that *if* the domain is properly analyzed, *if*

instruction is based on the formal analysis, and *if* assessments of both what is learned in the base domain and what is transferred to more remote domains are also grounded in the formal analysis, *then* a powerful idea like debugging can be taught and can have an impact on general problem-solving capacities.

In proposing *cognitive objectives* as the fundamental link between cognition and instruction, Greeno (1976) suggested that:

> . . . the explicit statement of instructional objectives based on psychological theory should have beneficial effects both in design of instruction and assessment of student achievement. The reason is simple: we can generally do a better job of accomplishing something and determining how well we have accomplished it when we have a better understanding of what it is we are trying to accomplish. (p. 123)

The work described in this paper represents a complete elaboration of this notion of cognitive objectives in the domain of teaching elementary school children (8 to 12 years old) how to debug LOGO programs. The cognitive objectives we developed were extracted from a formal task analysis of debugging skill that was used to produce a computer simulation that can debug a wide range of buggy LOGO programs. The model's rules (productions) were used as the basis for explicit instruction in debugging and for the assessment of learning and transfer. The debugging instruction, learning assessment, and transfer testing were embedded in conventional LOGO courses that included both graphics and list-processing components. In this paper, we describe the task analysis, the curriculum and assessment procedures based on the model, and the design and results of two extensive experiments based on the curriculum. Finally, we discuss the points of contact between our work and other investigations of debugging, analogical problem solving, and transfer of training.

## A COGNITIVE MODEL OF DEBUGGING SKILL

A detailed task analysis of LOGO debugging skills provides the basis for all of the subsequent debugging instruction, learning assessment, and transfer assessment. The analysis was intended to capture, in the form of a concrete model, the decision processes, knowledge, and subskills necessary for efficient debugging of LOGO graphics and list-processing programs with one or more semantic and/or syntactic bugs. In this section we describe the model at several levels of detail. In order to provide the required background, we start with a brief introduction to LOGO. Then, we provide a general characterization of the major phases of the debugging process, followed by a description of the computer simulation implementation of the model and an account of how the model debugs faulty

LOGO programs. Finally, we comment on the particular level of debugging skill that our model is intended to represent.

## LOGO in a Nutshell

LOGO is a sophisticated computer programming language designed to enable young children to create interesting graphics effects. Although best known for its graphics, LOGO is also a general-purpose, list-processing language with capabilities extending to the domains of music and basic word processing. In all of these domains, one can write procedures that utilize powerful programming structures such as subprocedures, variables, and recursion.

In LOGO graphics, the user is supposed to imagine that a "turtle" will move a "pen" around the screen to draw pictures. ForwarD, BacK, LeftTurn, and RightTurn are the commands necessary to move and turn the turtle. Each of these commands requires a numeric argument to indicate the distance to move (for FD and BK) or the number of degrees to turn (for LT and RT). In addition, PenUp and PenDown control the position of a pen: when the pen is down, any turtle movement leaves a trace of the turtle's path on the screen. LOGO also has commands to direct the flow of control; these include REPEAT, IF, and STOP.

Many of LOGO's features—procedures, variables, conditionals, recursion, graphics, and iteration—are illustrated in Fig. 1. The listing on the left shows five procedure definitions (BOX, ROOF, HOUSE, MOVE,

```
TO BOX :SIZE
  REPEAT 4 [FD :SIZE RT 90]
END

TO ROOF :SIZE
  REPEAT 3 [FD :SIZE RT 120]
END

TO HOUSE :SIZE
  BOX :SIZE
  FD :SIZE RT 30
  ROOF :SIZE
  LT 30 BK :SIZE
END


TO NEIGHBORHOOD :SIZE
  HOUSE :SIZE
  IF :SIZE = 20 THEN STOP
  MOVE :SIZE
  NEIGHBORHOOD :SIZE - 10
END

TO MOVE :SIZE
  RT 90 PU FD :SIZE + 10 PD LT 90
END
```

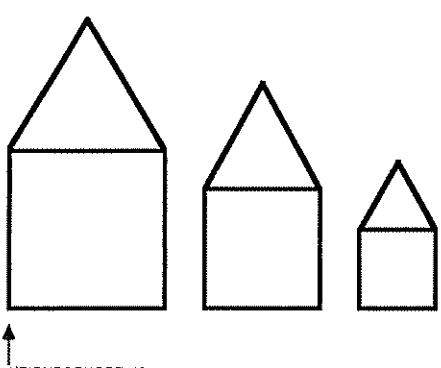



FIG. 1. An example of a LOGO graphics program. Procedures are delimited by "TO" and "END."

and NEIGHBORHOOD), all of which use various primitive commands for moving, turning, iteration, and conditionals. The panel on the right shows the graphic output that would be generated if the procedure NEIGHBORHOOD were called with an input of 40 and the turtle was initialized at the position indicated by the arrow.

## General Character of Debugging

Our analysis assumes the existence of four sources of information. Three are well defined: (a) a description of the goal (for graphics programs, a picture of the desired outcome, and for list-processing programs, a specification of the desired outcome), (b) an on-line listing of the buggy program, and (c) the output produced by the buggy program. The fourth information source, programming knowledge, is more variable; it depends on assumptions about the level of debugging skill being simulated. The model represents a performance theory—stated at the level of goals, productions, and operators—of how to debug the faulty program. The goal structure and production set are invariant. Operators vary with respect to how much they can extract from the information sources. This variation corresponds to different assumptions about the ability of the person whose debugging skills are being modeled. LOGO-specific knowledge is incorporated in most of the productions and operators, and general knowledge about debugging is incorporated in the model's goal structure and the productions that establish and traverse the goal structure.

In the following analysis, we distinguish between the *discrepancy* and the *bug*. The former refers to the difference between the program plan and the program output. The latter refers to the erroneous component of the program that caused the discrepancy. The goal of the debugging process is to detect and correct the discrepancy-causing bug. For example, if the goal drawing corresponding to Fig. 1 was a series of abutting houses, rather than spaced houses, then the discrepancy between the goal drawing and the program output would be described in terms of the "spread" or "location." The bug that caused the discrepancy would be the +10 in the subprocedure MOVE, which repositions the turtle after each house is drawn.

There are five phases to the debugging process. The first phase establishes four subgoals that, when completed, reassert the top goal. The five phases are:

1. *Program evaluation.* Run the program. Compare the program plan and the program output. If they do not match perfectly, then identify the bug, represent the program, locate the bug, and correct the bug.

2. *Bug Identification.* Generate a description of the discrepancy between the program plan and the program output. Based on the discrepancy description, propose specific types of bugs that might be responsible

for the discrepancy. Where multiple possibilities exist, do further discrepancy description and bug proposal. When only one possibility remains, move to the next phase.

- In its purest form, the discrepancy description makes no reference to the fact that the faulty output is program-generated. This is, the discrepancies are characterized entirely in terms of their static features.[1] Table 1 lists the most common types of discrepancy encountered when LOGO graphics and list-processing programs are debugged. The quotations presented in the second column are representative comments from children in our LOGO courses about the type of discrepancy shown in the first column. Note that one possible outcome of the bug identification step is knowing that the plan and output are not identical but being unable to describe the mismatch. However, in the case of syntax errors, the error message always provides a description of the discrepancy for the user (though the user may ignore it).
- Given the description of the discrepancy, the model makes inferences about which specific program components are capable of generating that type of discrepancy. The third column in Table 1 suggests some of the possible mappings. For example, if the discrepancy is spread, then it is likely to be caused by turning the wrong angle or moving the wrong distance. In addition to proposing these general types of programming errors, the model has a set of rules which propose further discrepancy description to discriminate between multiple possibilities. When only one possibility remains, the model continues with the next phase. However, the model may need to cycle through discrepancy description and bug proposal several times before a specific program command is identified as the bug (see the fourth column in Table 1). The result of this complex processing is a narrower search for the bug (e.g., "it shouldn't be left 90—it should be right 90 I think").

*3. Program representation.* Represent the structure of the program to investigate the probable location of the buggy command in the program listing.

- Knowledge of the program's structure may be the result of having written the program or of assuming that programs for certain types of plans will be structured in characteristic ways. For example, the model may be given knowledge that the program has a repeat structure because the user wrote the program or because the user observes that a picture is composed of several identical figures (typically programmed using a RE-

[1] It is possible that discrepancy descriptions might include temporal information, because in our procedure, the child watches as the program's output is dynamically generated. On the computer we used, Fig. 1 would take about 5 s to draw. Also, for list-processing output, the temporal order of different portions of the output is preserved by the output listing on the screen.

TABLE 1
Sample Discrepancy—Bug Mappings for LOGO Graphics and List-Processing Programs

| Discrepancy | Example description | Buggy component | Specific bug |
|---|---|---|---|
| Orientation | "this is going over here instead of down" | Angle | LT n or RT n |
| Size | "that line—it's too long" | Distance | FD n or BK n |
| Spread | "these are too close together" | Angle or Distance | LT n or RT n or FD n or BK n |
| Location | "this is supposed to be in the middle" | Distance | FD n or BK n |
| Extent | "lots too many squares" | Iteration or Recursion stop or Recursion interval | REPEAT n<br>IF :x = n THEN STOP<br>NAME :x +- n |
| Extra part | "it drew a line there" | Pen position or Program call | PU omitted or Extra call |
| Wrong part | "it drew corn instead of a stalk" | Program call | Switched call |
| Missing part | "I wanted a box there" | Pen position or Program call | PD omitted or Call omitted |
| Print variable | "it printed 'score' instead of the number" | Punctuation | Quoted variable |
| Not matching | "I put the right answer and it marked it wrong" | Nesting | READLIST or READWORD |
| Wrong value | "it printed the number instead of the place" | Variable name | Wrong variable name |
| How to | ERROR MESSAGE | Punctuation | Missing punctuation |
| What to | ERROR MESSAGE | Command | Missing command or Missing parenthesis |
| No value | ERROR MESSAGE | Initialization | MAKE "name value or No parameter |
| Don't know | "this mess" | ? | ? |

PEAT statement). Knowing that the bug is located within a REPEAT statement narrows the search in the program listing. In the case of syntax errors, the error message gives the user information about which procedure contains the bug.

*4. Bug location.* Using the cues gathered in the last two phases, examine the program in order to locate the alleged bug.

The efficiency of the bug location process depends on the outcome of the bug identification and program representation processes. At best, the model searches for a perfectly specified bug (both the buggy command and its arguments are specified) in a highly constrained set of possible bug locations. At worst, the model must perform a step-by-step examination of the program because it has no knowledge of the bug's identity and no cues about its location.

5. *Bug correction*. Examine the program plan to determine the appropriate correction. Replace the bug with the correction in the program listing and then reevaluate the program.

This reevaluation is slightly different from the initial test in that the model knows that a change has just been made. It first determines whether the correction fixed the original problem. If the correction worked, the model will determine whether there are any more bugs to fix; otherwise, it will debug the correction before proceeding.

