A MODEL OF NOVICE DEBUGGING IN LISP[1]

CLAUDIUS M. KESSLER
JOHN R. ANDERSON
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

ABSTRACT

This paper reports an investigation of novice programmers trying to debug one-line LISP functions. We present a model of debugging based on protocol data and introduce a production system simulation of the ideal novice debugger. We conclude with a discussion of the applicability of such a model to the teaching of programming in LISP.

INTRODUCTION

A large part of the professional activity of a computer programmer consists of debugging programs. In programming courses, however, debugging is hardly ever treated as a skill that needs to be taught. Although it is generally acknowledged that bugs occur frequently and slow down the programming of novices as well as experts, the novice must acquire the skill to correct bugs on his own. There is some help available in the form of debugging tools that exist in most programming languages. Frequently, however, these tools are based on an advanced understanding of the language, and novices do not even know about their existence. In many programming courses, structured programming is explicitly taught with the implication that it will make debugging of programs easier, but there is no actual instruction in debugging itself.

Debugging has been the target of previous research in cognitive science. However, the outcome of this research usually is geared towards developing programming tools for debugging. Rich & Waters (1), for example, propose a programmer's apprentice that, among other things, helps to debug programs. They represent the knowledge about programs as programming plans. Debugging consists of modifying plans so they fit a given programming situation. These plans can then be translated into code by the apprentice. The PROUST model by Johnson & Soloway (2) examines Pascal programs and discovers bugs if they violate certain programming structures such as looping constructs. In this way it develops a model of the programmer's intentions that allows it to pinpoint errors in the code. The basic knowledge of PROUST rests in programming plans similar to those discussed by Rich & Waters (1).

---

[1] This research has been supported by Contract MDA903-85-K-0343 from the Army Research Institute.

There have been a number of psychological studies looking at novice-expert differences in debugging (3, 4, 5, 6) and subjects' ability to deal with different kinds of bugs (7, 8, 9, 10, 11). However, to date, the only detailed process model of debugging is that proposed by Carver & Klahr (12). They developed an ideal production system model for debugging LOGO programs. The model is ideal in the sense that it is not based on data, but on a task analysis. It is supposed to capture the processes that go into debugging as done by an experienced, but not yet expert programmer. Carver & Klahr distinguish four phases in the debugging process: program evaluation, bug identification, bug location, and bug correction. The program evaluation phase compares the program output and the expected correct output. The bug identification phase creates one or more discrepancy descriptions which help to narrow the search for the program statement containing the bug. The bug location process actually searches the code for the buggy statement. Its effectiveness depends largely on the specificity of the descriptions created in the previous phase. In the bug correction phase, the buggy code gets replaced by new code, and a new evaluation is initiated. The productions in the model draw on four sources of information: the correct solution, the program output, the code, and knowledge of the programming language.

Studies on novice programmers tend to use a rather lenient definition of novice. Most so-called novices have received about a semester's course worth of instruction in the language they are tested in. True novices, people who have received little or no instruction in the language might show the difficulties inherent in debugging more clearly than these more experienced subjects. Furthermore, the effectiveness of instructional manipulations such as the timing of instruction, and teaching subskills can be more readily investigated if novices are tested in the process of acquiring a language.

In our study, we use true LISP novices, people with no previous instruction in LISP, to study the process of debugging. We want to present a theoretical outline of debugging based on data from protocol studies on the behavior of these novices. To make this theory more tractable, we formulated a production system model of debugging based on these results. The data consist of eight protocols of novice programmers who were taught the first two lessons of a LISP course sequence, which covered the basic LISP functions and the writing of simple function definitions in LISP. Instead of letting our subjects write functions, they were given buggy functions and they had to find and correct the errors.

METHOD

Materials

The buggy and the correct versions of the LISP functions we used in this experiment are listed in the Appendix. The functions were selected from lesson 2 of the LISP tutor (13 (chapter 2), 14). They were designed to be simple, but increasingly more challenging exercises in writing LISP function definitions. In each of the functions, one and only one bug was introduced. The bugs were taken from protocols of students that went through the LISP tutor. All of the bugs seemed to present difficulties to students having their first encounter with the material covered in lesson 2. Six bugs were used, each appearing in two different LISP functions, resulting in a total of 12 buggy functions. Six more functions were taken from lesson 2 in their correct form. They served as distractor items.

