ASSESSING CHILDREN'S LOGO DEBUGGING SKILLS
WITH A FORMAL MODEL*

SHARON McCOY CARVER
DAVID KLAHR
Carnegie-Mellon University

ABSTRACT
In this article, we propose a new way to assess children's acquisition of debugging skills in a LOGO environment. The assessment procedure is based on an explicit and precise model (in the form of a computer simulation) of good debugging skills. The model has four stages: 1) evaluating the program's planned and actual outcomes to determine that debugging is necessary, 2) identifying the bug by using descriptions of the discrepancy between the planned and actual outcomes to propose potential bugs, 3) locating the bug by using clues about the structure of the program to narrow the search, and 4) correcting the bug and retesting the program. We describe model-based measurements of the LOGO debugging skills actually acquired by students in a "typical" LOGO graphics course. Nine seventh- to nine-year-olds were given twenty-four hours of LOGO training over a three-week period. Students learned the editing and command generation skills prerequisite to debugging but were not able to interpret commands and use clues to identify, locate, and correct bugs. We conclude by discussing objectives for teaching the model's debugging skills directly.

1. PROGRAMMING, DEBUGGING,
AND COGNITIVE DEVELOPMENT

The possibility that the normal course of cognitive development can be substantially advanced by engaging children in particular activities has long had an appeal for parents and psychologists alike. The list of candidates for such mental exercise ranges from highly specific instruction (such as conservation training) to selected academic subjects (such as Greek, Latin or Geometry) to general classroom practices (such as "discovery learning"). The old question has recently

* This work was supported in part by a National Science Foundation Graduate Fellowship to the first author, and by grants from the National Science Foundation (BSN81-12743) and the Spencer Foundation to the second author.
reappeared in a new guise: "Does learning to program computers have a broad impact on children's thinking ability?" The implications of this question have been lucidly summarized by Pea and Kurland [1]. They comment on the widespread belief that

... through learning to program, children are learning much more than programming, far more than programming 'facts' It is said that children will acquire powerfully general higher cognitive skills such as planning abilities, problem-solving heuristics, and reflectiveness on the revisionary character of the problem solving process itself. This belief, although new in its application to this domain, is an old idea in a new costume which has been worn often before. In its common extreme form, it is based on an assumption about learning—that spontaneous experience with a powerful symbolic system will have beneficial cognitive consequences, especially for higher order cognitive skills. Similar arguments have been offered in centuries past for mathematics, logic, writing systems, and Latin [2-7].

1.1. Evaluating the Effects of Learning to Program

Many attempts to evaluate the cognitive consequences of learning to program have been made. This work is best exemplified by the LOGO assessments conducted at the Bank Street College of Education [1, 8, 9]. For example, Pea [9] addressed two of the many claims made by LOGO advocates [10-13]. The first part of Pea's study assessed the level of programming skills acquired by eight-to-nine-year-olds and eleven- to twelve-year-olds after about thirty hours of discovery learning with LOGO during one school year. Pea found that children performed at very low levels, even on such a fundamental aspect of programming as understanding primitive commands. Furthermore, when it came to writing programs, children could do little more than write chains of immediately executed commands, and they had great difficulty writing programs using variables. Finally, with respect to debugging, many children were able to debug syntax errors, but very few found procedural errors (misordered commands).

The second part of Pea's study was designed to determine whether learning LOGO programming enhanced general planning skills. The results are typical of the findings from most of the LOGO evaluations: First, most children do not learn much about programming or debugging in the twenty to forty hours of exposure that they usually get in these studies. Second, what little they do learn does not transfer to planning or problem-solving tasks in a nonprogramming context. Many similar evaluations have been reported, and they have, in general, failed to demonstrate any clear advantage for children who have been exposed to programming [14-18].

1 See [1] for a critical review of the literature
Nevertheless, although the LOGO claims remain largely unsubstantiated, it would be premature to conclude that they are false. The repeated failure to find any important effects of learning LOGO could result from the fact that most studies use a poorly defined independent variable, such as “hours of instruction” or “hours of exposure,” rather than demonstrated mastery levels of specific LOGO concepts. Therefore, transfer of debugging skill is assessed without a prior determination that the skill was ever acquired in the first place.

1.2. Basing Assessment on a Formal Model

One central aspect of learning to program is learning how to debug faulty programs, and this aspect has been identified as one of the “powerful ideas” that can generalize far beyond the programming context in which it is acquired [12]. But “debugging” is an ambiguous term, and this makes it difficult to evaluate claims for its cognitive consequences. It can be interpreted along a wide spectrum, ranging from an all-encompassing notion of self-improvement, to a more restricted view of eliminating faulty components in physical or mental procedures, to a constrained definition that is closest to its origins in computer programming: it is what one does to get a malfunctioning (buggy) computer program to work correctly. We believe that a search for transfer of debugging skills from programming experience is premature, because no one has yet made a clear statement of what “debugging skill” is, or how much of it is acquired during LOGO instruction.

In this article, we propose the following steps toward remedying this situation:

- Specify a model of debugging skill in terms of the cognitive processes that support it.
- Use the model to guide the measurement of what debugging skills LOGO students actually acquire in a “typical” LOGO curriculum.
- Account for the students’ debugging behavior in terms of the proposed model.

We believe that only after these steps have been taken will it be possible to design principled studies of transfer, and to conduct scientifically defensible evaluations of LOGO’s impact on cognitive development.

The rest of this article is organized as follows. In the next section, we give a brief description of LOGO. Then, in Section 3, we present a task analysis of the debugging process for LOGO programs. We then describe a study designed to probe children’s developing debugging skill during a LOGO course (Section 4). Next, we describe our evaluation of debugging skill acquisition as well as the acquisition of several important component skills (Section 5). Finally,

---

2 For example, juggling is one of Papert’s favorite domains for demonstrating learning via debugging, and Brown and Burton have convincingly demonstrated the need for debugging in children’s acquisition of arithmetic procedures [20].
in Section 6, we offer an explanation for the common finding that children are unable or unwilling to debug faulty programs, and we briefly describe a curriculum designed to foster debugging skills.

2. INTRODUCTION TO LOGO

LOGO is a sophisticated computer programming language whose complexity is hidden beneath a facade of simple, familiar commands. Young children with little training can create interesting graphics effects. LOGO can be used in "immediate" mode, where each command is executed as soon as it is entered, or it can be used to write programs of arbitrary complexity, which can subsequently be run in immediate mode or called by other programs. 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 an 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 primitives to direct the flow of control; these include `REPEAT n [list of commands]` which repeats the list of commands n times, `IF (conditional) THEN (command)` which is a simple conditional statement, and `STOP` which stops the execution of the current program and returns control to the calling procedure. In addition to the commands mentioned here, LOGO uses a simple screen editor for creating and revising programs.

The example programs in Figure 1 demonstrate the basic graphics capabilities of LOGO. The procedure definitions are listed on the left. For easier reading, some commands are indented and arranged on separate lines. The interactive calls and outcomes of the procedures are on the right. The starting position of the turtle is indicated by an arrow. In Figure 1a, the `REPEAT` statement is used to write programs to draw a square (SQU) and a triangle (TRI); then SQU and TRI are combined to make a program to draw a house. The last two commands, `LT 30 BK 40`, are not essential for drawing the figure but are added to return the turtle to its initial position. This positioning convention becomes important when more than one figure is drawn. In Figure 1b, each of the programs has

---

So named for the original physical turtle-shaped device that moved slowly around on the floor in the original versions of LOGO developed at the MIT AI lab. It is ironic that for most children now using LOGO, the origins of the term "turtle" are as obscure and generally irrelevant as those for "carriage-return."
Figure 1. Some simple LOGO programs. The left hand panels list the previously stored program, and the right hand panels illustrate the graphic effects of the commands listed. All drawings start with the turtle positioned at the lower left hand corner of the drawing, oriented to the north. a) LOGO program to draw a house comprised of a triangle atop a square, b) LOGO program to draw a house of variable size, c) Recursive LOGO program to draw a series of houses.
been revised to include a variable (:you, :me), so that one can draw houses of various sizes. Finally, Figure 1c illustrates how a recursive procedure can generate a neighborhood of houses of decreasing size.