## A Production-System Specification

In order to specify the model unambiguously and to demonstrate its sufficiency for debugging LOGO programs, it was implemented in GRAPES, a goal-restricted production system (Sauers and Farrell, 1982). The GRAPES model consists of a set of 84 productions that specify the action to be taken if certain conditions exist. The conditions include the goal the model is trying to achieve and the information currently available in working memory (the set of known facts). A production is selected and executed only when the appropriate conditions exist; thus the current state of the environment (current goals and knowledge) determines which actions will be performed. The actions include updating or adding to both working memory and goal memory.

The goals correspond to the steps in the debugging process listed above; productions represent general and specific search heuristics used for efficient debugging; operators represent both information-extraction skills (described earlier) and some subskills which are essential, but not central, to the debugging process (e.g., editing skills). The following sections briefly describe these three components of the model. These descriptions are followed by demonstrations of how the goals, heuristics, and operators work together to debug faulty LOGO progams. (A more detailed description, the full production system, and more examples of the model working in varied situations can be found in Carver, 1986).

### *Goals Direct the Solution*

The model's goal structure corresponds to the five phases described in the overview (illustrated in Fig. 2). The system has a set of productions for each goal to represent the different responses a debugger would have to the same goal in different situations. The "situations" are represented by the current contents of the system's working memory. Productions with test and evaluate goals start the system and evaluate the success of each debugging attempt (i.e., the match between the program plan and the program output). The describe and propose goals correspond to the bug identification phase; they satisfy the productions that describe the discrepancy between the program plan and the program's buggy output and

that propose possible bugs and ways to discriminate among them. Represent and specify correspond to the program representation phase; productions with these goals look for structural cues to the bug's location so that find, interpret, and check can actually isolate the bug using whatever cues they have about the bug's identity and location. Finally, the change and replace goals correspond to the bug correction phase; they fire productions that identify the appropriate correction and change the program listing accordingly.

*Heuristics Narrow the Search*

The system has two sets of debugging heuristics, one set for identifying the bug (phase 2) and one set for representing the location of the bug in the program (phase 3). Appropriate use of both sets of heuristics substantially narrows the search for the bug. Heuristics for identifying the bug correspond to the mappings between observed discrepancies and potential bugs (listed in Table 1). These heuristics are most useful in situations where several different types of bugs might be responsible for a particular type of discrepancy. In this case, the heuristic includes information for distinguishing among candidate bugs. For example, if the discrepancy has been identified initially as spread, then the model will request information about orientation because it has the knowledge that discrepancies described as both spread and orientation must have been caused by an angle bug, whereas those described only as spread discrepancies must have been caused by a distance bug.

Heuristics for representing the location of the bug involve knowledge of program structure types. For example, if the program is identified as having subprogram structure, the model would ask for information about which subprogram was likely to contain the error and it would confine its search to that subprogram unless no bug could be located there. If no

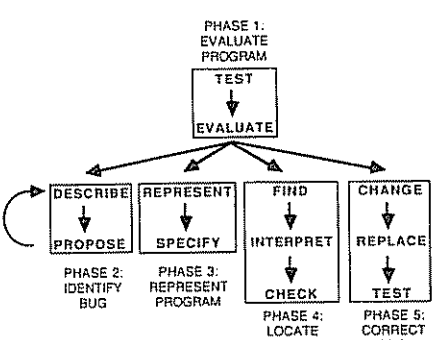



FIG 2. Goal structure of the debugging model

subprogram cue is available, the model will seek other structural cues, such as location with a REPEAT or IF statement or location after a particular command. For example, if the user can identify a correct command which was executed before the bug occurred, the model will use that command as a marker and begin its search after that command.

*Operators Process Information and Produce Behavior*

The debugging productions use 11 operators, or subskills, to process information available to the system. These operators are called by productions when it is necessary to process information from one or more sources or to take specific actions. In addition to the four initial information sources described earlier (program goal, program output, program listing, and knowledge of the programming language), an operator may use dynamic information contained in working memory, and operators may add information to working memory.

There are two classes of operators: (a) those that correspond to inspection of the buggy output and/or the plan and (b) those that correspond to maneuvering in the LOGO environment. Operators in the latter class (RUNning the program, ENTERing the editor, SKIPping to a particular location, READing a command, DELETing a command, and INSERTing a command) are automatically executed by the model, but operators in the former class are not. Instead, they are simulated by the user of the system. The user must MATCH the program plan and the output to determine whether a discrepancy exists, CONTRAST the two outcomes to describe the discrepancy, EXAMINE the buggy portion of the output, INTERPRET the effect of particular LOGO commands, and GENERATE the LOGO commands to create a particular effect.[2]

The user-simulated operators are important for simulating different levels of debugging skill. The model's solution to a debugging problem depends on the amount and accuracy of the information gathered about the debugging situation to guide the search for the bug. In the next section, we contrast debugging episodes simulating users with different amounts of knowledge. The more knowledge input to the model, the narrower the search for the bug.

## Simulating the Debugging Process with High and Low Information

This section contrasts two simulated attempts to debug the example program shown in Fig. 3. The simulations differ only in the amount of information the user gives the model about the bug's identity and loca-

[2] The rationale for this treatment of the "interface" operators is presented in the next section.

tion. (As noted earlier, LOGO has list-processing and string manipulation commands as well as graphics commands. They are illustrated in the following examples.)

First, we simulate a situation in which the debugger is a very knowledgeable user (Table 2). The model is provided with information about both the discrepancy and the program. The information, provided in response to the operators, is marked by → on the right-hand side of the trace. Here the user classifies the problem as list processing without an error message and then identifies the discrepancy type as *printvariable* since the variable JOB was printed instead of its value TV REPAIR. The model responds that the bug causing that discrepancy is likely to be incorrect punctuation. It also asks the user to input the name of the variable. The user is then asked a series of questions about the likely location of the bug. This user knows that the program LIVING has subprocedures and that the bug is likely to be in the procedure JOB. Since the model has been given knowledge about both the likely identity and the likely location of the bug, it locates the bug immediately—i.e., without having to interpret and check the outcomes of any commands. The model prompts the user to input the necessary fix, makes the specified change, and directs the user to retest the subprocedure JOB and then the main

A. Buggy Code

```
TO LIVING
  PRINT [DO YOU LIKE LIVING IN PENNSYLVANIA?]
  MAKE "LIVING READWORD
  IF EQUALP :LIVING "YES [WHERE] [WORKING]
 END
TO WHERE
  PRINT [WHERE DO YOU LIVE?]
  MAKE "WHERE READLIST
  (PRINT :WHERE [IS A NICE PLACE TO LIVE !)
 END
TO WORKING
  PRINT [TOO BAD, DO YOU LIKE WORKING HERE?]
  MAKE "WORKING READWORD
  IF EQUALP :WORKING "YES [JOB] [PRINT [I DON'T EITHER.]]
 END
TO JOB
  PRINT [WHAT IS YOUR JOB?]
  MAKE "JOB READLIST
  (PRINT "JOB [IS AN INTERESTING JOB.])
 END
```

B. Buggy Output

```
?living
DO YOU LIKE LIVING IN PENNSYLVANIA?
no
TOO BAD. DO YOU LIKE WORKING HERE?
yes
WHAT IS YOUR JOB?
tv repair
JOB IS AN INTERESTING JOB
?
```

FIG 3. An example list-processing bug. (A) Buggy program listing. (B) Buggy output. The last output line should be "TV REPAIR IS AN INTERESTING JOB." The bug is in the second print statement in the JOB procedure where "JOB should be the variable :JOB

TABLE 2
Trace of the Model Simulating an Efficient Debugger

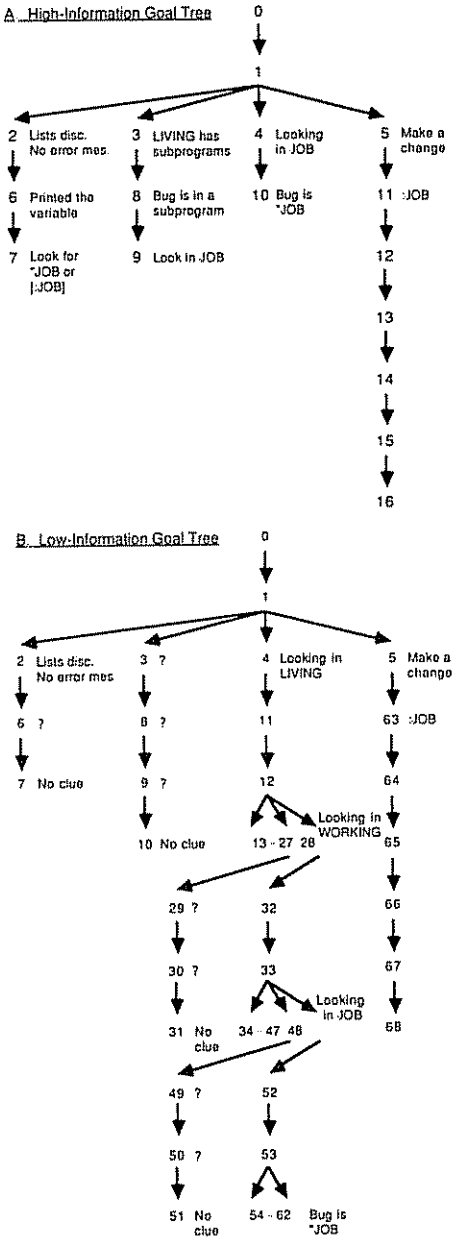
| | |
|---|---|
| Run the program LIVING. | →ok |
| Did the outcome match the plan [yes or no]? | →no |
| What type of discrepancy is there [graphics or lists]? | →lists |
| Did you get an error message [yes or no]? | →no |
| What is the discrepancy between the plan and outcome? [printvariable, wrongvalue, notmatching, extent, extrapart, wrongpart, missingpart, or ?] | →printvariable |
| The variable has probably been quoted or put in brackets. Use a colon when printing variables. | |
| What is the name of the variable? | →JOB |
| Look for PRINT "JOB or PRINT [ . . :JOB ]. | |
| Does the LIVING program have subprograms [yes or no]? | →yes |
| Is the bug in a subprogram [yes or no]? | →yes |
| Which subpart is wrong? | →JOB |
| The bug is "JOB in JOB. | |
| How should the fix be made? [change, delete, or insert] | →change |
| What should "JOB have been? | →:JOB |
| Run the program JOB. | →ok |
| Did the correction fix the problem? | →yes |
| Run the program LIVING. | →ok |
| Did the outcome match the plan [yes or no]? | →yes |

*Note* The model's prompts are listed on the left, and the responses we chose to simulate an effective debugger are listed on the right.

procedure LIVING. Since the fix was correct and no other bugs exist, the debugging episode is complete.

Figure 4A shows the goal tree generated during this high-information simulation. Each number refers to one subgoal, and the information content of the trace is included next to the corresponding subgoal number. Figure 4B shows the goal tree generated during a low-information simulation. In contrast to Fig. 4A, the user responds to the model's prompts with question marks, so the model has no clues to either the bug's identity or its location. In this case, the model starts searching for the bug in the main procedure and continues checking each command, in order of execution, until the bug is found. Each time a subprocedure call is encountered, the user is given another chance to provide information about the likely location of the bug. Once again, none is entered so the model continues its serial search until the user identifies "JOB" as the bug.

