HUMAN-COMPUTER INTERACTION, 1987-1988, Volume 3, pp. 351-399

Debugging: An Analysis of Bug-Location Strategies

Irvin R. Katz and **John R. Anderson**
Carnegie Mellon University

ABSTRACT

This article presents a series of four experiments investigating students' debugging of LISP programs. The experiments involve a population of students who know LISP reasonably well in that their errors are best classified as *slips* (Brown & Van Lehn, 1980). That is, students are unlikely to repeat the same errors either within their program or across programs (Experiment 1). The students' understanding of LISP is also reflected in their debugging behavior: They can usually fix a bug once they locate it. Students' difficulties are in locating the erroneous line of code. We observe that students use a variety of bug-location strategies during debugging (Experiment 2) and that the choice of strategy differs depending on whether students are debugging their own programs or other students' programs (Experiment 3). In addition, we observe that although the different bug-location strategies affect which lines of a program are searched, once students decide on a line, their ability to judge whether or not the line is correct and their ability to correct an error are not substantially affected by the strategy used to locate the line (Experiment 4). Finally, we argue that our results have implications not only for debugging in other computer languages, but for the general processes involved in troubleshooting as well.

Authors' present address: Irvin R. Katz and John R. Anderson, Department of Psychology, Carnegie Mellon University, Pittsburgh, PA 15213.

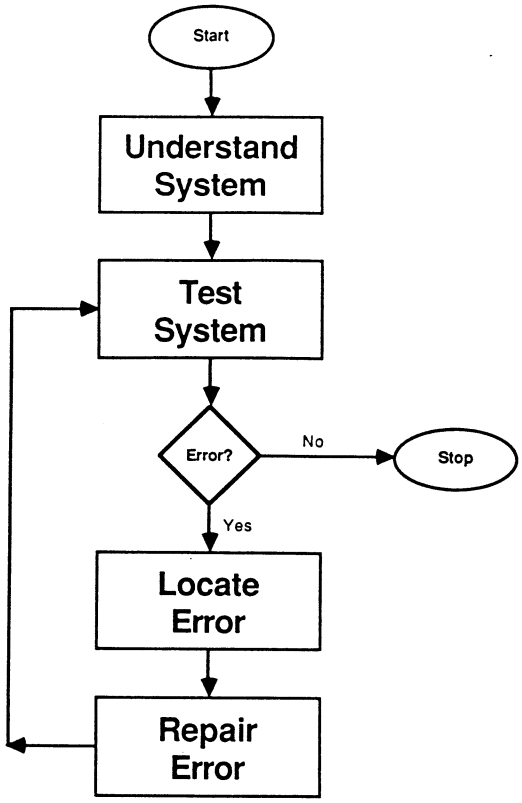
CONTENTS

1. INTRODUCTION

This article reports the work we have done thus far in trying to understand how students debug LISP programs. However, we do not regard the significance of this research as being limited to debugging in LISP or to debugging in general. We believe that the phenomena we are studying are just particularly clear instances of a very general problem-solving process-namely, troubleshooting. Troubleshooting permeates our everyday lives: The car won't start; the television's reception is bad; our computer programs don't work correctly; and so on. Figure 1 provides a high-level model of what is involved in troubleshooting.

Generally, a person must first come to understand or have a representation of the device being repaired. The person then usually tests the device in some way, if only to observe that incorrect behavior is produced. The person must then locate the error in some way and, once the error is found, repair it. After allegedly repairing the device, the person should test the device to be assured that correct behavior is now produced. If the device still acts incorrectly, further location and repair may be needed. Thus, according to this model, troubleshooting contains four steps (understanding, testing, location, and

Figure 1. **Simplified model of general troubleshooting.**



repair), repeating the last three steps if necessary. A similar characterization of troubleshooting has been suggested by Morris and Rouse (1985).

It is clear that this model is oversimplification of troubleshooting. Among other things, the model neither addresses the issue of how the processes interact nor how the processes are instantiated in a particular domain (i.e., for a particular device) or in a particular situation. This article fills in these details for the domain of computer program debugging.

Debugging may be thought of as just a specific instance of troubleshooting, but instead of fixing the problems in a device, the errors in a computer program are sought and corrected. Other researchers (e.g., Gugerty & Olson, 1986; Kessler & Anderson, 1986; Klahr & Carver, 1988) have shown that debugging may be characterized as consisting of subprocesses that correspond to the model of general troubleshooting just presented. Specifically, when debugging, one must: (a) test the program and detect that it isn't behaving incorrectly, (b) locate the erroneous line or lines of code, and (c) rewrite the buggy code. Also, if a person had not written the program originally (or had written it a long time ago), that person would probably need to first comprehend (or understand) the program.

Debugging is a good domain in which to study troubleshooting for two reasons. First, there is a great deal of flexibility in both what different subjects might do and how an experimenter can manipulate the task. Thus, troubleshooting may be observed in a variety of contexts, and subjects are free to troubleshoot in the same way they would in a normal programming environment. Second, most debugging requires subjects to go through each step of the troubleshooting process distinctly from each other. Thus, it is possible to investigate in detail each portion of troubleshooting. Other domains in which troubleshooting occurs may inherently simplify the character of the behavior. For example, if we were to study troubleshooting in the domain of TV-reception improving, we might not be able to observe separate steps because to improve a television's reception, a person normally just adjusts the fine-tuning or the antenna; the steps of location and repair merge into one set of actions.

This article presents a series of experiments in the domain of computer program debugging, designed to further specify the model of general troubleshooting already presented. In addition, these studies address some issues specific to debugging. In all of the studies, debugging is done in the LISP programming language, and the subjects are students enrolled in introductory LISP courses.

2. EXPERIMENT 1: AN EXPLORATION OF PROGRAMMING ERRORS

A prerequisite to understanding student debugging is knowledge about what kinds of bugs students produce and how these bugs reflect their state of knowledge. For instance, if most bugs reflect a fundamental lack of knowledge on the students' part, we should see very different debugging behavior than if most errors reflect slips (as defined by Brown & Van Lehn, 1980). Therefore, our first experiment consists of a series of observations about the errors students make while programming. The first study (Experiment 1a) is of the errors students make while working with a computer-tutoring system that teaches LISP programming (Anderson & Reiser, 1985; Reiser, Anderson, & Farrell, 1985); the second study (Experiment 1b) is of students' errors in a "natural" programming environment.

2.1. Experiment 1a: Errors While Using an Intelligent Tutoring System

Method

Subjects. The data analyzed were from two groups of students and the subjects of Experiment 3. The first group consisted of 13 students who coded

some basic LISP functions[1] (exercises from chapter 1 of Anderson, Corbett, & Reiser, 1987) and list-iteration functions (chapter 8). The second group consisted of 20 students who coded functions involving reading and printing (chapter 5), input-controlled iteration (chapter 6), and numeric iteration (chapter 6). In addition, the data from the 36 subjects of Experiment 3 were similarly analyzed. These subjects had coded four numeric-iteration functions (chapter 6). Sample problem descriptions and their solutions for problems involving basic LISP functions (Figure 14), list-iteration functions (Figure 15), reading/printing functions (Figure 16), input-controlled iteration functions (Figure 17) and numeric-iteration functions (Figure 24) are found in the appendix.

Procedure. The data were collected during the normal operation of the LISP tutor. This computer-tutoring system requires students to code a set of exercise problems for each lesson; the lessons correspond to the chapters of Anderson et al. (1987). In its usual capacity in a class, the system has students generate fairly standard programs; every student writes a program that is almost identical to every other student's program. Feedback to the student is only given when the student makes a coding error. As long as the student types in correct code, the tutor remains silent. In this way, when students are finished coding a problem, their result is a perfect program.

In addition to teaching LISP programming, this system also works as a data-collection device. The LISP tutor records information relevant to the student's coding of each function. In order to better explain the analysis performed and the data used, a brief explanation of the internals of the LISP tutor is necessary (see Anderson & Reiser, 1985, for more complete details).

The LISP tutor was built around a model-tracing paradigm. That is, the tutor has built into it an ideal student model, a production-system model of the different ways a student should write LISP code, that follows along with the actual student coding a function. Each production corresponds to the student either typing or planning a portion of the function. When the student deviates from the correct solution path, the tutor gives specific feedback and requires the student to try again (from the most recent production firing, not from the beginning of the function). For example, say a student should have typed (car arg), where arg is a variable name and car is a function that returns the first element of the list in arg. Instead, the student might make an error and begin to type (cdr arg), cdr being a function that returns the given list minus its first element. As soon as the student finishes typing (cdr, the tutor would respond with the feedback: "CDR will remove the first item from

[1] All LISP programs are functions in the sense that other programming languages use that word. In this article, the term *function* is used synonymously with *program*.

arg, but we want to return that first item. You will need a different function." Student errors that have such remediation are referred to as *diagnosed bugs.*

If the tutor doesn't recognize a student's input as correct or as a bug, the tutor displays "I don't understand that." and asks the student to try again. These errors, student inputs for which no feedback exists, are referred to as *undiagnosed bugs.* After entering two undiagnosed errors for a particular input, the student is given the answer that the tutor expected (via the ideal student model) and an explanation of why the answer is correct. Students may also obtain a particular answer and an explanation of the answer at any time while coding a function by typing an explain key. In sum, there are four actions a student may take that are relevant to each coding-production: entering a correct answer, entering a diagnosed error, entering an undiagnosed error, and typing the explain key. The tutor records (and time-stamps) each student interaction in terms of the productions possible, the student's input, and the tutor's response.

For the analysis, a 4 × 4 table of frequencies was calculated. For each opportunity to input to a particular production, we recorded the action taken by the student (i.e., correct response, diagnosed error, undiagnosed error, explain key) and the action taken on the previous opportunity with the same production. For example, if for a particular production the subject gave a correct response and had typed the explain key on the previous use of this production (which could have been in the same or a previous function), then we would add one to the number in the lower left-hand box of the frequency table (Figure 2).

Before discussing the results, what we mean by "opportunity to input to a particular production" needs clarification. Although students are given a number of tries to enter the correct code for each production, the frequency table only reflects the student's first response. Thus, if the first response is an error, it is recorded as such, the student receives feedback, and is prompted to try again. However, for purposes of our analysis, these responses that immediately follow feedback are excluded. The next opportunity would be the next time that production is needed in either the current problem or a later problem.

Results and Discussion

The frequencies for Lessons 1, 5, 6, and 8 are collectively shown in Figure 2. The entry for diagnosed-error/diagnosed-error contains two numbers. The first is the number of times the prior diagnosed error was the same as the current error; the second number is the number of times the prior error was different from the current error. Altogether, 13,793 pairs of responses are analyzed, 2,298 of which involve at least one error.

This figure shows three interesting characteristics. First, the probability of

Figure 2. **Frequency data of Experiment 1a.**

	Current Action			
Previous Action	Correct	Diagnosed	Undiagnosed	Explain-key
Correct	11495	507	343	58
		Same/Different		
Diagnosed	589	99/44	31	7
Undiagnosed	435	28	68	5
Explain-key	59	6	3	16

generating a correct answer after a correct answer was entered is high (.93). Thus, it is unlikely for students to make an error once they use a production correctly, but the chance of making an error is still greater than zero. Second, the probability of entering an error after generating an error is somewhat low (.22). These numbers suggest that students' errors are unstable. Out of the 2,298 pairs that involve at least one error, only 307 involve an error on both occasions (the sum of the numbers in the 3 × 3 submatrix made by excluding the row and column labeled *correct*). Of these, 143 involve diagnosed errors on both occasions and 99 of these are repeats of the same bug. This proportion (99 out of 143) might seem high, however, note that on the average, the tutor can recognize about two bugs per production. So, if both bugs were equally likely, we would have a 50:50 ratio. The observed value of 70:30 is not surprising given that some bugs are more frequent than others. In sum, there is no evidence in the data for the proposition that students tend to repeat their errors in the way we would expect them to if they had systematic misconceptions.