3. ANALYSIS OF DEBUGGING SKILLS

Even with designs as simple as those shown in Figure 1, programmers (especially children) rarely create faultless programs on the first attempt; debugging is frequently necessary. In this section, we describe our task analysis of debugging. It is intended to capture, in the form of a concrete model, the decision processes and sub-skills used by an experienced LOGO programmer confronting a graphics debugging problem. We offer two levels of description. First, in Section 3.1, we give a general characterization of the major components of the model. Next, in Section 3.2, we describe the actual production system implementation of the model and give two examples of it debugging a faulty LOGO program.

3.1. Overview of the Model

We assume that the model has access to the goal drawing, the buggy program, and the drawing that the buggy program produces. In the following analysis, we distinguish between the discrepancy and the bug. The former refers to the difference between the goal drawing and the program’s 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 Figure 1c was a series of three abutting houses, then the discrepancy between the goal drawing and the program output would be described in terms of the unintended space between the houses. The bug that caused the discrepancy would be the +10 in the FD :US +10 command.

In our model, there are four phases to the debugging process. The first phase establishes three subgoals that, when completed, reassert the top goal. The four phases are:

1. Program Evaluation. Run the program. Compare the goal drawing and the program output. If they do not match perfectly, then identify the bug, locate the bug, and correct the bug.
2. Bug Identification. Generate a description of the discrepancy between the goal drawing 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. When only one possibility remains, examine the program output to identify the specific bug.
• In its purest form, the discrepancy description makes no reference to the fact that the faulty drawing is program-generated. That is, the discrepancies are characterized entirely in terms of their pictorial features. Table 1 lists the most common types of discrepancy encountered in LOGO graphics debugging. The quotations presented in the second column are representative comments from children in our study about the type of discrepancy shown in the first column. Note that one possible outcome of this step is knowing that the two pictures are not identical but being unable to describe the mismatch.

• Given the pictorial 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 that propose further discrepancy description to discriminate between multiple possibilities. When only one possibility remains, the model examines the program output to determine the specific bug. The

<table>
<thead>
<tr>
<th>Discrepancy</th>
<th>Example Description</th>
<th>Buggy Component</th>
<th>Specific Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orientation</td>
<td>“this is going over here instead of down”</td>
<td>Angle</td>
<td>LT n or RT n</td>
</tr>
<tr>
<td>Size</td>
<td>“that line—it’s too long”</td>
<td>Distance</td>
<td>FD n or BK n</td>
</tr>
<tr>
<td>Spread</td>
<td>“these are too close together”</td>
<td>Angle or Distance</td>
<td>LT n or RT n or FD n or BK n</td>
</tr>
<tr>
<td>Location</td>
<td>“this is supposed to be in the middle”</td>
<td>Distance</td>
<td>FD n or BK n</td>
</tr>
<tr>
<td>Extent</td>
<td>“lots too many squares”</td>
<td>Iteration or Recursion stop or Recursion interval</td>
<td>REPEAT n IF :x = n THEN STOP NAME :X + - n</td>
</tr>
<tr>
<td>Don’t Know</td>
<td>“this mess”</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

*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, Figure 1a would take about five seconds to draw.*
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). However, the result of this complex processing is a narrower search for the bug (e.g., “right here—it shouldn’t be left 90—it should be right 90 I think”)

3 Bug Location Represent the structure of the program to investigate the probable location of the buggy command in the program listing. Using these cues, examine the program in order to locate the alleged bug.

- Knowledge of the program’s structure may be the result of having written the program or of assuming that programs for certain types of pictures 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 the picture is composed of several identical figures (typically programmed using a REPEAT statement. Knowing that the bug is located within a REPEAT statement narrows the search in the program listing.

- The efficiency of the bug location process depends on the outcome of the bug identification process and the search for cues to the bug’s location in the program listing. 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.

4. Bug Correction. Examine the goal drawing at the site of the bug to determine the appropriate correction. Replace the bug with the correction in the program listing and then reevaluate the program.

3.2. The Production System

In order to specify our model unambiguously and to demonstrate its sufficiency for debugging LOGO graphics programs, we implemented our model in GRAPES, a goal-restricted production system [21]. The forty-four productions in the system represent two kinds of knowledge: 1) general decision rules for generating subgoals and invoking operators, and 2) LOGO-specific heuristics for identifying and locating bugs. In the following sections, we will provide general descriptions of the system’s goals, the heuristics it employs, and the operators it invokes. These descriptions will be followed by traces of the model debugging a LOGO program.

3.2.1. The goals – The production system’s goal structure corresponds to the four phases described in the overview. A goal tree is shown in Figure 2. The test

5 A complete listing of the production system is available upon request. People with access to the ARPA net can address inquiries to KLAHR@psy.cmu.edu
and evaluate goals start the system and evaluate the success of each debugging attempt (i.e., the match between the goal drawing and the program output). The describe and propose goals correspond to the bug identification phase; they control the productions that describe the discrepancy between the goal drawing and the program's buggy output and that propose possible bugs and ways to discriminate among them. Locate and all of its subgoals (represent, specify, find, interpret, and check) are part of the bug location phase; they look for structural cues to the bug's location and then find the bug using the location cues plus whatever cues they have about the bug's identity. Finally, the change and replace goals correspond to the bug correction phase; they identify the appropriate correction and change the program listing accordingly.
3.2.2 The heuristics — The system has two sets of debugging heuristics, one set for identifying the bug and one set for locating the bug. Using both sets of heuristics narrows the search for the bug substantially. Heuristics for identifying the bug correspond to the mappings between observed discrepancies and potential bugs (listed in Table 1). These heuristics are most useful when there is more than one type of bug which can lead to a particular type of discrepancy. In this case, the heuristic includes information for distinguishing between them. For example, if the discrepancy has been initially identified 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 as only spread discrepancies must have been caused by a distance bug.

Heuristics for locating the bug involve knowledge of program structure types. If the program is identified as having either a repeat, a subprogram, or a recursion structure, then the model attempts to determine where the bug might be located within that structure. For example, if the program is identified as having a subprogram structure, the model would attempt to get 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.

3.2.3 The operators — The debugging process uses eleven operators, or sub-skills, 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. Four sources of information are available to the operators at all times: the goal drawing, the program output, the program listing, and knowledge of the programming language. In addition, an operator may use local information contained in a working memory element. Operators may also add information to working memory.

There are two classes of operators: 1) those that correspond to visual inspection of the buggy drawing and/or the plan (shown in the upper portion of Table 2), and 2) those that correspond to maneuvering in the LOGO environment. The latter set of operators (shown at the bottom of Table 2) are automatically executed by the model, but the former set are not. Instead, their operation is simulated by the user of the system. We have not attempted to program the operators that extract information from the drawings and/or the program listing because it would take us far afield of the central purpose of the model, which is to clarify the overall organization of the debugging processes as well as the interactions among the different heuristics.

