ED 152 321 IR 005 863

AUTHOR Wescourt, Keith T.; Hemphill, Linda
TITLE Representing and Teaching Knowledge for Troubleshooting/Debugging. Technical Report No. 292.
INSTITUTION Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY Advanced Research Projects Agency (DOD), Washington, D.C.; Office of Naval Research, Washington, D.C. Personnel and Training Branch.
PUB DATE 1 Feb 78
GRANT N00014-77-C-0124
NOTE 150p.

EDRS PRICE MF-$0.83 HC-$7.35 Plus Postage.
DESCRIPTORS Artificial Intelligence; *Computer Programs; *Concept Teaching; *Information Processing; *Instructional Design; Models; *Problem Solving; Programers; Programming Problems; Skill Development

ABSTRACT

The goal of the present project was to identify the types of knowledge necessary and useful for competent troubleshooting/debugging and to examine how new approaches to formal instruction might influence the attainment of competence by students. The research focused on the role of general strategies in troubleshooting/debugging, and how they might be represented and taught explicitly and directly in order to avoid the cost and other drawbacks of learning indirectly by observation and practice. Related work on troubleshooting/debugging was examined, and in conjunction with a logical analysis, contributed to a characterization of troubleshooting/debugging problems that emphasizes their generality across a number of technical fields and informal contexts. Further data gathered from students learning computer programming suggest that expert debuggers do not necessarily have superior general strategies; rather, their expertise derives from specific and sometimes idiosyncratic knowledge acquired through experience. An attempt to obtain a rigorous characterization of the differences and defects in the debugging strategy of students by applying a model-oriented data analysis method was unsuccessful. Another study was conducted to determine the effects of presenting a tutorial text which describes a few general heuristics designed to correct strategy deficits; results indicated a marginal increase in the apparent use of some of the heuristics by those who studied the text compared to a group who did not. The several methodological limitations and problems encountered suggest that, if the causes of differences in ability are to be specified in detail, and if the effects of direct problem-solving instruction are to be assessed, then it will be necessary to perfect model-based data analysis methods. (Author/DAG)

REPRESENTING AND TEACHING KNOWLEDGE FOR TROUBLESHOOTING/DEBUGGING

Keith T. Wescourt

Linda Hemphill

Contract No. N00014-77-C-0124, effective November 1, 1976.
Expiration Date: October 31, 1977.
Amount of Contract: $96,688.
Principal Investigator, Keith T. Wescourt, (415) 497-4117.
Contractor: Institute for Mathematical Studies in
the Social Sciences
Stanford University
Stanford, CA 94305

Sponsored by:

Office of Naval Research
Contract Authority No. NR 154-394
Scientific Officers: Dr. Marshall Farr and
Dr. Henry Halff

and

Advanced Research Project Agency
ARPA Order No. 3339
Program Code No. 61101E

IR 005 863

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Representing and Teaching Knowledge for Troubleshooting/Debugging.		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Nov. 1, 1976-Oct. 31, 1977
		6. PERFORMING ORG. REPORT NUMBER Technical Report No. 292
7. AUTHOR(s) Keith T. Wescourt and Linda Hemphill		8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Institute for Mathematical Studies in the Social Sciences, Stanford University, Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101E RR 042-06; RR 042-06-01 NR 154-394
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel & Training Research Programs Office of Naval Research (Code 458) Arlington, VA 22217		12. REPORT DATE February 1, 1978
		13. NUMBER OF PAGES 142
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

problem solving, debugging, troubleshooting, reasoning, instruction, complex learning, computer programming, artificial intelligence (AI), knowledge representations, heuristics

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

As society's dependence on technology increases, the need for competent technicians who can maintain and repair complex systems increases as well. Present methods of teaching troubleshooting/debugging remain primitive and expensive, relying on students to discover effective and efficient problem-solving methods by observation and practice in relatively unstructured environments. The goal of the present project was to identify the types of knowledge necessary and useful for competent troubleshooting/debugging and

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

to examine how new approaches to formal instruction might influence the attainment of competence by students. In particular, the research focused on the role of general strategies in troubleshooting/debugging and how they might be represented and taught explicitly and directly in order to avoid the cost and other drawbacks of learning indirectly by observation and practice.

Related work on troubleshooting/debugging was examined and in conjunction with a logical analysis contributed to a characterization of troubleshooting/debugging problems and problem-solving processes that emphasizes their generality across a number of technical fields and informal contexts. The analysis also suggests that debugging is a fundamental aspect of almost all learning and problem solving. One result of the analysis was the formulation of an information-processing model of a general troubleshooting/debugging strategy, which describes the types of reasoning processes needed, some of the factors governing selection of alternative processes in solving a problem, and an explicit control strategy.

Extensive examination of a corpus of data from students learning computer programming was undertaken, and some further limited debugging data were collected from both experienced and inexperienced programmers. These data are consistent with a hypothesis that expert debuggers do not necessarily have superior general strategies, but instead that their expertise derives from specific and sometimes idiosyncratic knowledge acquired through experience. Inexperienced programmers lack this knowledge, but in addition some of them have a defective general strategy as well. In an attempt to obtain a rigorous characterization of the differences and defects in the debugging strategy of the programming students, an effort was made to apply a model-oriented data analysis method reported in the literature. However, the method was unsuccessful for the data available and may have more basic limitations. As a consequence only informal conclusions about the defective strategies used by some inexperienced debuggers could be developed: (1) they are deficient in program testing and so fail to find bugs; (2) they do not collect or use available data about the effects of a bug to constrain their reasoning; (3) they have a low threshold for attempting minor and sometimes irrational repairs; and (4) they do not backtrack well from unsuccessful repair attempts.

A small-scale study was conducted to determine the effects of presenting a tutorial text, which explicitly describes a few general heuristics designed to correct these strategy deficits, to novice programmers. The data indicate a marginal increase in the apparent use of some of the heuristics by the programmers who studied the text compared to a group who did not. In addition, comments elicited from the students were generally favorable to presenting problem-solving strategies explicitly, as they were in the tutorial. However, the success of the groups in solving debugging test problems did not differ. There were several methodological limitations and problems encountered in the study which further confound the results. More general methodological issues for studies designed to investigate instruction in troubleshooting/debugging also became apparent. One of the most important is analysis of complex problem-solving data: if the causes of differences in ability are to be specified in detail and if the effects of direct problem-solving instruction are to be assessed, then it will be necessary to perfect model-based data analysis methods.

Summary

As society's dependence on technology increases, the need for competent technicians who can maintain and repair complex systems increases as well. Present methods of teaching troubleshooting/debugging remain primitive and expensive, relying on students to discover effective and efficient problem-solving methods by observation and practice in relatively unstructured environments. The goal of the present project was to identify the types of knowledge necessary and useful for competent troubleshooting/debugging and to examine how new approaches to formal instruction might influence the attainment of competence by students. In particular, the research focused on the role of general strategies in troubleshooting/debugging and how they might be represented and taught explicitly and directly in order to avoid the cost and other drawbacks of learning indirectly by observation and practice.

Related work on troubleshooting/debugging was examined and in conjunction with a logical analysis contributed to a characterization of troubleshooting/debugging problems and problem-solving processes that emphasizes their generality across a number of technical fields and informal contexts. The analysis also suggests that debugging is a fundamental aspect of almost all learning and problem solving. One result of the analysis was the formulation of an information-processing model of a general troubleshooting/debugging strategy, which describes the types of reasoning processes needed, some of the factors governing selection of alternative processes in solving a problem, and an explicit control strategy.

Extensive examination of a corpus of data from students learning computer programming was undertaken, and some further limited debugging data were collected from both experienced and inexperienced programmers. These data are consistent with a hypothesis that expert debuggers do not necessarily have superior general strategies, but instead that their expertise derives from specific and sometimes idiosyncratic knowledge acquired through experience. Inexperienced programmers lack this knowledge, but in addition some of them have a defective general strategy as well. In an attempt to obtain a rigorous characterization of the differences and defects in the debugging strategy of the programming students, an effort was made to apply a model-oriented data analysis method reported in the literature. However, the method was unsuccessful for the data available and may have more basic limitations. As a consequence only informal conclusions about the defective strategies used by some inexperienced debuggers could be developed: (1) they are deficient in program testing and so fail to find bugs; (2) they do not collect or use available data about the effects of a bug to constrain their reasoning; (3) they have a low threshold for attempting minor and sometimes irrational repairs; and (4) they do not backtrack well from unsuccessful repair attempts.

A small-scale study was conducted to determine the effects of presenting a tutorial text, which explicitly describes a few general

heuristics designed to correct these strategy deficits, to novice programmers. The data indicate a marginal increase in the apparent use of some of the heuristics by the programmers who studied the text compared to a group who did not. In addition comments elicited from the students were generally favorable to presently problem-solving strategies explicitly, as they were in the tutorial. However, the success of the groups in solving debugging test problems did not differ. There were several methodological limitations and problems encountered in the study which further confound the results. More general methodological issues for studies designed to investigate instruction in troubleshooting/debugging also became apparent. One of the most important is analysis of complex problem-solving data: if the causes of differences in ability are to be specified in detail and if the effects of direct problem-solving instruction are to be assessed, then it will be necessary to perfect model-based data analysis methods.

REPRESENTING AND TEACHING KNOWLEDGE FOR TROUBLESHOOTING/DEBUGGING

by

Keith T. Wescourt and Linda Hemphill

February 1, 1978

Institute for Mathematical Studies in the Social Sciences
Stanford University
Stanford, California

Acknowledgements

We wish to recognize the participation of Diana Egly, Alex Strong, Mary Dageforde, Roger Cole, and Marian Beard, all of who contributed to this research in several roles. We thank Drs. Marshall Farr and Henry Halff, Personnel & Training Research Programs, Office of Naval Research, and Dr. Harry O'Neil, Jr., Program Manager, Cybernetics Technology Office, Defense Advanced Research Projects Agency, for their support and encouragement throughout the project.

This research was sponsored by ONR Contract N00014-77-C-0124. Contract Authority No. NR 154-394.

I. Introduction

The increasing dependence of our society on technology is a phenomenon. Complex systems continue to perform new functions and become more sophisticated. For example, consider their role in modern commercial aviation. There are of course the modern jet aircraft incorporating dozens of electrical, electronic, and mechanical systems. But there are also the networks of radar and communication systems for controlling air traffic and the computerized scheduling and reservation systems for coordinating flights and access to them by passengers and cargo. It is difficult to imagine how the demands our society now places on commercial aviation could be satisfied without these complex systems. Such systems hav become equally indispensible throughout our society.

Error or failure is always a threat when relying on a complex system. The result might merely be inconvenience, as it would if an airline's reservation system lost track of a passenger's reservation. Or, it could be disaster, if, for example, an aircraft's radar failed in flight under conditions of poor visibility. Preventive maintenance and repair of complex systems is therefore an important concern. One response to the problem have been efforts to develop better types of technical data for both routine maintenance and repair procedures to accompany complex systems (Potter & Thomas, 1976). A second, complementary response, one with which this report is concerned, is to provide better training for the people responsible for testing and repairing complex systems.

If a system does not operate as it should either during testing or during actual use-- if the oil pressure warning light comes on in an aircraft, if ground radar incorrectly indicates the position of aircraft, or if the reservation system allows two passengers on the same flight to be assigned to seat 10A--, then a human technician must be summoned to solve the problem of locating and correcting the cause of the failure. This type of problem solving is referred to in different contexts as troubleshooting or debugging. The objective of "good" troubleshooting/debugging is to locate and correct the cause of failures efficiently, without undue cost of materials and time. An electronics technician does not want to replace several components in a circuit if he has reason to believe that only one of them is faulted and that he can identify and replace just that one in a reasonable amount of time. Similarly, a computer programmer faced with a program that generates incorrect results wants to make a relatively limited correction, one that does not entail recoding parts of the program that perform their function adequately.

Expert troubleshooters, those technicians (or technical consultants) who make difficult repair problems seem easy and "impossible" ones only difficult, have always been highly valued and are often regarded as artists, since their expertise is so poorly understood. Demand for their services can only grow as complex technology spreads. However, advancing technologies have introduced features such as built-in test systems, modular system organization, and miniaturization that make efficient troubleshooting of routine types of failures in even the most complex systems possible for technicians with more limited skill. Unfortunately, many newly trained technicians have difficulty even with routine problems and become competent only after they have had considerable field experience. Thus, maintenance costs are high and, in settings where there is a high-rate of personnel turnover, there tends to be a chronic shortage of competent technicians.