The contrasts between the model's behavior in the high- and low-


A. High-Information Goal Tree
0
1
2 Lists disc. No error mes.
3 LIVING has subprograms
4 Looking in JOB
5 Make a change
6 Printed the variable
8 Bug is in a subprogram
10 Bug is *JOB
11 :JOB
7 Look for *JOB or [:JOB]
9 Look in JOB
12
13
14
15
16
B. Low-Information Goal Tree
0
1
2 Lists disc. No error mes
3 ?
4 Looking in LIVING
5 Make a change
6 ?
8 ?
11
63 :JOB
7 No clue
9 ?
12
64
Looking in WORKING
10 No clue
13 - 27 28
65
29 ?
32
66
30 ?
33
67
Looking in JOB
31 No clue
34 - 47 48
68
49 ?
52
50 ?
53
51 No clue
54 - 62 Bug is *JOB

information situations are striking: the former required only 16 subgoals, while the latter required 68. The high-information simulation represents the ideal case in which the system's goals and heuristics were used efficiently to narrow the search until the bug was completely specified and its location was known. In the low-information situation, however, little use is made of the describe, propose, represent, and specify goals, so none of the search reduction heuristics are used and debugging proceeds by brute force, one command at a time. Most of the extra subgoals result from this serial search.

## Scope of the Model

The model is bounded along three important dimensions: Its place within the total set of processes that support programming, its interface with the external environment, and the range of debugging skills that it can simulate. In the following paragraphs, we elaborate each of these limitations.

### *Debugging, Not Programming*

This is not a model of the total programming process. Although the model contains a large amount of LOGO-specific knowledge in the discrepancy-bug mappings, it has no ability to take a particular programming goal and generate LOGO code to achieve that goal. It is intended to represent only one class of components of the full set of programming processes that a LOGO programmer might have.

### *Unmodeled Interface with the External Environment*

As noted in the previous section, several of the operators that control interaction between the debugger and the external world are not simulated. For example, the model has no perceptual front end that can compare program output with desired output in order to characterize the initial discrepancy. Instead, the results of processing by the MATCH and COMPARE operators are determined by inputs from the "user." Nor have we simulated INTERPRET, the operator that checks the result of a single command to see if it had the desired effect. This too is determined by user input. While the productions in our model determine *when* this information is required, and *what* to do with it, once it is available, there is no model of *how* that information is extracted from either memory or the environment. We do not believe that these unmodeled processes are

FIG. 4. Simulated goal trees for (A) High-information and (B) Low-information simulations. Because no clues to the bug's identity or location are gathered in the low-information simulation, the search (goal 4 and goals 11–62) is extensive compared to the two-step search of the efficient debugger.

simple, unimportant, or "hardware primitives" (see Palmer and Kimchi, 1986, for a discussion of the role of primitives in information-processing models). However, the creation of a perceptual front end that could construct encodings of discrepancies between intended and actual outcomes would be a very complex task that is tangential to our central purpose: the modeling of the basic goal structure of the debugging process. Our production system is limited to modeling the internal logic of the debugging process, primarily through the goal structure and the discrepancy-bug mappings described earlier.

A pragmatic advantage of this approach to modeling is that the "user interface" facilitates the simulation of debuggers with a wide variety of ability to extract relevant information from the programming environment. As noted earlier, a low-information debugger simply fails to provide any information relevant to the series of user prompts. The basic logic of the debugging process does not change in this situation, but there is a dramatic decrease in overall efficiency. By limiting the amount of information available to the model, we effectively disable the productions that could use that information. Thus the "user interface" provides a convenient mechanism for exploring various levels of debugging skill. (Two such examples were provided in the previous section, and several others are described in Carver, 1986).

*Skill Level of the Debugging Model*

For all its complexity, this model is not intended to characterize expert debugging skills. Instead, we have attempted to model a range of skill levels from rank beginner (who would fail to provide useful information to the operators) to good novice: the kind of debugging we would hope to achieve after a semester or two of LOGO instruction. This range is exemplified by the two debugging traces shown above.

One reason for focusing on this level, rather than on expert performance, is that it is not yet known how to fully describe such a skill. Studies of expert programmers (Gugerty & Olson, 1986; Jeffries, 1982) suggest that they may have three broad categories of discrepancy-bug mappings. At the lowest level are the relatively primitive and direct mappings of the kind included in our model. At the next level are bugs associated with incorrect implementation of standard combinations, conventions, and rituals (for example, standard iteration procedures or interactive prompts). At the third level are sophisticated schemes for dealing with the likely *causes* of bugs, such as those discussed by Spohrer and Soloway (1986). In addition, experts are able to search for multiple bugs simultaneously and to do multiple-step backward inferencing from the bug to its likely cause. Our model only treats one bug at a time and uses only one-step backward inferences (the discrepancy-bug mappings).

Even if we knew how to construct a model that had all these expert

skills, it would be very difficult to teach it to rank beginners, without first developing a learning hierarchy of simpler, more instructable skills. That is, we would have to develop an entry-level model comprised of simple, familiar elements. Such a model would probably look very much like the one presented here.

## APPLYING THE MODEL TO INSTRUCTION AND ASSESSMENT

It is clear that, even at the novice level, debugging is a complex cognitive skill requiring a mix of content-specific programming knowledge and general problem-solving heuristics. Many observers (e.g., Papert, 1980) have noted that when confronted with buggy programs, most children prefer to abandon them and start anew rather than debug them. We have suggested that they do so simply because they have acquired few, if any, of the requisite skills (Carver & Klahr, 1986). Indeed, in two pilot studies, we found that good debugging skills are not learned spontaneously in the context of either structured or unstructured LOGO curricula.

In our first pilot study, 9 second and third grade students given 24 h of structured LOGO graphics experience did not learn the central components of the model spontaneously (Carver & Klahr, 1986). In the second (unpublished) study,[3] we assessed 15 fifth grade students who had approximately 200 h of unstructured LOGO experience over an 18-month period. Subjects in both groups failed to gather effective clues about the identity and location of the bug; therefore, they relied heavily on serial search. Even their serial search was ineffective because they made frequent errors determining the effect of particular commands. Similar difficulties with debugging have also been demonstrated among LOGO teachers (Jenkins, 1986) and among adults learning other programming languages (Gould, 1975; Gugerty & Olson, 1986; Jeffries, 1982; Katz & Anderson, 1986; Kessler & Anderson, 1986). In general, it appears that, in the absence of explicit instruction in debugging, neither structured lessons nor open-ended "discovery" contexts are adequate for teaching children or adults anything beyond the most meager debugging skills. Therefore, we decided to use the model as the basis for a debugging curriculum, its assessment, and a set of transfer tests.

Each of the two studies described in this paper was designed to answer two questions:

1. Can the debugging skills used by the model be taught directly?
2. Can the debugging skills, once learned, be transferred to nonprogramming tasks requiring similar skills?

[3] We thank the staff of MIT's Media Technology Laboratory for providing access to their classrooms at the Hennigan School in Boston.

These issues were addressed in the context of three LOGO courses: a 50-h LOGO graphics and list-processing course taught to 22 8- to 11-year-old children in one school over a 6-month period (Study 1) and 25-h list-processing courses taught in a different school to two groups of 17 11-year-olds over an 8-month period (Study 2). The remainder of this section discusses the common features of these studies, and further sections describe the particulars of each study and the learning and transfer results from each.

## Principled Instructional Design

The main cognitive objective of the debugging curriculum was for students to acquire the same goal structure as the model, especially the initial phases where cues to the bug's identity and location are gathered to narrow the search for the bug. With only slight rewording of the goal structure shown in Fig. 2, particularly the interactive prompts the model gives the user, we produced a step-by-step debugging procedure to teach the students. In order to highlight the similarity between the model and the instruction, we show in Fig. 5 the debugging procedure students were taught in terms of the model's goal structure. The curriculum also included specific heuristics the model uses to map discrepancies onto likely bugs and to focus search on particular parts of the program. The discrepancy-bug mappings are equivalent to the knowledge in the propose productions, and the location clues are equivalent to the knowledge in the represent and specify productions. After being given several sample mappings, students were directed to keep written records of problems and their likely causes.

The curriculum was designed so that the first explicit instruction in debugging came after 6–8 h of programming experience. By this time, students' experience with the difficulty of debugging by serial search

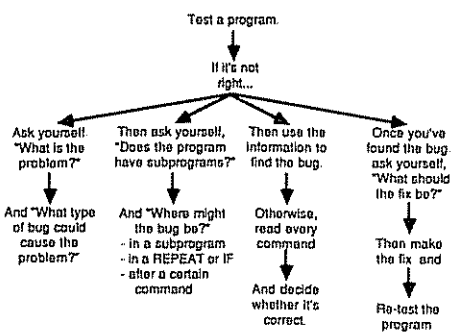



FIG. 5. Debugging curriculum corresponding to model's goal structure.

enabled them to appreciate the usefulness of the focused search skills being taught. Only one 40-min lesson was devoted exclusively to debugging; throughout the rest of the course, however, students were repeatedly encouraged to use the debugging procedures and challenged to find and record new clues.

## Principled Learning Assessment

Before the extent of transfer of a cognitive skill is assessed, it would seem prudent to establish that the skill was acquired in the first place. Nevertheless, as noted in Carver and Klahr (1986), this step is commonly omitted from LOGO transfer studies. In the studies described here, procedures for assessing the acquisition of debugging skills were derived directly from the model, and assessments were performed at several points during the LOGO course.

Students were asked to debug programs written and bugged by the experimenter. We provided the buggy programs so that the bugs would be the same for all students. In order to ensure that students understood what the buggy program was supposed to do, we gave them an opportunity, during a previous week, to write their own version of the program. During the debugging test, then, students were given the experimenter's buggy programs on-line and asked to fix all the bugs. The students were allowed to work until the program ran correctly or until one class period had elapsed, whichever came first.

The programs used for the debugging test were well structured; in other words, they made appropriate use of subprocedures and other LOGO substructures such as iteration, conditional statements, and recursion. In order to facilitate scoring, and to minimize the complications caused by interacting bugs, we planted bugs in such a way that the discrepancies they caused in the output would be fairly independent either in space (usually for graphics) or time (usually for list-processing). Although we used bugs that commonly occur in novice LOGO programs, we did not base our bug selection on an underlying theory of the processes that generate bugs in the first place (cf. Katz & Anderson, 1986; Spohrer, Soloway, & Pope, 1985). The only other criterion for bug selection was that there be a variety of discrepancy types in each program.

The model suggests several distinct measures of debugging skill. It predicts that increased knowledge of any form (discrepancy-bug mappings, discrepancy descriptions, program structure) will produce narrower search (fewer goals); developing debugging skill should therefore result in decreased debugging time. If all the knowledge provided to the model is accurate, the bug will be located and corrected in a single complete cycle (from the initial goal to test the program to the final retest goal). Developing accuracy in debugging should therefore result in fewer

debugging cycles needed to locate and correct bugs. These measures of speed and efficiency, along with some qualitative characterizations of debugging strategies, are the core of our analysis.