Two of the six bugs were syntactic bugs. One parenthesized a variable, the other quoted a variable. Four of the bugs were of a semantic nature. In one case, functions that should have been embedded were not. The three remaining bugs dealt with list combination problems. In one case, the 'list'-command was omitted, in the second, 'append' was used instead of 'list', and finally, a

'reverse'-command was omitted in a list manipulation that included several command steps.

While the first five bugs were completely isomorphic for both functions they appeared in, the 'reverse'-bug was an exception. In the SNOC-problem (see the Appendix), the 'reverse' that was deleted was the outermost 'reverse', while in the ROTATER-problem (see the Appendix), the innermost reverse was deleted. Table 1 demonstrates the output obtained from the six bug types.

Table 1
The Output of the Bugged Functions

```
=>(first '(a b c))
Error: eval: Undefined function  x

=>(replace 'rings '(ties hats pants))
(x hats pants)

=>(ftoc 32)
17.77777777777778

=>(sqr 2)
4

=>(back '(a b c))
((c b a) (c b a))

=>(snoc 'd '(a b c))
(d c b a)

=>(rotater '(a b c d))
(d d c b)
```

Design and Procedure

The subjects in our study were eight Carnegie-Mellon University undergraduates who participated for course credit or pay. They did not know any LISP and their previous college level programming experience was limited to at most one introductory Pascal class.

Two sequences of functions were made up for each subject. Each sequence consisted of six different bugs in a fixed order. The order in which the bugs were presented was the same as shown in the Appendix. The functions were assigned randomly to the first or second sequence, with the exception of SNOC and ROTATER. SNOC always was the last function in the first sequence, and ROTATER was the last function in the second sequence. This was done because our experience from tutoring lesson 2 indicated that the ROTATER problem was more difficult than any of the other problems in lesson 2. Furthermore, it seemed to us that the bug in ROTATER was harder to discover than the corresponding bug in SNOC.

The experiment was done over two days. On the first day, subjects went through an instruction booklet covering the material of lesson 1 of the LISP tutor. Basically, this included an introduction to LISP functions and the use of variables. They had to do exercises on a terminal using a LISP environment with

some added user-friendly features. Each subject was run individually.

On the second day, subjects went through an instruction booklet that explained how to write function definitions in LISP. The experimenter then gave an introduction to the verbal protocol technique, including an example of going through the process of writing a function call. Subjects were then asked to solve the problems in the instruction booklet on the terminal, using the same LISP environment as on day 1. Each subject received two sequences of nine functions, containing six buggy and three correct functions. While all of the bugs were presented in the same order in each sequence, the correct functions that were used as distractors were distributed semi-randomly, with the restrictions that no two correct problems could follow one another, and that they were neither in the first nor in the last position of a sequence.

The buggy functions were loaded into the LISP environment. Thus, the subjects were able to call the function they were working on. Before calling the function, however, they had to predict if the code was correct or not. The subjects were requested to talk aloud about their problem solving during the debugging process. The experimenter made sure that the subject did not stay silent for a prolonged period of time by prompting them to talk. The subjects were told that they should feel free to ask the experimenter for assistance if they found a problem unsolvable. Assistance was given in the form of different kinds of hints: if subjects followed a wrong path, they were told to back up, otherwise they were given an explanation of a part of the code. This explanation was the same for all subjects, and was constructed as to not give away the solution directly. This process was repeated until the subjects arrived at the correct solution. One of the verbal protocols was lost due to equipment failure.

In addition to the protocols, we obtained data from the terminal interactions which were recorded and time stamped. Each terminal interaction was either a completed command to run a function with some arguments or a function definition typed in by the subject.

RESULTS AND DISCUSSION

The average solution times and the number of terminal interactions for our eight subjects are shown in Table 2. These measures seemed more informative than the number of hints given per problem, since, due to the simplicity of the functions, there were hardly more than two hints given for each function. The time and terminal interaction data were submitted to analyses of variance with sequence and tasks as within-subject factors. There was a main effect of sequence for both times and terminal interactions (times: $F(1, 7) = 39.73$, $p < .001$; terminal interactions: $F(1, 7) = 8.28$, $p < .05$). Subjects took longer to work through the initial sequence of six tasks and typed in more statements overall. While there were no further significant results for the terminal interaction data, the solution times showed a main effect of tasks ($F(5, 35) = 13.16$, $p < .001$) and a significant interaction between sequence and tasks ($F(5, 35) = 9.34$, $p < .001$). However, if the first and last problems of the sequence are excluded from the Anova, the interaction between sequence and task goes away. Subjects speed up much more on the first problem, presumably reflecting the fact that they are adjusting to the task on the first problem. The last problem was the only problem that was not counterbalanced for the two sequences, with the more difficult problem always occuring in the second sequence. Individual t-tests for speedup between sequence 1 and 2 revealed a significant speed-up only for tasks 1, 2,and 4 (1: $t(7) = 5.49$, $p < .001$; 2: $t(7) = 3.31$, $p < .05$; 4: $t(7) = 3.02$, $p < .05$). Speed-up on tasks 3 and 5 falls just short of significance (3: $t(7) = 1.97$, $p < .09$; 5: $t(7) = 1.89$, $p < .1$). The speed-up for task 6 is quite weak statistically ($t(7) = .61$).