The research described in this report investigated the bases for competence and expertise in troubleshooting, as seen in the context of computer program debugging. The goal was to identify the types of knowledge necessary and useful for competent debugging and to determine whether new approaches to formal instruction might facilitate the attainment of competent debugging ability by new programmers.

Troubleshooting/debugging as a general aspect of problem solving

Situations that pose a problem of locating and correcting the cause of a failure are not limited to electronics, mechanics, and computer programming and do not necessarily include complex technology. In some contexts, the parallels are straightforward enough to have extended the common usage of the terms "troubleshooting" and "debugging". Management consultants are often called troubleshooters. Using the methods of operations research, they locate causes of inefficiency in an organization (corporations, agencies, etc.) and suggest corrections to its structure or procedures. The scope of these repairs is constrained by cost much as are those a technician can make in order to bring a device up to specifications.

Less obviously, the behavior of a teacher tutoring a single student shares features with that of a troubleshooter. In tutoring, the teacher asks questions evaluates the student's responses, and provides explanations in a continuing dialog (Figure 1). The purpose of some of his questions is to elicit answers that identify specific inaccuracies

1 T: Do you think it rains much in Oregon?

(Case selection: Oregon is a paradigm case of a first order causal model of rainfall.
Diagnosis: ask for a prediction about a particular case.)

2 S: No

(S's prediction is wrong)

3 T: Why do you think it doesn't rain much in Oregon?

(Diagnosis: ask for any factors.)

4 S: I'm not exactly sure - just hypothesizing - it seems to me that the surrounding states have rather dry climate, but I really don't know anything about the geography of Oregon.

(S's error is due to a proximity inference; S has no knowledge of relevant factors)

5 T: It does in fact rain a lot in Oregon. Can you guess what causes the rain there?

(Correction: inform student.
Diagnosis: ask for prior factors.)

6 S: Well, let me see - I have a feeling that there is a mountain range nearby and the ocean i.e. Pacific, I think probably borders Oregon somewhat?

(S names 2 factors, but does not mention their relationship to rainfall.)

7 T: Yes the Pacific borders Oregon how do you think it is involved in the heavy rainfall there?

(Diagnosis: T selects prior factor; holds other factor;
Rule: ask for intermediate factors.)

8 S: I haven't really got any idea - well not quite true; I would only be guessing. Does the air (moist air) from the ocean somehow get blown over Oregon and encounter a block of some sort which causes it to rise and cool?

(S is missing three steps that are in T's model: 1. why the air is moist, 2. why it is blown over Oregon, 3. why cooling results in rain)

Figure 1. Annotated dialog between a human tutor and student. From Stevens and Collins, 1977.

or omissions in the student's knowledge. Once these errors are detected, the tutor may provide explanations which he believes will correct them. Alternatively, as in the Socratic tutoring method, he may ask further questions designed to prompt the student to reason about other knowledge he has and thereby to correct himself. (See Collins, 1976, for an analysis of Socratic tutoring.) The tutor is thus debugging the student's system of knowledge (Stevens and Collins, 1977).

Troubleshooting/debugging problems also occur in[illegible] in a range of everyday contexts. Most commonly, people ar[illegible]d with balky cars or household appliances, and attempt some limited troubleshooting to avoid the expense and inconvenience of calling a repairman or at least to enable them to give him a good description of the problem if forced to call him. People also engage in informal debugging in developing instructions. For instance, if someone gets lost following directions you gave them for getting to your house, then you engage in debugging when you determine which step of your instructions were wrong or were executed incorrectly. If the instructions are lengthy, then it can be effort to check them step-by-step from the beginning against a mental image of a map or of the route you intended. Thus, to be more efficient, you might consider the location from which your friend called you when he found himself lost and its proximity to points along the intended route. The analysis serves to limit the section of your instructions you need to examine for the error. This type of reasoning behavior resembles that of a computer programmer, who uses the characteristics of a program's erroneous output to suggest where he should start tracing program code. Other informal situations that require debugging-like problem solving range from developing a new

(Figure 3 continued)

```
DEBUG      -> <[DIAGNOSE] + [REPAIR]>*
DIAGNOSE   -> <ASK | TRACE | "error">*
TRACE      -> [SELF-DOC*] + RUN*
SELF-DOC   -> ADD-PAUSE | ADD-PRINT | ADD-TRACE
ASK        -> "print definition" | "print value" |"print file"| ...
REPAIR     -> <RUN | EDIT | SOLVE>*
ADD-PAUSE -> ADD
ADD-PRINT -> ADD
ADD-TRACE -> ADD
EDIT       -> ADD | DELETE | CHANGE
RUN        -> "run statement of code" + "response" + [DEBUG]
ADD        -> "add statement of code" + "response" + [DEBUG]
DELETE     -> "delete statement of code" + "response" + [DEBUG]
CHANGE     -> "change statement of code" + "response" + [DEBUG]
```

like learning to ride a bicycle: you watch someone else and then climb on and try yourself. When you fall, you try to figure out why, and perhaps receive advice from a proficient bicyclist, such as "Look at the horizon, not the front wheel!" High motivation is required to learn troubleshooting/debugging in this way, since the frustrations one encounters are psychological analogies to skinned elbows and knees. The instructor's method of facilitating the process is largely empirical; he tries to identify the examples and exercises that result in better student performance on test exercises.

The indirect approach to teaching troubleshooting/debugging does work satisfactorily for some students: after all, it is the way in which existing competent troubleshooters acquired their skill. Other students having "fallen of the bicycle" more times than they can bear (or the educational system will allow) become drop-outs. In general however, the indirect method is less successful for teaching problem solving in technical, than in other subjects. The factor involved is the cost of resources required to generate examples and to allow students to work on exercises. In mathematics or subjects based on mathematics, most problems can be solved with paper-and-pencil and the only demands are on the instructor's imagination and energy and the student's time. Troubleshooting problems (and also design problems in engineering) require resources like equipment and space, which are scarce commodities in most educational settings. Since the cost of these resources varies directly with the amount of time used and number of errors made by students, there is an inherent pressure to limit student experience to a minimal number of simple, and less than realistic, problems. The limitations are most critical for students having difficulty, who fail to have experiences sufficient for learning the required knowledge and so either drop-out or fail. Even better students, however, may not get enough experience to become sufficiently competent by the time they finish formal instruction. Thus, new troubleshooters/debuggers must typically undergo a period of on-the-job training, which is expensive both because their productivity is low and because it requires the involvement of experienced technicians.[1]

A more direct approach to teaching troubleshooting/debugging

One approach to improving formal instruction in troubleshooting/debugging is to reduce the costs of the indirect method associated with providing examples to students and with operating and supervising student problem-solving laboratories (Finch, 1971). However, there is an apparent paradox in the indirect method that could indicate a need for a substantially different approach to instruction for some students. The paradox is that the learning by example and trial-and-error experience required by the indirect method may actually presuppose the very problem-solving strategies the student is attempting to learn (recall the analogy of tutoring as "debugging the student"). In effect, learning by the indirect approach requires the student to debug his strategy for how to debug.

Since people do learn to debug by observation and practice, no real paradox exists. Clearly, sophisticated strategies must evolve by bootstrapping from a primitive learning mechanism, which we is effective, though less than optimal, for inductive learning in simple

[1]For scientific professionals, the latter years of graduate education involve research experience that serves a similar function for developing problem-solving skills.

contexts. Students in technical disciplines bring to the classroom debugging strategies of varying effectiveness which they have induced by monitoring their attempts to solve the types of informal everyday troubleshooting/debugging problems we mentioned earlier. Some of them may already have effective general strategies and only have to learn how to apply them in a new problem domain. The indirect method works for them because their debugging strategies help them to learn efficiently from their experiences; they are proficient at debugging their own knowledge. However, those students with ineffective and inefficient initial strategies encounter a bootstrapping problem because efficient learning by induction presupposes some of the same strategies as debugging. Therefore, another approach to instruction in troubleshooting/debugging, which would be most advantageous to students of lower initial ability, is to try to teach more directly and explicitly the general strategies that students develop when they understand examples and try to solve problems themselves. Such instruction could help students to acquire an effective strategy for troubleshooting/debugging more rapidly and improve their general capability to learn by the indirect method to troubleshoot in a particular domain.

The are two aspects to developing an alternative, more direct approach for teaching troubleshooting/debugging. First, the strategies that students learn by observing competent problem solvers and by solving practice problems must be identified and articulated (i.e., represented). Second, a suitable pedagogy must be formulated. These goals are not necessarily independent, since pedagogical decisions can depend on the way the knowledge is represented and conversely, choices among alternative representations can depend on features of preferred or available teaching methods.

In the remaining sections of this report, we will discuss the ideas of others and ourselves about the nature of the troubleshooting/debugging process. We will describe our observations of computer program debugging behavior which bear upon these conceptions, and which also suggest the knowledge deficits that cause some inexperienced programmers to have difficulty with even simple debugging problems. We will conclude by presenting the results of a study designed to investigate whether such deficits might be corrected by direct instruction.

II. Understanding the troubleshooting/debugging process

Difficulties in studying troubleshooting/debugging

One reason that troubleshooting/debugging (and other types of complex problem solving) are taught indirectly is that it is difficult to gather the data needed to develop an empirically-based understanding of the problem-solving process. There are problems of observing a range of episodes and of the observer not interfering with the troubleshooter's behavior. Simple problems may be solved in minutes during a single "sitting", while complex problems may be solved over days or even weeks (e.g., the debugging problems faced by system programmers on large computer systems). Thus, it is much more difficult to observe the solutions to problems at the more difficult end of the spectrum. In any episode, there is the problem of observing the troubleshooter without causing him to depart from his normal procedures.

A general limitation in studying troubleshooting episodes is that much of the troubleshooter's time is spent in periods of thinking, during which there is no overt behavior to observe. Typically, it is difficult to infer what the problem solver is thinking from the behavior observed prior and subsequent to these quiet periods. Post hoc reports (e.g.,"Tell me how you solved that problem") tend to be edited and incomplete, appearing as idealized accounts which frequently conflict with observed behavioral data. More general self-reports ("Tell me how you troubleshoot") may also be contradictory and incomplete. There is a truism that being an expert at doing something does not necessarily imply being able to introspect on how one does it.

Troubleshooting/debugging seems to be an activity for which is not easy for most experts to describe their reasoning in either particular or general terms. The fictitious dialog in Figure 2 caricatures this inability.

There has also been a difficulty in analyzing and organizing the behavioral data and self-reports that can be obtained. Prior to the development of information-processing and cybernetics there was no adequate formalism for describing processes-- i.e., to represent procedural knowledge-- and thus for interpreting and integrating sets of observations in order to develop and test hypotheses relating the knowledge used by troubleshooters solvers and differences in troubleshooting episodes. While natural language has been used to represent propositional knowledge from the earliest times, it is a poor medium for expressing complex procedural knowledge. To convince yourself of this consider the typical comprehensibility of the assembly and operating instructions for various devices. Usually, one remains uncertain of his understanding until the device works (i.e., the instructions are understandable only if you already know the process). One apparent weakness of natural language for describing processes is its awkwardness and ambiguity for expressing complex conditional relationships between events. More generally, in natural language much of the knowledge being transmitted by the sender is implicit and must be inferred by the receiver. The demands for decoding the implicit knowledge may be more severe for procedural than for propositional knowledge. (Try to generate a sufficient description of how to drive a car that you can feel confident will be understood without questions by someone who has never driven one.) The limitations of natural language

O "How did you know the trouble was in the switch?"

E. "Because it worked intermittently when I jiggled the switch."

O "Well-- couldn't it jiggle the wire?"

E "No."

O "How do you ++know@ all that?"

E "It's ++obvious@."

O "Well then, why didn't I see it."

E "You have to have some familiarity."

O "Then it's ++not@ obvious, is it?"

Figure 2. Fictional dialog between an expert troubleshooter (E) and an observer (O) caricaturing the expert's difficulty in articulating the source of his expertise. From Zen and the Art of Motorcycle Maintenance, p. 135 (Pirsig, 1974).

may be partly responsible for the difficulties that problem-solvers seem to have introspecting: besides their difficulties in realizing how they troubleshoot, they may not be able to articulate what they are aware of.

Thus, understanding of the troubleshooting/debugging process has been hampered both by difficulties in making complete, valid observations and in systematically interpreting the data that can be obtained.