One final observation about the frequency table is that the frequencies from students in each lesson are distributed similarly despite the fact that the data were generated during the coding of very different LISP functions. In fact, the students in Lesson 1 weren't even coding their own LISP functions; they were merely generating calls to existing functions. Thus, whatever characteristics the figure shows, they must stem from a source that does not necessarily change with increasing programming knowledge.

Summary. The analysis seems to reveal that the errors produced during coding with the LISP tutor are relatively unstable; it is much more likely that a student enter a correct response after an error, rather than two errors in a row. In addition, once the student does enter a correct response to a production, it is unlikely that the student will make a mistake on that production again. Even when students make an error twice in a row, there is a good chance that the errors will be different.

2.2. Experiment 1b: Errors in a "Natural" Programming Environment

After observing the characteristics of the bugs students generate, we decided it would be useful to look at students programming in an unrestrained LISP environment to see if the same trends held. In addition, we wanted to gather protocols from the subjects to help us identify the source of these errors. In other words, we have seen quantitative evidence for bug instability; how we wish to see the qualitative explanation for such instability.

Method

Subjects. Subjects were 18 Carnegie Mellon University undergraduates enrolled in a LISP course. All had previously completed one introductory PASCAL course. The LISP course was divided into 10 lessons, ranging from basic LISP functions to recursion and search techniques. For the first 9 lessons, each student received a lesson booklet and attended a lecture. The lesson booklets, which were draft copies of Anderson et al. (1987), were used in lieu of a textbook. Each booklet described the new concepts for the lesson and gave functions for the student to write. Half of the class (structured section) completed each lesson with assistance from the LISP tutor (described in Experiment 1a) and the other half completed each lesson on their own (exploratory section). Of the students who participated in this study, 10 were from the structured section and 8 from the exploratory section. Preliminary analysis revealed no substantial difference betwen students in the two sections, so all results will be collapsed over the sections.

Design and Materials. Subjects wrote three LISP functions with the aid of a human tutor (discussed next). The first and third functions, BREADTH and BEFORE, are instantiations of a general graph-searching function format presented in Winston and Horn (1981). In this format, the variable of primary interest is *queue,* the control variable for the search loop. This variable contains a list of nodes of the network that have not yet been examined. After being examined, a node is removed from the queue and its immediate successor nodes are added to the queue. The function stops either when all nodes have been examined (i.e., the queue is empty) or a particular node is found. In the case of BREADTH, the function is looking through a predefined network, in a breadth-first fashion, for the first node with a particular attribute. In BEFORE, the function searches a network of course prerequisites and outputs all courses that are the prerequisites for a given course. The second function that subjects coded, PRE-REQUISITE, sets up the course prerequisite network that BEFORE searches. The problem descriptions and ideal solutions for these three functions may be seen in Figures 18, 19, and 20 of the appendix.

Procedure. The data were collected while the subjects completed a lesson on search techniques. For this lesson, the subjects received a booklet (as usual), but instead of completing the lesson on their own or with the LISP tutor, all subjects received the help of a human tutor. The function of the human tutor was to answer any questions the subject might have, assist the subject (minimally) in writing the three functions that comprised the lesson, and act as a verbal-protocol prompter. In addition to verbal protocols, a time-stamped transcript of everything the subject typed into the LISP environment was collected. The LISP environment used was the same one in which the exploratory students did their previous lessons and in which the structured students tested each problem in a lesson after having written the problem with the help of the LISP tutor.

Results and Discussion

The modal time to complete the lesson was 2 hr. Subjects' final solution to each problem were typically similar to the solutions shown in Figures 18, 19, and 20, with any differences being relatively minor and not changing the overall flow-of-control of the ideal solutions.

Figure 3 shows the frequency of each bug we observed subjects generating in the three programs. For sake of readability, these bugs have been placed into 5 categories according to their possible origins:[2] (a) goal errors, which show as missing pieces of code; (b) misrepresentation errors, which show as a misunderstanding about the problem statement or confusion over the roles of variables; (c) intrusion errors, which show as pieces of code that would have worked in a previous program, but are inappropriate in the current program; (d) misconceptions, which reflect a misunderstanding about an aspect of LISP; and (e) syntactic errors. An exhaustive set of examples of these bugs may be found in Figures 21, 22, and 23 of the appendix. A discussion of the nature of these errors follows. For purposes of the current discussion, only the results pertaining to the bugs generated by subjects are presented. The debugging of these bugs is discussed in Experiment 2.

From a descriptive point of view, the data show two obvious results: (a) that the bugs are very local and (b) that bugs do not repeat. The fact that the bugs are local means each bug is associated with only one line of code. None of the bugs produced span a number of lines, which would show that subjects had a misunderstanding about some programming or algorithm construct.

In addition to generating local bugs, subjects did not often generate the same bugs neither within a program at different places nor between programs. For example, although some subjects missed the `setq` (i.e., the "argument for function call: `setq`" bug of Figure 3) at one place where a variable was being set, they used `setq` correctly at other places in the same program or used `setq`

[2] For a more detailed discussion of these bug categories, see Katz and Anderson (1986).

Figure 3. **Bugs generated in each program. *indicates that errors were possible.**

	Program Name		
	Breadth	Prerequisite	Before
Goal Errors			
argument function call:			
setq	1	1	6
car	*	*	1
expand	1		2
cons		*	2
list	4		
prog	*	1	*
missing label	*	*	1
missing result update	*		1
missing (go loop)	*	3	1
missing member test			2
missing queue update	*		1
missing loop variable update		3	
missing cdr step	*		1
(of queue update)			
missing append step	1		2
(of queue update)			
missing exit test	1	*	*
missing argument to:			
equal	2		*
putprop		1	
member			1
missing has-pre-req putprop		1	
Total	10	10	21
Intrusion			
argument for function call:			
return	10	3	*
list for expand			7
nil returned	*		3
(go loop) inside cond			1
Total	10	3	11
Misrepresentation Errors			
member test/action mismatch			2
result initialized as queue			1
result updated as queue			2
extra list			1
t returned	10		*
example for variable	1		*
depth-first search	1		
parameter for (car queue)	10		*
no loop		1	

Figure 3. **Continued**

	Program Name		
	Breadth	Prerequisite	Before
'loop' for variable name	*	1	*
pre-req-for for has-pre-req		2	
Total	22	4	6
Misconceptions			
arguments in wrong order:			
cons		*	1
putprop		3	
cons for append	*	1	1
missing quote	2	*	*
extra quote	4	8	*
putprop has-pre-req separately		3	
didn't add pre-req-for		11	
Total	6	26	2
Syntactic Errors			
missing)	9	1	5
extra)	1	1	*
missing (	*	1	*
extra (	*	1	*
missing ( )	*	1	*
extra ( )	*	*	2
Total	10	5	7

correctly in a later or earlier program. Of the 58 bugs generated in the first program, only 9 were generated by the same subject in later programs; of the 48 bugs generated in the second program, only 4 were generated by the same subject in the last program.

With the exception of one subject who was confused about cons and append and a few subjects who had problems with putprop (a function that had just been introduced), subjects clearly did not hold many firm misconceptions about LISP. Their errors reflect this fact in that particular subjects did not generate the same bugs either within a program or across programs. If a subject did have a misconception, as the few subjects did, every time there was an opportunity to generate the bug, the subject should do so. Because subjects only occasionally generated particular bugs, some other explanation of the origins of the bugs is necessary.

Although subjects didn't hold many firm misconceptions, it should not be inferred that subjects held no misconceptions. It is entirely possible that some bugs were caused by the subjects misunderstanding something about search

functions or about LISP in general. The reason the bugs didn't repeat might be because students corrected their misconception after one instance of feedback (i.e., correcting the bug). Or, subjects might have been unsure about what to type, had entertained some competing hypotheses about what is correct, and had picked the wrong action. On the next try, subjects would have a better chance of picking the correct hypothesis.

However, the fact that many of the bugs appeared for the first time in later programs after the subject had coded, without error, the same or a similar line in a previous function, plus the fact that bugs did not repeat within programs, suggests a third explanation: Subjects had the correct LISP and programming knowledge, they just failed to retrieve or apply the knowledge correctly. These retrieval or application failures would have been caused by some inconsistent mechanisms such as working-memory failure or set effects.

2.3. Discussion of Experiments 1a and 1b

The results reveal evidence for the generation of errors primarily by some inconsistent mechanism rather than due to misconceptions on the part of the subjects. Anderson and Jeffries (1985), who were looking at a different set of errors, also observed the inconsistency of programming errors. They found that the frequency of such errors increased with working memory load. Their hypothesis was that errors were due to loss of information from working memory. This hypothesis is consistent with certain categories of errors we observed from Experiment 1b such as goal and syntactic errors. The other major category of error from Experiment 1b also produces an inconsistent pattern. These are the errors produced when the subject misunderstands the problem. The subject is unlikely to misunderstand the next problem statement that creates the same error.

We do not mean to imply that subjects never have misunderstandings about LISP or that their errors cannot reflect these misunderstandings. However, it is apparent that misunderstandings are quite infrequent for our population of subjects solving the kind of problems we give them. The errors we find are quite different than the misconceptions studied by Spohrer and Soloway (1986), although a possible reason for this difference might be the method of instruction used (standard classroom instruction vs. an intelligent tutoring system). The fact that misunderstandings are so infrequent means that when we examine subjects' debugging behavior in Experiments 2, 3, and 4, we are looking at subjects with adequate knowledge to correct the bugs if they can find them. This characteristic of our subject population greatly simplifies analysis.

3. EXPERIMENT 2: AN EXPLORATION OF DEBUGGING

Having now characterized the source of errors in programming, the next agenda item is to see how subjects go about debugging. In this experiment, we

were interested in observing the range of possible behaviors for subjects debugging LISP functions. To allow the subjects as much freedom as possible, the functions were coded and debugged in an unrestrained LISP environment.

3.1. Method

The subjects, design, materials, and procedures for this study were described under Experiment 1b.

3.2. Results and Discussion

The schematic protocols of eight subjects debugging BEFORE were used. The protocols covered from when BEFORE was initially entered to when it was completely debugged. The eight subjects were chosen for observation because they all generated bugs leading to BEFORE returning "nil." Unlike many error messages, this symptom did not print to a specific area of the code as being incorrect. Thus, subjects were forced to think about their program in detail and look for more information with which to track down the actual bug. In addition, because many of the observed bugs in BEFORE could produce this symptoms, and it was often the case that a few of the nil-producing bugs existed in a function at one time, it was possible to look at the relative difficulty of debugging various bugs that produce the same symptom.

Some of the bugs that gave the nil symptom were more difficult to find than others. That is, although the subjects seemed to find some bugs quickly and without any help from the human tutor, other bugs required extensive tutor intervention. However, it was rare for a subject to need assistance in fixing a bug once it was located. For these subjects, the trouble was in finding the bug rather than correcting it, but other researchers have found problems in the fixing stage of debugging (e.g., Jeffries, 1982; Kessler & Anderson, 1986). This result is consistent with our earlier observation that the bugs manifested by this subject population did not reflect any real misconceptions about LISP.

The observations are discussed in terms of the troubleshooting model covered in the introduction (Figure 1). Specifically, we focus on the location and repair stages of debugging. The first stage shown in the model, understand, was not observed in the current situation, although other researchers have observed such a process when a subject debugs another's program (Jeffries, 1981, 1982; Kessler & Anderson, 1986). That is, in this experiment, subjects did not spontaneously look over their program before testing it, perhaps because they felt that they already had a good understanding of what they wrote.