The main effect of tasks seems to be tied to task 1 in the first sequence and to the last task in the second sequence. An analysis of variance for

Table 2
Mean Completion Times and Terminal Interactions (in Parentheses)

Bug	First Pass	Second Pass
1 Parentheses	28:19 (7.62)	5:49 (3.75)
2 Quote	6:23 (3.50)	3:03 (2.87)
3 No Embedding	6:48 (4.87)	2:56 (2.75)
4 No Combiner	13:33 (5.00)	4:53 (5.00)
5 Wrong Combiner	8:59 (6.87)	4:42 (3.37)
6 Missing Reverse	10:39 (6.37)	9:26 (3.75)

Note: N = 8 for each cell. Times in minutes and seconds.

sequence 1, with the first task excluded, shows only a marginal effect of tasks ($F(4, 28) = 2.28$, $p < .09$). Similarly, an analysis of variance for sequence 2 with the last task (ROTATER) excluded, does not show a main effect of tasks at all ($F(4, 28) = 1.79$, $p < .16$).

The reason subjects took so long to complete the first problem was that they did not really understand how to write function definitions in LISP. They had extracted virtually nothing from reading the problem instructions. This finding ties in with other research on learning to program in LISP, e.g. (15). Anderson (16) found that subjects show over a 50% speed-up from the first function they code in lesson 1 of the LISP tutor to the second function. In our experiment, most of the time in the first problem was taken up by the experimenter giving a hands-on demonstration of a LISP function definition. Once subjects seemed to have grasped the LISP function definition form, their behavior in problem 1 followed pretty much the same course as in the other problems.

The ROTATER problem proved to be the most difficult in the experiment. As mentioned before, there are two factors contributing to this. First, the function is more complex than any other of the 12 problems, and second, the bug in ROTATER was harder to find than the bug in the related problem, SNOC. The process of debugging ROTATER will receive more consideration below.

The quantitative data give a fairly obvious picture of the subjects' performance. Subjects take an extraordinarily long time to do the very first problem, then spend about the same time on all the other problems in the first sequence. When the bugs are repeated, subjects speed up considerably, with the exception of the last problem, ROTATER which has a slightly different bug and which we had predicted would be more difficult. Since the terminal interaction data do not add any further information, they will not be considered here.

A THEORY OF DEBUGGING

In order to find an explanation for the pattern of results we obtained, and to get a better idea of the processes that go on in debugging, we considered protocol data from our subjects. The six bugs we used seemed to be sufficiently distinct to produce no transfer of debugging from one bug to the other. Yet, the protocols lead us to believe that most subjects used the same debugging strategy for all problems. We now want to characterize this general debugging strategy we could extract from the protocols.

The debugging process could essentially be broken down into four episodes.

These were code comprehension, bug detection, bug localization, and bug repair. The first two of these episodes were usually short, and there is not much evidence about the mental processes taking place in the protocols. The bug localization and bug repair episodes provide the bulk of our protocol data. We now turn to a discussion of the processes going on in the different episodes.

The Debugging Episodes

Code Comprehension. Most subjects started out by trying to understand what the code was doing. The reading of the code at this time was rather superficial. Subjects often gave wrong judgments when asked to predict if the code was correct before they were allowed to run it. In the best case, they came out of this comprehension process with correct hypotheses of the code's behavior; in the worst case, they skipped the code comprehension and just took a guess about the correctness of the function. In general, they formed wrong hypotheses about the code, either because they did not try hard enough to understand it, or because they did not have the necessary LISP knowledge available at this point.

Bug Detection. The next step subjects took was to run the code. This was an obvious step, which usually led the subjects to detect the error. Subjects made very few comments during this stage. Since bug detection includes the process of describing the difference between the desired and the obtained results, it is possible that our tasks were sufficiently simple that subjects did not need to go through an elaborate process in order to compare the correct and the buggy answer.