Table 2 lists, for each operator, its function, the user prompt that is given when the operator is executed, and the allowable responses from the user (which simulate the result of the unmodeled operator). A brief description of each operator follows:
<table>
<thead>
<tr>
<th>Operator</th>
<th>Function (Prompt)</th>
<th>User Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATCH</td>
<td>Did the outcome match the plan? or Did the simulation match the plan?</td>
<td>yes or no</td>
</tr>
<tr>
<td>CONTRAST</td>
<td>What is the discrepancy between the plan and the outcome?</td>
<td>orientation, size, spread,</td>
</tr>
<tr>
<td></td>
<td>Is ______ discrepant? (orientation or size)</td>
<td>location, extent, or ?</td>
</tr>
<tr>
<td>EXAMINE</td>
<td>What is the discrepant ______ on the outcome? (angle or distance)</td>
<td>(______)</td>
</tr>
<tr>
<td></td>
<td>Are the similar figures of equal size?</td>
<td>yes or no</td>
</tr>
<tr>
<td></td>
<td>How many repetitions are there on the outcome?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>What was the size of the last part?</td>
<td>(______)</td>
</tr>
<tr>
<td></td>
<td>What is the interval on the outcome?</td>
<td>(______)</td>
</tr>
<tr>
<td>INTERPRET</td>
<td>Simulate the command ______.</td>
<td></td>
</tr>
<tr>
<td>GENERATE</td>
<td>What should the command ______ have been?</td>
<td>(______)</td>
</tr>
<tr>
<td>RUN</td>
<td>Run the program ______.</td>
<td></td>
</tr>
<tr>
<td>ENTER</td>
<td>Enter the program ______.</td>
<td></td>
</tr>
<tr>
<td>SKIP</td>
<td>Skip to the command ______.</td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>Read the next command.</td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete the command ______.</td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>Insert the command ______.</td>
<td></td>
</tr>
</tbody>
</table>

- The MATCH operator is called to process information from the goal drawing and the program output and then input its judgment about the match between the two. The system questions the user, "Did the outcome match the plan?" or "Did the simulation match the plan?", and expects a yes or no response. A yes response causes the element (match yes) to enter working memory; likewise, a no response yields the element (match no).

- CONTRAST is a discrepancy description operator that processes information from the goal drawing and the program output. The system asks "What is the discrepancy between the plan and outcome?" and requests one of the following answers: orientation, size, spread, location, extent, or ? (as in Table 1) A working memory element of the form (discrepancy response yes) is then added to working memory. CONTRAST can also indicate whether a specified
discrepancy exists or not. If a working memory element such as (description must be about size) exists, then the system's query is "Is size discrepant?" A yes or no response is expected and the resulting working memory element is of the form (discrepancy size response).

- **EXAMINE** is a bug proposal operator that seeks specific information about the bug using LOGO knowledge to guide processing of information from the program outcome. The system asks the user a question such as "What is the discrepant angle on the outcome?" and labels the user's response as the bug (e.g., the bug is [RT 120]).

- **INTERPRET** is a bug location operator that simulates the effect of the current command using LOGO knowledge. The system instructs the user to simulate the command and then asks "Did the simulation match the plan?" This question is a call to the MATCH operator described above.

- **GENERATE** is a bug correction operator that uses LOGO knowledge to determine what command would be necessary to accomplish the desired effect. The system asks the user "What should the command bug have been?" and creates a new working memory element from the user's response (e.g., the correction is [RT 90]).

- Six of the operators are associated with physical action in the LOGO environment. The model automatically carries out these operations on its representation of the LOGO program, while notifying the user that it is doing so. These operators include RUNning the program, ENTERing the editor to view the program listing, SKIPping to a particular location in the program listing, READing a command, DELETing a command, and INSERTing a command.

3.2.4 Examples — In this section, we present two examples that illustrate how the model could approach a specific problem. The programmer's plan was to draw a garden picture (Figure 3a). The program's outcome was, however, discrepant from the plan (Figure 3b) because there is a bug in the program (Figure 3c). The two examples differ in the amount of information about the bug's identity and location.

In the first example, we simulate a situation in which the debugger is a very knowledgeable user. The model is provided with a lot of information about both the discrepancy and the program. The information, provided in response to the operators, is shown in Table 3. Here the user identifies the discrepancy type as spread. The model then proposes either an angle or a distance bug, and requests information about the orientation so it can distinguish between the two possibilities. The user notices that orientation is also discrepant, so the model proposes that the bug is an angle and asks the user to estimate the discrepant angle. The user estimates the angle to be RT 25. The user also tells the model that the program is structured using subprograms, so the model asks the user which subprogram went wrong. The user identifies the buggy subprogram as FLOWER, so the model skips in the program listing to that subprogram. It then requests
information about the structure of the subprogram and is told that the subprogram contains a buggy repeat statement. So, when the find goal is reached, the model knows to look within a REPEAT statement in the FLOWER subprogram for the bug. It also has a clue that the bug is RT 25. In fact, there is no such command in that location, but the model identifies the RT 20 command as sufficiently close to identify it as the bug. The model then requests the user to input the correction, which it makes, and then rerun the program to check the correction. The correction is accurate so debugging is complete. Figure 4 shows the goal tree generated during this debugging cycle.

The second example illustrates the model's behavior when the user knows nothing about the discrepancy between the plan and the outcome and nothing about the structure of the program. The user runs the program and knows that a discrepancy exists. When asked about the type of discrepancy, the user responds with a ? so the model cannot propose the bug's identity. When asked about the program's structure, the user again answers with a ? so the model tries to get information about color cues. Again the user responds negatively so the model must search for the bug by iterating through the program listing.
Table 3. High-information Trace

<table>
<thead>
<tr>
<th>Action</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run the program GARDEN.</td>
<td>→ ok</td>
</tr>
<tr>
<td>Did the outcome match the plan? [yes or no]</td>
<td>→ no</td>
</tr>
<tr>
<td>What is the discrepancy between the plan and outcome? [orientation, size, spread, location, extent, or?]</td>
<td>→ spread</td>
</tr>
<tr>
<td>Is orientation discrepant? [yes or no]</td>
<td>→ yes</td>
</tr>
<tr>
<td>What is the discrepant angle on the outcome?</td>
<td>→ (RT 25)</td>
</tr>
<tr>
<td>How is the GARDEN program structured? [subprogram, recursion, repeat, or?]</td>
<td>→ subprogram</td>
</tr>
<tr>
<td>Which subpart is wrong?</td>
<td>→ (FLOWER)</td>
</tr>
<tr>
<td>How is the FLOWER program structured? [subprogram, recursion, repeat, or?]</td>
<td>→ repeat</td>
</tr>
<tr>
<td>How many times is the discrepancy repeated?</td>
<td>→ 10</td>
</tr>
<tr>
<td>The bug is (RT 20). What should the command (RT 20) have been?</td>
<td>→ (RT 36)</td>
</tr>
<tr>
<td>Run the program GARDEN</td>
<td>→ ok</td>
</tr>
<tr>
<td>Did the outcome match the plan? [yes or no]</td>
<td>→ yes</td>
</tr>
</tbody>
</table>

The first command encountered (PLANTER) is not in the model's list of primitives so it is assumed to be a subprogram call. The model then locates the subprogram listing, asks about its structure, and, learning nothing, iterates through the subprogram commands. For each command, the model asks the user to hand simulate its effect and determine whether that effect matched the plan. Once the commands in the PLANTER subprogram have been exhausted, the model returns to the main program to interpret its next command. That is also a subprogram (FLOWER) so the model must again locate the subprogram listing, ask about its structure, and, learning nothing, iterate through its commands. Upon reaching the REPEAT statement, the model treats it like a subprogram except that its listing follows it directly rather than being stored as a separate program. The first command in the REPEAT statement is then determined by the user to be the bug. The model requests a correction which, once it is input by the user, will be substituted into the program listing in the place of the bug. Finally, the model directs the user to rerun the program to be sure the correction was accurate. Since it was, the debugging episode was complete.