Locating Bugs. When looking for a bug, the subjects seem to be working from a mental representation of the program as well as the listing of the

program itself. This mental representation is probably built during the coding of the function and maintained by the salience of the actual, written code. The representation appears to contain information regarding the subjects' intentions in writing a line or whole section of code as well as presuppositions concerning the code in general (e.g., the only possible exit from the function is the return statement).

There were three general strategies that the subjects used to locate a bug: (a) a simple mapping from the program's behavior to the bug, (b) hand-simulation, and (c) causal reasoning.

The simple mapping strategy merely means that the program's buggy behavior pointed to the bug that produced this behavior. In the current situation, the buggy behavior invoking this strategy was an error message. A typical example of this strategy is when a subject saw the error message "ERROR: eval: undefined function course." The subject wondered aloud why the computer thought that "course" was a function (course was a variable name), and started looking at instances of the word couse in the program. No reasoning about the program was necessary beyond realizing that because the error messages referred to the word course, the bug must have something to do with where that word appeared in the program. This strategy may become more difficult to use when a program has a large number of occurrences of a particular variable.

Hand-simulation consists of the subject executing the program (with sample parameter values—similar to Kant & Newell's, 1984, test-case execution) as the computer would and looking for inconsistencies between what occurs in the actual function and what is expected as per the subject's representation of the function. Researchers studying debugging in other languages have also observed this strategy. For example, Klahr and Carver's (1988) "brute-force" strategy observed in children debugging LOGO programs or Jeffries (1981, 1982), who observed this strategy being used by both expert and novice PASCAL programmers. In addition, the use of simulation in electronics troubleshooting may be inferred because factors that would seem to affect the difficulty of simulating a system (e.g., the existence of feedback loops; Rouse, 1979) cause a degradation in troubleshooting performance.

However, unlike the novices in Jeffries's (1982) experiment, for our subjects, simulation did not seem to be a frequently used strategy and, in fact, many subjects switched from simulation to causal reasoning in the middle of simulating. A problem noted with simulation is that subjects may believe that they are executing the program as written, but may, in fact, be executing the program as intended. For example, several subjects generated the "argument for function call: setq" bug in Line 9 of BEFORE (Figure 20). When simulating the function, some of these subjects would interpret the line as if it read "(setq visited (cons (car queue) visited))" instead of what was

actually there: "(cons (car queue) visited)". Thus, the subjects incorrectly read the actual line as updating the result variable visited. It is unlikely that this misinterpretation is caused by a general misconception with setq, or other LISP functions, for two reasons: all subjects correctly used setq elsewhere, and all subjects were able to correct the line once they determined (or were told) that the bug existed at that line. Curiously, this misreading of the code did not seem to appear when subjects used the causal reasoning strategy.

The causal reasoning strategy involves looking at the information obtained in the testing of the function (e.g., the program's output or the results of tracing the subfunction EXPAND) and reasoning about what might be causing the bug. In this strategy, it is the subjects' representation of the program, as well as general LISP knowledge, that seems to guide the search for the bug, as opposed to hand-simulation in which the actual, written codes guide the search. This strategy, in slightly different forms, has also been observed in subjects debugging PASCAL and LOGO programs (Gugerty & Olson, 1986) and in subjects troubleshooting electronic circuits (called the "evaluative strategy" by Rassmusen & Jensen, 1974). As an example of causal reasoning, consider one subject who had generated the bug "missing result variable update" (Figure 23). Upon seeing the nil symptom, he started reading over the program (simulating) from the beginning. When he reached the cond-clause ((null queue) (return visited)), he stated, "visited has nothing in it when it returned, maybe we didn't add anything to visited." The subject then looked at the appropriate section of code and realized that he had forgotten to update the result variable.

As it happened, not updating the result variable was not the bug that caused the program to return nil. The function returned nil because, due to a missing close parenthesis, the (go loop) statement was skipped. The subject eventually found this bug by tracing the helping function, EXPAND, and seeing that BEFORE never looped. It seems that the fact that the function would potentially not return through the return statement of Line 5 (Figure 20) was not initially encoded in the subject's representation of the function. Only when forced by other information was the subject able to elaborate his representation to include this possibility.

Why are some aspects of the program represented initially in the subject's mind but other aspects are only added after other possibilities are refuted? One answer might be familiarity. If a person often makes parenthesis errors, such an error might well be the first thing checked when debugging future programs, perhaps even independently of whether or not a program's behavior actually reflects such bugs. As time goes on, people may learn more and more of these mappings between bugs and behavior, until most of their debugging looks like simple mapping. It is just these "compiled reasoning" mappings that Klahr and Carver (1988) used in teaching children debugging skills. For our subjects, it might have been the case that their past programming

assignments were of sufficient simplicity that parenthesis errors were rare or obvious when made (and, for the structured subjects, the LISP tutor handled parenthesis balancing). As a result, they did not have a sufficiently refined representation of the parenthesis level at which a statement occurs and, therefore, found certain parenthesis bugs difficult to locate. The importance of subjects' representation of the program is addressed further in Experiment 3.

Repairing Bugs. As reported in Experiment 1b, subjects' troubles did not stem from problems with their general knowledge of LISP, but rather from the application of this knowledge to code a program. This result is also reflected in the subjects' ability to fix bugs once they were located. Subjects had little difficulty in correcting errors once it was known at which statement in the function the error occurred. Only 8 out of 18 subjects made mistakes when correcting a bug; each subject make only one error. Thus, out of the 47 errors that were corrected in BEFORE, only 8 miscorrections were made. No miscorrections were made in either BREADTH or PRE-REQUISITE.

Summary. The protocols show that subjects produce a wide range of behaviors when debugging their programs. From the observations, it is clear that debugging is not a single activity, but a set of activities, each component of which may be performed differently depending on the situation. Specifically, in the current situation, subjects showed a variety of strategies in how they went about locating the bugs in their code.

With these two experiments as background, we now report two experiments that focus on the issue of the processes involved in locating a bug. The first experiment is concerned with how these bug-location strategies might vary depending on whether subjects are debugging their own program or someone else's.

4. EXPERIMENT 3: DEBUGGING YOUR OWN VERSUS ANOTHER'S PROGRAM

Programmers often report that instead of debugging someone else's program, they would rather write their own version. These programmers are implicitly stating that the task of debugging another person's code is more difficult than the combined tasks of writing and debugging their own program. Are the tasks of debugging your own versus another's code really so different? The notion is intuitively appealing as well as supported by an informal observation made of subjects' behavior in the exploratory experiment (Experiment 2). Specifically, we speculate that subjects might vary in their choice of bug-location strategies.

In her study of expert and novice programmers, Jeffries (1982) reported seeing little use of the causal reasoning strategy by her subjects, who were debugging programs prepared by the experimenter. In contrast, the causal

reasoning strategy was observed quite frequently in Experiment 2, in which subjects were debugging their own programs. Although there are many differences between Experiment 2 and Jeffries's study (e.g., different computer languages, different debugging situations, and different length of programs), the difference in authorship of the programs being debugged seemed the most interesting distinction to focus on. Thus, Experiment 3 was designed as a direct look at the relationship between authorship and performance.

4.1. Method

Subjects were asked to write and debug four LISP functions. Subjects debugged their own functions (OWN task) as well as other subjects' functions (OTHER task). For each of the four trials, subjects wrote either a target function or a filler function. After writing a target function, subjects debugged that function. After writing a filler function, subjects debugged another subject's target function. There were four experimental conditions which differed in the order of debugging tasks that subjects performed: (a) OWN, OWN, OTHER, OTHER; (b) OTHER, OTHER OWN, OWN; (c) OWN, OTHER, OWN, OTHER; and (d) OTHER, OWN, OTHER OWN. In addition to time-stamped keyboard transcripts of the subjects' interaction with the system, concurrent verbal protocols were collected while some subjects were debugging the functions.

Subjects. Subjects were 36 Carnegie Mellon University undergraduates who were recruited from three introductory LISP courses. Fourteen, 11, and 12 subjects were taken from each class, respectively. The data from one subject in the third class was excluded because the subject refused to follow instructions. The experiment was completed as an optional course assignment. The experiment occurred approximately halfway through each LISP course, as Lesson 6 out of 12 total LISP lessons. All three courses used the same textbook (draft copies of Anderson et al., 1987) and assigned the same homework problems. Students in the first class normally completed each homework problem on their own. Students in the other two classes completed most of the problems with the aid of the LISP tutor, and one problem per lesson was completed on their own. Thus, students in all three classes were exposed to almost identical LISP materials.

The computer experience of the subjects differed depending on which course they were recruited from. The first class consisted of students who had previously taken at least one introductory PASCAL course, although some students had more experience than just one prior programming course. A prerequisite for the second class was that students had taken exactly one introductory PASCAL course previously. A prerequisite for the third class

was that this class be students' first formal computer language course. However, there did seem to be an amount of overlap in ability between the groups, possibly because not all of the students paid attention to the courses' prerequisites.

Finally, concurrent verbal protocols (Ericsson & Simon, 1984) were collected from 13 of the subjects as they debugged each function. The protocol subjects were evenly distributed among the four experimental conditions: four subjects from the first condition and three subjects each from the other three conditions.

Materials. There were six LISP functions used in this experiment. The problem descriptions and code for the functions are shown in Figure 24 of the appendix. Four of the functions (FACTORIAL, CREATE-LIST, LIST-SKIP, and NUM-SUM) were target functions; they were all debugged by subjects, although each subject wrote only two of these functions (as described next). Two of the functions (NEXT-PRIME and NEXT-PRIME-BOUND) were filler functions; they were written by all subjects before debugging another subject's function.

Subjects wrote each function with the aid of the LISP tutor. As was the case in Experiment 1a, this tutor normally gives feedback to students as they are coding a problem such that the final result is a perfect program. For this experiment, the tutor was altered to accept certain errors in coding as if they were correct.

The bugs that were allowed by the system were chosen from keyboard transcripts of students from past classes writing the same functions. The frequency with which these bugs appeared suggested that subjects in the experiment would be likely to generate the bugs. There were three types of bugs used in this experiment, which were distinguished by how they usually affected a program's behavior. The bugs chosen would have (if they had not been corrected by the LISP tutor) caused a program either to: (a) go into an infinite loop (actually, the LISP environment in which programs were executed was guarded against true infinite loops because an error message was displayed when a function iterated more than 250 times), (b) return the initial value of the result variable (e.g., FACTORIAL returning 1 or CREATE-LIST returning a list containing the argument to the function), or (c) return some other incorrect or incomplete result (e.g., CREATE-LIST return (1 2 3) when the argument was 4 or FACTORIAL returning 0). In all, it was possible for subjects to generate a maximum of about seven bugs per program.