Bug Localization. This episode consisted in actually finding the piece of code that was responsible for the error. For most subjects, this was a difficult problem solving process. The difficulty of this episode was enhanced, of course, if subjects had done the previous episodes improperly or not all. However, subjects usually could locate the bug in the code after going through some iterations of creating and rejecting hypotheses.

Bug Repair. This phase proved to be difficult independent of what had been going on in the previous phases. Even if subjects had found the bug and knew the faulty part of the code, it proved to be a hard task to come up with the correct code. This was the phase where most hints were given. If subjects made a correction that turned out to be wrong, there was a danger that they would get lost when they tried to correct their own initial correction. In these cases, the experimenter intervened and put them back on the track of the original function. Thus, while we estimate that this phase took longest for all our subjects, the time spent in correcting an identified error might be underestimated by our experimental procedure.

Evaluation

The sketch of the debugging process outlined above is supported by six of the seven protocols we obtained. There were no debugging episodes in these protocols that could not be accommodated by this scheme. There were of course additional processes going on, that in our view were less central to the debugging process. For example, subjects often looked for LISP function definitions they could not retrieve in memory, or they did checks on the syntax of the code they were writing. Subjects also made meta-comments about how hard or easy certain problems were.

The debugging process as described so far is by no means the only way to debug, even at a beginner's level of proficiency. In fact, one of our subjects displayed an entirely different strategy. The first thing he did when reading the problem was to actually generate the code in his mind. He then compared his code with the code actually presented. To the extent that he generated the correct code, his strategy proved more efficient than the one outlined so far. This strategy would work only for simple problems such as ours, of course.

Before we look into some issues concerning specific bugs, we have to

address the question where the savings in time are coming from when going from the first to the second sequence. First we must note that sequence and bugs are confounded in our experiment, so that this result may be of limited validity. We believe, however, that we would have obtained some speed-up even with randomized repeated bug sequences, for an obvious reason: our subjects could remember at least some of the bugs. The protocols show some anecdotal evidence of bug recognition on the second pass. Since we did not explicitly give a recognition test, we do not know how frequently bugs actually were recognized. As the error data will show, bug repetition did not generally facilitate comprehension of the code. As it stands, we do not know if bug recognition is the only cause for the speed-up. However, the fact that we could not observe any speed-up within the two sequences favors the bug recognition explanation.

Bug Data

We conclude this section with a closer look at the bugs and the errors they triggered in our subjects. While we try to interpret the data in terms of our theory, it should be remembered that bug type was confounded with presentation sequence in our experiment. We therefore cannot exclude the possibility that the error distribution we obtained is at least in part due to the specific presentation sequence we used. Our main interest here is in the qualitative nature of the errors, and not so much in their distribution over tasks. Table 3 gives an overview of how the errors subjects made were distributed over the six bug types. The first column gives the comprehension errors, where subjects assumed that the function was working correctly, before they actually ran the code. The second column gives the errors in the bug location phase, and the third column gives the wrong repairs. Repair errors were only counted when a bug had been located correctly, and when a subject actually typed in the code as a correction for the bug. If an error had been made in the location phase, the

Table 3
Number of Errors for each Bug

Bug	First Pass			Second Pass			Overall		
	C	L	R	C	L	R	C	L	R
Parentheses	2	1	3	4	0	0	6	1	3
Quote	2	1	0	1	0	0	3	1	0
No Embedding	0	0	2	0	0	0	0	0	2
No Combiner	1	1	4	1	0	2	2	1	6
Wrong Combiner	1	2	3	1	0	2	2	2	5
Missing Reverse	0	2	3	1	1	1	1	3	4
Sum	6	7	15	8	1	5	14	8	20

Note: N = 7 for each cell. C = Comprehension, L = Location, R = Repair. See text for further explanations.

wrong repair that followed it was not counted as a repair error. The table gives the number of subjects that went wrong on a given problem in a given phase. Repeated errors within a phase were not counted. It should be noted that our subjects found the debugging process much harder than this error measure indicates. Subjects often made repeated errors on a problem, and considered many erroneous alternatives before they settled for one.