6 Primitives include FD, BK, LT, RT, PU, PD, PC, HT, HOME and two basic shape programs RightCircle and DIAMond
The contrasts between the model's behavior in the high- and low-information situations are striking: the former required only twenty production firings, while the latter required sixty-eight. With respect to the relative complexity of the subgoal trees, shown schematically in Figure 5, we see the same kind of difference: Nineteen sub-goals versus fifty-two. The high-information simulation represents the ideal case in which the bug is completely specified and its location is known. The system's goals and heuristics were used efficiently to narrow the search for
Figure 5. Comparison of schematic goal trees for high-information (a) and low-information (b) traces. Note that (a) is the same goal tree shown in Figure 4, with much of the detail suppressed.
the bug. In the low-information situation little use is made of the describe, propose, represent, and specify goals so none of the heuristics for narrowing the debugger's search are used and debugging proceeds by brute force, one command at a time. Most of the extra production firings and sub-goals result from this serial search. For the purpose of this simulation, we assumed that the interpret operator correctly identified the bug. If this had not been the case, the difference between the two traces would have been even more striking because repeated debugging cycles would have been necessary.

4. ASSESSMENT OF DEBUGGING SKILLS

The formal task analysis of LOGO debugging provides a context in which to assess children's performance. How much of the debugging skill we have specified do children acquire in a "normal" LOGO course? In order to address this question, we designed and implemented a LOGO course that enabled us to assess both the acquisition of debugging skills (the model's goals, heuristics, and debugging operators) and the conventional LOGO subskills (the model's general operators). Debugging skills were evaluated at three times during the LOGO course by asking students to describe the problem with the output of several programs and then asking them to view and debug the programs. We assessed the other operators assumed by the model at three times also. The ability to INTERPRET commands (i.e., to predict the behavior they would cause) was assessed using a paper and pencil task; students were given programs and asked to draw their outcomes. We used a turtle target game to assess students' ability to GENERATE commands to cause specific behavior: students had to give the turtle one turn command and one distance command to make it reach a target. The ability to maneuver within the LOGO programming environment (RUN, ENTER, SKIP, READ, DELETE, and INSERT) was tested using a program editing task: students were given a hard-copy of a program with edits marked on it and they were asked to make the edits.

4.1. Method

Nine children (5 females and 4 males) ranging in age from seven years one month to eight years nine months participated in the study. They were recruited from the communities surrounding Carnegie-Mellon University by advertising a free LOGO course. None of the children knew any computer programming languages, but most of them had limited experience either with computer games or with programmed instruction. Eight of the subjects came in pairs and one came individually to twelve two-hour LOGO classes over a three-week period during the summer. All lessons were taught by a twenty-four-year-old female experimenter who was an experienced LOGO instructor. The instruction used an Apple IIe computer with Terrapin LOGO. Skills in command interpretation,
command generation, maneuvering within the LOGO environment, and debugging were tested three times during the course.

4.2. Lessons

Classes were taught in a small classroom/laboratory in the CMU Children's School. Each session began with a short warm-up time devoted to free programming. Any new material was then introduced and the students did the activities contained in their lesson packets. They were encouraged to explore variations on the current theme in order to understand a concept's flexibility. One snack break was provided during each two-hour session. Tests were generally given during the second hour except during sessions 4, 8, and 12 when students took one test during each hour. Students were using the computer together with their partner at all times except snack time and the time used for written tests. Students set the pace of the classes, and exploration was encouraged. Any extra time at the end of the session (typically no more than 10 to 15 minutes) was devoted to free programming, i.e., the students could create whatever they wanted.

- Session 1 included instruction in basic turtle manipulation commands (Draw, Forward, Back, Circle, Left, Right, PenUp, PenDown, PenColor, Background, and HOME). A turtle step ruler and an angle wheel were available for use throughout the course unless otherwise specified. In addition, students were taught how to choose left from right turns by putting their hands together, pointing them in the same direction the turtle was pointing, and then deciding which hand (left or right) would have to move in order to point in the turtle's desired direction.

- Session 2 reviewed the basic turtle manipulation commands in the context of a turtle target game and turtle mazes. Students were taught to estimate the number of degrees in a turn by first identifying an imaginary "90° line" (Figure 6), then determining whether the turn should be greater or less than 90°, and finally estimating from there. An imaginary "180° line" was included to emphasize the upper boundary for estimates. During Session 2, students took the first command generation test.

- Session 3 introduced the REPEAT command and the commands for editing programs. Students then took the first maneuvering test.

- No new material was introduced during Session 4. Students took both the first command interpretation test and the first debugging test.

- Session 5 introduced the idea of calling programs within programs and then allowed free programming of the students' own designs.

- Session 6 focused on the possibility of using variables to make shapes of different sizes. Students also took the second command generation test.

- Session 7 combined the ideas from Sessions 5 and 6 to create variable programs within programs. Students also took the second maneuvering test.
No new material was introduced during Session 8. Students took the second command interpretation test and the second debugging test.

Session 9 introduced tail recursion. The recursive structure was described as having three parts: a DO line to tell the turtle what to do with the variable, a STOP line to help the turtle decide when it's finished, and a GO line to insert a copy of the program with a new variable. For example, in Figure 1c, the second line of the program would be characterized as the DO line, since it does the basic action to be repeated; the third line would be called the STOP line, since it contains the termination test, and the fourth line would be called the GO line, because it is the recursive call with a decremented variable. Students practiced altering various parts of this structure to produce different effects and were encouraged to program their own designs.

Session 10 expanded the idea of recursion to include imbedded recursion by reviewing the recursive structure and adding a second DO line following the GO line. Students experimented with alterations and their own programs. They then took the final command generation test.

Session 11 focused on LOGO music programming. The LOGO scale was introduced along with the basic program statement, PLAY (notes) (durations). Students were encouraged to write both familiar melodies and original compositions. Students also took the final maneuvering test during Session 11.

No new material was introduced during Session 12. Students took the final command interpretation and the final debugging test.
4.3. Tests

Students were tested either in pairs or individually, depending on the test. For tests of maneuvering and command generating, the paired students alternated problems so that each one did half of the test and watched the other student do the other half. The one unpaired student did whole tests. Tests of interpreting commands and debugging were administered during the same sessions. For paired students, one did the debugging test while the other did command interpretation, and then they switched. For these tests, all the students did whole tests. The turtle step ruler and the angle wheel were available only for the debugging test.

4.3.1 Debugging – Debugging skills were tested using actual computer debugging exercises; each test consisted of six problems. Students were given a drawing (in color on graph paper) of each program’s intended outcome and were asked to run the program to see what it actually produced. Students were told that there was only one wrong command in the program. They were required to locate the discrepancy point on the screen and asked, “What’s the problem?” Their responses revealed whether they thought it important to search for clues to the bug’s identity and location prior to viewing the program (i.e., whether they would follow the model’s goal structure) as well as whether they had sufficient knowledge of the heuristics necessary to discover accurate clues. Students were then asked to locate the error in the program and to correct it in any way they could. Their ability to actually debug the program revealed whether they could use any clues they discovered. The turtle step ruler, angle wheel, correct drawing, graph paper, and pencil were available as external aids. In addition, the experimenter helped the students keep their places in the program if requested to do so. These helps were made available so that problems with basic operations would not interfere with debugging ability. The experimenter recorded the students’ description of the bug, their search method, and their debugging attempts as well as any additional comments they made. Since programs on later tests could be more sophisticated, no attempt was made to control for the number or type of commands.