In order to debug functions, subjects were given the use of an editor. The main purpose of this editor was to keep the subject from creating any more bugs in the function than already existed, thus assuring that subjects would be looking for only those bugs generated during coding rather than bugs

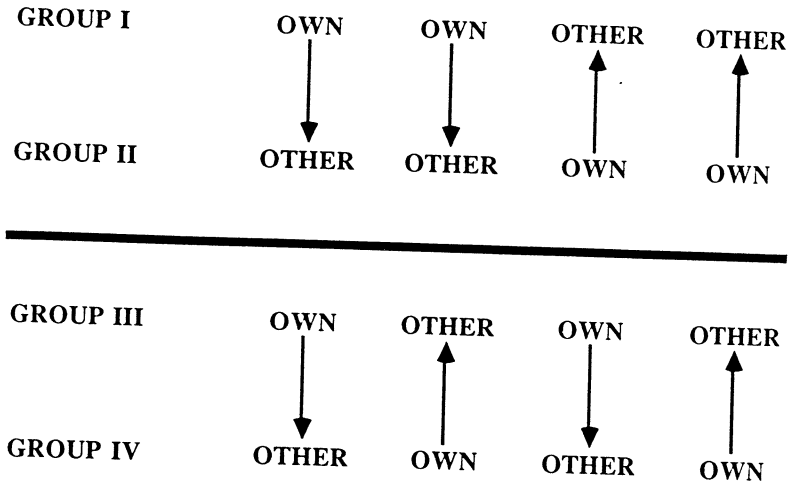
introduced during debugging. This restriction was necessary to insure that yoked subjects would always be looking for the same bugs. To implement this restriction, subjects were allowed to make any changes to the function that they wished, although if the function didn't work (as determined, internally, when the subject exited the editor), the editor would restore any changes that would not have specifically fixed the bugs in the program. For example, say a subject's function has two bugs in it. During an editor session, the subject might fix one of the bugs, but change some other parts of the function that is unrelated to either bug. When the subject leaves the editor, the only change to the function that would be kept would be the actual bug fix. The code associated with the other change would be restored to what it was before the subject started the current editor session, and a message would be printed to tell the subject that the editor had done such restorations. The current version of the function was always displayed to the subject upon leaving the editor and reentering the LISP environment.

Clearly, this editor restricted the range of possible debugging behaviors that a subject might exhibit. Specifically, subjects were limited in the possible debugging strategies they could use. The editor did not allow subjects to add lines that would print variables nor did it allow them to make successive approximations to the bug fix. However, these restrictions should have caused subjects to act more homogeneously; it would be more likely that subjects performed the same whether or not they were debugging their own function. Thus, the limits caused by the editor actually worked against our hypothesis that people perform differently in the two debugging tasks.

Design. For each subject, the experiment consisted of four trials in which the subject wrote one LISP function and then debugged one LISP function. Subjects debugged either a function they had just written or a function written by a previous subject. In the latter case, the OTHER debugging task, subjects wrote a filler function (either NEXT-PRIME or NEXT-PRIME-BOUND; see Figure 24) before receiving the other subject's function for debugging. Subjects debugged two of their own functions and two of another subject's functions. Subjects in the same class always debugged the four programs in the same order, although each subject probably saw the programs with different bugs in them. Two classes debugged FACTORIAL, CREATE-LIST, LIST-SKIP, and then NUM-SUM; the other class debugged CREATE-LIST, FACTORIAL, NUM-SUM, and then LIST-SKIP. Thus, any results collapsed over program should be relatively independent of attributes of the particular programs, but not entirely. Completely counterbalancing the order of functions was not feasible due to the limited number of subjects available.

There were four experimental conditions distinguished by the order of debugging tasks subjects performed. Figure 4 shows these conditions and how

Figure 4. Design of Experiment 3.



they interacted. As shown by the arrows in the figure, the subjects in Groups I and II and in Groups III and IV passed their functions (i.e., a LISP function plus any bugs generated) to each other. The first Group I and III subjects from each class debugged programs written by the experimenter and seeded with two bugs each; these data were excluded from all analyses. In general, a Group I subject would debug two of his own functions and then two functions of the previous Group II subject, while a Group II subject would debug the first two functions of the previous Group I subject and then two of her own functions, which, in turn, would be passed on to the next Group I subject. A similar relationship held between subjects in Groups III and IV. This partial yoking reduced the chance that any differences found between the two tasks are due to people debugging different bugs. Because the editor did not allow bugs to be introduced during debugging, yoked subjects were always looking for precisely the same bugs.

Procedures. Subjects completed the experiment either alone or in groups of two, but all worked separately. Before beginning the experiment, subjects were trained on the use of the computer tutor and on the editor. The subjects from two of the LISP classes needed no training on the tutor because they had been using the tutor for their class. The other group of subjects received a small amount of instruction on the use of the tutor and then used the LISP tutor to write two functions, which they had written on their own previously in the class. All subjects then received about 15 min of training on the editor in which they had to perform a set of editing procedures. An informal criterion was used to determine when a subject was proficient enough on the editor to begin the experiment.

After the training, subjects were told that they would be writing four numeric-iteration functions with the aid of the LISP tutor. Subjects were told:

"The tutor has been altered to be a little more forgiving than usual in the range of solutions it will accept. As a result of this extra freedom, the tutor may not recognize when you make an error. Thus, the programs you write may contain bugs and, afterwards, you will use the editor and the LISP window to debug your programs." Subjects then wrote their first function for the experiment with the aid of the LISP tutor.

Once this function was written, a LISP environment was made available to subjects. At this point, subjects were asked to debug a function. If a subject was to debug the function just written, he or she was told to "make the function you just wrote work, if it doesn't work already, by using the LISP environment and by using the editor you saw previously" (subjects were able to easily switch between the editor and the LISP environment). If a subject was to debug another subject's function, he or she was given the problem description for that function (Figure 24) and told essentially the same instructions, except that the subject was to debug a function that the computer had just loaded in and to forget about the function he or she had just written. Sometimes it turned out that the subject was asked to debug a completely correct program. In these cases, the subjects' task was trivial: They merely had to test the program once (for every function, before being allowed to go on, the subjects were required to call each function successfully at least once) and then go on to write the next program.

For all subjects, the experimenter was available to answer questions about editor commands, although the experimenter usually did not watch as the subjects debugged or wrote their functions. If subjects had not fixed the function within 20 min, the experimenter told them what bugs remained and watched as they fixed those bugs. After correcting the first function, subjects wrote their second LISP function. This procedure continued for four trials. Subjects were then debriefed on the purpose of the experiment, and an informal interview usually ensued.

4.2. Results and Discussion

The modal time to complete the experiment was 2 hr, with times ranging from ½hr for quite experienced programmers to over 3 hr for relatively computer-naive subjects. Figure 5 shows some performance comparisons of the subjects with different backgrounds. The subjects from the first class were less likely to generate an error in the two programs that they wrote than were subjects of the other two classes, $\chi^2(1, N = 72) = 5.05, p < .03$. Also, given that a subject is debugging a program with a bug (recall that some programs may contain no errors), the subjects with more experience were able to find the error without any assistance (nonsignificantly) more often than the other subjects, $\chi^2(1, N = 77) = 1.31$, *n.s.* However, the current series of studies

Figure 5. **Comparisons of subjects from the three classes: number of incorrect programs and errors found without assistance.**

	Relative Computer Experience		
	High	Moderate	Low
Erroneous programs	13	15	17
Correct programs	15	7	5
Errors found	15	15	17
Errors not found	6	12	12

was not designed to investigate differences due to background and, in other comparisons, expert–novice differences do not appear. Thus, the remaining results were collapsed over the three groups.

The first hypothesized source of differences between debugging your own versus another's program was in the relative understanding of the program in each task. Jeffries's (1982) subjects, who debugged the experimenter's programs, spent a great deal of time comprehending the program: in contrast, the subjects of Experiment 2, who debugged their own programs, did not spend time just looking at the program. Thus, a subject debugging another's program should spend some time looking at the program before beginning debugging, but should spend less time (if any) looking at his own program. This prediction was substantiated. Comprehension time was taken as the time between a person first seeing a program and when the person began to type. Due to a technical error, this measure was not available for two subjects. People debugging their own program (M = 37 sec) spent about 1 min less just looking at the program compared with when they were debugging another person's function (M = 101 sec), a difference which is statistically significant (within-subjects comparison), $F(1, 28) = 17.09$, $p < .0003$.

To investigate relatively how well people performed on each task, the time to debug the first bug in a program was compared for each subject. Because subjects differed in their computer expertise, a within-subjects comparison was used. Seventeen subjects were excluded from the analysis because there were no data on their debugging times for one of the tasks (their own debugging or other debugging). In addition, the time to just the first bug was used so that programs with more than one bug could be included in the analysis. The mean debugging times are shown in Figure 6. There is a significant advantage for persons debugging their own program, $F(1, 18) = 8.84$, $p < .009$. This advantage is almost entirely due to people being unsuccessful in finding the bug in the other person's program (i.e., the subject didn't find at least one bug in the program within the 20-min time limit; such occurrences were scored as the subject taking 20 min). In fact, only one

Figure 6. **Min to debug the first bug in a program.**

	Task		
	Own Program	Other's Program	*n*
All subjects	9.9	15.0	19
Successful subjects	9.8	10.4	8

subject (of the 19 represented in the figure) debugging his own program could not find any bugs, whereas 10 subjects debugging other people's programs could not find any bugs, although they were able to find the bugs in their own programs. Clearly, subjects were better overall at finding bugs in their own programs, which might reflect that subjects understood their own programs better than other subject's programs.

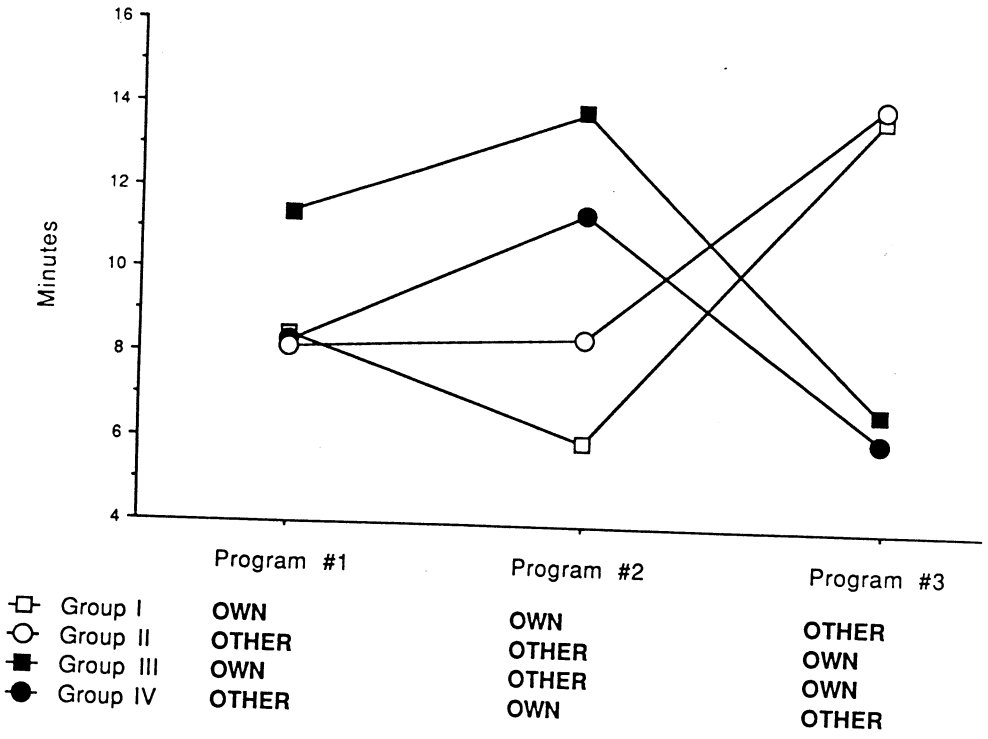
What about the subjects who were successful at debugging (i.e., those that found at least one bug within 20 min)? Our predictions were based on subjects who were able to find at least one of the bugs in a program. It is unclear what subjects are doing when they go over the 20-min time limit: Did they need just a little more time, had they given up, or had they been floundering and just guessing? Thus, all further analyses considered only those subjects who were able to find at least one bug in the program they were debugging.