As can be seen, there is a 50% reduction of errors between the first and the second sequence. Interestingly, this reduction occurs only for the errors in the location and the repair phase, while errors in comprehension stay at the same level. In addition, the errors were not distributed evenly across tasks. The comprehension errors concentrated on the syntactic bugs, while the repair errors occured mainly with bugs that had to do with list manipulation. Locating a bug once an error had been detected was the easiest problem in all of our tasks. Below we discuss the bugs in the order they were given to the subjects, noting is in which particular debugging episode subjects had difficulty with the bug.

Parenthesized Variable. This bug was very difficult to detect, even when it occured for the second time. Of all six bug types given to them, subjects most often accepted this bug as a correct solution (see Table 3). Once the bug was found (frequently with hints from the experimenter), correction was a problem only the first time around, when subjects were still unsure about how to write and evaluate function definitions.

Quoted Variable. This bug also turned out to be hard to detect. However, most subjects noted the quote in the function body as being rather unusual. Subjects seemed to detect the bug just because nothing else was wrong with the function. Again, correcting the bug was easy once it was detected.

No Embedding. This turned out to be the easiest bug for most of our subjects. They recognized immediately that to translate the mathematical formula, the LISP code had to be embedded. Some subjects had to think hard about how the correct embedding was to be done, but all came up with the correct solution in the end.

No Combiner. There was some problem in detecting this bug, but it turned out to be even harder to fix it. Subjects had trouble with the concept of creating a list to output two numbers, and often needed a hint to find the bug. Once the bug was found, there was the problem of finding the right combiner. Most subjects iterated at random through the three combiner functions they knew, without having an idea about their differences.

Wrong Combiner. Essentially, this problem showed the same characteristics as the previous one. However, this time some subjects had to be alerted to the fact that the combiner used produced a wrong parenthesization. Once parenthesization was detected as the problem, localization was not too difficult. To fix the problem, some subjects went through the same kind of combiner iteration they had used in the previous problem.

Missing Reverse. There are actually two bugs to consider. In SNOC, the outer 'reverse' was left out, while in ROTATER, it was the inner 'reverse' (see the Appendix). Nobody had problems in detecting the error in SNOC. The main problem in SNOC was to detect the systematic relation between the correct answer and the output. Some subjects seemed to bypass this analysis and went straight to changing the code. This resulted in an inadequate correction, leading the subjects back to consider alternatives. At this point, two of the subjects decided to rewrite the code by using an (*append* <*list*> (*list* <*atom*>)) construction. The other subjects went back to the code, and discovered, with or without hints from the experimenter, that all they had to do was to reverse the output.

ROTATER posed a different problem. By the time they got to this problem, subjects had gathered some experience in debugging. Most of the subjects quickly located the error as being in the second argument to *append*. Opinions on how to

fix this error differed widely, however. Most subject found the correct solution after exploring several erroneous paths and getting hints from the experimenter. The complexity of the code left more correction paths open for the subjects than the previous problems. This seems to account for the additional time needed to solve this problem.

To summarize, it seems that we are able to attribute most of the typical errors on a particular function to one of the debugging episodes we described. The debugging process we presented thus should be able to serve as an ideal model for novice debugging. The errors students make could be thought of as deviations from this ideal model. The next section will describe a computational model of the debugging process.

A PRODUCTION SYSTEM MODEL OF DEBUGGING

Overview

We developed a simulation of the student debugger written in GRAPES (17), a goal-restricted production system language intended to implement an aspect of the ACT* theory (18) [2]. GRAPES is distinguished from other production system languages by its goal structure - productions must match to particular goals before they can fire. One of the actions of a production can be to actually create goals and subgoals. Thus, the goal-structure itself is created by productions. This feature of GRAPES makes it particularly useful for modeling the goal-directedness of human problem solving behavior. Figure 1 gives the goal structure for the model.