4.3.2 Interpreting commands – Command interpretation skills (the INTERPRET operator in the model) are defined as the ability to predict what a given program will do (i.e., to behave like the LOGO interpreter). These skills were tested using a written test, because the computer cannot record what students believe the effect of a command will be. Students were given programs printed on graph paper. They were instructed that each square of the graph paper was ten turtle steps on a side and fifteen steps on the diagonal. The students’ task was to draw what the program would make the turtle draw. The instructor demonstrated the procedure for one sample program before the first test. Each test consisted of six programs. As with debugging tests, later tests included more sophisticated programs, so that tests were not controlled in terms of the number
and type of commands. The first test included programs with no hierarchical structure and programs with REPEAT statements. The second test included programs with variables, as well as programs similar to those on the first test. The third test added programs with subprograms, tail recursion, and imbedded recursion.

4.3.3 Generating commands -- Command generation skills (the GENERATE operator in the model) are defined as the ability to generate a command to achieve a particular local action by the turtle. The most basic of these skills are to make the turtle turn and move specific angular and linear distances. The command generation test consisted of a series of twenty-four trials in which the computer presented the student with a display containing the turtle and a target. The student's task was to type in one turn command and one move command to get the turtle from its initial position to the target. This task includes angle and distance estimation as well as selection of RT versus LT. Four independent variables were used: the initial orientation of the turtle at the center of the screen (0° to 300° in 60° increments), the angular disparity between the turtle's orientation and the target (15° to 180° in 15° increments), the direction of the angular disparity (RT or LT), and the distance between the turtle and the target (20 to 90 steps in 10-step increments). Because judgments of RT versus LT, angle, and distance were assumed to be independent, we did not test all possible combinations of these variables. A LOGO program presented the stimuli and recorded responses. As soon as the students entered their "program," the screen was cleared, and the next trial was presented. There was no feedback.

4.3.4 Maneuvering in the LOGO environment -- Before children can attempt debugging, they must be able to maneuver within the LOGO environment. This maneuvering includes moving back and forth between the interactive and editing modes, moving within the edit mode to the desired location, and making changes to the program text. This non-trivial task requires children to learn a minimum of ten new commands, seven of which require simultaneous pressing of two keys ("control" plus a letter).

Maneuvering skills (the RUN, ENTER, DELETE, INSERT, etc. operators in the model) were tested using a program editing task. Students were given a computer printout of a program with five changes marked on it in red ink. The test required students to enter the editor, make the changes, exit the editor, and then run the corrected program to see what "surprise" drawing it would make. Each editing test consisted of four tasks each of which had five changes. Three of the changes were insertions, one at the beginning of a line, one in the middle of a line, and one at the end of a line. The other two changes were corrections, one of a whole unit (e.g., 90 → 85) and one of a partial unit (e.g., 90 → 80). The experimenter recorded each keystroke the subject made.

This procedure is very similar to those used in investigations of how adults use text editors [22, 23]. It is remarkable that this relatively complex task is just one of many components to be acquired by our young students in the overall LOGO context.
5. DESCRIPTION OF DEVELOPING SKILLS IN TERMS OF THE MODEL

The goal of this investigation was to assess the debugging goals, heuristics, and operators as well as the component LOGO operators that children actually acquire during a typical LOGO course. Figure 2 depicts the general results in terms of the proposed debugging model; the black boxes indicate the parts of the model with which the children had difficulty. In the test phase, children were able to MATCH the goal drawing and the program output to determine whether or not they were the same. However, they made little use of the describe, propose, represent, and specify goals. The CONTRASTs and EXAMINations they did make were vague and rarely discriminated between potential bugs. Also, they rarely attempted to use location heuristics. Even when identification and location clues had been mentioned, they were seldom used to narrow the search for the bug, so children had to locate and change the bug using step-by-step examination of the program. Despite good skills in maneuvering within the LOGO environment (RUN, ENTER, SKIP, etc.), the serial search was tedious because of the children’s poorly developed skills at INTERPREting commands. However, children were generally good at GENERAting the command to correct the bug once it had been located. The following sections discuss the results of the separate tests which support these generalizations.

5.1. Search for Cues to Identify the Bug

The first part of the debugging test evaluated the children’s ability to compare the goal drawing and the program output to extract cues which could guide search for the bug within the program. Children’s answers to the question “What’s the problem?” reveal their ability to MATCH the two pictures, their ability to describe the discrepancy (CONTRAST) between them, and their knowledge of what bugs cause discrepancies (identity heuristics and the EXAMINE operator). At this point children might also be able to infer something about the program structure (if it had repeated figures for instance).

5.1.1. Matching – Children were given a drawing of the goal for the program and they watched each program produce its output; however, most of the programs ran so quickly that the children were basically comparing two static figures. They had no difficulty distinguishing between those figures that were identical and those that were not. The dynamic working of the MATCH operator was most evident in instances where the goal drawing contained circles, which are drawn very slowly. In these cases, the children could discover the mismatch even before the program had finished running (e.g., “I see the problem already”).

---

8 Because the image of the turtle icon must be regenerated at each orientation around the circle.
or they would identify a partial match before the program had reached the buggy command (e.g., "I don't see anything wrong yet," for a program that started correctly but did not repeat enough).

5.1.2 Discrepancy description – Children's descriptions of the discrepancy between the goal drawing and the program output (before entering the editor to view the program) were of three types: Type 0 comments contained information only about the point of the discrepancy, Type 1 comments contained information about the properties of the discrepancy also, and Type 2 comments also contained information about the identity of the bug.

Table 4 shows the distribution of the fifty-four scores (9 subjects × 6 problems) for each test. There is a sharp reduction in the proportion of discrepancy-point-only comments: from 27 percent on the first test to none on the third. However, only about half of the discrepancy descriptions given were good enough to isolate the problem as one of the five types listed in Table 1 (15 of 28 on Test 1, 25 of 43 on Test 2, and 18 of 37 on Test 3). The other half were either confusions between two types of discrepancy description or failures to specify the type of discrepancy. For example, spread and extent discrepancies were frequently confused; students would say "it's too squeezed together" when there were actually missing parts or "I have to add to it" when the parts were too close together. Failures to specify the type of discrepancy frequently occurred with orientation, location, and size problems; a student could say "it's because this needs to be down here" referring to any of these three types of discrepancies. Some of the subjects also had difficulty distinguishing the discrepancy point (the locus of the bug) from the areas of discrepancy (the misplaced parts of the plan). While the CONTRAST operator appears to be active in the children's debugging process, it does not reliably distinguish between the various types of discrepancy.