Note that the model predicts that, given sufficient time, there should be no difference in overall success rates of good and poor debuggers: even the brute-force approach in Fig. 4 eventually found the bug. In actual practice, we would expect the longer times required by children who followed the less-focused search strategies to be more prone to error, fatigue, and distraction, and therefore they would get lower performance scores. However, the use of detailed *process* analysis provides a much more direct assessment of the extent to which children have acquired the skill we are attempting to teach.

## Principled Transfer Assessment

The goal of the transfer assessments is to discover whether the knowledge elements acquired from the debugging instruction can be applied in new instances. Our model provides a basis for making specific predictions about transfer effects and for choosing tasks where debugging skills are likely to be useful. Three types of noncomputer transfer tests were designed, all of which involved detection and correction of errors in a set of written instructions about how to achieve a well-specified goal. The three types were intended to produce a range of difficulty and to vary the class of basic operations involved in following the instructions. The tests also varied in the type of discrepancy information presented to the subjects. The easiest problems involved directions for arranging something (setting a table, building with blocks, or arranging furniture). Discrepancy information for this class of tasks consisted of two illustrations, similar to, but usually more complex than, the two representations for LOGO graphics debugging. The next easiest problems involved directions for distributing something (paying wages, delivering trees, or ordering food). Discrepancy information for these tasks was presented in tabular form. Finally, the most difficult problems involved directions for traveling somewhere (playing golf, visiting airports, or running errands). Discrepancy information for the route-following problems was presented in terms of a text description of where the person wanted to be, where the person wound up after following the (buggy) directions, and a relatively complex map showing streets and landmarks. The tasks were always presented in order of increasing difficulty so that students would not do poorly on an easier test purely as a result of being frustrated by a harder one. All of the transfer tasks are similar to program debugging in three ways:

1. Instructions given at the beginning of each transfer session and the cover story for each item were designed to highlight the debugging nature

of the tasks. The instructions for transfer tests mimic the program debugging situation: "Today I would like you to read three stories. In each story, someone gives someone else directions. The person follows the directions perfectly, but something goes wrong because one of the directions is wrong. Your job is to find the problem with the directions and fix it so that next time it will be done correctly."

2. Information about the desired and actual output was provided before the written directions could be viewed, just as discrepancy information is available from test runs in debugging situations. As noted above, in two of the three test types, this information was pictorial; in the third, however, it was tabular. From the pictures and tables, subjects could gather clues about the identity of the bug and its probable location just as they could in the program debugging situation.

3. Lists of instructions were structured like LOGO subprocedures by adding headings between sections of instructions to label their purpose. Subjects could use the headings to determine which sections of the instructions were likely to contain the bug just as they could use the subprocedure names to guide their search for program bugs.

The following example shows how the model's brute-force strategy (low-information search) and selective-search strategy (high-information search) would solve one of the transfer items. Figure 6 shows the plan and outcome for the furniture arranging problem. Table 3 lists the accompanying directions. Before viewing the figure, students read the following cover story.

> Mrs. Fisher was moving into a new house with the help of two movers. She asked them to arrange the furniture in her house and gave them a list of directions to follow. The movers followed the instructions *perfectly*, but there was *one* problem with the directions so the furniture was not arranged correctly.
>
> The next page shows the way Mrs. Fisher wanted the furniture to look and the way it looked after the movers arranged it. Use these pictures to help you find the problem with Mrs. Fisher's directions. Then fix the directions so the movers could arrange the furniture correctly.

Comparison of the two floor plans yields a bug identity clue that there is a table out of place. Closer inspection may reveal that the table is in the living room. One might also notice that the table has been placed between two chairs in both drawings and hypothesize that the confusion resulted from a misunderstanding of which two chairs. The structure of the directions makes use of location clues possible. In most cases, the directions are divided into three parts; here, one part describes how to arrange the dining room, one part the living room, and one part the kitchen.

Someone using a brute-force strategy (as many children did) would tediously read each line and check the picture to make sure it was correct

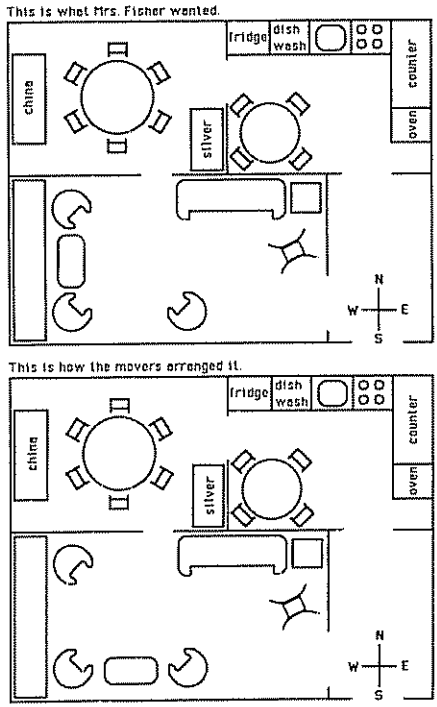



FIG 6 Example transfer test: Furniture arrangement discrepancy

until the incorrect direction was located. A solver who knows to look for a misplaced table might scan the directions until reaching one describing the placement of a table. This would lead to false alarms on the lines describing the three other tables in the home (especially the two which are described prior to the misplaced table). A solver who knows to look in the directions for the living room will ignore the dining room directions and focus only on the living room ones. Depending on the other available information, the solver might check each of the living room commands or only the ones referring to tables. A solver who noticed that the table was between two chairs could scan for a phrase about a table between chairs.

Solvers with all of these strategies could locate the bug and add the information to define which two chairs should surround the coffee table. Here, as in the base domain of program debugging, it is the search process, not the success rate, that distinguishes the different strategies. However, solvers who search more of the directions might be more likely to false alarm and therefore be less successful.

TABLE 3
Example Transfer Test: Buggy Directions

Here are the directions Mrs. Fisher gave to the movers.

To arrange the dining room,
Center the china cabinet on the west wall.
Place the silver cabinet in the south-east corner.
Put the table in the center of the room.
Arrange the 6 chairs around the table evenly.

To arrange the living room,
Place the cabinets against the west wall.
Place one chair in front of each end of the cabinets.
Place the square table in the north-east corner
Put the sofa on the north wall, next to the square table.
Place another chair on the south wall, across from the sofa.
Put the coffee table between the two chairs.
Put the rocker on the east wall, next to the square table.

To arrange the kitchen,
Put the refrigerator in the north-west corner
Put the dishwasher to the right of the refrigerator.
Put the sink to the right of the dishwasher.
Put the stove to the right of the sink.
Place the counter next to the stove and along the east wall.
Put the oven along the east wall, next to the counter.
Place the table in the south-west corner of the room.
Arrange the 4 chairs around the table evenly.

Change or add one thing to fix Mrs. Fisher's directions.

## Data Collection

The primary goal of assessing skill acquisition and transfer is to understand the detailed mechanisms and internal structures of the cognitive processes involved. Several methods were used to ensure collection of data that would facilitate this understanding. Students were encouraged to think aloud so that the goals, strategies, and knowledge influencing their solutions could also be recorded (Ericsson & Simon, 1984). Students' behavior on all tests was videotaped. For debugging tests, the videotape contained a visual record of all screen activity. For the transfer tests, the camera was focused on the subject's test paper(s). In both cases, the videotape also contained a record of elapsed time and an auditory record of all verbalizations by the subject and the experimenter. In order to give students an opportunity to attempt as much of each test as possible during the allotted time and thereby maximize the amount of data collected, the experimenter intervened to provide help when impasses were reached. The amount and type of interventions were also recorded and included in the analysis.

## STUDY 1

The primary goal of Study 1 was to assess the extent to which students could learn debugging skills from explicit instruction in a LOGO programming course and transfer them to debugging of noncomputer directions.

### Design and Procedures

The LOGO curriculum for Study 1 was implemented at a Montessori school during the 1985–1986 school year. The second author, who was an experienced LOGO teacher, served as the instructor. All of the third through sixth grade children in the school participated.[4] Of the original class of 24 students, 22 (8 females and 14 males) ranging in age from 8;2 to 11;8 successfully completed the 50-h course. (Two left the class.) All instruction took place in a dedicated classroom equipped with two Apple IIc computers running Apple LOGO II. Students came to computer classes in sets of four and worked in pairs. Each pair of students had two 1-h LOGO classes per week for 25 weeks.

All lessons were taught in a guided discovery manner and included time for self-initiated projects. The intervention of the teacher in the students' work was kept to a minimum, but new commands and ideas were introduced in a structured way and beginning activities for using them were initiated by the teacher. An entire lesson on the benefits of using subprocedures (decomposition, reusability, and compartmentalization) was included. Since the memory load of early programming is high, reminders of all commands and concepts were posted on a large bulletin board, within easy view.

#### *Assessing Learning*

Study 1 involves a between-subjects comparison. All pairs received the same LOGO instruction including explicit instruction in debugging. However, half of the pairs began with graphics and then moved into list-processing, while the other half took the two mini-courses in the reverse order (depicted in the upper portion of Fig. 7). There were no significant differences between the students taking graphics first and those taking list processing first in terms of age, sex, standardized achievement scores, or access to computers at home.

Debugging performance was measured at three times during each mini-course (1, 2, and 3 for graphics and 4, 5, and 6 for list-processing in Fig. 7), so students took a total of 6 debugging tests. The first debugging test was taken after special attention to subprocedures but before debugging instruction (indicated by the vertical bar in Fig. 7). These first tests (number 1 for Group A and number 4 for Group B) therefore measure the level

[4] The Montessori philosophy emphasizes multilevel classrooms; at this particular school, the third through sixth grades were combined into a class with one teacher.

of skill prior to explicit debugging instruction. Tests were not counterbalanced with test time; they corresponded, instead, to the concepts being learned at that period in the course. At any point during the second mini-course, better performance on graphics tests by students taking graphics second than by students taking graphics first can be attributed to learning in their list-processing mini-course. Likewise, better peformance on list-processing tests by students taking it second than by students taking it first can be attributed to learning in their graphics mini-course.

Each debugging test contained six bugs. For the graphics test, five of the bugs were semantic bugs while only one was a syntactic bug. Syntactic errors include misspellings, inappropriate punctuation or spacing, and other errors which interrupt the running of the program. Semantic errors do not stop the program from running but do cause faulty output. Since syntax tends to be more of a problem for list-processing, those tests contained three syntactic and three semantic bugs. We expected that the children would have difficulty giving think-aloud protocols in the cognitively demanding debugging situation. For this reason, children worked in pairs and were encouraged to talk to their partners while debugging.