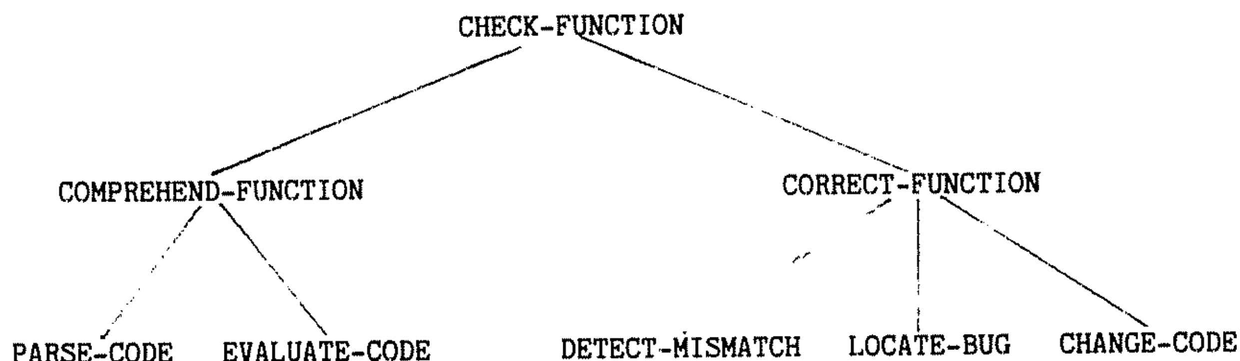

Figure 1
The Goal Tree

The model is started up by giving it a problem and initializing a top goal. When the top goal is initialized, two subgoals are set up. The first subgoal demands an evaluation of the function in order to comprehend it, while the second one demands to find and correct the bug. To satisfy the comprehension goal, the function is first parsed into the smallest units that can be

---

[2] The actual productions used and traces of their performance can be obtained by writing to Claudius Kessler, Department of Psychology, Carnegie-Mellon University, Pittsburgh, Pa 15213.

evaluated. The function is then evaluated according to the rules of LISP, with the result of every LISP function call being stored in working memory. The output that a given function call would produce is the final result of the comprehension phase.

Under the second subgoal, code correction, further subgoals are set up that in turn detect a bug, locate it, and finally change the code. This phase starts out with a goal to check the output for correctness. If it is found to be incorrect, the checking productions return a description of the discrepancy between the output and the correct answer. Under the localization subgoal, the evaluation results of all LISP function calls can then be compared to the discrepancy description, and the buggy part of the code can be located in this way. Once the bug is located, a final goal to change the buggy part of the code is instantiated. If a change can be made, the production system goes again through the evaluation cycle. If the output now matches the correct answer, the production system halts with a success message. If not, the system can go through another debugging cycle if its knowledge base indicates another possible change. If all possibilities are exhausted, the system halts with a failure message.

Examples

The production system was designed so it could handle all the bugs in the Appendix. These bugs could appear in any user-defined function as long as no other than the basic LISP functions were used. In order to see how the model works, examples of the functions FIRST, SQR, and ROTATER are given. In addition to the general working principles of the productions system, the FIRST function illustrates how syntactic bugs are handled, while SQR and ROTATER demonstrate how two aspects of suboptimal debugging that we observed in our protocols have been built in the model.

FIRST. The top-level production sets up the subgoals to parse the function and to match the output and the correct answer. Following LISP syntax, the parsing productions parse the code into the basic functions "car" with the argument "(x)" and, since "x" is in parentheses, it is further parsed as a function with no argument. Obviously, "x" is not a basic LISP function, and the evaluation productions cannot evaluate it. Thus, an error message is returned as the result of the first subgoal and the location of the error is marked as "x".

Since there is no answer to be matched, and the error is already located, the only thing that is left to do under the second subgoal is to change the code. This is done by the matching production that fires when there is a syntax error. It corrects the code, and on the second iteration of the debugging process, the code is recognized as correct.

SQR. In this case, the first subgoal returns with an evaluation. The result is a number, not a list. The match productions then evoke the DETECT-operator which gives the discrepancy description that there is no list. The discrepancy description is then used in the locate-productions to locate the error as a statement to make a list. Under the change-goal, a combiner function is inserted in the code. In this case the production system has incomplete knowledge about combiner functions. It randomly picks a combiner that has not been used yet. This turned out to be a typical novice strategy. Since there are only 3 possible statements, the debugging is guaranteed to succeed on the third iteration at worst.

ROTATER. Again, the first subgoal is completed with the return of an evaluation. The match-productions again evoke the DETECT operator which returns with a discrepancy description pointing to the part of the answer list that is wrong. The locate-productions then locate the part of the code which is responsible for the wrong part of the answer list. The system then goes off into a simple means-ends analysis, trying to substitute code so it gets the correct second element of the list. This is another novice strategy we observed. Since

this approach does not produce the correct code, it is abandoned. Our students also tended to give up this strategy when it did not lead to immediate success. It then sets a goal to use the functions in the code in order to find the right answer, a strategy that was given as a hint when students got lost. In this way, it finally arrives at inserting the missing reverse. The function is then checked again and is found correct.