5.1.3 Bug proposal – The identity heuristics and the EXAMINE operator jointly specify the bug. The easiest bugs to identify were those in the most structured programs, particularly recursive programs for which students had learned to describe programs in terms of a 3-fold structure (DO, STOP, and GO). Once the drawing was identified as recursive (having parts increasing or

<table>
<thead>
<tr>
<th></th>
<th>Discrepancy Point Only</th>
<th>Discrepancy Description</th>
<th>Identification of the Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>14 (.27)</td>
<td>28 (.55)</td>
<td>9 (.18)</td>
</tr>
<tr>
<td>Test 2</td>
<td>6 (.11)</td>
<td>43 (.81)</td>
<td>4 (.08)</td>
</tr>
<tr>
<td>Test 3</td>
<td>0 (.0)</td>
<td>37 (.71)</td>
<td>15 (.29)</td>
</tr>
<tr>
<td>TOTAL</td>
<td>20 (.13)</td>
<td>108 (.69)</td>
<td>28 (.18)</td>
</tr>
</tbody>
</table>
decreasing in size), the bug could be attributed to one of the three parts of the
program and search for the bug could focus on that part.

The most difficult bugs to identify involved repeated bugs where the outcome
was "too squashed" or "not spread apart enough"; such a description could refer
to bugs in several places: not enough rotation before the next iteration, or not
enough separation between parts before the next iteration. The children assumed
that all spread discrepancies were the result of an angle bug. So they entered the
program and changed the first angle. Even when they reran the program and saw
that they had failed to correct the error, many of them did not attempt to change
the distance until they had tried several other corrections of the angle. We inter-
pret this as a difficulty with the bug identity heuristics.

Children also have difficulty with the EXAMINE operator. Most of the bug
descriptions (23 of 28) did not include specific arguments and half (8 of 16) of
the correction descriptions indicated only whether the change should increase or
decrease the argument but did not specify the change.

5.2. Search for the Bug with Cues

Once the comparison of the goal drawing and the program’s outcome is com-
plete, the programmer must enter the program editor to view the buggy program.
The information gained by discrepancy description and bug proposal will only
be useful if it can discriminate between the possible discrepancy-causing bugs.

Since children’s search for cues to the bug’s identity were poor and they
rarely knew enough about the program structure to use cues to locate the bug,
their actual use of the cues was minimal. Only three cues were used on the first
test, one on the second, and nineteen on the third. The increase in cue use on
the third test is mostly due to the two recursive programs whose structure is well-
specified. One of the programs did not stop so three of the students checked the
STOP line immediately upon entering the program editor; the other program was
incrementing the variable by an insufficient amount so five of the students
looked directly for the GO line, which contains the increment statement.

Cues which do not rely on such structure within the program included color
changes (i.e., knowing that the bug occurred either before or after the color
changed) and looking for a specific command. The latter of these can lead to er-
roneous fixes. On one problem, four students described the bug as ‘‘a wrong
turn, instead of left it should be right.’’ Three of these four students entered the
editor, moved down to the first LT in the program, changed it to RT, exited,
ran the program, and were surprised at the added damage they had done. Upon
reentering the program, they saw the second LT and realized they had changed
the wrong one. The fourth child paused at the first LT but changed only the
second; and when queried about how she knew which one to change, she replied,
‘‘it would have to be the second to last one (turn)’’ She was using two cues: the
specific command and counting the turns.
Many potential cues—such as counting line segments or exploiting the figure’s symmetry—were never used by these students. The cues already mentioned were used only in certain situations and not in others where they would have been appropriate. Most of the students used a few cues, but three of the nine students never used any.

5.3. Search for the Bug without Cues

Lacking accurate cues to the bug and its location or failing to use correctly specified cues leaves a programmer to search the program serially. This search requires interpreting each command in turn to discover whether it is the bug, determining the appropriate correction once the bug has been found, and maneuvering within and between modes to make the necessary change and check the program. These supporting components of debugging skill were assessed in isolation from the debugging context.

5.3.1. Interpreting commands — Command interpretation tests were designed to evaluate the INTERPRET operator independently from the debugging situation. They were the most difficult tests of all: for only 19 percent of the 162 problems (9 children doing three tests with 6 problems each) did the children make correct drawings. We did not calculate a total numerical score because the programs for each test have different numbers of lines and different amounts of depth (or hierarchy). Instead, we studied each error in the students’ drawings in order to uncover students’ general difficulties. Each error in an incorrect drawing was classified into one of three types: measurement errors, placekeeping errors, and misconceptions.

Measurement errors involved angles and distances; they included turning left instead of right, turning 45° instead of 30°, and moving sixty steps instead of forty-five et cetera. Angle errors occurred most often when children had to turn angles other than 90° from orientations other than 0°. The command generation tests directly examined errors such as these (see Section 5.3.2)

Placekeeping errors included both redoing and skipping commands as well as doing the wrong number of iterations in a REPEAT statement or forgetting the final turn in a REPEAT statement. Redoing and skipping usually occurred in programs having duplicate lines or commands. Losing track of the current variable value in a recursive procedure was another common placekeeping error. Few students used external placekeeping strategies such as crossing off lines of completed code, or keeping a tally of repeats completed.

Misconceptions varied widely from student to student. The most frequent of these were 1) turning to the absolute angle (as if starting from a heading of 0°) rather than turning from the actual current heading, 2) drawing circles beginning to the turtle’s left rather than in the direction it was heading, and 3) assuming that when the turtle moves backwards it also turns around. Many of the misconceptions dropped out naturally but others remained since the students had no negative feedback.
The following example from a child's first command interpretation test shows two measurement errors, a placekeeping error, and a misconception (Figure 7). The first measurement error occurs at line 5; the student drew FD 20 instead of FD 30. In line 6, the student's turn reveals a misconception; she turns to absolute 90° instead of turning RT 90° from her current heading of 225°. She completes the second FD 30 correctly but then makes a placekeeping error: she apparently looked back to line 5 when in fact she had completed line 7. As a result, she redoes lines 6 and 7. In the process, she makes a second measurement error by turning left instead of right in line 6. The remaining lines are executed properly.

Many students attempt to interpret REPEAT statements and recursive functions by guessing the patterns they will form. Guessing is obvious when the student performs one iteration of the REPEAT statement and then repeats the resulting shape without taking the current turtle heading into account. For example, Figure 8 shows the drawings of a student who used pattern guessing when told to draw D2 using 20 as the value for :HI. The student followed one iteration of the REPEAT statement and then drew three more identical shapes beside it. The instructor asked the student to try the drawing again paying special attention to the heading of the turtle (i.e., without guessing) The student returned
almost immediately with the second drawing (another guess); the student's third attempt was successful.

Imbedded recursion causes great difficulty because subjects get confused about the execution of the second DO line. Some students stop with the STOP statement rather than popping up to the next higher copy of the program so the second DO line is never executed. Other students execute the first DO line recursively (or guess the pattern correctly) but execute the second DO line only once. Figure 9 shows the drawing of a student who was asked to draw F3 using 50 as the value for :ME. Despite some difficulty with keeping track of the variable values, the student manages to work his way through the recursive calls until he reaches the STOP statement. He then comes to the unexecuted DO line and substitutes the initial variable value; he views this line as separate from the recursive part of the program so he executes it only once.

Errors made by the INTERPRET operator would have severely handicapped the children's debugging ability. Such errors could cause misidentification either of correct commands as buggy or of buggy commands as correct. During the debugging tasks, however, the children were able to use rulers and angle wheels.
TO F3 :ME
HT
FD 10 LT 90 FD :ME BK :ME RT 90
IF :ME = 0 THEN STOP
F3 :ME - 5
BK 10 RT 90 FD :ME BK :ME LT 90
END

Figure 9. Example of command interpretation error with imbedded recursion

to help them, and the experimenter corrected any mistakes they made in command interpretation.