*Assessing Transfer*

Study 1 includes a within-subjects pretest/post-test comparison of performance on the transfer tasks decribed earlier (depicted at the bottom of Fig. 7). A midtest was also included to monitor transfer after the first

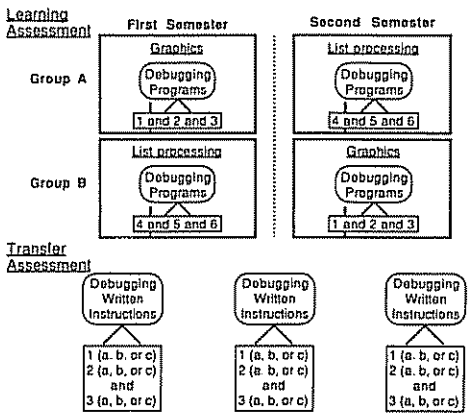



FIG 7. The design of Study 1. For the learning assessment, the boxes represent LOGO experience, with debugging tests given a total of six times for each group. The vertical bar indicates when the explicit debugging lesson was given. For the transfer assessment, the ovals indicate that transfer tests were given before the first mini-course, between mini-courses, and after the second mini-course. At each transfer test time, each student took one version (a, b, or c) of each of three test types (1, 2, and 3).

mini-course only. At each of the three test times, each student took three types of transfer tests (1, 2, and 3 in Fig. 7), all of which involved debugging a written set of instructions about how to achieve a well-specified goal. One-third of the students took each version at each test time (a, b, or c in Fig. 7). We predicted that students' ability to debug these non-computer tasks would improve as a result of learning debugging in LOGO.

## Learning Results

Since the two groups of subjects took the two mini-courses in different order, we could compare the debugging strategies and performance on the same tests of students who had no prior debugging instruction or experience with students who had debugging instruction and experience in another LOGO domain. For example, all of the students had the same amount of LOGO graphics experience when they took test 1 (see Fig. 7), but Group B had previous debugging instruction and experience in LOGO list-processing. If the students in Group B learned the general goal structure of debugging in list-processing, they should be able to apply it in graphics because our model shows that the goal structure is identical for debugging LOGO graphics and list-processing programs. Similarly, if the same students have learned effective bug identity and location heuristics in list-processing, at least some of them should be applicable in graphics since our model has several discrepancy-bug mappings (primarily those dealing with syntactic errors) and program structure cues that apply to both domains. In addition, the subskills required by the debugging process, such as editing and running programs, are similar in graphics and list-processing. There was no comparison group (a group that got no explicit instruction in debugging); however, our results can be compared to the results from the pilot studies. We present evidence that when given debugging instruction based on the performance model, students were able to acquire effective debugging skills. Without such instruction, students in the pilot studies debugged poorly.

The goal of the learning assessment was to determine which of the debugging skills students were able to acquire from the direct instruction provided in both LOGO minicourses. Debugging episodes were transcribed directly in terms of the model's goal structure. Each statement and action was categorized by goal type. Transcripts were divided into episodes based on the test goal. A new cycle began each time the subjects ran a program or ran a series of programs without doing anything else in between. The time at the start of each cycle was entered on the transcript and the order of comments and actions was preserved by numbering each entry. In addition, all experimenter interaction with the pair of students was recorded and numbered in sequence with the other events.

Figure 8 shows one cycle of a typical transcript. (One such transcript

was generated for each cycle of each debugging episode for each problem for each student pair.) The example begins with a test of the program SEASHORE. A negative evaluation is indicated by the comment "Oh no!" The discrepancy was described as "the boat side is too long," and the bug was proposed as "it went FD too much." The students know that the program representation includes subprocedures, and one student asks, "Which subprogram should we try?" The other specifies the buggy subprogram as "boat." They edit the subprogram BOAT and scan for a FD command with a large argument. FD 90 is isolated and understood to be the incorrect move. "It should be 40," says one student (probably since the FD command corresponding to the other side of the boat is FD 40). The students then replace 90 with 40 and exit the editor to retest the program.

Several related aspects of debugging skill were extracted from the debugging transcripts in order to precisely characterize the students' acquisition of different parts of the model's skills: speed, efficiency, clue gathering, and search strategies. For the purpose of these analyses, the within mini-course test scores have been averaged across students to get a score for each mini-course as a whole. First, the debugging speed was measured by dividing each student pair's total debugging time by the number of bugs fixed (out of a possible six). In cases when the students corrected all six bugs, this included all debugging time up to but not including the final run (when the program worked correctly). When the students did not correct all six bugs, the time was measured up to but not including the program run that confirmed their last correct fix. This adjustment excludes time at the end of the session spent on bugs that were never corrected, thus the speed measure included only time spent on bugs that were fixed. The model makes no predictions about absolute debugging time, but the time would be expected to decrease as debugging skill

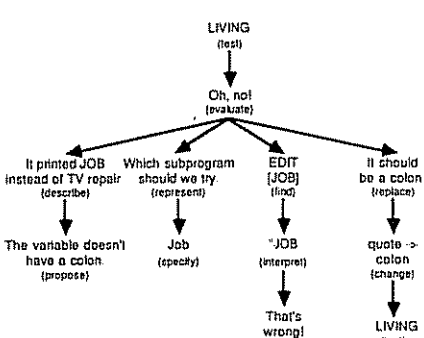



FIG. 8. Sample program debugging transcript.

improves since strategies would shift from brute force to more focused search, which requires fewer subgoals. (More detailed descriptions of the results and statistical analyses for Study 1 are presented in Carver, 1986.)

We expected that students would require less time to correct each bug in the second mini-course than in the first if their strategies shifted from brute-force to more focused search, which requires fewer subgoals. In fact, mean time per bug decreased from 9 min 6 s in the first mini-course to only 5 min 25 s in the second ($F(1,76) = 42.69$, $p < .01$). As debugging skill improves, students should also take fewer cycles (each isolated test goal initiates a cycle) to fix each bug. Debugging efficiency improved significantly by about one cycle per bug from the first to the second mini-course (from a mean of 2.9 to a mean of 2.05 cycles per bug, $F(1,76) = 12.64$, $p < .01$). In the second mini-course, several pairs of students averaged close to a perfect score of one cycle per bug and a few actually took fewer than one cycle per bug because they fixed several bugs in one procedure without exiting to retest the program in between.

The goal structure of efficient debugging, as presented to the students in our curriculum, stresses the value of seeking cues to narrow the search. As students' understanding of that goal structure and knowledge of discrepancy-bug mappings and of location cues increases, they should begin to make more comments about the bug's likely identity and/or location before suggesting a command as the bug. Analysis of the students' comments during program debugging provides some support for this prediction. Accuracy of students' discrepancy descriptions was nearly 100% in both mini-courses, whereas the accuracy of the bug proposals and bug location comments remained constant at about 75%. The important change was that students were somewhat more likely to make these statements *before* beginning to search for the bug in the second mini-course. The proportion of discrepancy comments made prior to search increased from less than 65% on tests in the first mini-course to over 80% on tests in the second. Similarly, the proportion of early bug proposals increased from only 25 to over 50% and the proportion of early bug location comments increased from 75 to 85%. However, these trends fall just short of significance, primarily due to the small $N$: students made very few comments overall, describing the discrepancy aloud for about half of the cycles, but proposing bugs for only between ¼ and ⅓ of the cycles. Location descriptions were more frequent than bug proposals but were still offered for less than half of the cycles.

Increasingly focused search should also cause a decrease in unnecessary search, i.e., the number of correct subprograms the students erroneously edit and the number of correct commands they erroneously identify as the bug. The subprogram structure of the buggy programs was easy for the students to recognize because a list of subprogram names was

displayed on the computer screen at the beginning of the test. Students rarely misjudged which subprogram did which part of the program because the subprogram names were related to their function. The mean number of times subjects looked into a program that did not contain the bug ranged from 2 to 3 per test (i.e., per six bugs) in both mini-courses. Most of these errors resulted from forgetting the names of the programs or forgetting what subprograms existed. For list-processing, there was a significant decrease from the first mini-course to the second in both the amount of brute-force search (reading and checking each command in a program) and the number of false alarms (correct commands which were misidentified as the bug). Brute-force search decreased from 4.33 per test to 0.33, $F(1,4) = 8.47, p < .01$, and total number of false alarms decreased from 29 to 9, $F(1,4) = 10.91, p < .01$. For graphics, there were corresponding reductions, but they were not significant (from 9.3 to 4.6 for brute-force search and from 68 to 50 false alarms).

In summary, following the one lesson that focused on debugging, students' debugging speed and efficiency improved. They began to use the new strategies we had taught, especially by asking themselves which subprocedure was likely to contain a particular bug. They could, however, have made more use of the list of discrepancy-bug mappings. They memorized some of the more common mappings early, but many students used their problem–cause mapping chart only as a last resort. Nonetheless, they used brute-force strategies less often and made fewer false alarms. In fact, most of them did better than the LOGO teachers tested by Jenkins (1986) on the same programs.

## Transfer Results

The goal of the transfer analysis was to show that the focused search strategies learned in the LOGO environment would transfer to similar debugging situations not involving programming. For each transfer problem, subjects' discrepancy-description, bug-proposal, and bug-location comments were transcribed, as well as the type of scanning strategy they used to locate the discrepancy initially. Each line the subject read and each time the subject flipped back to look at the plan and outcome were recorded so that the search process could be quantified and the search strategy characterized.

Four qualitatively different search strategies are possible. The worst strategy is to read or simulate haphazardly (a few lines here, a line or two there) or to simulate nothing. (Simulation refers to actually interpreting the instruction to determine what effect it would have; usually this process involves referring to the diagrams or tables.) This strategy is unlikely to find the bug because the subject may or may not even read the buggy line let alone bother to check it against the desired outcome. A slightly

better strategy, *brute force*, is to read every line and simulate every instruction. This strategy is likely to find the bug but processes many unnecessary lines. Better yet is using a *self-terminating brute-force* strategy: to read and simulate everything until the bug is located and then disregard the rest. The best strategy is to *focus selectively* only on the appropriate subsection of the program, maybe even only on the part near the bug.

Subjects' reading and simulating strategies for the first pass through the written instructions were categorized for each of the three items on the pre-, mid-, and post-tests. Figure 9 shows percentage distributions across all transfer tests of the reading/simulating strategy combinations for each test time. There were no differences between Groups A and B so the data have been collapsed across groups. The total number of combinations per test time should have been 66 (22 subjects times three tests at each time); the actual totals are slightly lower due to several videotape failures.