For overall debugging times, if only successful subjects are considered (Figure 6), no advantage for one's own program is shown, $F(1, 8) = .23$, n.s. However, the predictions concern differences in the strategies used in each task, and average times are not suitable for demonstrating such strategy differences (Siegler, 1987). Thus, two analyses that were more sensitive to strategy differences were performed: an analysis of the learning effects over the course of the experiment and an analysis of the verbal protocols collected while subjects were debugging.

To investigate possible learning effects, we compared the time to debug each program for the four experimental conditions (Figure 7). Here, time to debug refers to times for only the first bug found in a program. This measure allows us to perform an analysis collapsed across all programs with bugs. The alternative would have been to plot separate graphs for programs with one, two, three, and so on bugs, and would have resulted in each graph reflecting only a small number of observations.

We had predicted that subjects would use one type of strategy when debugging their own program and a different type of strategy when debugging another's program. Thus, if we assume that the tasks of debugging your own versus debugging another's program are distinct, we should see separate learning trends for each task. Although there should be transfer from performing a task on one program to performing that task on another

Figure 7. **Min to debug the first bug across the three programs.**



program, there should be no between-tasks transfer. A 4 × 3 (Condition × Program[3]) between-subjects analysis of variance (ANOVA) was performed. Because the experimental conditions differed only in the order of the debugging tasks performed,[4] we would expect a Condition × Program interaction, which was indeed the only significant effect found, $F(6, 21) = 3.21$, $p < .03$.

In looking at the graph of the data (Figure 7), it is clear that Groups I and II are very similar as are Groups III and IV, but the two sets of groups are different from each other. This similarity between groups is not surprising because each pair of groups are complements of each other in terms of the tasks performed (Figure 4). Groups I and II perform the same task twice, then a different task, whereas Groups III and IV alternate tasks. In fact, in

[3] The debugging times for the fourth program were not included as there were no subjects in one of the groups who had debugged this program and only one subject from the remaining three groups. The reason for the small number of subjects debugging this last program is that most people did not generate any bugs.

[4] From one of the LISP classes, the data of subjects who had debugged the programs in a different order from subjects in the other two classes were excluded from the analysis because only data from subjects in experimental Groups I and II were available. There were not any data from the subjects in the other groups because the subjects in Groups III and IV did not generate many bugs in the second or third programs, and those few subjects who did get an opportunity to debug generally were unsuccessful at finding the bugs.

collapsing the groups by task pattern, a 2 × 3 (Task Pattern × Program) ANOVA revealed the obvious interaction as the only significant effect, $F(2, 27) = 10.75$, $p < .0004$. An explanation of the interaction becomes clear upon examination of the different groups performance for the second and third programs.

For the second program, the two groups taking the least amount of time (Groups I and II) are also the groups performing the same task as performed in the first program. Group I subjects had debugged their own program for the first and second program; Group II subjects had debugged other subjects' programs for the first and second programs. In contrast, Groups III and IV took more time than the other groups to debug the second program. Thus, for the second program, the groups that performed a particular task for the first time were (nonsignificantly) slower at debugging than the groups that had performed the same task twice (Groups I and II: 7.7 min; Groups III and IV: 12.7 min; $F(1, 7) = 2.34$, *n.s.*).

The trend of the two debugging tasks being different becomes more pronounced in the third program. Groups I and II now take significantly longer to debug the third program compared to Groups III and IV (14.2 vs. 6.6 min; $F(1, 11) = 24.09$, $p < .0005$). Subjects in Groups I and II had performed their particular debugging tasks for the first time in the third program, whereas subjects in the other two groups had performed each type of debugging task once on previous programs.

The different learning trends for each task suggest that subjects are performing (and, therefore, learning) different behaviors in each debugging task. To pinpoint the differences in behaviors, an analysis of the subjects' protocols was performed.

For purposes of this analysis, we divided debugging strategies into two types: those that reflect *forward reasoning* and those that reflect *backward reasoning*. Forward-reasoning strategies are those in which the subjects' search stems from the actual, written code. Two examples of such strategies are: (a) comprehension (Jeffries, 1982; Kessler & Anderson, 1986) where a subject finds bugs while building a representation of the program (usually paraphrasing the code in the process) and (b) hand-simulation (Experiment 2; Gugerty & Olson, 1986; Jeffries, 1982; Klahr & Carver, 1988) where subjects evaluate the code as if they were the computer. Backward-reasoning strategies are those in which the subjects search starts from the incorrect behavior of the program. Examples of such strategies include: (a) simple mapping (Experiment 2; Klahr & Carver, 1988) where the program's output points directly to a specific line (or type of line) being incorrect and (b) causal reasoning (Experiment 2; Gugerty & Olson, 1986) where the subject searches backward from the program's output, using their knowledge of the program and programming to find the error.

The analysis consisted of deciding whether a particular protocol reflects a

Figure 8. **Strategies used separated by debugging task.**

	Task	
	Own Program	Other's Program
Forwards reasoning	2	8
Backwards reasoning	5	2

forward- or backward-reasoning bug-location strategy. Specifically, if a subject refers to successive lines of the program, without considering the output of the program, the protocol may be categorized as reflecting forward reasoning. If the subject makes predictions about the location of the bugs in a program based on the program's output or uses the program's output to limit his search of the program, that subject's protocols may be categorized as reflecting backward reasoning. As it turned out, subjects tended to use either one type of strategy or another, therefore, the categorization was an easy one to make. Only two subjects, one debugging his own function and one debugging another's function, both referred to the output of the program and to successive lines of code. Because in both cases the output of the program seemed to be main piece of information used, both protocols were coded as reflecting backward reasoning.

In performing this high-level analysis, each debugging trial of a subject was categorized separately. Of the 52 protocols collected (13 Subjects × 4 Debugging Trials Each), 8 were excluded because of technical failures in the recordings and 27 were excluded because either the programs being debugged didn't contain any bugs or the subjects did not successfully find at least one of the bugs in a program. The results of the analysis on the remaining 17 protocols are shown in Figure 8. As predicted earlier, subjects debugging their own functions tended to use a backward-reasoning strategy whereas subjects debugging another's function tended to use a forward-reasoning strategy, $\chi^2(1, N = 17) = 4.5, p < .05$. When searching their own program, subjects used the program's output and their knowledge of the program to guide their research; in contrast, when searching another subject's program, the subjects' search was guided by the actual, written code. Note that this effect is not merely a demonstration that people debugging another's program first try to comprehend it. By forward reasoning, we refer to the several different strategies just outlined. Some of the subjects debugging another's program discovered bugs through hand-simulation of the program rather than during their initial period of comprehension.

Summary. Subjects debugged programs differently depending on whether or not they had written the program. Overall, subjects were more successful

at debugging their own programs. In addition, the practice data suggested that subjects performed the two debugging tasks differently. An analysis of the subjects' verbal protocols showed that subjects debugging their own programs tended to use a backward-reasoning strategy but used a forward-reasoning strategy when debugging other subjects' programs.

In this experiment, as in previous ones, bug-location strategy has appeared to be the debugging phase of main interest. In the final experiment, we investigated what effects the different bug-location strategies might have on the other phases of debugging.

5. EXPERIMENT 4: BUG-LOCATION STRATEGIES

The final experiment was more controlled than previous ones. Here, we directly manipulated the bug-location strategy subjects could use in order to see the effects of bug-location strategies on the other components of debugging. In particular, we were interested in seeing if the two types of bug-location strategies, forward reasoning and backward reasoning, would have different effects on debugging performance.

5.1. Method

Subjects. Subjects were 27 Carnegie Mellon University students who completed the experiment as an alternative to one of the tutor lessons, as described previously. The subjects were recruited from two different LISP classes, 18 from one class and 9 from the other. All of the subjects had had prior programming experience on the order of one or two college courses.

The experiment consisted of two subexperiments: a manipulation of the strategy subjects were forced to use and an investigation of the trainability of bug-location strategies. The bug-location strategy was manipulated by giving subjects in different groups the same problems, but with different debugging interfaces.

Stimuli. Every subject saw 12 programs (two groups of six). The first four programs in each group had one bug each; the sixth program in each group did not contain any bugs. The fifth program in each group had two bugs each, but because of technical problems, subjects' performance on these problems is not discussed. All of the problems were numeric-iteration programs (Chapter 6, Anderson et al., 1987), similar to those written by subjects in the previous experiment (Experiment 3; Figure 24).

Strategy Conditions. There were three groups of subjects defined by the debugging strategy used on the first set of six programs. One group used what

Figure 9. **Sample stimulus program from hand-simulation condition.**

How the program appeared to subjects:

```
(defun factorial (num)
      (let (<COUNTER-INITIALIZATION>
            <RESULT-INITIALIZATION>)
           (loop
            (cond (<TEST>
                   <ACTION>))
            <COUNTER-UPDATE>
            <RESULT-UPDATE>)))
```

The actual (correct) program:

```
(defun factorial (num)
      (let ((count 0)
            (prod 1))
           (loop
            (cond ((equal count num)
                   (return prod)))
            (setq counter (1+ counter))
            (setq prod (* prod counter)))))
```

seemed to be a typical forward-reasoning strategy, hand-simulation (Experiment 2). A second group used a typical backward-reasoning strategy, which is similar to causal reasoning (Experiment 2), and is referred to as *working-backwards*. A third group was allowed to look at the code in any way they wished. For the second set of programs, all subjects could look at the program in any way they wished. The first set of six programs is referred to as *training* programs; the second set is referred to as *transfer* programs.

The strategies controlled how subjects went about locating the erroneous line of code in a function. Subjects were allowed to see only one line of code at a time. The other lines of the program showed only place-holders such as <COUNTER-UPDATE> or <EXIT-CONDITION>. The strategies controlled, to some extent, the order in which subjects looked at lines of code.

In hand simulation, subjects were required to look at each line of code in the same order LISP would execute the program. Figure 9 is an example of what a stimulus program looked like before subjects chose a line of code to check. Subjects were told to execute the program using given parameters that were displayed, which would cause the correct version of the function to loop once (i.e., the first time the exit test, at <TEST>, is executed it would be false, but after executing the variable-update lines, the test would be true and the function would exit). At each line of code, subjects were asked to evaluate the line (the computer kept track of the values of variables in the program and

Figure 10. **Top-level menu for working-backwards condition.**

```
What do you wish to check:
  1 Check if the function stops at the right time
  2 Check if a variable equals the answer when the function stops
  3 Check if the function returns the variable
```

displayed these to subjects) and to judge the correctness of the line. If the line was buggy, they were asked to type in the correct line of code. Subjects were given minimal feedback (correct or incorrect) on each question. For typing in correct code, subjects were given two tries to enter the code before being told the answer. These questions (evaluate, judge, and record if necessary) were asked for every line of code. To make all the strategy conditions similar, these questions were asked in the same way no matter which bug-location strategy was used. Thus, the strategies differed only in the way subjects chose a particular line, but not in what they did with each line of code.

The interface for the working-backwards strategy was designed to guide the subject through a reasoning process: Given the buggy output of the function, what line(s) could contain bugs? The top level of the interface was a menu of possible ways of characterizing the buggy output (Figure 10). After choosing one of these items, the subject was given a menu of the particular lines of code that were relevant to what the subject wanted to check. For example, if the subject wanted to check if the function stops at the right time, her or she would be given a choice of looking at the counter-initialization, test, and counter-update, which were the only lines of code relevant to the program stopping. From this menu, the subject would choose one of the lines of code (or return to the top-level menu) and would be asked the same questions for each line, as already described.