GENERAL DISCUSSION

The model completely specifies the processes necessary to debug the functions in the Appendix. It does so by breaking down the debugging processes into episodes. These episodes closely match the performance of human novices on the same tasks. While we tried to implement some of the ineffective strategies human novices show, we did not impose other limitations on the production system. Our system, for example, does not have any working memory limitations that lead to a buggy performance. Anderson & Jeffries (19) have shown that these working memory limitations account for a large part of the inferior performance of novices. It is possible and highly likely that the knowledge our subjects had about LISP was in a state where it was highly capacity demanding. Thus, working memory limitations probably account for some of the performance we observed in our subjects. This goes for forgetting the exact specification of a LISP function as well as for forgetting intermediate results or goals that have been established.

Another aspect of our system that may not be true of the human novices is the strict goal hierarchy. The behavior of our novices was not as goal-directed as our system is. Subjects often floundered and went off into some trial-and-error behavior until they were brought back on track by the experimenter. We think that our model nevertheless captures all the steps a novice is in fact going through when debugging a problem flawlessly.

In our protocols, we observed several ways in which our subjects' performance was suboptimal. Our model captures some of the ways in which a novice can perform ineffectively. It further allows us to identify the subskill in which the performance flaw is occuring. We know for example that detecting a bug does not guarantee that it also can be corrected, and vice versa, that a person who has enough knowledge of LISP to write functions correctly, does not necessarily recognize a given bug. Since the model attributes performance lapses to different episodes, it could be useful as a diagnostic tool for identifying weaknesses in a beginning programmer's knowledge.

The research reported in this paper also has implications for the teaching of debugging. It became clear that debugging is a skill that does not immediately follow from the ability to write code. Rather, it consists of several subskills that can and must be taught in addition to instructions about how to write programs. These subskills include the ability to evaluate code correctly, to be able to locate errors by parsing the code and matching it with the results obtained, and the ability to generate correct code to fix the bug.

In our model, we could simulate the debugging process as we extracted it from the protocols, including some inefficient strategies our novice subjects were using. With more data on the debugging behavior of novices on different functions, we should get a more complete picture of the possible strategies and novice inefficiencies associated with debugging subskills. An extended production system could model these skills, incorporating the ideal debugger and the strategy differences and errors of the novices. As such, it would form the core of a debugging tutoring system that could complement programming instruction.

REFERENCES

1. Rich, C., & Waters, R. C. (1981). Abstraction, inspection, and debugging in programming (Tech. Rep. AI Memo 634). Massachussetts Institute of Technology, Boston, MA.

2. Johnson, L., & Soloway, E. (1984). Intention-based diagnosis of programming errors. Proceedings of the 1984 Conference of the AAAI.

3. Youngs, E. A. (1974). Human errors in programming. International Journal of Man-Machine Studies, 6, 361-376.

4. Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1983). What do novices know about programming? In B. Shneiderman & A. Badre (Ed.), Directions in Human-Computer Interaction. Norwood, NJ: Ablex Inc.

5. Jeffries, R. (1981). Computer program debugging by experts. Paper presented at the Psychonomics Society Meeting.

6. Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. Paper presented at the American Educational Research Association Annual Meeting.

7. Gould, J. D., & Drongowski, P. (1974). An exploratory study of computer program debugging. Human Factors, 16, 258-277.

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

9. Atwood, M. E., & Ramsay, H. R. (1978). Cognitive structures in the comprehension and memory of computer programmers: An investigation of computer program debugging (Tech. Rep. TR-78-A21). U. S. Army Research Institute.

10. Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and loopin constructs: An empirical study. Communications of the ACM, 26, 853-860.

11. Katz, I. R., & Anderson, J. R. (1985). An exploratory study of novice programmer's bugs and debugging behavior. Unpublished Manuscript. Carnegie-Mellon University, Pittsburgh, PA.

12. Carver, S. M., & Klahr, D. (in press). Children's acquisition of debugging skills in a LOGO environment. Journal of Educational Computing Research.

13. Reiser B., Anderson J. R., & Farrell, R. (1985). Dynamic student modelling in an intelligent tutor for LISP programming. Proceedings of the International Joint Conference on Artificial Intelligence.

14. Anderson, J. R., Corbett, A. T., & Reiser B. J. (in press). Essential LISP. Reading, Ma.: Addison-Wesley.

15. Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. Cognitive Science, 8, 87-129.

16. Anderson, J. R. (1985). Production Systems, Learning, and Tutoring. In D. Klahr, P. Langley, & R. Neches (Ed.), Self-modifying Production Systems: Models of Learning and Development. Cambridge, Ma.: Bradford Books/MIT.