Information-processing models

Over the past twenty years researchers in information-processing psychology concerned with understanding intelligent human behavior and those in artificial intelligence (AI) interested in developing "intelligent" computer systems have developed new formalisms for representing knowledge. Semantic networks (Quillian, 1969; Woods, 1975), production systems (Newell, 1975), procedural networks (Brown, Burton, Hausmann, Goldstein, Huggins, & Miller, 1977; Sacerdoti, 1975), logical calculi (Nilsson, 1971), and process grammars (Miller & Goldstein, 1976a; Woods 1970) are the new "languages" used to represent the declarative and procedural knowledge underlying intelligent behavior in a range of tasks. These formalisms have enabled the development of sufficient ("strong") computational models for certain well-structured problem domains, such as logical proof, games, and puzzles. There are now computer programs that can solve such problems as well or better than most human problem solvers. Strong computational models have also been used to simulate human problem-solving behavior, including its variability and errors, in an analysis-by-synthesis approach to interpreting behavioral and introspective data (Newell & Simon, 1972).

Beyond their application in automated problem solvers, the knowledge representations that have been developed provide a framework for analyzing observations and for articulating partial models of less well understood types of problem solving like troubleshooting/debugging. That is, even if it is not yet possible to write a general program to troubleshoot faults in circuits or one to debug other programs, it may be possible represent the top-level organization such a program would need and some of its more specific data-structures and procedures. Such "weak" models are a basis for directing attention to aspects of the process that are not yet understood and their logical relationships to those that are and for interpreting new data in order to expand our understanding.

Over the past several years, psychologists and computer scientists working the the field of AI at MIT have conducted research on information-processing models of programming and debugging. As a consequence of their work they have come to adopt a view that debugging is a fundamental aspect of most, if not all, complex human learning and problem solving (Goldstein, 1975; Miller & Goldstein, 1976a; Papert, 1971; Ruth, 1974; Sussman, 1973). The position is based on their informal analyses of human programming behavior and on their attempts to develop "intelligent" programs for writing programs and for solving other types of problems. People learning to program and even experienced programmers designing programs <u>knowingly</u> code and attempt to execute programs that are inadequate. They may be unsure about the effects of a particular contruct or of the interaction of familiar constructs in combination. When the program fails, by reasoning from the way it failed they can modify it to function correctly. As a simple example, a statistical program may involve printing a table with a complicated format that depends on the parameters of the data to be analyzed. The programmer writing the program may have difficulty calculating the format parameters needed to align the headings and entries in the table. He may therefore proceed by estimating the format and then executing the program. The errors he observes enable him to modify his original estimates to produce a correct format.

This notion of the generality of debugging goes beyond our earlier comments about the range of situations in which debugging-like behavior is required; it says more strongly that problem solvers consciously create debugging problems for themselves as part of a general planning strategy. Debugging is seen as a natural complement to design in the process of planning and implementing a program. Either because it is more efficient or because human information-processing capabilities (e.g., in "working memory") limit the complexity of the design process, programmers implement programs with an expectation that they will have to debug them-- i.e., debugging is not necessarily an afterthought forced on programmers.

There is an alternative view of debugging that it is a regrettable outcome of poor design and that programmers can and should strive to eliminate all debugging through rigorous design. This position is popular among advocates of "structured programming" (Dahl, Dijkstra, & Hoare, 1972). We disagree with this viewpoint. While rigorous initial top-down program design is certainly desirable, it is unrealistic to demand and expect flawless design for complex, innovative programs. Our own informal observations of skillful professional programmers indicate that despite conscientious efforts at top-down design, they inevitably start implementing and testing programs before the design is complete. It seems that there are too many complex interrelationships in most programs and that they can be understood and implemented more easily by debugging than by abstract logical analysis. From the programmer's perspective, there is a strategic tradeoff between the costs of design and debugging such that it is most efficient to integrate the two so as to minimize the the maximum complexity at any point.

From their studies of programming, the researchers at MIT have generalized the constructive role of debugging in learning and problem solving using the following logic. A computer program is a representation of a plan, a sequence of legal operations in an environment that when executed will accomplish a goal (i.e., solve a problem). For example, a sorting program is a plan for accomplishing the goal of arranging a list of values in a desired order from an arbitrary initial order. However, writing and executing a program is only one way of expressing and following a plan. Plans were developed and executed by people to solve problems long before computers existed and have been embodied as mechanical and electro-mechanical systems. Programs are just a general way of representing plans. That is why programs can be used to simulate some of the behavior of people and of mechanical and electronic systems.

Plans then, like programs, may also first be formulated with some ignorance of whether particular actions will be effective. If execution of the plan proves it inadequate and if the plan is to be used again, then the information obtained from the failure can be applied to modify the plan. However, even if the plan was for a unique problem and will never be used again, debugging the plan is useful. In designing new plans, parts of old plans for somewhat related problems may be used and so a "library" of correct plans can help the problem solver. Furthermore, one can see that there must be "plans for planning"—general strategies for making design and debugging decisions in planning solutions to particular problems (e.g., whether to synthesize part of a design or borrow it from a design in one's plan library). Plan failures provide feedback that can be used to debug not only the faulty plan, but also the strategy used to design it in the first place.

Sussman (1973) developed many of these ideas about planning and debugging in the course of formulating a computational model called HACKER, a program that solves problems in the paradigmatic "blocks-world" domain. Given a problem of rearranging some of the blocks on a table, HACKER in its naive starting state designs a solution of simple actions (pick up, put on). Depending on the problem, its initial solution may succeed or fail (where failure is defined by action sequences that are redundant or impossible in the blocks world). In case of failure, HACKER works to debug the plan (not always successfully). It also stores information about correct plans and about bugs that it can use in designing solutions for subsequent problems.[2]

Mark Miller and Ira Goldstein at MIT (Miller & Goldstein, 1976a) have attempted to formalize the relationship between design and debugging in problem solving using what they call <u>planning grammars</u>, which are representations of design and debugging strategies. Employing both context-free grammars and augmented transition networks (ATN)

[2]See Sacerdoti (1975) for an alternative view that correct plans can be implemented in incremental stages of design and execution without debugging.

(Woods, 1970), they have written systems of rules that describe the process of creating and executing LOGO graphics programs (Figure 3). They have proposed that planning grammars can serve two functions: (1) interpreting and comparing the behavior of different programmers and (2) developing "intelligent" systems for assisting programmers in designing and debugging their programs. They have explored the second use in their SPADE system (Miller & Goldstein, 1976b), which records a programmer's planning decisions with respect to a planning grammar. The record is used to advise the programmer of conflicts and omissions in the structure of the program and of his options any point in the planning process (Figure 4).

<u>A general characterization of troubleshooting/debugging problems</u>

Our examination of research on troubleshooting/debugging has led us to formulate a characterization of troubleshooting and debugging general to a range of problem domains.

We define troubleshooting/debugging as a type of problem solving focused on either an abstract plan or a <u>procedural system</u>. A procedural system is a physical entity that embodies a plan and can execute it to accomplish its goal. A characteristic of plans and procedural systems is that they can be represented as hierarchies of functional subparts, each subpart having a specific role in achievement of the overall goal. For instance, a plan for building a table includes subplans for obtaining a design, obtaining materials, assembling the wood and hardware, and finishing the assembled table. Each of these subplans consists of smaller subplans. The plan for obtaining materials might include subplans for borrowing a truck, selecting a lumber supplier,

```
P1:  SOLVE        -> PLAN + [DEBUG]
P2:  PLAN         -> IDENTIFY | DECOMPOSE | REFORMULATE
P3:  IDENTIFY     -> PRIMITIVE | DEFINED
P4:  DEFINED      -> USE-CODE & GET-FILE
P5:  DECOMPOSE    -> CONJUNCTION | REPETITION
P6:  CONJUNCTION -> LINEAR | NONLINEAR
P7:  LINEAR       -> SET | SEQ
P8:  SEQ          -> [SETUP] + <MAINSTEP + [INTERFACE]>* + [CLEANUP]
P9:  SET          -> <STEP>*
P10: SETUP        -> STEP
P11: MAINSTEP     -> STEP
P12: INTERFACE    -> STEP
P13: CLEANUP      -> STEP
P14: STEP         -> ADD | SOLVE
P15: REPETITION   -> ROUND | RECURSION
P16: ROUND        -> ITER-PLAN | TAIL-RECUR
P17: ITER-PLAN    -> "repeat step" + SEQ
P18: TAIL-RECUR   -> STOP-STEP + SEQ     + REC-STEP
P19: REC-STEP     -> "recursive program call"
P20: STOP-STEP    -> "stop program call"
```

Figure 3. Miller and Goldstein's (1976b) content-free grammars for planning and debugging programs.

(Figure 3 continued)

```
DEBUG      -> <[DIAGNOSE] + [REPAIR]>*
DIAGNOSE   -> <ASK | TRACE | "error">*
TRACE      -> [SELF-DOC*] + RUN*
SELF-DOC   -> ADD-PAUSE | ADD-PRINT | ADD-TRACE
ASK        -> "print definition" | "print value" |"print file"| ...
REPAIR     -> <RUN | EDIT | SOLVE>*
ADD-PAUSE -> ADD
ADD-PRINT -> ADD
ADD-TRACE -> ADD
EDIT       -> ADD | DELETE | CHANGE
RUN        -> "run statement of code" + "response" + [DEBUG]
ADD        -> "add statement of code" + "response" + [DEBUG]
DELETE     -> "delete statement of code" + "response" + [DEBUG]
CHANGE     -> "change statement of code" + "response" + [DEBUG]
```

SPADE-0 Scenario (part a)

The preliminary version of SPADE is little more than a bookkeeper for the user's subgoals. It is currently being implemented by assigning an interpretive procedure to each grammatical primitive.

```
1a. What is the name of your top level procedure?
1b. >WW

2a. Rule for WW is: SOLVE -> PLAN + [DEBUG].
    Rule for WW-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
    What now?
2b. >DECOMPOSE

3a. Rule for WW-4 is: LINEAR -> SEQUENTIAL | SET.
    What now?
3b. >SEQUENTIAL

4a. Rule for WW-5 is: SEQ -> [SETUP] + <MAIN + [INTERFACE]>* + [CLEANUP]
    Do you wish to include the optional constituent SETUP?
4b. >No

5a. How many occurrences of <MAIN + [INTERFACE]> do you want?
5b. >3

6a. Do you wish to include the optional constituent CLEANUP?
6b. >LATER
```

LATER is a command understood by SPADE. It means that the user wants to terminate this path, but that it must be recommenced later. SPADE keeps track of conjunctive goals, reminding the user of the pending goal list if asked. Here it records the information required to return to, and eventually complete, the application of the SEQ rule. It can then go on to other goals.

Figure 4. Dialog with programmer's assistant based on the grammar shown in Figure 3. From Miller and Goldstein, 1976b.

(Figure 4 continued)

SPADE-0 Scenario (part b)

7a. Your pending goals are POLE, ... and WELL. What NOW?
7b. >POLE

SPADE applies the SOLVE rule recursively.

8a. Rule for POLE is: SOLVE -> PLAN + [DEBUG]
Rule for POLE-1 is: IDENTIFY | DECOMPOSE | REFORMULATE.
What now?
8b. >IDENTIFY

9a. Rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.
What now?
9b. >PRIMITIVE

10a. Rule for POLE-3 is: PRIMITIVE -> VECTOR | ROTATION | PENSTATE
What now?
10b. >VECTOR

.
.
.

>FORWARD 100

11a. POLE completed. Pending goals are: ROOF, WELL, and WW-17 (CLEANUP of WW). What now?
11b. >WELL

.
.
.

12a. Rule for WELL-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
What now?
12b. >DECOMPOSE

Here we have substituted a grammar which contains rules for conjunction but not repetition. This allows us to illustrate the manner in which SPADE avoids interrogating the user when no actual decision is required.

13a. Rule for WELL-4 is: DECOMPOSE -> CONJUNCTION.
(Forced.)
Rule for WELL-5 is: CONJUNCTION -> LINEAR | NONLINEAR
What now?

selecting a hardware supplier, and scheduling the trip to make the purchases. As another example, an electronic power supply consists of subcircuits such as amplifiers, voltage regulators, etc. In turn, each of subcircuit consists of more basic subcircuits, and so on until the level of primitive components-- transistors, resistors, diodes, wires, etc. Similarly, a computer program will typically have subprograms for input, output, initialization, sorting, etc.