The final interface, the neutral strategy, was similar to working-backwards in that the lines of code were the leaves in a hierarchy. However, in this interface, the top-level menu was location-based. That is, subjects chose which area of the code (initializations, exit-condition, updates) they wished to look at and then were given a choice of the two lines of codes in that area. For example, if a subject chose to look at the updates, he or she was given the choice of looking at the counter-update or result-update. The subject would then be asked the same questions already mentioned. After answering the questions and assuming the line was correct, the subject would be returned to the top-level menu.

As stated previously, all subjects were transferred to the neutral strategy during the second group of six programs. The purpose of this switch was to see the effect of different prior training: Would subjects trained in different strategies behave differently when allowed do what they wanted? To allow

subjects as much freedom as possible, but still remain in the current paradigm, the neutral strategy was used. Unlike for the first six programs, subjects were no longer required to evaluate each line of code (recall that this question was included in the neutral and working-backwards interfaces to make them similar to the hand-simulation interface). Thus, for each line, subjects had only to judge if the line was correct and enter the correct code if the line contained a bug.

Procedure. Subjects were given a brief description of the experiment and then completed three practice problems to familiarize themselves with the various interfaces they would be using. The general scheme of what a subject did for each program is as follows. First, the subject read a description of what the program was supposed to do and saw two examples of the program being used. The subject was asked by the system to judge whether or not the program worked properly. If the program was buggy, the subject would then try to locate one of the bugs using one of the bug-location strategies described previously.

After finding a bug, and correcting it, subjects were shown the same two examples of the program being used, now with the bug fixed, and were asked to judge if the program was working properly (which it was because the programs discussed here all had one bug each). Subjects then went on to the next program.

This procedure continued for all 12 programs. Subjects were then debriefed on the purpose of the study, and usually an informal discussion of the experiment ensued.

5.2. Results and Discussion

Our primary goal was to observe what effects the forced bug-location strategies would have on subjects' performance during the various components of debugging (e.g., bug location and bug repair). The simplest prediction is that variables related to bug location should be affected by the different strategy conditions while those variables related to other components of debugging should be unaffected.

We were also interested in seeing if subjects would carry over any of the strategies they were trained on into the case where they were free to use any strategy, or combination of strategies, they wished. If the strategy conditions have any effect on free debugging, we would expect to observe the same effects in both the training (forced strategy) and transfer (free debugging) phases of the experiment.

Bug Location. There are three measures related to searching the program for the bug: (a) the time it takes to choose each line to check; (b) the number

Figure 11. **Transfer problems: sec to choose each line and number of lines searched.**

	Strategy Conditions		
	Working-Backwards	Hand-Simulation	Neutral
Seconds to choose	14.0	8.4	13.0
Number of lines	1.0	1.7	1.4
n	9	9	9

of lines inspected before finding the buggy line; and (c) most importantly, a trace of the subject's search through a program—which lines the subject inspected and in what order.

The results of the first two measures for the transfer problems are shown in Figure 11. The measures for the training problems are not shown because these measures are so intertwined with the way each interface was implemented that any results obtained would be suspect. For the transfer problems, neither measure shows a significant effect (time to choose: $F < 1$, *n.s.;* number of lines: $F(2, 24) = 1.17$, *n.s.*). Although not reliably different, the time it took subjects to choose each line is in the correct direction if subjects are using the same debugging strategy that they were trained with. Specifically, the hand-simulation condition chose lines slightly more quickly than either of the other two groups, which is what would be expected because choosing the next line in hand-simulation follows a simple rule: Choose the next line in the execution of the function (and in these functions, the next line is usually the next one serially). Clearly, however, a more sensitive measure of each subject's search of the programs is needed: a trace of the subjects' search behavior.

To analyze subjects' search through the various programs, we obtained a rating of how well a particular search (i.e., the identity and order of each line checked) reflects the two bug-location strategies introduced in the experiment. To perform this analysis, three models were defined. Each model predicts which lines (and in what order) subjects will check if the subjects are using a particular bug-location strategy. Thus, the models give us a way of objectively stating whether or not a particular search by a subject is consistent with a particular debugging strategy. What we then did was rate how all of the searches performed by subjects in a particular condition match each of the strategies. Each of the nine subjects in a condition searched four programs, thus there were 36 searches performed per condition, and our ratings are the percentage of these 36 searches that fit each of the models (a separate rating was calculated for each model). Before proceeding to the actual analysis, each model is discussed in more detail.

The three models reflect the debugging strategies subjects might be using to search the programs. The first two models were based directly on the two debugging interfaces: working-backwards and hand-simulation; a model for the third interface, the neutral condition, was not created because that interface did not force any particular debugging strategy. The third model was based indirectly on the hand-simulation strategy, and the motivation for this model is discussed next.

The model based on the working-backwards strategy, referred to as the *causal model* (to avoid confusion with the experimental condition), which predicts subject performance by defining groups of program lines and asserting that subjects will search the lines in one group before proceeding to a line in another group. The three groups are identical to the lines associated with the three hypotheses of what might be wrong with a program as shown in the top-level working-backwards menu (Figure 10). However, it should be noted that both the order in which the groups are checked and the order of lines checked within a group are not predicted by the model. Similarly, subjects may skip lines in a group (or skip a group altogether) and still be categorized as reflecting the model. The model only defines the groups of lines and predicts that groups will not overlap during a subject's search of the program.

The model based directly on the hand-simulation interface, the *program-order* model, is different from the previous model in that it predicts the exact order in which subjects will search the program. Specifically, to be categorized as fitting this model, a subject would search the lines of the program in the same order the computer would execute those lines, given the sample function call that subjects are shown (these function calls always resulted in the correct program looping once). Thus, the model predicts the following sequence of lines: counter-initialization, result-initialization, test, counter-update, result-update, test, and action (Figure 9). As with the previous model, subjects may skip lines and still fit the model as long as the lines they search preserve the just-mentioned sequence.

The final model, based indirectly on the hand-simulation interface, is referred to as the *serial order* model. Similar to the program-order model, this model defines a sequence of lines subjects will check and asserts that subjects will check the lines in the same order as the sequence, although lines may be skipped. This sequence reflects the serial order of the lines on the screen: counter-initialization, result-initialization, test, action, counter-update, and result-update (Figure 9). Thus, the model differs from the program-order model in the relation between the action of the condition and the variable updates. In the serial-order model, the action must be checked before either of the variable update lines; in the program-order model, the action must be the last line checked. The serial-order model was created based on the intuition that some subjects might interpret the hand-simulation interface incorrectly,

Figure 12. **Percentage of searches fitting each model.**

Training Problems			
	Strategy Conditions		
Models	Working-Backwards	Hand-Simulation	Neutral
Causal	82%	0%	40%
Program-order	9%	100%	33%
Serial-order	9%	50%	53%
Transfer Problems			
	Strategy Conditions		
Models	Working-Backwards	Hand-Simulation	Neutral
Causal	60%	10%	16%
Serial-order	0%	40%	20%
Program-order	20%	20%	16%

thinking that the interface was forcing them to search the program serially, instead of searching in the order the computer would execute the program.

For the analysis, we wanted to obtain ratings of how well program-searching in each condition reflects each of the models—nine ratings overall. We calculated separate ratings for each model because a particular search may be consistent with more than one model. To get a rating of the match between a particular model and the searching done in a particular condition, we categorized each of the 36 searches done by subjects in the condition (9 Subjects × 4 Programs) as either fitting the model, not fitting the model, or ambiguous (i.e., consisting of less than four lines searched). The rating we use is the percentage of searches that fit the model out of the total number of unambiguous searches. Thus, each of the nine ratings may range from 0% to 100%.

The fact that some searches may be considered consistent with more than one model is somewhat problematic; so to reduce the amount of overlap, searches consisting of less than four lines were excluded from the analysis (i.e., categorized as ambiguous). In general, the lines that subjects looked at when they only checked three lines or less could be categorized as fitting all of the models. Thus, if we had included these searches, the percentages shown next would have all been much higher, and any effects would have been obscured. In addition, the removal of these small searches increases our chances of categorizing the search correctly: The more lines checked during a search, the less likely we are to say incorrectly that the search fits a particular model.

The straight-forward prediction is that the subjects trained on a particular strategy should act consistently with the model derived from that strategy;

their behavior should fit the appropriate model. Thus, subjects in the working-backwards condition should be more consistent with the causal model than they are with program-order or serial models. In contrast, subjects in the hand-simulation condition should be more consistent with the program-order or serial models. As shown at the top of Figure 12, this prediction holds when subjects are being forced to use a particular debugging strategy. In particular, a convincing effect appears if we focus on the two strategy conditions and the first two models. The working-backwards subjects are quite consistent with the causal model (82% of the debugging done by subjects in this condition fits the causal model), but are not as consistent with the program-order model (9%). The reverse is true for the hand-simulation subjects, none of whom act consistently with the causal model and all of whom are consistent with the program-order model. These are the results we would expect because each model was based on the interface used in that condition: The hand-simulation interface completely forces the subjects to be consistent with the program-order model, whereas the working-backwards interface only partially forces the subjects to be consistent with the causal model.

The results for the transfer problems are more interesting (Figure 12, bottom). When subjects were able to search the program in any order they wished, they still behaved (for the most part) as if they were using the debugging strategy that they were trained with during the first part of the experiment. This result is easiest to see in the case of the working-backwards subjects who were quite consistent with the causal model again, but did not act consistently with either of the "order" models. The hand-simulation subjects did not behave as the program-order model would predict, although they did act consistently with the serial-order model. These subjects did not seem to pick up on what we had been intending to train—that they should search the program by executing each line of code in the same order the computer would—but instead went to a straight serial-order search. In any event, these subjects clearly did not fit the causal model as well as they fit the serial-order model, as was predicted.

For the neutral subjects, in both the training and transfer problems, there does not seem to be a clear preference for a particular strategy. Each search is about equally likely to match any of the strategies, although there does seem to be a slight preference for the serial-order strategy, which intuitively seems to be the easiest strategy to use, if not the most efficient one.

To sum so far, for the two groups who were explicitly being trained on bug-location strategies, the analysis of the subjects' search showed that subjects continued to search using the strategy they were trained with, even when the interface no longer forced them into a particular strategy. Thus, if debugging strategies affect other components of debugging, we would expect to see any effects appear in the transfer programs as well as in the training programs.

Figure 13. **Judging and recoding times in sec. Numbers in parentheses are mean errors per subject.**

	Judging		
	Strategy Conditions		
	Working-Backwards	Hand-Simulation	Neutral
Training programs	7.8	4.8	3.6
	(4)	(6)	(10)
Transfer programs	19.0	15.0	11.0
	(4)	(11)	(15)
	Recoding		
	Strategy Condition		
	Working-Backwards	Hand-Simulation	Neutral
Training programs	38.0	43.0	25.0
	(6)	(6)	(11)
Transfer programs	51.0	53.0	41.0
	(16)	(15)	(15)
n	9	9	9

Other Components of Debugging. The two variables considered in this section are: (a) how well subjects were able to judge that a buggy line (once selected by the bug-location strategy) was indeed incorrect, and (b) how well subjects were able to correct that buggy line of code. As stated previously, we predict that the particular debugging strategy used (either forced by the interface or one a subject might be using in the transfer programs) will have no effect on those components of debugging unrelated to line selection.

The top of Figure 13 shows the time and error data from subjects judging the correctness of the buggy line. In both the training and transfer programs, there is a main effect for strategy, although the effect is only marginally significant for the second set of program—Training: $F(2, 24) = 4.77$, $p < .02$; Transfer: $F(2, 24) = 2.73$, $p < .09$. These effects are tempered, however, by the opposite (nonsignificant) trend in the error data. Thus, subjects using different strategies show different speed-accuracy tendencies. The neutral subjects are fastest, but also most prone to making an error, whereas the working-backwards subjects are slowest, but most accurate. It is interesting to note that the same speed-accuracy trend occurs both when the subjects are forced to use a strategy and when all subjects are transferred to the neutral strategy. This result lends further support to the proposal that subjects are carrying over the debugging strategies they were trained with into the problems in which they are free to use any strategy.