17. Sauers, R., & Farrell, R. (1982). GRAPES User's Manual ONR Technical Report. Carnegie-Mellon University, Pittsburgh, PA.

18. Anderson, J. R. (1983). The Architecture of Cognition. Cambridge, MA: Harvard University Press.

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

APPENDIX

The Bugged and the Correct Function Versions

BUG: Parenthesized Variable

This function is called first. Given any list, it should return the first element of that list. For instance, (first '(a b c)) should return a.

wrong: (defun first (x) (car (x)))

correct: (defun first (x) (car x))

This function is called extract.It should return the second element of a list. For instance, extract called on (a b c) should return b.

wrong: (defun extract (x) (car (cdr (x))))

correct: (defun extract (x) (car (cdr x)))

BUG: Quoted Variable

This function is called replace. It should replace the first element of a list with a new element. This function takes two parameters - the new element and the list. For instance, (replace 'rings '(ties hats pants)) should return (rings hats pants).

wrong: (defun replace (x y) (cons 'x (cdr y)))

correct: (defun replace (x y) (cons x (cdr y)))

This function is called pair. It takes as its argument an element x and a list y. It should return a list consisting of two elements, x and the first element of the list y. For example, (pair 2 '(4 5 6)) = (2 4).

wrong: (defun pair (x y) (list 'x (car y)))

correct: (defun pair (x y) (list x (car y)))

BUG: Wrong Embedding

This function is called "ftoc". It takes as its argument a degree reading in fahrenheit and should return the celsius equivalent. The formula for converting fahrenheit (f) to celsius (c) is: c = ((f - 32) / 1.8)).

wrong: (defun ftoc (x) (difference x 32)(quotient x 1.8))

correct: (defun ftoc (x) (quotient (difference x 32) 1.8))

This function is called "ctof" . It should convert celsius degrees to fahrenheit degrees. The equation to convert celsius (c) to fahrenheit (f) is: f = (c * 1.8) + 32).

wrong: (defun ctof (x) (times x 1.8) (plus x 32))

correct: (defun ctof (x) (times (plus x 32) 1.8))

BUG: No Combiner

This function is called sqr. It should return a list of the perimeter and the area of a square, given the length of one side. So, (sqr 2) should return (8 4).

wrong: (defun sqr (x) (times x 4) (times x x))

correct: (defun sqr (x) (list (times x 4) (times x x)))

This function is called polar. It takes one argument that is a radius of a circle that is situated at the origin of a cartesian co-ordinate plane and another argument that is the angle away from the x axis. The radius and angle are measurements in a polar co-ordinate system that are converted to cartesian (x and y) co-ordinates and then returned in a list by this function. The x co-ordinate is the radius times the cosine of the angle and the y co-ordinate is the radius times the sine of the angle. For example, (polar 10 60) = (5.0 8.66).

wrong: (defun polar (x y) (times x (cos y)) (times x (sin y)))

correct: (defun polar (x y) (list (times x (cos y)) (times x (sin y))))

BUG: Wrong Combiner

This function is called back. It should return two copies of a list, where each copy is the original reversed. Thus, (back '(a b c)) should return (c b a c b a).

wrong: (defun back (x) (list (reverse x) (reverse x)))

```
correct: (defun back (x) (append (reverse x) (reverse x)))
```

This function is called pal. It takes a single list as an argument and should return a palindrome that is twice as long. A palindrome is a list that reads the same forward and backward. For instance, (a b c c b a) would be the palindrome made from (a b c).

```
wrong: (defun pal (x) (list x (reverse x)))

correct: (defun pal (x) (append x (reverse x)))
```

BUG: Missing Reverse

This function is called snoc. It is the opposite of cons. Instead of inserting an item into the front of a list, it should insert the item at the end. So, (snoc 'd '(a b c)) = (a b c d).

```
wrong: (defun snoc (x y)  (cons x (reverse y)))

correct: (defun snoc (x y)  (reverse (cons x (reverse y))))
```

This function is called rotater. Its argument is always a list. It should return a list that is the same as the argument except that the former last element becomes the new first element. Thus, it rotates the list one to the right. For example, (rotater '(a b c d)) = (d a b c).

```
wrong: (defun rotater (x) (append (last x) (reverse (cdr x))))

correct: (defun rotater (x) (append (last x) (reverse (cdr (reverse x)))))
```