A feature of such functionally defined hierarchies is that the subparts at each level are independent in that each is a "black box" from the viewpoint of the others; it doesn't matter how each does what it does, as long as it fulfills its role in attaining the overall goal. For instance, in assembling the table, the details of the subplan by which the materials were obtained are irrelevant as long as the materials are all there when the assembly subplan is executed. Similarly, structurally different but functionally equivalent voltage regulator circuits can be interchanged in a power supply and different sorting algorithms can be interchanged in a program.[3]

The subparts at each level of the functional hierarchy have a teleological structure. In the simplest, linear structure, the action of each subpart depends directly on that of one other subpart and affects directly one other subpart. Of course, the action of a subpart can indirectly affect all the subparts subsequent to it in the

[3] The relationship between subparts at a level of the hierarchy can be more complicated than this, since it is possible for them to be functionally discrete, but still share physical structure. For example, two subprograms for input and output may share ("call") a type-conversion subprogram at a lower level. This overlap is incidental in that shared structure can be replaced by redundant copies, but important in that a defect in the shared structure may affect the function of all superordinate parts.

teleological structure. More complicated structures have multiple interfaces and feedback paths between subparts. When a subpart contains a fault, then its action will be incorrect for at least some of the possible actions of immediately prior subparts. Its faulty actions may inhibit subsequent subparts from operating and thus terminate the operation of the entire plan/system or may propagate through them and distort the actions of the plan/system.

Troubleshooting/debugging involves reasoning about the actions of the plan/system and its teleological structure at each level of its functional hierarchy in order to localize the fault to a minimum number of subparts (ideally one) at that level. The actions and structure of the suspect subpart(s) are then used to localize the fault at the next lower level of the functional hierarchy, and so on until the cost of repairing a subpart(s) is less than the cost of further localization.

Expected cost plays several roles in debugging. It not only determines the level at which repair is attempted, but also serves to order logically equivalent debugging actions. Cost depends on how the structure of the plan/system affects measurements of the actions of and the ability to repair a particular subpart. It also is determined from the debugger's idiosyncratic experiences. For example, if a car idles unevenly, an experienced mechanic may jar the carburetor in case a piece of dirt is lodged in one of the small internal passages <u>before</u> he has done any tests on the ignition timing, spark plugs, or engine compression which might logically determine that the problem is actually in the carburetor. His attempted repair in this case is inexpensive enough to allow testing of a hypothesis developed by induction ("uneven idle has in past experience been associated with dirt in the carburetor") rather than by deduction ("the observations that have been made logically determine that the problem must be in the carburetor").

As an illustration of how cost thresholds enter into reasoning of debugging, consider a simple home troubleshooting problem. Suppose you wake up during the night and decide to go to the kitchen for a snack. When you move the switch on your beside lamp to the "ON" position, the lamp fails to light. Given that you are motivated to discover the cause of the failure and, if possible, effect a repair what would constitute an effective and efficient tack. If there have been previous problems with the lamp that you have traced to an intermittent short in its switch, you might operate the switch several times in an attempt to "unshort" it temporarily. That is, you might identify the symptom and immediately recognize a possible cause that your experience suggests may be more likely than other possible causes and that has an inexpensive (if temporary) repair. If you had no such reason for suspecting the switch, then you must reason about the circuit (procedural system) that contains the lamp. The lamp circuit has a simple linear teleology consisting of the external power supply to the house, a fuse or circuit breaker, the wall outlet, the lamp plug and cord, the lamp switch, the light bulb, and several intervening sections of wiring. The light bulb will not light (the initial symptom you observed), if there is a fault in any of the components prior to it.

One aspect of an effective general troubleshooting/debugging strategy is to make observations that, given the structure of the system, are logically sufficient to exclude or include subparts from a location hypothesis, which is simply a description of where the fault could possibly be located in the system. The actions of any subpart in a linear teleological structure can serve to refine the hypothesis in one of two ways. If the actions are normal at point A, then the fault must be in a subpart subsequent to that point; if the actions are abnormal at A, then the fault must be in a prior subpart. Thus, if all subparts can be observed with equal cost and if the debugger has no special knowledge relating the observations he can make to the likelihood of possible faults (e.g., the lamp switch has an intermittent short, bulbs fail 5 times as often as fuses, etc), then an optimal strategy is to make observations that will repeatedly halve the scope of the location hypothesis until the fault is isolated to a single subpart; this minimizes the expected number of observations that are required to localize to a single subpart. Thus, in our example, because the wall outlet is near the middle of the lamp circuit, the first observation would be to see if the lamp is plugged in and, if so, whether another electrical device connected to the same outlet operates correctly.

Suppose that the lamp proves to be plugged in and furthermore that an electric clock is plugged in at the same wall outlet so that you can easily (without getting out of bed) observe whether it is still operating. If it is, then the fault can not be in the house power supply, the fuse, or any of the connecting wires prior to the wall outlet. If the clock is not working, then the fault _is_ in one of those subsystems (barring two independent failures in the lamp and clock). Let's assume the clock also is not working. Breaks in house wiring are ordinarily uncommon, and so it is most likely that either the power supply has failed or the fuse has blown. Since the fuse box is in the basement, it is "costly" to check, relative to looking out your window to see if the street lights are still working. If those lights are off, then the power has failed. If they are on, then you can replace the fuse.)If that doesn't solve the problem, then get your snack, go back to sleep, and call an electrician in the morning, because the problem is in the internal wiring of the house.)

This is an efficient way to isolate the fault, though given slight changes in the situation other solutions might become better. For instance, if your bed is next to a window, then the easiest observation to start with (before looking at your clock) might be to look out at the street lights. Of course, if they are on, then you know only that the power supply is intact--only one subsystem has been eliminated compared to the three or four eliminated by checking the clock whether it is working or not. The strategy for making observations seems general in itself, but in this episode requires knowledge of the lamp circuit and is affected by idiosyncratic knowledge and by parameters of the situation that determine the costs of making observations and repairs.

Representing a strategy for troubleshooting/debugging

In order to understand more precisely how different types of knowledge are used in troubleshooting/debugging, we developed an information-processing model for a general debugging strategy, like the one illustrated in the above example. The model is general in the sense that it is intended to describe the overall structure of successful debugging episodes by different individuals for different problems in a range of subject domains. Variations in the structure of any episode are due to characteristics of domains and problems and differences in the domain-specific knowledge of individuals. The model identifies the points at which these factors produce variations in problem-solving behavior. It is a very "weak" model in that is far from a sufficient computational model of troubleshooting/debugging in any domain. However, it is intended to be a logically sufficient description of a top-level organizational structure for a strong model. Our model draws upon prior research on debugging mentioned earlier, particularly the planning grammars of Miller and Goldstein (1976a, b)

The model is a representation of procedural knowledge and we have chosen to express it here as a type of procedural network (Figure 5). We considered, but dismissed, the possibility of using a production system formalism to represent the model. The primary factor in this decision was that production systems hide the control structure of a procedure by distributing it across the individual productions. A second factor was that production systems incorporate semantic tests at every point in the control structure-- they presuppose that all procedures are invoked conditionally-- while we found that we wanted to identify both conditional and unconditional calling relationships. The procedural network overcomes both of these difficulties. First, it explicitly represents the overall control structure of the model. Second, by annotating the connections between procedures, conditional and unconditional flow of control are conveniently distinguished. An ATN formalism also has a natural way of distinguishing conditional and unconditional paths of control. but we found it somewhat less heuristic for communicating the entire top-down structure of the model. We want to emphasize that the model could have been represented as a production system, but with less efficiency and comprehensibility.

The notation in Figure 5 requires some explanation. Each node

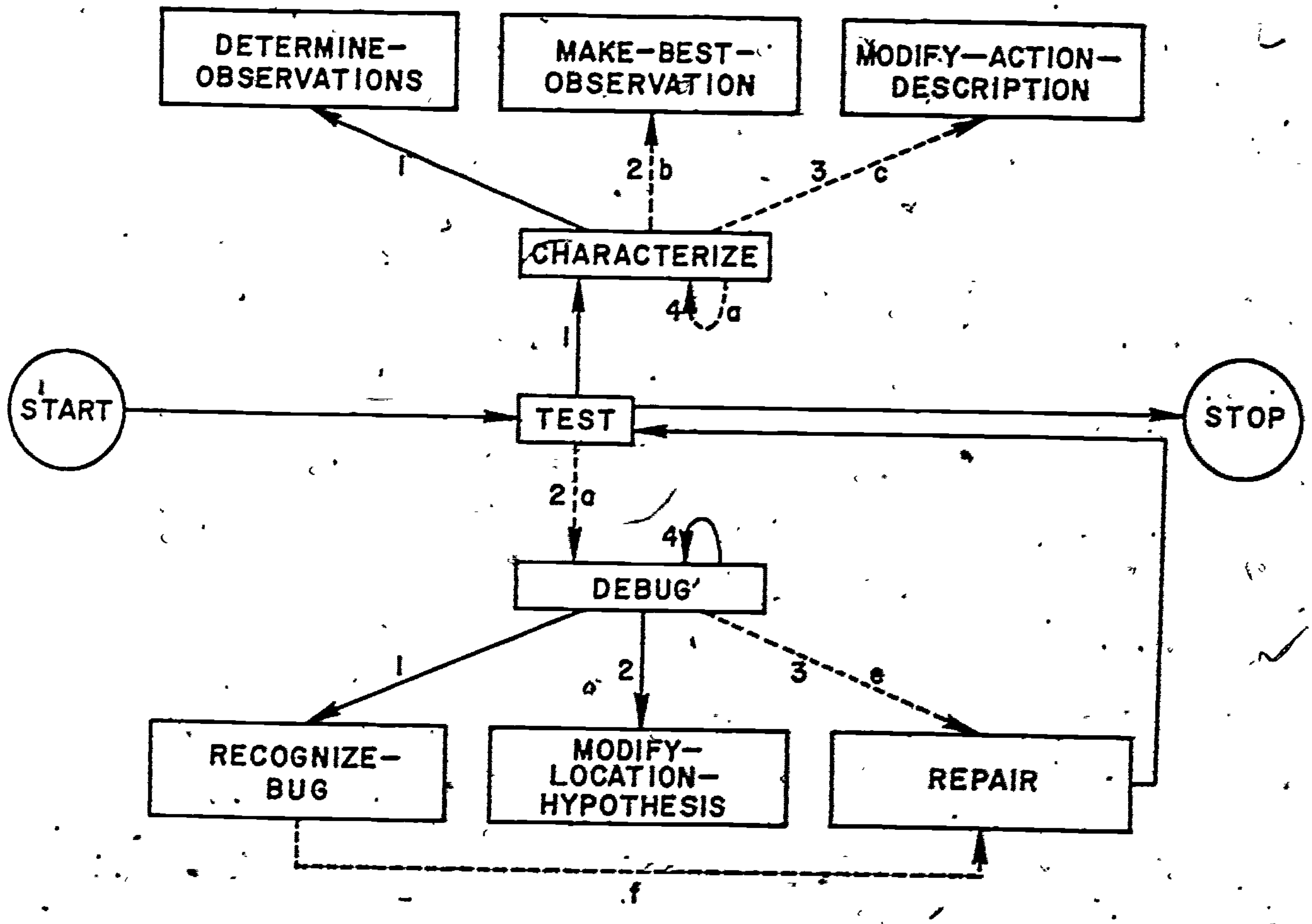


Figure 5. Procedural network for the top-level structure of a general strategy for troubleshooting/debugging.

in the procedural network is the name of a procedure defined by its function. An arrow from one node to another indicates that the procedure at the tail calls, or passes control down to, the procedure at the head. If a node has more than one arrow emanating from it, the calls from it are always made in the order denoted by the integers labeling the arrows. Solid arrows represent unconditional calls, while dashed arrows are calls made only if some semantic tests are first satisfied. Each dashed arrow is labeled with a letter. Table 1 summarizes the semantic tests for each of these labeled dashed arrows in Figure 5 and lists the global registers and data structures used by the model.[4] In tracing flow of control in the network, the following convention is in force: when a procedure finishes (when all its subordinate procedures finish) it passes control back up to the superordinate procedure that called it; that superordinate procedure then calls its next subordinate procedure (if any).