There were striking improvements in students' performance on the transfer tests. On the pretests students' behavior was very much like the brute-force strategy of the debugging model. The top left cell shows that the predominant strategy (45%) was to read all the commands and simulate none, but use of this weakest of all strategies dropped to 20% by the midtest and 12% by the post-test. For reading, use of brute force dropped from 58% on the pretest to 25% by the midtest. Analysis of the column marginals in Fig. 9 reveals that most of the reading strategy shift took place by the midtest: pre-mid-post $\chi^2$ (4) = 35.55, $p < .01$; but pre-mid $\chi^2$ (2) = 29.21, $p < .01$; while mid-post $\chi^2$ (2) = 2.49, n.s. Two-thirds of the students simulated none of the lines on their first reading of the directions on the pretest. Some students had shifted to a more focused simulation strategy by the midtest, but nearly half of the students were still not

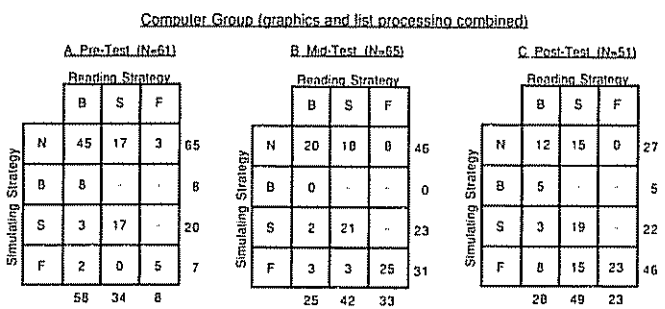



FIG. 9. Change in search strategies on transfer task in Study 1. Numbers indicate the percentage of trials on which each strategy combination was used. N, No search; B, brute-force search (every instruction); S, self-terminating search (every instruction up to the bug); and F, focused search (only instructions near or similar to the bug).

simulating any commands. By the post-test, nearly half of the students were using the most advanced simulation strategy. For simulating, strategy shifts occur at the midtest and continue through the post-test: pre-mid-post $\chi^2$ (6) = 51.01, $p < .01$; pre-mid $\chi^2$ (3) 26.62, $p < .01$; mid-post $\chi^2$ (3) = 12.89, $p < .01$.

In addition to improved strategies, the students made more correct fixes on later tests. The percentage of subjects who made the correct change increased from an average of 33% on the pretests to 56% on the midtests to 64% on the post-tests, $F(11,16) = 5.53$, $p < .05$. The students' mean search time decreased significantly from the pretest to the post-test as a result of the shift to the selective search strategy (from 4 min 50 s on the pretest to 3 min 41 s on the midtest to 3 min 4 s on the post-test, $F(1,16) = 6.87$, $p < .025$). The improvements on these tasks are primarily a result of increasing use of location clues.

Also, the number of students who read and simulated lines after the initial bug was identified increased from the pretest to the post-test from an average of 2 students (out of 22) on the pretests to an average of 8 students on the midtests to an average of 11 students on the post-tests ($\chi^2$ (4) = 13.77, $p < .05$). One thing that students have clearly learned from debugging experience is that a fix may be wrong or may make things worse. The production system model always instructs the user to recheck the program once a fix has been made. Even though retesting is not easy for debugging noncomputer directions, students demonstrated that they knew a very important goal: to check the fixes. They were also able to transfer that part of their strategy despite having to tailor it to a new situation.

## Discussion

Our thesis is that following explicit debugging instruction, students will develop effective debugging strategies in the LOGO context and transfer them to nonprogramming contexts. Study 1 provided evidence that third through sixth grade students who were taught debugging skills in the context of a 25-h LOGO graphics mini-course were better at debugging in a subsequent list-processing mini-course than classmates who took list-processing first. Also, those who were taught debugging in the list-processing mini-course did better on debugging tests in a subsequent graphics mini-course than those who took the graphics course first. All of these students demonstrated better debugging strategies, took less time, and were more accurate on transfer tests of debugging in nonprogramming contexts after debugging instruction than before.

The improvements students demonstrated between the first mini-course and second mini-course in both graphics and list-processing is evidence that learning did take place. However, it is not possible to trace

the within-mini-course improvement because the study used a between-subjects design and did not counterbalance tests within each mini-course. Also, because the students worked on the debugging tests in pairs, it was not possible to trace an individual student's acquisition and subsequent transfer of debugging skills.

There are alternative hypotheses that could account for the improvements on the transfer tests. One is that students would improve on the post-test as a result of their natural development over the time span of the LOGO course. The control for the effect of maturation was built into the treatment group since the age range of the students was 3 years while the span of the study was only 6 months. A comparison of the older half of the students with the younger half revealed that the older half of the subjects were not more accurate and did not demonstrate better search or checking strategies than the younger subjects. The older subjects were significantly faster than the younger subjects on the pretest ($t$ [64] = 3.41, $p <$ .05), but there was no midtest or post-test difference. Another alternative hypothesis is that improvement on the post-test is due merely to learning how to do that type of test. Study 1 had no control group without LOGO experience between transfer test sets, so the improvement on the transfer tests could be a practice effect. Study 2 was designed to facilitate better learning assessment and to test the practice effect hypothesis. In addition, we had the opportunity to test one group of students 4 months after the end of their LOGO course to determine the stability of the transfer effect.

## STUDY 2

### Design

For Study 2, we implemented two list-processing LOGO courses. All 34 of the sixth graders in a private girls' school participated. This study includes a within-subjects design with individually administered tests. We also used isomorphic tests that could be counterbalanced with test time so that individual students' improvement could be monitored more precisely. Half of the subjects were randomly assigned to Group I ("LOGO first") and half to Group II ("Study Hall first"). The subjects in Group I attended a LOGO list-processing class for two 40-min periods per week for the first semester, while the subjects in Group II were in study hall. During the second semester, the treatments were reversed. (See Fig. 10.)

As with Study 1, all lessons were taught in a guided discovery manner and included time for self-initiated projects. Reminders of all commands and concepts were included on handouts instead of being posted on bulletin boards.

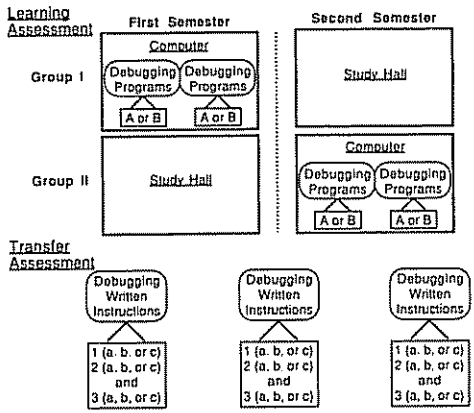



FIG. 10. The design of Study 2. Each group took a one semester LOGO course. Group I took the LOGO course during the first semester, while Group II served as a control group and took Study Hall. During the second semester, the treatments were reversed. During each course, each student took two debugging tests. The transfer tests were given as in Study 1.

*Assessing Learning*

During each list-processing course, the subjects took two programming and two debugging tests (one set shortly after the debugging instruction and one near the end of the course) to monitor their developing debugging skills. Two isomorphic program descriptions were used so that they could be counterbalanced with test time. In addition, two sets of eight bugs (five semantic and three syntactic) were created and also counterbalanced with the two test times and the two isomorphic programs. The sets of bugs were isomorphic in the sense that they caused similar discrepancies at the same point in the program output. (Table 4 shows the discrepancies and bugs for sets A and B.)

At each test time, each student was given one class period to write a program according to one of the program descriptions. During a subsequent class period, the student was asked to fix a program written (according to the same description) and bugged (with one of the sets of bugs) by the experimenters. At the second test time, each student used the other program description and was asked to fix a program with the other set of bugs.

*Assessing Transfer*

Debugging skills in a noncomputer context were assessed at the beginning of the year, in the middle of the year (after half of the subjects had

TABLE 4
Isomorphic Bug Sets

| Set A | Set B |
|---|---|
| Semantic bugs | |
| D: Doesn't wait for input (uses old value) | D: Doesn't wait for input (uses READLIST) |
| B: Forgot MAKE "variable READLIST | B: Quoted READLIST |
| D: Correct response doesn't match | D: Correct response doesn't match |
| B: Switch READLIST/READWORD | B: Switch EQUALP/MEMBERP |
| D: :FRIEND gets printed | D: FRIEND gets printed |
| B: Move bracket after variable | B: Switch " to : |
| D: Questions out of order | D: Asks wrong question |
| B: Reverse the calls | B: Reverse conditional [1][2] → [2][1] YES → NO |
| D: Wrong name value in salutation | D: Correct response doesn't match |
| B: Variable name used twice—change one | B: Used wrong variable name—change it |
| Syntactic bugs | |
| D: I don't know how to WHAT | D: PRINT didn't output to IF |
| B: Need brackets around list | B: Need brackets around PRINT in IF |
| D: _____has no value | D: _____has no value |
| B: Change : to " in MAKE | B: Misspelled variable name |
| D: Too much inside ( )'s | D: I don't know what to do with_____ |
| B: Forgot PRINT | B: Forgot ( ) around PRINT |

*Note.* Each set contains five semantic and three syntactic bugs. For each, we list the observed discrepancy (D) between the actual and the desired output and the bug (B).

been given explicit debugging instruction in the LOGO context), and at the end of the year (after the rest of the subjects had taken the LOGO course). At each test time, students took the transfer test battery used in Study 1. Once again, we predicted that students' ability to debug these noncomputer tasks would improve as a result of—and only to the extent of—learning debugging in LOGO. In the following sections, we report the results of the learning assessments for Group I only and the results of the transfer assessments for both groups.

## Learning Results

As in Study 1, students' performance on quantitative measures of debugging skill improved during the course. The students' mean debugging speed decreased from 9 min 36 s per bug on the first debugging test to 4 min 43 s on the second, $F(1,33) = 13.00$, $p < .001$. Students also took fewer cycles to fix each bug on their second test. The mean number of cycles students took to fix each bug decreased from 3.02 on the first test to 1.60 on the second, $F(1,33) = 8.81$, $p < .01$.

The protocol data did not reveal any improvement in students' use of bug identity clues. On both debugging tests, students averaged slightly more than one discrepancy description per debugging cycle and were

accurate 90% of the time. This high accuracy rate, similar to that found in Study 1, is not surprising since describing the discrepancy between the goal output and the actual output requires no knowledge of programming or debugging. In contrast, students did not propose a bug on every debugging cycle. There was a slight but nonsignificant decrease in the percentage of cycles on which they proposed bugs (from 63% on the first test to 50% on the second). This suggests that students were still having difficulty finding clues to the likely bug before looking at the program code. One reason for this difficulty was that students made ineffective use of their lists of discrepancy-bug mappings. They memorized some of the more common mappings early in the course, but rarely recorded new mappings after the debugging lesson and consulted their listing only after prompting.

In contrast, students' use of location clues did improve. The proportion of cycles on which students commented about the structure of the program increased from only 19% on the first test to 34% on the second test, $F(1,32) = 7.79, p < .01$. Accuracy of these comments increased slightly from 74 to 95%. In conjunction with the increase in comments about program structure, the frequency with which they specified the bug location increased slightly from 67 to 82% and their accuracy rate remained constant at about 60%. In general, students' comments reveal a slight increase in the clues they gathered about the bug's likely location, but their actual search behavior demonstrates increasingly narrow search more dramatically.

The number of times the students looked into a subprocedure that did not contain the bug (or information relevant to the bug) should decrease as the students learn to use location clues. The number of subprocedures students entered (per bug fixed) decreased from 4.42 on the first test to 2.26 on the second test, $F(1,32) = 10.68, p < .01$. Part of this decrease can be attributed to a decrease in the number of correct subprocedures students erroneously edited (from 1.13 per bug fixed on the first test to .46 per bug fixed on the second, $F(1,32) = 5.09, p < .05$). Increased attention to the structure of the program may also contribute to the decrease since students may less often need to look into correct subprocedures to find information about flow of control. Students should also make fewer false alarms as they learn to use clues to the bug's identity. The mean number of correct commands which were misidentified as the bug (false alarms) and subsequently changed from 7.29 on the first test to 2.41 on the second, $F(1,32) = 16.14, p < .001$.