Finally, Figure 13 (bottom) shows the time and error data from subjects entering the code to fix a buggy line. There is again a slight, but not significant, tendency for the neutral subjects to be faster and make more errors. Except for this tendency, there are no significant effects of strategy either when subjects are forced to use a strategy or when they are free to use any strategy they wish.

Summary. The results suggest that the process of locating a bug is fairly independent of other processes involved in debugging. The interfaces directly manipulated how the subjects would search the programs, and these interfaces had an effect (not surprisingly) on the order and identity of the lines in the program subjects inspected. In addition, when subjects were not forced by the interface to use a particular search strategy, they still behaved as if they were using the strategy they were trained on. On the other hand, the different bug-location strategies had little effect on other aspects of debugging. That is, the different strategies did not provide subjects with information that could help or hinder their later performance. The only effect was a change in the tradeoff between speed and accuracy in judging whether a line of code is correct. There was no apparent effect in correcting a line of code.

In their task analysis of troubleshooting, Morris and Rouse (1985) claimed that the ability to use a strategy to locate errors is distinct and separate from the abilities to perform tests and repair components. This claim is supported by the results of Experiment 4. We found no substantial interaction between strategy use, error judgment (component testing), or error correction (component repair).

6. CONCLUSIONS

Recall that when debugging a program (or troubleshooting a system), a person may perform four distinct actions: (a) understanding the system being worked on, (b) testing that system, (c) locating the erroneous component of the system, and (d) repairing that component.

In order to debug a program, a person must have some knowledge about the program. The type of information a person has about a program clearly should depend on the method used to gain the knowledge. Experiment 3 suggested two ways in which information about a program might be collected: (a) by planning and coding the program; and (b) through an explicit comprehension process, if the program was written by someone else. Understanding is an important debugging stage because it is the resulting knowledge that a person uses to locate the bugs in the program. Depending on what a person knows about the program, different debugging strategies might be used. For example, subjects debugging their own programs demonstrated the knowledge needed to use the program's output to locate a bug. In contrast,

subjects debugging another's program opted for the strategies that use the actual, written code to locate bugs.

One explanation for this strategy difference might be the different type of information contained in authors' and others' mental representations of the programs. For instance, when writing their own program, a person has knowledge of their intentions (the function of each part of the code) and of the way the algorithm works (Kant & Newell, 1984). However, for people comprehending another's program, Pennington (1987) showed that expert programmers initially represent the program in therms of its control flow, which is how the program is executed rather than the reasoning behind the execution. It may be that, in order to effectively use a backward-reasoning strategy, knowledge of why an algorithm was implemented in a particular way is necessary. In contrast, having primarily knowledge of how a program is executed may lead to a forward-reasoning strategy. Therefore, the reason we see strategy use changing depending on authorship might be because programmers use the debugging strategy that reflects their knowledge of the program.

After understanding the program and if bugs were not found during this time, the program is usually tested. The information gained from these tests could be simply that the program doesn't work or, more specifically, how the program doesn't work. For our subjects, however, the decisions about how to test the programs were simple to make because of two aspects of the debugging situation: (a) the programs were short (on the order of 10 lines) and not complex in terms of the different correct behavior possible, and (b) the problem descriptions for the programs usually contained good examples of test cases. With longer and more complex programs (or systems of programs), this debugging stage might become more challenging as the obvious tests might not reveal the errors in such programs.

Once it has been determined that a program doesn't work, the bug causing the incorrect behavior must be found. This stage of debugging, location, is really the focus of current research. In the exploratory work (Experiment 2), it was revealed that there were several strategies that subjects used to find errors. These strategies have also been observed in subjects debugging in PASCAL (Gugerty & Olson, 1986; Jeffries, 1982), LOGO (Gugerty & Olson, 1986; Klahr & Carver, 1988), as well as in subjects troubleshooting electronic devices (Morris & Rouse, 1985; Rassmusen, & Jensen, 1974). As stated previously, the strategies that subjects used seem to depend on what information the subjects have about the program. For example, subjects unfamiliar with a program were shown to prefer a forward-reasoning strategy as compared to the authors of the programs who were more likely to use strategies that made use of the output of the program (Experiment 3).

The fact that these strategies have been observed in subjects performing other, more complex, troubleshooting tasks suggests that our other results

may extend to these tasks, even though we investigated subjects working only in LISP. Not only do we replicate previous debugging work done in other languages and in electronics troubleshooting, but the type of programs subjects were working with (simple iterative routines) are quite similar to corresponding routines in more procedural languages such as PASCAL. The findings should generalize to other languages because subjects were primarily using those features of LISP that are shared with many other programming languages.

Although the understanding stage interacts with the bug-location strategies used, it was shown in Experiment 4 that the strategies do not provide any knowledge that would affect either the judgement of the correctness of a line of code or the recoding of a buggy line. The only effect found was that subjects using a backward-reasoning strategy were more careful (i.e., took longer and made less errors) when judging lines compared with subjects using a forward-reasoning strategy or any strategy they wished.

Finally, after finding the buggy line(s) of code, the error must be repaired. As with the testing stage, the repair stage did not play a major role in the debugging observed. Usually it was the location of the bug that caused the subjects difficulty rather than the correction of the bug. The ease with which errors were fixed is likely due to the nature of the errors involved. Our subjects usually generated (and, therefore, debugged) unsystematic and unstable errors; rarely did errors stem from misconceptions about LISP (Experiment 1). Thus, once an error was spotted, it was easily fixed (Experiment 2). In addition, all of the bugs generated by subjects were local; no global, algorithm-level bugs were produced. Based on our observations, the repair stage of debugging seems to consist of a local recoding of the buggy line. That is, subjects merely regenerate the line of code based on their original intentions in writing that line. Bug repair might become more difficult when global errors are generated. If the bugs being sought are algorithmic in nature, it seems clear that subjects might be able to detect the incorrect lines, but be unable to see an immediate solution.

Acknowledgments. We thank Sharon Carver, Wayne Gray, Claudius Kessler, David Klahr, Brian Reiser, and Robert Ward for their comments on various portions of this work.

Support. This material is based on work supported under a National Science Foundation Graduate Fellowship to Irvin R. Katz and Contract MDA903-85-K-0343 from the Army Research Institute and the Air Force Human Resources Laboratory to John R. Anderson.

REFERENCES

Anderson, J. R., Corbett, A. T., & Reiser, B. J. (1987). *Essential LISP.* Reading, MA: Addison-Wesley.

Anderson, J. R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction, 1,* 107–131.

Anderson, J. R., & Reiser, B. J. (1985, April). The LISP tutor. *BYTE,* pp. 159–175.

Brown, J. S., & Van Lehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4,* 379–426.

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

Gugerty, L., & Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In E. Soloway & S. Iyengar (Ed.), *Empirical studies of programmers* (pp. 13–27). Norwood, NJ: Ablex.

Jeffries, R. (1981, November). *Computer program debugging by experts.* Paper presented at the meeting of the Psychonomics Society, Philadelphia, PA.

Jeffries, R. (1982, March). *A comparison of the debugging behavior of expert and novice programmers.* Paper presented at the annual meeting of the American Educational Research Association, New York.

Kant, E., & Newell, A.(1984). Problem solving techniques for the design of algorithms. *Information Processing & Management, 20*(1–2), 97–118.

Katz, I. R., & Anderson, J. R. (1986, June). *An exploratory study of novice programmers' bugs and debugging behavior.* Poster presented at the 1st annual Empirical Studies of Programmers Workshop, Washington, DC.

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in LISP. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 198–212). Norwood, NJ: Ablex.

Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology, 20*(3), 362–404.

Morris, N. M., & Rouse, W. B. (1985). Review and evaluation of empirical research in troubleshooting. *Human Factors, 27*(5), 503–530.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*(3), 295–341.

Rassmusen, J., & Jensen, A. (1974). Mental procedures in real-life tasks: A case study of electronic trouble shooting. *Ergonomics, 17*(3), 293–307.

Reiser, B. J., Anderson, J. R., & Farrell, R. G. (1985, August). *Dynamic student modelling in an intelligent tutor for LISP programming.* Paper presented at the meeting of the International Journal Conference on Artificial Intelligence, Los Angeles, CA.

Rouse, W. B. (1979). A model of human decision making in fault diagnosis tasks that include feedback and redundancy. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-9,* 237–241.

Siegler, R. S. (1987). The perils of averaging data over strategies: An example from children's addition. *Journal of Experimental Psychology: General, 116*(3), 250–264.

Spohrer, J. G., & Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. In E. Soloway & S. Iyengar (Ed.), *Empirical studies of programmers* (pp. 230–251). Norwood, NJ: Ablex.

Winston, P. H., & Horn, B. K. P. (1981). *LISP.* Cambridge, MA: Addison-Wesley.

HCI Editorial Record. First manuscript received December 8, 1987. Revision received July 15, 1988. Accepted by Ruven Brooks. —*Editor*

APPENDIX: SUPPLEMENTARY FIGURES

These figures are for those readers interested in the particular stimulus programs used in the experiments. Also, Figures 21, 22, and 23 give examples of all of the bugs mentioned by descriptive name in Figure 3.

Figure 14. **Sample basic LISP function problems and solutions.**

Write a function call that will take the list *(c d e)* and returns the first element, which is *c*.

Solution:

```
(car '(c d e))
```

Write a functional call that takes the lists *(3 2)* and *(b c)* and produces the complex list *((3 2) (b c))*.

Solution:

```
(list '(3 2) '(b c))
```

Figure 15. **Sample list-iteration problem and solutions.**

Define a function called *list-sum*. Given a list of numbers, *list-sum* returns the sum of those numbers. For example,

(list-sum '(5 10 -4 27)) returns *38*.
(list-sum '( )) returns *0*.