Since a general strategy for troubleshooting/debugging is a plan for solving problems, its representation as a procedural network can be viewed in the same way as the plans and procedural systems the strategy can be applied to. That is, the levels of the network are levels of the strategy's functional hierarchy. The hierarchy is incomplete in that it does not extend down to the primitive procedures needed to solve problems in any specific domain. The teleological structure of the hierarchy is complex (not linear) and is represented in part by the ordering on arrows emanating down from a node. A second part of the

---

[4]Since much of the communication among procedures is by global structures, the recursive procedure calls in the network in most cases do not increase the memory demands of the model beyond those that would be imposed by iterative calls.

Table 1

Summary of Global Data Structures and Registers and of Procedure Invocation Semantics of the Troubleshooting/Debugging Model Illustrated in Figure 5.

Data structures and registers

ACTION-DESCRIPTION :	list of propositions describing the relation between observed and normal plan/system actions.
LOCATION-HYPOTHESIS:	description of parts of plan/system where a fault may possibly exist,
ERROR? :	TRUE if no error has been detected or if a repair had been made and not yet tested; FALSE otherwise.
DELTA-I :	unidimensional value that is a function of the changes in the ACTION-DESCRIPTION over time.
I :	threshold value to which DELTA-I is compared to judge the expected payoff of determining further observations.
O :	threshold that determines minimum payoff of observations made.
R :	threshold that determines maximum cost of repairs made.

Procedure invocation semantics

arc-label (from Figure 5)

a: if ERROR? = TRUE
b: there exists at least one observation with an expected payoff greater than O.
c: if MAKE-BEST-OBSERVATION was called.
d: if ERROR? = FALSE and DELTA-I > I.
e: if the cost of repairing the parts denoted by the total LOCATION-HYPOTHESIS is less than R.
f: if the cost of repairing the part assumed to have been recognized as the location of the fault is less than R.

teleological structure is implicit in the semantic tests for conditional procedural calls. The tests involve global registers and data structures that are accessed and modified by procedures throughout the network. Thus, the invocation of conditional procedures and, in fact, the actions of both conditional and unconditional procedures depends not only on the actions of the calling procedure but on any procedures that have previously modified the registers and data structures. We make this point to emphasize that while the procedures contained in a general troubleshooting/debugging strategy may seem obvious, the relationships between them are not and may therefore cause the greatest difficulty in understanding and inducing how to apply the strategy by observing it in action.

We will now proceed to elaborate the model, describing the calling semantics and function of each procedure and indicating the different types of knowledge required and how they become available to the problem solver.

TEST. Every time a plan is executed or a system is activated, it is implicitly being tested. For instance, whenever you switch on a light, you are testing it and the circuit of which it is a part. If the light fails to go on, then debugging is initiated. More clearly, a technician engaged in routine maintenance consciously tests a system to see if he can gather data which may cause him to reject the hypothesis that the system is fully operational. Thus, the top-level procedure in the model is TEST. The model is always started at TEST. At that point, a register ERROR? is FALSE, indicating an assumption that there is no error in the system being tested. This is also reflected by the initial value of a data structure ACTION-DESCRIPTION which is NULL. TEST is also called by REPAIR.

TEST invokes CHARACTERIZE unconditionally. It subsequently calls DEBUG only if ERROR? is TRUE upon return from CHARACTERIZE.

CHARACTERIZE. The function of CHARACTERIZE is to collect data that allow modification of the ACTION-DESCRIPTION. If it adds a clause to the ACTION-DESCRIPTION that describes a discrepancy between observed and normal actions, then it sets ERROR? to TRUE if it was previously FALSE. This corresponds to detecting a bug during testing.

CHARACTERIZE does its work via three subprocedures, DETERMINE-OBSERVATIONS, MAKE-BEST-OBSERVATION, and MODIFY-ACTION-DESCRIPTION. The call to MAKE-BEST-OBSERVATION is conditional on whether there is a potential observation whose payoff (a function of its cost and expected information return) exceeds a minimum threshold, which we will denote O. This means that an observation is not made if it is too expensive or if it is not expected to alter the ACTION-DESCRIPTION significantly. The initial value of O is set by TEST and depends on the expected cost of a subsequent system failure if a bug is not detected and repaired. CHARACTERIZE also may call itself conditionally, if ERROR? is FALSE and a register DELTA-I, which reflects the rate of information change in the ACTION-DESCRIPTION is above a threshold I. This means that when CHARACTERIZE is called by TEST, either initially or after a repair, observations will be made as long as the ACTION-DESCRIPTION changes by the addition of propositions asserting that observed actions are normal or by the deletion of propositions asserting discrepancies noted previously between observed and normal actions. In general, this implies that characterization during testing continues until there are no more potential observations whose payoff exceeds O. Thus, testing does not necessarily continue until the debugger is logically certain the system is error-free, but only until his confidence leads him to believe that further observations have a higher cost than failure to detect a possible error would have.

DETERMINE-OBSERVATIONS. The first procedure called by CHARACTERIZE is DETERMINE-OBSERVATIONS, which identifies a set of potential observations. The observations are determined with respect to the current LOCATION-HYPOTHESIS, a data structure describing a subhierarchy of the plan/system which is initialized to the entire hierarchy and modified subsequently by the procedure MODIFY-LOCATION-HYPOTHESIS by deduction involving the ACTION-DESCRIPTION. The LOCATION-HYPOTHESIS represents the part of the plan/system to which a detected bug has been logically isolated or conversely may be viewed as the part of the plan/system which is not known to be bug-free given any prior observations. Each observation identified by DETERMINE-OBSERVATIONS has a potential effect on the ACTION-DESCRIPTION which can further reduce the extent of the plan/system denoted by the LOCATION-HYPOTHESIS.

Observations may be experimental, involving manipulations of the plan/systems parameters (if any). For example, they may require an electronics technician to change the external control settings of a device or a programmer to alter the data input to a program. In some domains, like management consulting, no experiments are possible and observations must be "naturalistic."

DETERMINE-OBSERVATIONS accesses the debugger's knowledge of the plan/system's functional hierarchy and its teleological structure in order to identify points where informative observations can be made. In some contexts (e.g., electronics troubleshooting), there may be external sources of that information (technical data). Otherwise the hierarchy must be built up from the lowest level using knowledge about primitive subparts and the laws that describe their interrelationships. Knowledge about higher-order subparts derived in this way may be stored in memory in a "library" which may allow the debugger to recognize that subpart if it appears in subsequent episodes.

<u>MAKE-BEST-OBSERVATION</u>. MAKE-BEST-OBSERVATION is second procedure called by CHARACTERIZE. As noted in the discussion of CHARACTERIZE, its call is conditional on there being at least one potential observation with an expected payoff exceeding 0. MAKE-BEST-OBSERVATION performs the observation with the highest payoff as determined from its cost and its potential for affecting the ACTION-DESCRIPTION. An observation expected to return a large amount of information may be passed over for a less productive one if the latter's cost is much lower.

MAKE-BEST-OBSERVATION accesses the debugger's knowledge of how to make observations (e.g., use of measurement equipment) and of their expected cost. Most knowledge of these costs is probably acquired through experience and may be stored in the same library as the knowledge used to recognize higher-order subparts. That library may also contain the knowledge of likely outcomes of observations used to estimate the effect of an observation on the ACTION-DESCRIPTION. This latter knowledge supplements information about the possible outcomes deduced from knowledge of the functional hierarchy of the plan/system.

<u>MODIFY-ACTION-DESCRIPTION</u>. This procedure modifies the ACTION-DESCRIPTION according to the observed actions and is called only if MAKE-BEST-OBSERVATION was called. The modification involves adding a

proposition to the description noting either a normal action or a discrepancy from a normal action. In testing subsequent to a repair, it may also involve deleting or modifying a proposition already in the description. Generation of the proposition requires access to knowledge for deducing the normal actions of subparts and structures of subparts.

DEBUG. DEBUG is the controlling procedure once an error has been detected. It is called by TEST if CHARACTERIZE has returned with ERROR? equal to TRUE. It calls the procedures RECOGNIZE-BUG, MODIFY-LOCATION-HYPOTHESIS, REPAIR, CHARACTERIZE, and itself. The call to REPAIR is conditional. Further details about DEBUG will be given following the description of its subprocedures.

RECOGNIZE-BUG. RECOGNIZE-BUG is a powerful procedure in that it can radically alter the overall strategy of logically localizing a bug at progressively lower levels of a plan/system's functional hierarchy. It accesses the ACTION-DESCRIPTION and matches it against a knowledge library of bugs and associated ACTION-DESCRIPTIONS encountered in past episodes with identical or similar plan/systems (idiosyncratic experiential knowledge). If a sufficient match is obtained to a known bug and the cost of repairing that bug is is less than a threshold R, then RECOGNIZE-BUG immediately calls REPAIR. If the cost of the repair is too high to be attempted at that time (R increases as a function of the number of times DEBUG has been called), then the old LOCATION-HYPOTHESIS is saved and a new LOCATION-HYPOTHESIS is set to be the level of the hierarchy at which the subpart containing the recognized bug is defined. This has the effect of focusing subsequent characterization on a "suspect" subpart. For example, when a mechanic first examines a car with an uneven idle, the ACTION-DESCRIPTION is "uneven idle" and the initial LOCATION-HYPOTHESIS includes the entire ignition and fuel systems. If he has knowledge that "uneven idle" is frequently due to dirt in a carburetor passage, and is familiar with the "trick" of jarring the dirt loose by striking the carburetor on the outside, then he may immediately try that repair. If he is not familiar with that inexpensive repair (or if he is and it doesn't seem to work) and is not yet ready to disassemble the carburetor or use a chemical cleaner, then he can set the LOCATION-HYPOTHESIS to be "fuel system" so that he can make further observations which will indicate whether or not there is some problem in the carburetor. If the problem is logically localized to the carburetor, then an appropriate repair will be made with the savings of not having made unnecessary observations to exclude the ignition system. However, if one of these observations on the fuel system should make the LOCATION-HYPOTHESIS logically inconsistent with the ACTION-DESCRIPTION (as detected by MODIFY-LOCATION-HYPOTHESIS), then the previous LOCATION-HYPOTHESIS must be restored and modified. Thus, for example, if further observations prove conclusively that there is no fault in the carburetor, then the LOCATION-HYPOTHESIS containing the ignition system and entire fuel system is restored and the problem-solving process continued from that point.

<u>MODIFY-LOCATION-HYPOTHESIS</u>. This procedure accesses the ACTION-DESCRIPTION and using knowledge of the plan/system's teleology deduces whether any of the subparts in the LOCATION-HYPOTHESIS logically can be excluded as candidates for containing the bug. This is illustrated by our earlier example of troubleshooting when your bedside lamp fails to light. Initially the LOCATION-HYPOTHESIS includes all the elements of the circuit. When the observation is made that the electric

clock is still working, the ACTION-DESCRIPTION becomes "light inoperable, current available at wall outlet."

MODIFY-LOCATION-HYPOTHESIS deduces from this that the fault cannot be in the external power supply, the fuse, or the intermediate wiring and modifies the LOCATION-HYPOTHESIS accordingly.

When the LOCATION-HYPOTHESIS is reduced to a single subpart at a level of the functional hierarchy, it is reset to contain the subparts in the level immediately below that subpart. For instance, in troubleshooting a circuit, if the LOCATION-HYPOTHESIS is reduced to "voltage regulator", it is reset to the level of the hierarchy comprising the immediate subparts of the voltage regulator. Thus, the bug is localized to progressively simpler (and structurally smaller) parts of the plan/system.

As noted in the discussion of RECOGNIZE-BUG, if MODIFY-LOCATION-HYPOTHESIS should deduce that the ACTION-DESCRIPTION is inconsistent with a bug anywhere in the parts of the plan/system denoted by the LOCATION-HYPOTHESIS, then a prior call to RECOGNIZE-BUG produced a false recognition and the LOCATION-HYPOTHESIS prior to that call is restored.

<u>REPAIR</u>. If the cost of repairing (replacing or modifying) the subpart(s) denoted by the LOCATION-HYPOTHESIS is less than the repair cost threshold R, then DEBUG calls REPAIR. REPAIR accesses the debugger's knowledge how the subpart(s) are designed and implemented to function as intended. For an electronics technician this may be something as basic as how a transistor is supposed to be connected and for a programmer how to write a format statement. On the other hand, a programmer may rewrite an entire sorting procedure if he determines that

there is a bug in the existing one and believes it is more efficient to rewrite than to try to localize the bug further. REPAIR also accesses knowledge about specialized "tools" like soldering irons or computer file editors needed to accomplish repairs in different domains.