The children’s reliance on the experimenter for assistance in command interpretation was scored on a 0-2 scale for each debugging task (0 indicating step-by-step help in reading through the program, 1 indicating occasional help, and 2 indicating no help). For each test, a total of two points for each of nine students for each of six problems was possible (a total of 108). Total scores per test were then compared across tests to identify improvement. From test 1 to test 3, there was about a 50 percent increase in the proportion of the maximum possible score (43, 49, 65), although given the small n, this improvement was not significant. By the third test, children were achieving only 65 percent of the maximum “independence score.” Their dependency was primarily manifested in continued requests for help with the angle wheel and with keeping their place in the program as they read through it.
Given the poor performance on command interpretation in a non-debugging context, it is not surprising that the same kinds of errors—measurement errors, placekeeping errors and misconceptions—occur within the debugging context. However, the debugging context (Section 4.3.1) evokes two additional kinds of errors. First, children's prediction of turtle behavior was often heavily influenced by their expectations (the goal for that command). For example, one child was hand simulating a program's actions; when she came to the erroneous FD 80, she only drew FD 40 (the correct distance). When questioned about how far forward she had gone, she replied, "that's wrong then; it should be something like 40." The second additional type of interpretation error observed during the debugging test was the tendency to skip commands which called other programs (i.e., were not primitives). The children seemed to assume that the turtle would ignore any commands which they did not understand.

5.3.2 Generating commands — Another important component of debugging skill is the ability to generate commands appropriate to the desired outcome. This includes generation of move and turn commands plus generation of organized structures of these commands (such as repeats and recursion). In this study, we could assess command generation skills in two contexts: the target game described in Section 4.3.3 and the actual debugging tests.

Our command generation tests were designed to test only the generation of move and turn commands. This included separate measures of the ability to judge left from right turns, the ability to estimate angles, and the ability to estimate distances. Each answer was scored in three parts: the direction, the angle, and the distance. The proportion of correct direction judgments for each initial orientation was compared across tests. For both the angle and the distance estimates, the mean estimate and the mean error for each actual value were compared across tests.

Children's ability to correctly choose left from right turns improved markedly during the LOGO course. On the first test, children made almost perfect judgments when the turtle was oriented at 0° (i.e., where \textsc{forward} would move the turtle toward the top of the screen), an orientation identical to their own. Despite their knowledge of a concrete strategy for choosing left from right, as the turtle's orientation diverged from 0°, either to the right or to the left, children's judgments were less often correct. Their judgments were worst when the turtle was oriented at 180° (e.g., turtle pointed down), an orientation directly opposite their own. Left-right judgments at 180° improved from 33 percent correct to 85 percent correct during the course. By the third test, judgments at all six turtle orientations had reached a high level (see Figure 10). They are not all at the 100 percent level because even when children use a correct method for choosing which turn to make, there is some probability that they will misidentify the direction (i.e., call their left their right or vice versa).

Fairly accurate estimation of both distance and angles started with the first test and continued throughout the course. Angle estimates (Figure 11) were
most accurate for 90° and 180° turns, and they were more accurate for angles less than 90° than for those greater than 90°. Distance estimates (Figure 12) increased linearly with distance, averaging about 25 percent too low.

In the actual debugging context, children's ability to GENERATE commands was evidenced by the number of corrections they made once the bug had been identified. Once the children found the bug, they were fairly good at making the necessary changes. For the fifty-four problems on each test, the students made a total of ninety-nine, eighty-one, and seventy changes on the three tests. By the third debugging test, children rarely made erroneous corrections. However, this slight improvement in making corrections was not significant.

Children rarely used cues to guide their corrections; 3, 6, and 6 cues were used on the three tests (N = 54 for each test). The cues used were of two types: symmetry cues and graph paper cues. For example, one student used a symmetry cue to solve a problem where the second half of the boat deck should be forty turtle steps not eighty because the first half was forty turtle steps. Another student counted the number of squares on the graph paper plan to determine the same correction.
Figure 11. Angle estimation on the three command generation tests.
 a) Mean angle estimate vs. actual angle, b) Mean angle error vs. actual angle.
Figure 12. Distance estimation on the three command generation tests.
   a) Mean distance estimate vs. actual distance,
   b) Mean distance error vs. actual distance.
5.3.3 Maneuvering in the LOGO Environment – The maneuverability tests were designed to test debugging operators such as RUN, SKIP, DELETE, and INSERT which enable the programmer to interact with the LOGO system. The number of keystrokes used to complete each task was divided by the minimum number of keystrokes for that task to get a measure of efficiency which could be compared across tests.

On the first maneuvering test, which was given the day the children learned to use the editor, children made an average of 75 percent more keystrokes than necessary to complete the editing tasks. By the second test, they made only 57 percent more keystrokes than necessary, and by the third test only 45 percent. Children became significantly more efficient at maneuvering during the LOGO course \((F(2,57) = 6.11; p < 0.01)\).

Children's extra moves can be characterized in two ways (see Table 5). First, they can be described as either errors or redundancies. Errors cause some damage and require the child to back up or make a correction. For example, forgetting to put a space between a command and its argument would cause the program to stop. If the child made this error, he had to go back to that position in the program to add a space. Also, holding down an arrow key or a control character so that it repeats can cause disastrous effects which require correction or back-up. On the other hand, redundancies cause no damage, but they are not efficient. Leaving an extra space or using an arrow key repeatedly rather than using (control) E wastes keystrokes. Children's improving efficiency is due to a decrease in both errors and redundancies. Children made 48 percent, 39 percent, and 27 percent more keystrokes than necessary because of errors on the three tests, and they made 27 percent, 18 percent, and 18 percent because of redundancies. They are not merely learning to avoid damaging effects but also to use the quickest methods.

<table>
<thead>
<tr>
<th>Type of extra keystroke</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>75%</td>
<td>57%</td>
<td>45%</td>
</tr>
<tr>
<td>Errors</td>
<td>48%</td>
<td>39%</td>
<td>27%</td>
</tr>
<tr>
<td>Redundancies</td>
<td>27%</td>
<td>18%</td>
<td>18%</td>
</tr>
<tr>
<td>Between modes</td>
<td>4%</td>
<td>10%</td>
<td>6%</td>
</tr>
<tr>
<td>Insert at beginning</td>
<td>14%</td>
<td>10%</td>
<td>9%</td>
</tr>
<tr>
<td>Insert in middle</td>
<td>12%</td>
<td>8%</td>
<td>6%</td>
</tr>
<tr>
<td>Insert at end</td>
<td>12%</td>
<td>16%</td>
<td>6%</td>
</tr>
<tr>
<td>Change part</td>
<td>13%</td>
<td>6%</td>
<td>9%</td>
</tr>
<tr>
<td>Change all</td>
<td>21%</td>
<td>6%</td>
<td>9%</td>
</tr>
</tbody>
</table>
The extra moves can also be characterized in terms of where they occur in the editing task. Children initially made fewer extra keystrokes when they moved between interactive and edit modes (4%, 10%, and 6% on the three tests) than when they actually made the edits. There was no difference between the five types of edits. On the third test, the extra keystrokes were evenly distributed across the six parts of the editing task.

5.4. Acquisition of Debugging Skills: Summary

After twenty-four hours of LOGO experience, which is more than students receive during a typical year-long LOGO course, our subjects had not developed effective debugging strategies. In fact, they rarely debugged programs when they were not required to do so. In terms of the model, children lacked the goal structure and heuristics for discrepancy-bug mappings and locating the bug in the program listing. When debugging was required, the children used a simple debugging method (serial search through the program listing) even when they had identified appropriate cues and had begun to develop its basic operators and a few heuristics. These acquisitions were made primarily without specific instruction, though the test sessions could be considered instruction to some degree. The children did acquire simple programming skills useful for debugging, including maneuvering within the LOGO environment and generating turn and move commands.