Solution:

```
(defun list-sum (lis)
        (let ((sum 0))
            (loop
               (cond ((nul lis) (return sum)))
               (setq sum (+ (car lis) sum))
               (setq lis (cdr lis)))))
```

Figure 16. **Sample reading/printing problem and solution.**

Write a function called ***read-check*** that accepts one argument, which must be a list. The function should type the prompt ***(type an expression)*** and read an input from the user. The function should return ***t*** if the input is a member of the argument list, and return ***nil*** otherwise. For example,

```
=>(read-check '(a b c d))
(type an expression)b
t
```

Solution:

```
(defun read-check (lis)
       (print '(type an expression))
       (and (member (read) lis) t))
```

Figure 17. **Sample input-controlled iteration problem and solution.**

Write a function called ***read-square*** that does not accept any arguments. The function should contain a loop that prints the prompt ***Enter the next number:***, reads a number and prints the square of the number. The function should return the atom done when the user types something that is not a number. For example,

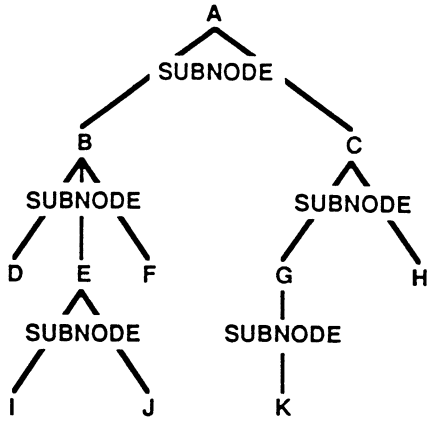
```
=>(read-square)
Enter the next number: 5
25
Enter the next number: -11
121
Enter the next number: x
done
```

Solution:

```
(defun read-square ( )
       (let (inp)
         (loop
            (print "Enter the next number:")
            (setq inp (read))
            (cond ((not (numberp inp)) (return 'done)))
            (print (* inp inp)))))
```

Figure 18. **Problem description and ideal solution for BREADTH. The numbers appearing to the left of the function below are not part of the actual function, but merely a way of referring to the lines of the program. The same is true for the numbers shown in PRE-REQUISITE and BEFORE.**

Write a function called breadth which takes one argument: a root node in a tree. The function should perform a breadth-first search of the tree and return the first node that has the property success with the value t. The function should return nil if no such node is found. Nodes will be connected to nodes below them by a label called subnode. Below we have illustrated a simple network.




```
1  (defun breadth (root)
2     (prog (queue expansion)
3           (setq queue (list root))
4      loop (cond
5                ((null queue) (return nil))
6                ((get (car queue) 'success) (return (car queue))))
7           (seq expansion (expand (car queue)))
8           (setq queue(cdr queue))
9           (setq queue (append queue expansion))
10          (go loop)))
   (defun expand (course) (get course 'subnode))
```

Figure 19. **Problem description and ideal solution for PRE-REQUISITE.**

Write a function pre-requisite which takes two arguments: a course name (an atom and a list of courses which are the prerequisites of the first argument). The function should add each of the prerequisites onto the property list of the given course under the property has-pre-req and should add the first argument to each of the prerequisites' property lists under the property pre-req-for. Example:

```
=> (pre-requisite 'Automata '(Systems CompuFund Complang))
nil
=> (pre-requisite 'CompuFund '(MathII Writing))
nil
=> (pre-requisite 'Systems '(CompFund DataStruct))
nil
=> (pre-requisite 'MathII '(MathI))
nil
```

```
1   (defun pre-requisite (course prereq-lis)
2      (prog ( )
3            (putprop course
4                     (append prereq-lis (get course 'has-pre-req))
5                     'has-pre-req)
6      loop (cond
7               ((null prereq-lis) (return nil))
8               (t (putprop (car prereq-lis)
9                           (cons course (get (car prereq-lis) 'pre-req-for)
10                          'pre-req-for)))
11           (setq prereq-lis (cdr prereq-lis))
12           (go loop)))
```

Figure 20. **Problem description and ideal solution for BEFORE.**

Write a function called before which, when given a course name, returns a list consisting of that course's prerequisites, each of the prerequisites' prerequisites, and so on. The function should return a list of unique course names (i.e., there should be no duplicate course names in the returned list). Also, the order of the courses in the returned list is not important. Unlike the previous problem, this involves visiting all the nodes in the tree rather than looking for a particular node. Also, we want to return a list of all nodes visited rather than a particular node. We will build this list up as we expand various nodes in the tree.

```
⇒  (before 'Automata)
   (Systems CompFund CompLang MathII Writing MathI DataStruct)
⇒  (before 'DataStruct)
   nil
⇒  (before 'CompFund)
   (MathII Writing MathI)
```

```
1   (defun before (course)
2      (prog (queue expansion visited)
3            (setq queue (expand course))
4            (setq visited ( ) )
5      loop  (cond ((null queue) (return visited)))
6            (cond
7               ((member (car queue) visited)
8                (setq queue (cdr queue)))
9               (t  (setq visited (cons (car queue) visited))
10                  (setq expansion (expand (car queue)))
11                  (setq queue (cdr queue))
12                  (setq queue (append queue expansion))))
13           (go loop)))
    (defun expand (course) (get course 'has-pre-req))
```

Figure 21. **Examples of BREADTH bugs.**

Bug Name	Line Number and Example Code
Goal Errors	
argument for function call:	
setq	9 (append queue expansion)
expand	7 (setq expansion(car queue))
list	3 (setq queue root)
missing append step	<line 9 omitted>
missing exit test	<line 5 omitted>
missing argument to equal	6 ((equal (get (car queue) 'success)) (return (car queue)))
Intrusion Errors	
argument for function call:	
return	5 ((null queue) nil)
Misrepresentation Errors	
t returned	6 ((get (car queue) 'success) (return t))
example for variable	3 (setq queue (list 'a))
depth-first search	9 (setq queue (append expansion queue))
parameter for (car queue)	6 ((get root 'success) (return root))
Misconceptions	
missing quote	6 ((get (car queue) success) (return (car queue)))
extra quote	6 ((get '(car queue) 'success) (return (car queue)))
Syntactic Errors	
missing)	6 ((get (car queue) 'success) (return (car queue)))
(following lines become clauses in the cond)	7 (setq expansion (expand (car queue))) 8 (setq queue . . .) . . .
extra)	6 ((equal (get (car queue) 'success)) t) (return t))
(return becomes a clause in the cond)	

Figure 22. **Examples of PRE-REQUISITE bugs.**

Bug Name	Line Number and Example Code
Goal Errors	
argument for function call:	
setq	11 (cdr prerq-lis)
prog	<line 2 omitted>
missing (go loop)	<line 12 omitted>
missing loop variable update	<line 11 omitted>
missing argument to putprop	<line 9 omitted>
missing has-pre-req putprop	<lines 3, 4, and 5 omitted>
Intrusion Errors	
argument for function call:	
return	7 ((null prereq-lis) nil))
Misrepresentation Errors	
no loop	<line 2 omitted>
	6 (cond
	<lines 11 and 12 omitted>
'loop' for variable name	7 ((null loop) (return nil))
pre-req-for for has-pre-req	4 (append prereq-lis
	(get course 'pre-req-for))
	5 'pre-req-for)
Misconceptions	
arguments in wrong order:	
putprop	9 'pre-req-for
(always occured with	10 course)
not adding pre-req-for)	
cons for append	4 (cons prereq-lis
	(get 'course 'has-pre-req))
extra quote	4 (append pre-req-lis
	(get 'course 'has-pre-req))
putprop has-pre-req separately	<lines 3, 4, and 5 omitted>
	10a (putprop course (car prereq-list)
	'pre-req-for)
didn't add pre-req-for	9 course
Syntactic Errors	
missing)	10 'pre-req-for))
(lines 11 and 12 become	11 (setq prereq-lis (cdr prereq-lis)
clauses in the cond)	12 (go loop))))
missing (	7 (null prerq-list) nil)
(occurred with argument	
for function call: return)	
extra (	8 ((t (putprop (car prereq-lis)
missing ( )	7 ((null prereq-list) return nil)

Figure 23. **Examples of BEFORE bugs.**

Bug Name	Line Number and Example Code
Goal Errors	
argument for function call:	
setq	9 (t (cons (car queue) visited)
car	9 (t (setq visited (cons queue) visited)
expand	3 (setq queue course)
cons	9 (t (setq visited (car queue))
missing label	5 (cond ((null queue) (return visited)))
missing result update	9 (t
missing (go loop)	<line 13 omitted>
missing member test	<lines 7 and 8 omitted>
missing queue update	<lines 11 and 12 omitted>
missing cdr step	<line 11 omitted>
missing append step	<line 12 omitted>
missing argument to member	7 ((member (car queue))
Intrusion Errors	
list for expand	3 (setq queue (list course))
nil returned	5 loop (cond ((null queue) (return nil)))
(go loop) inside cond	12 (setq queue (append queue expansion))
	13 (go loop)))))
Misrepresentation Errors	
member test/action mismatch	7 ((member (car queue) visited)
	8 (setq visited (cons (car queue) visited))))
	9 (setq expansion (expand (car queue)))
	10 (setq queue . . .)
	. . .
result initialized as queue	4 (setq visited (expand course))
result updated as queue	9 (t (setq visited (append (cdr queue) (expand (car queue))))
extra list	4 (setq visited (list ( )))
Misconceptions	
cons arguments in wrong order	9 (t setq visited (cons visited (car queue)))
cons for append	12 (setq queue (cons queue expansion))
Syntactic Errors	
missing)	12 (setq queue (append queue expansion)))
(go loop becomes a cond clause)	13 (go loop))))
extra ( )	3 (setq queue (expand (course)))

Figure 24. **LISP functions and problem descriptions of Experiment 3.**

Define a function called **factorial** that has one parameter. This function computes the factorial of its parameter. The factorial of a number is the product of all the integers between 1 and the number multiplied together. So, **(factorial 5)** = 1 × 2 × 3 × 4 × 5 **120**. This function should accept the parameter value 0 in addition to the positive integers. (The factorial of 0 is defined as 1.)

```
(defun factorial (num)
  (prog (count product)
        (setq count 0)
        (setq product 1)
   loop (cond ((equal count num) (return product)))
        (setq count (1+ count))
        (setq product (times count product))
        (go loop)))
```

Define a function called **create-list** that has one parameter. This function returns a list of all integers between 1 and the value of the parameter, in ascending order. So, if the value of the parameter is 8, create-list should return **(1 2 3 4 5 6 7 8)**. You should count DOWN in this loop, rather than up, so the integers are generated in DESCENDING order. That way you can generate the list conveniently (without having to flip it.)

```
(defun create-list (num)
  (prog (count result)
        (setq count num)
        (setq result (list num))
  loop  (cond ((equal count 1) (return result)))
        (setq count (1- count))
        (setq result (cons count result))
        (go loop)))
```

Define a function called **list-skip** that has two parameters. This function returns a list of EVERY OTHER number between the 1st and 2nd parameter. The list always includes the first parameter and includes every other number that is less than or equal to the second parameter. For example, **(list-skip 2 8)** returns **(2 4 6 8)**, and **(list-skip 2 9)** returns **(2 4 6 8)**. You should count UP in this function (so you can handle the bounds conveniently). You should use the lesson 2 function **snoc**, that has two arguments. **snoc** inserts its first argument at the END of its second argument (which must be a list).

```
(defun list-skip (low high)
  (prog (count result)
        (setq count low)
        (setq result (list low)
  loop  (cond ((greaterp count (difference high 2)) (return low)))
        (setq count (plus count 2))
        (setq result (snoc count result))
        (go loop)))
```

Figure 24. **(Continued)**

Define a function called **num-sum** that has one parameter. Num-sum inputs a series of numbers and returns the sum of the numbers. The parameter indicates how many numbers to input.

```
(defun num-sum (num)
  (prog (count sum)
        (setq count 0)
        (setq sum 0)
   loop (cond ((equal count num) (return sum)))
        (setq count (1+ count))
        (setq sum (plus (read) sum))
        (go loop)))
```

Define a function called **next-prime** that has one parameter. This function returns the first prime number that is greater than or equal to the value of the parameter. For example, **(next-prime 24) = 29.** A predicate **primep** has been defined for you to use in this problem. It takes one argument and returns **t** if the argument is prime and **nil** otherwise. NOTE: When you test **next-prime** in the LISP window, use an argument smaller than 1369. The predicate **primep** does not work correctly with larger arguments.

```
(defun next-prime (num)
  (prog (count)
        (setq count num)
  loop  (cond ((primep count) (return count)))
        (setq count (1+ count))
        (go loop)))
```

Define a function called **next-prime-bound** that has two parameters. This function returns the first prime number greater than or equal to the first parameter but not greater than the second parameter. If there is no prime number within that range, the function returns **no-prime.** So, **(next-prime-bound 54 59)** returns **59** and **(next-prime-bound 62 66)** returns **no-prime.** You can use the predicate **primep** that takes one argument and returns **t** if it is prime and *nil* otherwise.

```
(defun next-prime-bound (start bound)
  (prog (count)
        (setq count start)
  loop  (cond ((primep count) (return count))
              ((equal count bound) (return 'no-prime)))
        (setq count (1+ count))
        (go loop)))
```