REPAIR sets ERROR? to FALSE and calls TEST. If the repair corrects the fault then that call to TEST will eventually call STOP, terminating the problem-solving process. If the repair is incorrect, the call to TEST will eventually invoke DEBUG again.

<u>Continuing DEBUG</u>. If the cost of calling REPAIR exceeds R, then DEBUG calls CHARACTERIZE and then itself. Upon the return from CHARACTERIZE, the ACTION-DESCRIPTION will have been updated if an observation was made. If one was not, then DEBUG modifies both R and O. It increases R, so that there is a chance that REPAIR can be called even though the LOCATION-HYPOTHESIS cannot change because the DESCRIPTION-HYPOTHESIS is not modified. It decreases O, so that if REPAIR still cannot be called there is a chance that an observation will be made on the next call to CHARACTERIZE. Thus, when the process gets stymied, it frees itself either by making more expensive repairs than usual or by making observations that are more expensive or less informative than usual.

<u>Further comments on the model</u>. The strategy we have outlined here is a competence rather than a performance model. Deficiencies in any of the knowledge required may cause it to fail. In particular, the knowledge of each level of the plan/system's functional hierarchy and its teleological structure is crucial for modifying the ACTION-DESCRIPTION and the LOCATION-HYPOTHESIS. Note that it is not necessary to know the hierarchy from top-to-bottom but instead only down

to the level at which one is willing to pay for a repair. Thus, in working on a circuit one may understand (from technical data) the functioning of the voltage regulator with respect to other subcircuits at that level of analysis, but not understand the internal structure of the voltage regulator. The available knowledge is sufficient for localizing a fault to the voltage regulator and this may be adequate if one is willing to replace that entire subcircuit (knowing that only one primitive component may have failed).

The only explicit error recovery mechanism in the model is for false recognitions by RECOGNIZE-BUG that cause an inappropriate jump to a lower level of the hierarchy. The model backtracks from these errors by saving and restoring earlier copies of the LOCATION-HYPOTHESIS. Thus, these errors increase costs, but will not directly lead the process to complete failure in the way other knowledge deficiencies may.

Explaining the expertise of expert debuggers.

Given that a general strategy and different types of domain-specific knowledge underlie troubleshooting/debugging behavior in the ways suggested by our model, we can ask about the contribution each makes to expert performance. Is the expert an expert because he develops a superior general strategy and adheres to it religiously? Or, does his expertise stem more from his extensive knowledge of the problem domain, including the fundamental declarative knowledge, specialized procedures for making observations and repairs, and idiosyncratic libraries of information about important recurring high-level subsystems and about the bugs frequently associated with observed patterns of symptoms? The introspections of the expert in the dialog in Figure 2 are consistent with the latter explanation. He attributes his easy and efficient solution of a problem to his "familiarity" with the fact that an observed symptom is (almost) always associated with a particular fault, although he cannot articulate how he was able to access that fact.

In terms of our model for a debugging strategy, the expert in the dialog achieved a solution by calling his RECOGNIZE-BUG procedure, bypassing some of the progressive top-down localization and characterization which slowly converge on a fault by deductive reasoning. He supplemented his deduction with identification. Localization by identification is also exemplified by the mechanic who first jars the carburetor to attempt to remedy an uneven idle. These examples illustrate how by using a library of knowledge acquired through experience, the expert can choose to focus at a low level of plan/system's functional hierarchy without deducing the location of the bug from observations made at higher levels. However, since the information in the library may sometimes be applied in inappropriate contexts (false matches to ACTION-DESCRIPTIONS), the expert must backtrack and integrate observations made at several levels in the system much more frequently than required when localization is strictly top-down and deductive. Failures in backtracking can reduce efficiency by causing the expert relying on identification to make redundant observations.

This explanation of debugging expertise seems to be consistent with data we collected from five programmers with different degrees of experience.[5] They ranged from one with a masters degree in computer

[5] When we say "programmer" we do not necessarily mean an individual who was trained and works specifically as a programmer. For

science and several years of advanced programming experience to a total novice who had no formal instruction in programming and only a few weeks of self-instruction. Within this range were students and professionals with from one to several years of instruction and practical programming experience.

We asked these programmers to debug a short BASIC program and to write a commentary on their reasoning as they worked. They had access to a BASIC interpreter for running and modifying the bugged program. They were provided with the program description, listing, and sample input-output shown in Figure 6. It is a sorting program designed to interact with a user at a terminal. It accepts numbers one at a time, acknowledging each by printing "BON APPETIT", until the user types a zero signifying the end of the list. The list sorted in ascending order is then printed. The program was written by a member of the research team and is deliberately obscure so that it would not be completely trivial to experienced programmers despite its brevity. A more elegant solution is possible using fewer variables and less complicated parameters for FOR..NEXT loops. However, the program shown is correct except for a single character. Line 100 should be X = C - J + I instead of X = C - J + 1. Note that such an error could be the result of a simple reading or typing error in entering the program, rather than a design error. However, it is not the type of error that can be detected by the parser or runtime system of a programming environment; it is a logical error that causes the program to produce incorrect results when it is executed, as can be seen in Figure 6. In fact, the effects of the

---

our purposes a programmer is anyone who writes programs incident to his job or (his activities a student).

Description

This program inputs numbers (up to 100 numbers), sorts each number into ascending order as it is input, and prints the ordered inputs when a key value of zero is input.

```
10 DIM N(100)
20 C = 1
30 INPUT N(C)
40 IF N(C) = 0 THEN 180
50 FOR I = 1 TO C
60 IF N(C) >= N(I) THEN 140
80 D = I
90 FOR J = D TO C
100 X = C - J + 1
110 N(X + 1) = N(X)
120 NEXT J
130 N(I) = N(C + 1)
140 NEXT I
150 PRINT "BON APPETIT"
160 C = C + 1
170 GOTO 30
180 FOR Q = 1 TO C - 1
190 PRINT N(Q)
200 NEXT Q
210 END
```

Sample input/output:

```
*RUN
EXECUTION OF YOUR PROGRAM

        :77
BON APPETIT
        :15
BON APPETIT
        :34
BON APPETIT
        :10
BON APPETIT
        :88
BON APPETIT
        :0
:
10
15
0
77
88

EXECUTION COMPLETED AT LINE 210
```

Figure 6. Debugging problem given to five programmers of varying experience.

bug depends on the input and if the user enters a list in perfect reverse order (e.g., 81, 54, 33, 12), then the program correctly sorts it. A complete ACTION-DESCRIPTION of the bugged program at its top level is that if an incoming number belongs at the beginning or the end of the existing list it is appended correctly, but if it needs to be inserted between two previous values, it is lost and replaced by a zero.

All of the five programmers who participated were able to debug the program, although the amount of time they required varied from less than an hour for the most expert to several days (a few hours each day) for the novice. Also, they varied in how much of the program code they modified in order to eliminate the bug. Figures 7a, 7b, and 7c are segments of the written commentaries generated by the expert, an intermediate programmer, and the novice, respectively. The concepts the expert uses to describe the program reflect his specialized knowledge of sorting algorithms, which he used to identify functional subsegments of the program. He immediate tried to identify bugs at a low level of program structure -- in storage allocation and in incrementing counter variables. Although he examined the sample output at an early point, he does not articulate an ACTION-DESCRIPTION. The structure of the episode reflects repeated attempts to use specialized knowledge to predict and search for likely types of bugs: an emphasis on recognition as opposed to localization. His repair, not shown in the commentary, was to rewrite the nested sorting loops in a more straightforward form, eliminating the unnecessary variables. Thus, he made a judgement that it would be more efficient to substitute a block of code, than to try to isolate the bug further and then make a more minimal modification.

Figure 7b is the commentary of a programmer with several years

Read description.

Check DIM statement.

Look at output.

Bon appetit?

C points to next empty word in N.

When input = 0, then 180; print N(1) to N(C-1) or stop. Look for C ← C+1. Found on 160.

Sort takes place between 50 and 140. If $N_c \geq$ all of the N_i's, nothing is done.

Get to 80 if $N_c < N_i$.

The J loop (90-120) is supposed to move all of the entries between N_{i+1} and N_c up one.

It does some sort of inversion. I think this is unnecessary, and also the variables D and X. I am going to try it out on the terminal.

Attached is the program I typed in, before I tried to run it. (Note that it differs from my scribbled notes. I didn't look at the notes while I was typing it in.)

I tried sorting the numbers

9, 4, 6, 1, 7, 13

and it worked.

Figure 7a. Debugging commentary on sorting program by a formally trained, highly experienced programmer.

Think about what it should do. Visualize number coming in and finding its way to the top of list.

Note that 0 is in the wrong place and 34 is missing -- in fact, 0 is in the 34 place.

34 is the 3rd input (check first to see if it's first or last -- no).

Now go to program.

Since 0 was wrong, check the branch to 130. Don't see anything wierd there -- output loop looks OK.

Look at sort loop (50-140 I guess), which skips if current (new input) is geq.

Inner loop looks complicated, in fact, bizarre. Have to puzzle it through -- purpose is to shove everybody down to insert new guy. I knew that because it's a loop (should have known it anyway).

100 puzzling -- next one in the list. Push it one lower down. Looks bad -- N(x+1) ← N(x) will lose the previous value of N(x+1). (hypothesis): Test with given inputs by tracing values of everything. Not working -- I'm not getting anything pushed anywhere except out of sight -- the 15 is moving higher instead of the 77. Try again.

Note: D is unnecessary since it doesn't change inside the J loop -- I wouldn't be destroyed if 90 said for J = I to C. Think about that: is this strategy reasonable -- yes, I have a place where the current input is lower than this element, so I have to change everybody from here on up. So J = I to C is fine.

But I immediately get N(3) ← 15 which is dumb. (Think about whether BIP thinks X = C-J+1 gives 2 or if 0, then array error would have occurred, so must be 2.)

Seems that N(2) = 77 N(3) = 15 which is backwards. Keep going. C changes to 3 but that should have zapped the 15 in N(3). No -- missed 130 which stuffs 15 back into 1!! Seems that I get 2, N(2) compares. NO: OK because N(C) is no longer the current input.

Looks like I just lost the 77 but that's impossible, so have to try again with 34 input.

Stymie.

Wait. Strategy makes sense -- put new one up high, move the others up one at a time. N(4) gets the 34, so N(3) is available for 77 to move to. Value of X should be 2 to make that move. X ← C-J+1 is 3.

NO. The first time was wrong. Back up to 90 again with I = 2 (i.e., comparing 34 with 77).

Figure 7b. Excerpt from debugging commentary on sorting program by a programmer of intermediate experience.

No, wait a second, I guess it's O.K. for it to print 0; it should just print it first. Maybe that's where the problem is. Even though I still haven't figured out how the program is supposed to work, I'll check out the parts that deal with switching from the ordering procedure itself to printing the final output, and see what I can find.

Of course! Zero is going to be the last value entered, so it will have the highest numbered subscript and get printed last. That's certainly one of the problems with this program, although it may not be the only one. I'll have to figure out how the whole thing is supposed to work. But I have a hunch that if line 40 ("IF N(C)=0 THEN 180") gets moved down between lines 160 and 170 (the end of the main subscript-reassigning loop) then the program will work the way it should. This move should assign 0 the lowest subscript before telling the machine to print out, assuming the rest of the program was written correctly. Let me check that out...

```
160 C=C+1                          --so that's how the subscript
                                     gets incremented.
```

(Break--overnight)

I just realized that although 0 was initially assigned the highest subscript (I think), it was not printed last when the numbers were printed out in subscript order (lines 180 to 200). This means that the subscripts of some of the other numbers (namely 77 and 88) were higher than the subscript for 0 by the time the printing was done. I'm going to try to run through the program mentally, feeding in the numbers used in the run shown here, to see what this program is doing.

```
when you get to INPUT 34,
                                  N(1)=15      N(2)=77
N(1) no longer has a value→N(2)=15      N(3)=77←34 is lost here
N(2) no longer has a value→N(3)=15      N(4)=77
```

Although I still don't grasp it completely, the strategy of this program seems to be to reorder the subscripts of the input numbers by comparing each new number one at a time with each of the numbers which have already been typed in and:

--If the new number N(C) is less than the number it is being compared to, N(I), this increment the subscript of each of the numbers that are greater than or equal to N(I) and give the new number the old subscript of N(I).

--If the new number N(C) is greater than the old number N(I) to which it is being compared, leave the subscripts alone and compare N(C) with the next old number.