As in Study 1, after the one lesson focusing on debugging, students in Group I began to develop effective debugging strategies. In particular, they restricted their search to a smaller portion of the program and to a more specific type of bug. They were slightly more likely to mention

accurate clues about the bug's likely identity and location on the second test, and they were less likely to change inappropriate types of bugs and search inappropriate locations. On the first debugging test, only a few students fixed all eight bugs within one class period and all students needed extensive help from the experimenter both to identify and locate bugs. On the second test, most students successfully corrected all of the bugs with less help with gathering clues. We conclude that focused search had become an effective debugging strategy for them in the LOGO context.

## Transfer Results

Students in both groups spent about 30 s looking at the discrepancy information before turning to the directions at all three test times. Their comments were restricted to orientation to the type of discrepancy information ("This is what they wanted and this is what they got.") and descriptions of the discrepancy ("This table should be on the west side."). As with the LOGO debugging, most of the discrepancy descriptions were correct. Students rarely made comments, however, about the likely identity of the buggy directions, presumably because they had inadequate knowledge about the discrepancy-bug mappings in the transfer domains. Students did make comments about the likely location of the bug on the distribution tests where the discrepancy information was presented in a table with section headings. Here, the location clue was the most salient, and the accuracy of students' comments was high for both groups. The difference between the computer and control groups emerges only when the actual search process is considered.

Each subject's reading and simulating strategies were categorized for each of the three items taken at each test time by the second author and a research assistant (inter-rater reliability was 86%). Figures 11a–11f show the strategy usage for the two groups at the three test times. For each test, each subject's reading and simulating strategies were categorized, so each figure includes three strategy combinations for each subject. For example, the upper left cell in Fig. 11a shows that the brute-force reading/no simulating strategy combination accounted for 47% of the episodes among the 17 Group I subjects taking three tests each ($N = 51$). This cell represents the worst strategy combination. The bottom right cell (focused reading/focused simulating), which represents that best strategy combination, accounted for 8% of the pretest episodes by Group I. (Recall that the corresponding figures for Study 1 pretests are 45 and 5%.)

Analysis of the column and row marginals in Figs. 11a and 11b reveals that students in Group I shifted toward more focused reading and simulating strategies ($\chi^2$ (2) = 21.87, $p < .01$ for reading and $\chi^2$ (3) = 27.16, $p < .01$ for simulating). Comparable analysis for the Group II data shown

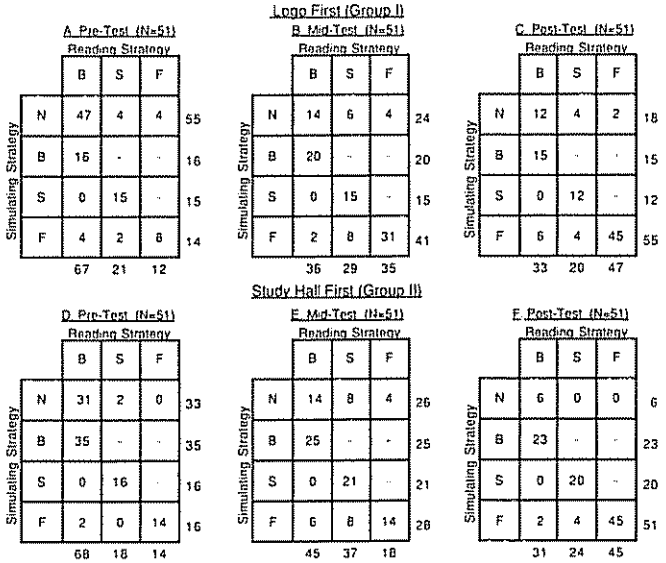



FIG. 11. Change in search strategies on transfer task in Study 2. Numbers indicate the percentage of trials on which each strategy combination was used by the experimental and control groups. N, No search; B, brute-force search (every instruction); S, self-terminating search (every instruction up to the bug); and F, focused search (only instructions near or similar to the bug).

in Figs. 11d and 11e does not show any significant strategy shift. However, analysis of the column and row marginals for Figs. 11e and 11f does show a significant shift for Group II after their LOGO course ($\chi^2$ (2) = 16.92, $p < .01$ for reading and $\chi^2$ (3) = 19.30, $p < .01$ for simulating). Similar analysis of Figs. 11b and 11c shows no decrement in performance for Group I despite a semester away from LOGO ($\chi^2$ (2) = 3.54 n.s. for reading, and $\chi^2$ (3) = 3.95, n.s. for simulating).

In addition to strategy choice, the changes subjects made were scored as either correct or incorrect. In order to be scored as correct, subjects had to debug the directions rather than merely add to them. The proportion of Group I students who accurately debugged the directions increased from 33% on the pretest to 55% on the midtest and post-test, $F(1,16) = 4.12$, $p < .1$. In contrast, the accuracy rate of Group II remained constant (41 and 39%) from the pretest to the midtest, but improved to 65% after their LOGO course, $F(1,16) = 7.62$, $p < .025$.

## Tracing Individual Learning and Transfer Patterns

Although it is well established that amount of positive transfer is cor-

related with amount of initial learning (Ellis, 1965), as Gick and Holyoak (1987) point out, most recent studies of transfer of problem-solving skills do not analyze this relation. The individual assessments in Study 2 enabled us to go beyond the aggregate results to test whether the subjects whose debugging skills improve most in the debugging course are also the ones whose strategies improve most on the transfer test. In order to facilitate such correlational analysis, we converted our strategy classifications into a numerical score by giving 0 points for No Strategy, 1 point for Brute Force, 2 points for Self-terminating Search, and 3 points for Focused Search. Since each subject gets two strategy scores (reading and simulating) for each of three tests at each test time, possible scores range from 0 to 18. We are not arguing that these strategies are, in fact, equidistant on a strategy continuum but rather that this simple scoring will reflect the strategy differences. In fact, analyses of variance on these quantitative strategy data show the same pattern as the $\chi^2$ analyses of the categorical data showed. The Group I mean score increased from 7.00 on the pretest to 11.06 on the midtest, $F(1,32) = 9.28, p < .01$. The means for Group II did not increase significantly (7.76 on the pretest and 9.35 on the midtest).

The correlation between changes in strategy scores from pretest to midtest and changes in debugging efficiency from test 1 to test 2 for students in Group I was $r = .52$ ($p < .05$). All students either improved on both measures or failed to improve on both; there were no students who improved in LOGO debugging but got worse on the transfer task and no students who got worse in debugging but improved on the transfer task. Similarly, there was a significant correlation of .58 between students' change in strategy scores relative to their decrease in debugging time per bug fixed. For Group II, with the exception of two students, the change in strategy scores from the pretest to the midtest clustered around zero; in other words, with no LOGO experience there was no change in LOGO debugging efficiency and no change in transfer test search strategy.

## GENERAL DISCUSSION

The aim of these studies was to test the thesis that (a) children can learn high-level thinking skills from computer programming if the component skills are precisely specified and taught directly, and (b) once the skills have been learned, they can be evoked and applied in other domains that share relevant features. We specified a model of effective debugging skill that emphasized the importance of gathering clues to a bug's identity and its location in order to focus search for the bug in the program code. Based on this model, we designed one explicit debugging lesson. This lesson was inserted in what were otherwise conventional LOGO courses, followed by numerous opportunities to apply the procedure suggested in

that lesson. We then used the model to assess students' learning and transfer of debugging skills. Our results indicated that students in the LOGO courses did acquire focused search strategies that increased their debugging efficiency and decreased their debugging time. On the transfer tests, they shifted from unfocused (serial) to focused search strategies. In Study 2, we found a positive correlation between the amount of improvement in LOGO debugging and the amount of strategy shift on the transfer tests and that this effect did not decline over a 4-month period containing no further programming experience.

## Basis for Effective Transfer

These findings are in contrast to the largely negative results from previous studies of the transferability of high-level problem-solving skills from computer programming experience (e.g., Garlick, 1984; McGilly et al., 1984; Mohamed, 1985; Pea, 1983). We discuss the basis for the effectiveness of this transfer in terms of three issues: the role of the detailed task analysis, the importance of multiple instances in the base domain, and the distinction between near and far transfer.

### *Detailed Task Analysis*

We believe that the key to our students' acquisition and transfer of debugging skills was the careful task analysis of debugging skill components and the explicit debugging curriculum derived from that analysis. Preliminary reports from Clements (1987) and Perkins (1987) also emphasize the importance of precise instruction in metacognitive strategies. Although we have attributed the effectiveness of this approach primarily to the task analysis, we do not mean to confuse sufficiency—which is all our data can logically support—with necessity. Many other factors involved in the implementation of any instructional experiment, such as the skill of the teacher (who was the same throughout these studies) or the repeated guidance in application of the debugging procedure, may also play an important role in facilitating transfer. Furthermore, it remains to be seen how effectively this detailed task analysis can be performed in other instructional areas, both within and beyond the domain of programming. Several successful examples of fine-grained analysis of learning and transfer have been reported in such domains as text editing (Singley & Anderson, 1985) and learning to operate a complex device (Kieras & Bovair, 1985; Polson & Kieras, 1985), but these studies did not derive a set of instructional goals from their production-system analyses. The full set of issues can only be resolved by further exploration of the use of formal cognitive objectives in instructional design and assessment.

*Multiple Instances for Effective Learning in the Base Domain*

Gick and Holyoak (1987) summarize the literature showing that people need at least two different examples of an underlying strategy or concept in order to induce a schema sufficiently general to facilitate transfer to a new domain. If we adopt a coarse "grain size" in interpreting our studies, then we can view each of the LOGO topics (lists or graphics) as an exemplar of the debugging strategy. This interpretation would lead to the prediction that exposure to both graphics and lists would yield substantially higher transfer scores than exposure to only one of them. But in Study 1, much of the transfer from programming to nonprogramming domains occurred by the midtest, after only one "example," and Study 2, which only taught lists, produced an equivalent amount of transfer. However, this grain size equates three very large aggregates—graphics debugging instruction, list-processing debugging instruction, and the set of transfer problems—with Gick and Holyoak's two base-domain problems and the target transfer problems, respectively. The vast difference in temporal and problem grain sizes seriously weakens the comparison.

A more accurate rendering, one that is consistent with the Gick and Holyoak position, views each curriculum as providing not only explicit instruction in the basic schema for focused search (i.e., the goal tree), but also numerous and varied examples of the schema in operation (i.e., all the instances of actual program debugging). Thus, by the time students encountered our midtest transfer problems, they had constructed a sufficiently general debugging schema for analogical transfer to occur. In addition to repeated opportunities for developing the schema, our procedure utilized another one of the important factors that, according to Gick and Holyoak, facilitate transfer: the "similarity of the learner's mental representation of the training and transfer tasks." That is, the underlying goal structure of debugging in both the base and the transfer domains was the same.