6. TOWARD A PRINCIPLED BASIS FOR TEACHING DEBUGGING SKILLS

As we noted at the outset of this article, it is impossible to subject the broad claims about programming’s impact on cognitive development to a rigorous evaluation without taking several preliminary steps. First, we need to specify what is learned about the programming task itself. Our model of debugging is a first step in that direction for one of the important sub-domains of programming. Next, we need to use the model as the basis for evaluating just how much of the skill is acquired in different instructional contexts. In this article, we reported one such evaluation of a “typical” LOGO curriculum, and discovered that few of the requisite skills were acquired. We believe that the next step is to use the model as the basis for a debugging curriculum: a LOGO course that includes intensive instruction in all of the component skills embodied by the model. In this concluding section, we will address these two points: why children don’t debug, and how we might teach them to.

6.1. Why Don’t Children Debug?

Our debugging model assumes that the child has already made the decision to debug a program, rather than to simply abandon it and start over. However, our
informal observation is that children prefer to restart rather than to debug, and this is consistent with Papert's impressions of debugging avoidance:

Children often develop a "resistance" to debugging. . . . The child plans to . . . draw a certain figure, such as a house or a stick man. A program is quickly written and tried. It doesn't work. Instead of being debugged, it is erased Sometimes the whole project is abandoned. Sometimes the child tries again and again and again with admirable persistence but always starting from scratch in an apparent attempt to do the thing "correctly" in one shot. The child might fail or succeed in making the computer draw the picture. But the child has not yet succeeded in acquiring the strategy of debugging [12, pp. 113-114].

Papert attributes this resistance to deeply-ingrained, culturally-transmitted attitudes toward the self.

It is easy to empathize. The ethic of the school has rubbed off too well. What we see as a good program with a small bug, the child sees as "wrong," "bad," "a mistake." School teaches that errors are bad; the last thing one wants to do is pore over them, dwell on them, or think about them. The child is glad to take advantage of the computer's ability to erase it all without any trace for anyone to see. The debugging philosophy suggests an opposite attitude. Errors benefit us because they lead us to study what happened, to understand what went wrong, and, through understanding, to fix it. Experience with computer programming leads children more effectively than any other activity to "believe in" debugging [12, p. 114].

We have little disagreement with Papert's "debugging philosophy," but we believe that he is mistaken in attributing the problem to inappropriate attitudes or self-concepts. Children don't debug because they don't know how to. There are three reasons why they fail to acquire this skill: 1) debugging is a complex skill; 2) it requires extra capacity; and 3) it is rarely taught directly. Below, we elaborate each of these points.

6.1.1 Debugging is a complex skill — Debugging's complexity is obvious from the large number of goals, heuristics, and operators necessary to describe our model. The bug identification and bug location productions represent a minimal set of heuristics for finding bugs; without these search shortcuts, the model's debugging is laborious. Well-developed operators are essential for accurately comparing the actual output with the goal drawing and for interacting with the LOGO system; without such operators, the model makes frequent errors and requires many cycles to correctly debug a program.

Studies with adults have shown that skilled programmers attempt to use the goal structure of a program as an aid to bug isolation. For example, Jeffries' study of expert and novice Pascal programmers showed that growing expertise
involved developing a hierarchical representation of programs [24]. Jeffries also found that experts had accumulated a set of familiar patterns that they used to relate flaws in the output to potential bugs. Our model represents such knowledge in the propose productions. Spohrer, Soloway and Pope make the interesting suggestion that failure to maintain the appropriate goal hierarchy ("goal drop-out" and "merged goals") is a common source of bugs [25]. Atwood and Ramsey found that debugging was difficult for adult novice FORTRAN programmers because they lacked useful heuristics for seeking cues from faulty output which could narrow their search for the bug [26].

Although goal structures play an important role in adult programming, a hierarchical conceptualization of the solution to a programming problem is very rare at the low level of programming skill typically reached by children. For example, in Pea's study (described in Section 1.1), children were able to debug syntax errors effectively but were not able to locate semantic bugs such as misordered commands [9]. Part of their difficulty was a result of their tendency to approach programs as long chains of direct commands rather than as hierarchical structures. Nevertheless, because we wanted our model to represent an experienced debugger, we did include the capacity to effectively use knowledge about the goal structure of the buggy program. If the responses to the represent operators indicate that the program has a specific structure and that the bug appears to be local to a particular component of that structure, then the model immediately constrains its search to that location, as in the high-information trace shown earlier. (It is common LOGO practice to name subprocedures according to the function that they serve, so knowing that a bug is in subprocedure X is equivalent to knowing goal X is not being correctly achieved.)

6.2 Debugging skills require extra capacity — Development and use of debugging skills requires memory capacity sufficient to keep track of available cues. Typically, the context in which students experience a need for debugging is one in which their attention is directed toward the problem at hand: getting the program right. This focus leaves them little capacity to learn about the debugging process itself.

Pea and Kurland give a multi-stage characterization of the acquisition of programming skills [1]. The beginner is simply a "code generator" who focuses on individual commands rather than developing a structured program. Next, the student begins to think in terms of higher level units, becoming a "program generator" who can create and debug complex programs. Finally, the student becomes sufficiently familiar with the language that he can distance himself from the coding processes to consider the general problem-solving aspects of programming such as elegance, efficiency, and optimization; he has become a "software developer." Our evidence suggests that even after twenty to thirty hours of instruction, most children are barely out of the first stage; they are still struggling to acquire the basic LOGO operators. Only after these operators are well-learned do programmers have a sufficient capacity to excel in debugging.
6.2. How Could Children Learn to Debug?

We are currently attempting to teach children effective debugging skills. Our methodology is designed to address the above-mentioned reasons why children typically fail to debug well. Debugging is an explicit part of our LOGO curriculum. It is introduced in the context of familiar programs so that attention can be focused on debugging as opposed to basic programming. Also, students have access to several types of external aids for basic LOGO operations; all the commands are printed on bulletin boards around the classroom; angle wheels, coordinate charts, etc. are provided; heuristics for programming (such as symmetry and angle addition) are stressed. All of these aids reduce the load caused by basic programming. Our instructional objectives for teaching debugging are derived directly from our model. Students are taught a series of steps for debugging which correspond exactly to the model's goal structure. Also, students are taught to record the discrepancy-bug mappings they encounter so that they are available in future debugging situations. Heuristics for locating bugs are also stressed. Throughout the course, debugging skills are being assessed during the students' own programming as well as while the students debug programs we have written. In addition, transfer of debugging skills to a second LOGO programming domain and to debugging in real life situations is being assessed.

The goal for our research was to develop a methodology for studying complex skill acquisition and transfer and to use it to study children's debugging skills. The methodology involves first analyzing the desired skills using a formal model, then assessing the skills as they develop naturally. Next, the model can be used to design instruction to facilitate skill development. Finally, the acquisition of the desired skills can be assessed according to the model and transfer tests can be chosen which require similar skills. The current paper covers only the first half of this agenda. Discussion of the second half is forthcoming [27].

ACKNOWLEDGMENTS

We thank our colleagues John Anderson and Bob Siegler for comments on an earlier draft.
REFERENCES


11. S. Papert, Teaching Children Thinking, Programmed Learning and Educational Technology, 9, pp. 245-255, 1972.


Direct reprint requests to:

Dr. David Klahr
Psychology Department
Carnegie-Mellon University
Pittsburgh, PA 15213