Figure 7c. Excerpt from debugging commentary on sorting program by a novice programmer.

of practical experience, but little formal instruction. There is more evidence of a general debugging strategy and less use of specific knowledge about sorting than in the commentary in Figure 7a. Before looking at the structure of the program, the programmer tries to describe the bug's symptoms in the program output-- to develop an initial, top-level ACTION-DESCRIPTION. The observation "that 0 is in the wrong place" (she incorrectly assumes that the 0 in the output is the 0 the used typed to end his input) leads her to locate and characterize the segment of the program designed to stop the input cycle when a 0 is typed by the user (she sets her LOCATION-HYPOTHESIS to that segment). When she (mentally) tests that segment and observes no evidence of a bug, she focuses on the nested loops that sort the array (resets the LOCATION-HYPOTHESIS to the top-level). Her reasoning in using the zero in the output represents a call from DEBUG to RECOGNIZE-BUG in which an incorrect ACTION-DESCRIPTION was matched to information in a library of bugs and their manifestations.

The programmer's experience allows her to judge that some of the code in the sorting loop is "bizarre", and thus a likely location of the bug. She characterizes that code by mentally tracing execution and observing how an actual (as opposed to abstract) set of numbers are moved within the array. She has some difficulty in generating an ACTION-DESCRIPTION from her observations and therefore tries to apply some knowledge she does have about sorting when she decides that line 110, N(X + 1) = N(X), looks suspicious (another call to RECOGNIZE-BUG). In some sorting algorithms, such a transfer of values might lose the contents of an array location, but in this case the line is correct. (One of the other intermediate programmers also suspected this line).

Thus, she has to backtrack from this attempted identification of the error and resumes her characterization of a segment of the nested loops. Eventually, she did identify and repair the bug. Like the expert, she tried to use specific experiential knowledge to shortcut a top-down analysis via identification, but did not have the knowledge needed to succeed on that basis. (She might have solved the problem more quickly, but unlike the more expert programmer, chose to localize the bug within the sorting loops rather than rewrite them completely.)

Figure 7c is one of six pages of commentary generated by the novice programmer (who later displayed better than average programming skills for his degree of experience). He worked on the program in four separate sessions and eventually did debug it. His commentary reveals how his unfamiliarity with the fundamentals of the BASIC programming language and of program organization made it an effort for him to perceive the program at a high level. He begins by spending considerable effort characterizing the code line-by-line with no good idea of what he is looking for, since he fails to generate an ACTION-DESCRIPTION beforehand. In fact, he did not report looking at the input-output data before the second session. Prior to his successful solution he attempted several "irrational" minor repairs based on his misunderstanding of single lines of code and their function in the program. Like the most expert programmer, he searched for bugs by examining the code, but unlike the expert he had no basis for making rational predictions for what the bug might be.

In general, these data are consistent with the view that the expert debugger is an expert-- that he solves problems with minimal expense-- because he has a great deal of experiential knowledge that allows him frequently to follow cost-saving alternative pathways within a general debugging strategy, as represented in our model by the procedure RECOGNIZE-BUG. It is not seem necessary to postulate that he has a general strategy superior to that of somewhat less skilled debuggers in order to explain his expertise. Instead, he simply seems better able to exploit the benefits of an identification substrategy which even novices try to use.

Weaknesses in the debugging of inexperienced programmers

The commentary in Figure 7c shows that an inexperienced programmer can have considerable difficulty with a debugging problem because of the effort required to understand how the program is supposed to accomplish its intended function. Of course, programmers most often encounter their debugging problems in programs which they themselves designed and implemented, and thus can understand. However as we noted earlier, programmers sometimes knowingly implement and run programs that are incorrect, finding it more efficient to develop correct code by debugging, than to derive the correct code initially by logical analysis. In these cases, problems in debugging can arise because of difficulties in knowing how to design code for repairs, rather than in locating the bug. Sometimes, presumed understanding of some code can actually impede programmers' debugging of their own programs. If they write code they are certain is correct and manage to insert a bug in it, then (1) that code is the last place they will look for the bug, despite observations that might indicate that it is a likely location and (2) when they do look at the code, they may miss the bug, because they see what they intended the code to do and not what it actually does. Thus, a programmer debugging his own program may lose some objectivity, while one debugging another's program may have fundamental problems understanding how the program is supposed to work. There are some programmers in real contexts who are faced with the problems of debugging programs written by someone else: for example, consultants and members of teams working together on a large project. They lose the advantage a designer's knowledge of his program, but by the same token are less prone to "blindness." They may face situations where they have difficulties debugging a program because they don't have the knowledge needed to understand it, rather than because they have inadequate debugging strategies. In other troubleshooting/debugging domains, like electronics and mechanics, technicians routinely face problems with devices unfamiliar to them. In these situations, they must turn to technical data for the devices or be able to synthesize the device's structure from the bottom up, if they are to effect a repair.

We have proposed that expert debuggers have general, top-down debugging strategies, but that their expertise is defined by their mental libraries of domain- and problem-specific knowledge gained through their experiences. Inexperienced programmers obviously lack comprehensive libraries. But is this the sole source of their difficulties, or are their general debugging strategies also deficient so that they do not make the most effective use of the specific knowledge they do have. This is an important question from the viewpoint of instruction, since it would be more feasible to try to teach a well-defined general strategy, than a large, ill-defined corpus of specific knowledge. The commentary of the novice programmer debugging on the sorting program does seem to reflect a strategy less efficient than those we found in the commentaries of more experienced programmers who also had difficulties with the problem. However, the knowledge required just to understand that program was so far beyond the experience of the novice that it could have been the case that he had a good strategy available but had trouble executing it.

In an attempt to determine whether inexperienced programmers have difficulties debugging because they lack an effective general debugging strategy, we examined programming data collected from students learning to program. The data originated from three groups who had participated in the BASIC Instructional Program (BIP) 1975, a CAI system for teaching introductory BASIC programming to people with no prior computer experience. In all there were data from 100 college students, who wrote on the order of 40 short BASIC programs each during 10-15 hours of terminal time in BIP. The original use of the data had been in evaluating BIP's effectiveness for teaching programming and in examining the way students used some of BIP's subsystems for writing and debugging their programs.

The data are quite comprehensive records of students' interactions at the terminal, which we will call chronologies here. The information contained in the chronologies for the three different groups varies somewhat, since analyses of the earlier versions had suggested improvements in format and content. For instance, the first and second groups of chronologies do not directly indicate the order in which lines of code were entered by student; the code was recorded on the chronology when the student listed or ran his program and it was possible to determine small changes in the code by comparing successive listings. In the third group of chronologies, each time the student typed a line of code it was written to the chronology and, in addition, when he listed and ran the program, the order in which the lines had been entered was stored with the listing. Since we found this information to be useful, our analyses focused primarily on the third group of chronologies.

Figure 8 is an excerpt from a chronology. In general, chronologies record the sequence of BIP commands and lines of program code typed by students when they worked on their programs. The commands include:

LIST	- lists the student's program
RUN	- executes the program
DEMO	- executes a model solution stored for the task the student is working
HINT	- prints a hint stored for the task
TRACE	- executes the student's program and prints for each line the values of any variables that changed
FLOW	- executes the program one line at a time, showing how variables change, and using the CRT to indicate the flow of control graphically
MORE	- executes the program and the model solution on test values and compares their output in order to judge whether the student's program is correct

Lines of code entered are denoted by the keyword LINE, or SYNTAX ERROR if the student typed an incorrect line that could be detected by the parser. Each entry in a chronology includes the time at which the command was typed by the student. It does not always include the exact response of BIP to that command. For example, while LIST does put the program listing on the chronology, HINT only puts the hint number, not the text of the hint.

The chronology data constitute an indirect window onto students' reasoning as they designed and debugged their programs. For example, if a student ran a DEMO after partially coding his program, it might be

```
run
        5/9/77 10.16:51
        same program
        output: TYPE IN TWO ODD NUMBERS, THE LOWER ONE FIRST.
        input: 1
        input: 5
        output  THERE ARE 7 NUMBERS BETWEEN THEM.
        completed at line 42

flow
        5/9/77 10:17 17

        output: TYPE IN TWO ODD NUMBERS, THE LOWER ONE FIRST.
        input: 21
        input  23
        output: THERE ARE 25 NUMBERS BETWEEN THEM.
        aborted at line 42.

list
        5/9/77 10:19:09
        same program

line
        5/9/77 10:19:42
        41 PRINT "THERE ARE ",C," NUMBERS BETWEEN THEM"

demo
        5/9/77 10:19:47
        order   program listing
        30      01 C=0
        19      10 PRINT "TYPE IN TWO ODD NUMBERS, THE LOWER ONE FIRST."
        2       15 INPUT L,H
        3       20 IF L=H OR L>H THEN 100
        7       26 IF L/2-0.5<>INT(L/2) THEN 150
        20      27 IF H/2-0 5<>INT(H/2) THEN 155
        9       30 FOR I=L TO H STEP 2
        10      35 C= C+1
        11      40 NEXT I
        31      41 PRINT "THERE ARE ",C; " NUMBERS BETWEEN THEM"
        27      42 STOP
        29      100 IF L>H THEN 110
        13      101 PRINT "YOU TYPED IN THE SAME NUMBER TWICE, TRY AGAIN
                                                                  WITH "
        14      102 PRINT "DIFFERENT NUMBERS."
        18      103 GOTO 15
        16      110 PRINT "YOU SHOULD TYPE THE LOWER NUMBER FIRST, TRY
                                                                 AGAIN."
        17      111 GOTO 15
        25      150 PRINT "THE LOWER NUMBER WAS NOT ODD, TRY AGAIN."
        23      151 GOTO 15
        24      155 PRINT "THE HIGHER NUMBER WAS NOT ODD, TRY AGAIN "
        26      156 GOTO 15
        28      999 END
        *
        input  21
        input: 23
```

Figure 8. Excerpt from a BIP chronology. The student was debugging a program at this point.

(Figure 8 continued)

```
line
        5/9/77 10:21:03
        41 PRINT "THERE ARE ";C," NUMBERS BETWEEN ";L ;" AND ";H

run
        5/9/77 10:21:08
        order    program listing
        1        01 C=0
        2        10 PRINT "TYPE IN TWO ODD NUMBERS, THE LOWER ONE FIRST."
        3        15 INPUT L,H
        4        20 IF L=H OR L>H THEN 100
        5        26 IF L/2-0.5<>INT(L/2) THEN 150
        6        27 IF H/2-0.5<>INT(H/2) THEN 155
        7        30 FOR I=L TO H STEP 2
        8        35 C= C+1
        9        40 NEXT I
        23       41 PRINT "THERE ARE ",C;" NUMBERS BETWEEN ",L ;" AND ";H
        11       42 STOP
        12       100 IF L>H THEN 110
        13       101 PRINT "YOU TYPED IN THE SAME NUMBER TWICE, TRY AGAIN
                                                                   WITH "
        14       102 PRINT "DIFFERENT NUMBERS "
        15       103 GOTO 15
        16       110 PRINT "YOU SHOULD TYPE THE LOWER NUMBER FIRST, TRY
                                                                 AGAIN."
        17       111 GOTO 15
        18       150 PRINT "THE LOWER NUMBER WAS NOT ODD, TRY AGAIN."
        19       151 GOTO 15
        20       155 PRINT "THE HIGHER NUMBER WAS NOT ODD, TRY AGAIN."
        21       156 GOTO 15
        22       999 END
        *
        output  TYPE IN TWO ODD NUMBERS, THE LOWER ONE FIRST.
        input. 21
        input. 23
        output  THERE ARE 2 NUMBERS BETWEEN 21 AND 23
        completed at line 42

list
        5/9/77 10 21 32
        same program

line
        5/9/77 10 21 45
        01 C=-2
```

that he has become confused about the problem or alternatively that he is designing and implementing the program in sections and is ready to examine the requirements of a new section. Looking past the DEMO in the chronology, if the student changes code he had already entered, we would opt for the first interpretation; if he entered some new code, we would choose the second. Tracing through a chronology and trying to reconstruct what the student's strategy was resembles the task of an archaeologist working to infer the values and motivations of a society from physical artifacts.[6]

Our initial examination of the chronologies was directed at identifying debugging episodes involving logical bugs. These are bugs that let the program execute but cause it to produce incorrect output (e.g., GOTO an incorrect line number), as opposed to those that are syntactic or context-free and are detected by BIP's parser or runtime system (e.g., GOTO a non-existent line number). Further, we looked for episodes where it seemed that the student considered the program to be completed at the time he detected the bug, as opposed to episodes where the debugging seemed to be integrated into design-- i.e.. where the student was trying to discover how some unfamiliar programming construct worked. These distinctions had to be inferred by looking at how the program was coded and by how readily the student seemed to change the program. One sure clue that a student thought a program was complete was his calling BIP's solution checker with the MORE command. Since BIP requires that a student RUN his program before MORE will be executed, typing MORE implies the student had RUN his program and thought it was correct.