*Transfer: Near or Far?*

In assessing instructional effectiveness, one cannot escape the vexed issue of the extent to which transfer is to "far" vs "near" domains. With respect to the transfer tasks used in these studies, two questions can be raised: (1) Are they examples of "near" or "far" transfer from the base domain of debugging LOGO programs? The answer to this question is implicit in any evaluation of the extent to which there really are general cognitive benefits of learning to program. (2) Is there a principled basis for deciding that, within the different types of transfer tasks, some are "closer" to the base domain than others? If so, then we would expect better transfer to the closer tasks, and if we found it, we would have additional support for the original cognitive analysis.

Answers to both questions hinge on our ability to construct measures of task similarity among the base domain and the three versions of the transfer tasks. But even within the limited set of transfer tasks used in the present study, this is very difficult to do. The three general types of transfer items included arrangement tasks, involving physical layouts of furniture, block structures, or table settings; distribution tasks, in which different amounts of money, food, or other items were allocated to different classes of people; and route-following tasks, in which someone followed a set of map-following directions based on landmarks and relative turns. With respect to the first question, it could be argued that these transfer tasks are all rather close to the base domain. The tasks were constructed so that the original goal structure for debugging programs would apply to all of the transfer tasks, and the instructions to be followed for each task were formatted so as to emphasize their decomposability into subcomponents (see Table 3). However, at the level of task-specific knowledge, the transfer tasks are much less like the original programming tasks than are the well-known problem isomorphs in which transfer has been shown to be so difficult (cf. Gray & Orasanu, 1987). Differences abound in the nature of the discrepancies, in the discrepancy-bug mappings, and in the pragmatic knowledge about the specific transfer domain. In this regard, all of our transfer tasks are much less similar to LOGO than those used in many other LOGO transfer studies.

A similar problem arises with respect to the question of relative distance from LOGO to the different transfer tasks. One might argue that the furniture-arrangement and route-following tasks are closer to LOGO graphics problems than the distribution tasks because they both present discrepancy information in a two-dimensional spatial array. Indeed, the route-following task might seem to be closest of all, because it requires that the subject follow a set of turn and move commands very much like those given to the Turtle in a LOGO program. However, the route-following task could also be considered the most distant from LOGO, inasmuch as it uses very complex maps (taken from commercially published road maps) and requires the ability to read map conventions, route numbers, and convoluted topologies. Or, one could argue that distribution tasks, because they usually involve a bug in an instruction about relative quantities (e.g., "deliver half as many elm and cherry trees as tulip trees"), tap underlying skills at translating "story problems" that are only remotely related to what is taught in a LOGO course.

Although some recent information-processing analyses have successfully identified the production (in a production system of the tasks under analysis) as the countable element in the identical-elements approach (e.g., Singley & Anderson, 1988), we believe that in tasks like the ones studied in our work, there is no principled way to allocate relative influ-

ence to productions at different levels, or "grain sizes," in counting the number of identical elements. Thus, while we would expect a near perfect overlap between the production sets for generating goal trees corresponding to the base and transfer tasks (indeed, that was our intent in constructing the transfer tasks), it is not clear how to quantify the amount of domain-specific overlap between our base and transfer tasks.

Before we can decide whether any particular study represents an instance of "near" or "far" transfer, we need two types of additional analyses. First, we must perform a complete analysis of the elementary steps in each domain. Second, we must elaborate approaches like Singley and Anderson's (1985) procedure for estimating the relative contribution of general and specific components to the overall transfer scores. In evaluating Thorndike's (1906) critique of the "doctrine of formal discipline," Nisbett, Fong, Lehman, and Cheng (1987) conclude that:

> Thorndike was partially correct, after all, in that transfer applies only insofar as there are common identical elements But the identity lies at a much higher level of abstraction than he suggested . . ."

We concur with this general observation and further suggest that the quantification of this higher level of abstraction remains a crucial problem in understanding transfer of training.

## REFERENCES


Angell, J. R (1908). The doctrine of formal discipline in the light of principles of general psychology. *Educational Review*, 36, 1–14.

Bassok, M., & Holyoak, K. (1987). *Schema-based interdomain transfer between isomorphic topics in algebra and physics*. Working paper, University of Pittsburgh, LRDC.

Carver, S. M. (1986). *LOGO debugging skills: Analysis, instruction, and assessment*. Unpublished doctoral dissertation, Department of Psychology, Carnegie-Mellon University.

Carver, S. M., & Klahr, D. (1986). Assessing children's LOGO debugging skills with a formal model. *Journal of Educational Computing Research*, 2(4), 487–525.

Clements, D. H. (1987) Componential employment and development in LOGO programming environments. In *Proceedings of the Biennial Meetings of the Society for Research in Child Development*. Baltimore, MD: SRCD, April 1987.

Clements, D. H. & Gullo, D. F. (1984) Effects of computer programming on young children's cognition. *Journal of Educational Psychology*, 76(6), 1051–1058.

Dalbey, J., & Linn, M. (1984). Spider world: A robot language for learning to program. In *Proceedings of the American Educational Research Association Conference*. New Orleans, LA: AERA, April 1984.

Ellis, H. C. (1965). *The transfer of learning*. New York: Macmillan

Ericsson, K. A., & Simon, H. A (1984). *Protocol analysis: Verbal reports as data*. Cambridge, MA: The MIT Press.

Garlick, S. (1984). *Computer programming and cognitive outcomes: A classroom evaluation of Logo*. Unpublished honors dissertation The Flinders University of South Australia.

Geva, E., & Cohen, R. (1987). Transfer of spatial concepts from LOGO to map-reading. In

*Proceedings of the Biennial Meetings of the Society for Research in Child Development*. Baltimore, MD: SRCD, April 1987.

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

Gick, M. L., & Holyoak, K. J. (1987). The cognitive basis of knowledge transfer. In S. M. Cormier & J. D. Hagman (Eds.), *Transfer of learning. Contemporary research and applications*. New York: Academic Press.

Gorman, H., Jr., & Bourne, L. E., Jr. (1983). Learning to think by learning Logo: Rule learning in third grade computer programmers. *Bulletin of the Psychonomic Society*, 21(3), 165–167.

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man–Machine Studies*, 7, 151–182.

Gray, W. D., & Orasanu, J. M. (1987). Transfer of cognitive skills. In S. M. Cormier & J. D. Hagman (Eds.), *Transfer of learning: Contemporary research and applications*. New York: Academic Press

Greeno, J. G. (1976). Cognitive objectives of instruction: Theory of knowledge for solving problems and answering questions. In D. Klahr (Ed.). *Cognition and instruction*. Hillsdale, NJ: Erlbaum.

Gugerty, L., & Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers*. Norwood, NJ: Ablex.

Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. In *Proceedings of the American Educational Research Association*. New York, NY: AERA, March 1982.

Jenkins, E. A., Jr. (1986). *An analysis of expert debugging of LOGO programs*. Working paper, Department of Psychology, Carnegie–Mellon University

Katz, I. R., & Anderson, J. R. (1986). *An exploratory study of novice programmers' bugs and debugging behavior*. Paper presented at First Workshop on Empirical Studies of Programmers, June 1986. Washington, D.C.

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in LISP. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers*. Norwood, NJ: Ablex.

Kieras, D. E., & Bovair, S. (1985). *The acquisition of procedures from text: A production-system analysis of transfer of training* (Technical Report 16 (TR-85/ONR-16). University of Michigan.

Linn, M. C., & Fisher, C. W. (1983). The gap between promise and reality in computer education: Planning a response. In *Making our schools more effective: A conference for California educators*. San Francisco, CA: ACCCEL, December 1983.

Mandinach, E. B., Linn, M. C., Pea, R. D., & Kurland, D. M. (1986). The cognitive effects of computer learning environments. *Journal of Educational Computing Research*, 2(4), 409–427.

Mayer, R. E. (1988). *Teaching and learning computer programming: Multiple research perspectives*. Hillsdale, NJ: Erlbaum

McGilly, C. A., Poulin-Dubois, D., & Shultz, T. R. (1984). *The effect of learning LOGO on children's problem-solving skills*. Working paper, Department of Psychology, McGill University.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13, 307–325.

Mohamed, M. A. (1985). *The effects of learning LOGO computer language upon the higher cognitive processes and the analytic/global cognitive styles of elementary school students*. Unpublished doctoral dissertation, School of Education, University of Pittsburgh

Nisbett, R. E., Fong, G. T., Lehman, D. R., & Cheng, P. W. (1987). Teaching reasoning. *Science*, 238, 525–631.

Olson, G. M., Sheppard, S., & Soloway, E. (Eds.) (1987). *Empirical studies of programmers: Second workshop*. Norwood, NJ: Ablex.

Palmer, S. E., & Kimchi, R. (1986). The information processing approach to cognition. In T. J. Knapp & L. C. Robertson (Eds.), *Approaches to cognition: Contrasts and controversies*. Hillsdale, NJ: Erlbaum.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic Books.

Pea, R. D. (1983). Logo programming and problem solving. In *Proceedings of the American Educational Research Association Conference*. Montreal, Canada: AERA, April 1983.

Pea, R. D., & Sheingold, K. (1987). *Mirrors of minds: Patterns of experience in educational computing*. Norwood, NJ: Ablex.

Perkins, D. (1987). Instructional strategies for problems of novice programmers. In *Proceedings of the American Educational Research Association Conference*. Washington, DC: AERA, April 1987.

Polson, P. G., & Kieras, D. E. (1985). A quantitative model of the learning and performance of text-editing knowledge. In L. Borman & B. Curtis (Eds.), *Proceedings of CHI '85 Human Factors in Computing Systems*. New York: Association for Computing Machinery.

Reed, S. K., Ernst, G. W., & Banjeri, R. (1974). The role of analogy in transfer between similar problem states. *Cognitive Psychology*, 6, 436–450.

Sauers, R., & Farrell, R. (1982). *GRAPES User's Manual*. Department of Psychology, Carnegie-Mellon University.

Simon, H. A., & Hayes, J. R. (1976). The understanding process: Problem isomorphs. *Cognitive Psychology*, 8, 165–190.

Singley, M. K., & Anderson, J. R. (1985). The transfer of text-editing skill. *International Journal of Man–Machine Studies*, 22, 403–423.

Singley, M. K., & Anderson, J. R. (1988). A keystroke analysis of learning and transfer in text editing. *Human-Computer Interaction*, in press.

Soloway, E., & Iyengar, S. (Eds.) (1986). *Empirical studies of programmers*. Norwood, NJ: Ablex.

Spohrer, J. G., & Soloway, E. (1988). Analyzing the high frequency bugs in novice programs. In E. Soloway & S. Iyenger (Eds.), *Empirical studies of programmers*. Norwood, NJ: Ablex.

Spohrer, J. G., Soloway, E., & Pope, E. (1985). Where the bugs are. In L. Borman & B. Curtis (Eds.), *Proceedings of CHI '85 Human Factors in Computing Systems*. New York: Association for Computing Machinery.

Thorndike, E. (1906). *Principles of teaching*. New York: Seiler.

(Accepted January 11, 1988)