[6] It has the same potential pitfall that the researcher's own world view restricts the interpretations he might see.

Most of BIP's programming tasks involve interactive programs that process input from a "user." In programs where flow of control was conditional on user input, we found many episodes where the student ran his program with inputs that did not cause a bug to manifest itself and then typed MORE. In most cases, the inputs used by the solution checker did detect the bug, although sometimes the solution checker incorrectly accepted a student program with a bug in it. Thus it became evident early in our examination of the chronologies that many students did not recognize the need to test programs across conditions that would exercise the different branches of conditional control structures.

Our original plan was to analyze the debugging episodes we found by parsing them with context-free debugging grammar similar to that found in Miller and Goldstein's (1976b) (Figure 3) planning/debugging grammar for LOGO programming. One feature of the context-free grammar rules is that a particular higher-order node (left-hand side of rule) may be expanded in terms of alternative lower-order nodes (right-hand side). One of our goals after parsing the episodes was to examine alternative expansions of a higher-order node to determine what semantics of the context determined the choice among alternatives. Thus if there were a rule

    repair := replace-code | modify-code

(the "|" is read as "or") we would be looking for features of a context that predicted when a bug was repaired by replacing old code and when it was repaired by some minimal editing of existing code. By determining the semantics, a more powerful ATN grammar then could be developed for describing the episodes. Once the general grammar describing the debugging strategies was formulated, our plan was to characterize the differences between students in terms of alternative subsets of the general grammar they employed and, in particular, to see if the poorer debuggers were those with degenerate versions of the debugging grammar.

The effort to derive a grammar encountered problems immediately. At the lowest level, where we were trying to identify rules mapping onto the chronology keywords (RUN, LIST, DEMO, LINE, etc) and the timing information, we found an unexpected degree of variability both within and between students. For instance, by examining several episodes we might derive

```
test-repair := RUN + <long latency> |
               RUN + <long latency> + test-repair
```

(the "+" is read as "and then") as a general rule of the grammar. However, in some other episodes we would then observe students changing a line of code, listing the program, and then changing that line again without ever having RUN the program to test the first repair. We soon realized that because the programs were on the average short (a maximum of about 30 lines) that student might have been testing the programs by looking at a listing and mentally tracing its execution rather than running it. We could have added this alternative to the rule for test-repair, except that LIST + <long-latency> occurred in other rules as well. In fact, LIST following a repair was a common "cliche" in students' behavior: evidently each time they changed some code, many students listed the program and looked at it briefly, simply to verify that BIP had inserted the code as they intended. Although, the time spent for such a visual check is less on the average than that spent mentally executing a program, the observed times overlapped enough to make use of the time data to distinguish these cases unreliable. In other contexts, LIST often did not occur when it was expected; we hypothesize that for shorter programs, an earlier listing could still have been on the CRT after a few intervening events. Thus, LIST was not a reliable indicator of when the student had been examining the program, and when it did occur even examination of the surrounding context was insufficient to determine the type of thinking the student was engaged in. It soon became clear that even the lowest level rules in the debugging grammar would be complicated by alternative and optional patterns of keywords, and that the same patterns would be included in several rules. In most episodes, the only way to piece together what a student's strategy had been was to integrate semantic clues from throughout the episode, and even that involved making sometimes tenuous inferences. We found therefore that it does not seem possible to derive reasonable debugging grammars, of the type proposed by Miller and Goldstein, for describing a range of episodes in the BIP student chronologies.

Even though we were unsuccessful at describing the debugging strategies of different BIP students in terms of a unifying information-processing model, the episodes we examined were very informative with respect to identifying weaknesses in the debugging of these inexperienced programmers. As we mentioned, there were frequent failures to test programs thoroughly when they were first run. This failure generalized to testing after repairs as well. Although there were ambiguous cases, in most instances the subsequent context made it clear that RUNs we judged we should have found, were not being replaced by mental execution of the program. As a result of inadequate testing, students failed to detect bugs in their programs. Not surprisingly, we also observed that even when students did detect bugs by running their program, they did not rerun the program with varying inputs, which by exercising different parts of the program's control structure would produce output useful for localizing the part of the program containing the bug (i.e., they did not CHARACTERIZE).

One of the most striking failures to test and characterize that we found in the chronologies involved the program shown in Figure 9. It is one student's attempted solution to BIP's task CALCULATOR, which specifies an interactive program for (1) getting two numbers from the user, (2) getting a numerical code corresponding to one of the four primary arithmetic operations (+, -, *, /), and (3) printing to the terminal the result of applying the specified operation to the two numbers. The student's program has a fundamental flow-of-control bug(s), which results in execution "falling through" the code for computing and printing the results (lines 80 to 150). Thus, for example, if the user typed a "1" to specify addition, the program branches correctly to line 120 to do the addition, but incorrectly continues on to do subtraction, multiplication, and division. Similarly, for subtraction, the multiplication and division are computed as well. Only for division, the final branch in the control structure, does the calculator compute and print only what it is supposed to. To correct the program, three lines, 125, 135, and 145, all of which should be STOP or GOTO 199, must be inserted. The student who wrote the program failed to debug it; in fact, he failed to detect the bug at all, although he called the solution checker and had it reject the program six separate times!

The major cause of the student's difficulty was that every time

```
10 PRINT "THIS IS A CALCULATOR"
20 PRINT "TYPE 1 TO ADD, 2 TO SUBTRACT, 3 TO MULTIPLY, AND 4 TO DIVIDE"
30 INPUT C
40 PRINT "NOW CHOOSE A NUMBER"
50 INPUT X
60 PRINT "NOW CHOOSE ANOTHER NUMBER"
70 INPUT Y
80 IF C = 1 THEN 120
90 IF C = 2 THEN 130
100 IF C = 3 THEN 140
110 IF C = 4 THEN 160
120 PRINT "THE SUM IS "; X + Y
130 PRINT "THE DIFFERENCE IS "; X - Y
140 PRINT "THE PRODUCT IS ", X * Y
150 PRINT "THE QUOTIENT IS "; X / Y
199 END
```

Figure 9. A student's solution to BIP's task CALCULATOR. The program has a recurring flow-of-control bug, which the student failed to detect because of inadequate testing.

he typed RUN to test his program, he specified "4" as the operation code. Not once did he test it with another operator. Since, "4" for division is the only case in which the program works correctly, he never saw the bug manifest itself in the output. In between running the program and calling the solution checker, we found that he used LIST and spent long periods before his next RUN. Assuming that he looked at the listing during these periods and because three similar lines were missing, we can conclude that he did not understand how to design conditional control structures and had not simply made a careless error. However, if he had RUN the program just once with a code other than "4", the erroneous output could have served to help him understand the defect in his design.

We found many other examples of inadequate testing and characterization. In fact, there was evidence that even when they ran the program and it produced incorrect output, some students did not realize that there was a bug in the program. In these cases, the students called the solution checker immediately following their RUN of the program, suggesting either that they had not analyzed the output or that they did not understand what the program they wrote was supposed to do.[7] Based on independent observations we made, we believe that in many cases the students did not analyze the output. A member of the research team spent about 20 hours observing (and assisting) course consultants and students discussing problems for the introductory ALGOL programming class at Stanford. These students probably have a higher aptitude for programming on the average than the BIP students and work on programming

[7] The solution checker at that time did not attempt to tell the student what it had found wrong, so that it was not called as way to obtain information.

tasks more complex than those in the BIP curriculum. They usually came to the consultants when they had trouble debugging their programs. In a large number of cases, students had not looked at their output other than to note that the program did not work. Instead, they described their debugging as going through the program line-by-line looking for a mistake, even though they had not thought about what was wrong. For errors trapped by the ALGOL runtime system (e.g., illegal memory reference) their debugging was even more irrational, since they did not attend to system diagnostics which could have identified the type of statement containing their error or, in some cases, the actual line containing it. Thus, the general debugging strategy of the ALGOL students we observed was deficient in testing and characterization in much the same way as that inferred from the chronologies of the BIP students.

Another type of poor debugging strategy we observed in the chronologies involved students making a series of several minor, sometimes completely non-functional, modifications to their programs in a very short period of time. Most often, this behavior was seen in the same episodes where there was no attempt to characterize the bug by running the program with varying inputs. A related failure was that attempted repairs that did not correct a bug were not undone at once and evidently were forgotten. As a result, "almost correct" programs sometimes were rendered less correct during debugging as the student compounded the original bug with others resulting from the ineffective repairs.

In order to substantiate some of the inferences we had drawn from the chronologies, we collected written debugging commentaries from inexperienced programmers working on staged debugging problems. The procedure was similar to that under which the commentaries were obtained from programmers debugging the sorting program. Four students who had completed 10 hours in the BIP course about one-half year earlier participated in the study. Each worked to debug a series of programs within the BIP programming environment. The programs themselves were selected from the chronologies and involved different types of bugs: computation, assignment, flow-of-control. This meant that the students were debugging programs written by other inexperienced programmers as solutions to problems they had themselves attempted in BIP.

The students were instructed to maintain a written record of their thoughts as they tried to debug the programs. In particular, they were told that whenever they decided to take an action-- LIST or RUN the program, or make a repair-- they should record their reasoning. BIP chronologies were saved for the debugging sessions and in addition the sessions were conducted on hardcopy terminals, instead of CRTs, so that exact typescripts of the interactions could be obtained. For each debugging problem, the students were given a listing of the program and a description of what is was supposed to do, but were given no sample input-output data. A copy of the program was preloaded into their program space in BIP, so that they themselves did not have to type it in in order to run or manipulate it. They had at most an hour to work on each problem.

In describing the results of this study, two general observations must first be noted. The subjects had not done any programming since the time they finished BIP and their behavior and the commentaries indicate they had forgotten features of the BASIC language and of how to use BIP. Therefore, much of their effort, especially on the first few problems, was spent using the BIP manual and trying to relearn fundamentals. Second, the subjects had trouble maintaining an ongoing commentary. They would work on the problem for a while and afterward write, rather than write as they were thinking. The observer provided constant prompts to remind them to write and they were encouraged to write and not to concern themselves with working quickly. Nonetheless, the commentaries are fragmentary records at best and are more retrospective accounts of what the subjects were thinking than they are real-time records.

The commentaries substantiate and elaborate our observations on the inadequate debugging strategies we saw in the earlier chronologies. Again, the most salient deficits were in testing and characterization, in obtaining and using information from a bugged program's input-output relationships. From the commentaries we could determine that when students listed and examined a program, they were not substituting mental execution for an actual computer run, but were scanning individual lines of code for errors. Now, since the subjects were debugging programs written by someone else, it is not necessarily a bad strategy to list and examine the bugged program in a global way in order to determine its overall organization. However, there were several cases in which subjects reported looking at lines of code for errors, when they had not yet run the program and seen how the error manifested itself, as illustrated by the following excerpt:

> I've done this program before, so I feel confident that I'll be able to find the bug. After one reading I've no idea what the problem is. I just looked at the two input statements, they look OK. I just looked at the 50 statement. Nothing looks wrong there. I'll run the program to see if there's a problem.

I just read the output on miles per gallon. Thought: I've got it! The machine divides before it subtracts. I'll try putting in parentheses around E-B to see if it will subtract first.

This subject recognized the program and thought he could find the bug just by looking at it. He examines the program listing line-by-line without success. Then he runs the program, sees the nature of the error, and is immediately able to locate the bug.

The commentaries indicate the mechanism for the quick and apparently unmotivated repairs we had seen students make in the chronologies. Consider the following excerpt:

This equals business in 160 to 230 is confusing stuff. Seems to me they're double assigning things. H and L are being given two values. I think maybe 160 and 170 can be deleted. Try and see.

The subject, without having run the program, examined the code and saw something that looked "confusing." Consequently, without any sound reason for doing so, she deletes two lines. This compounded the bug in the problem, so that when she tested the program (for the first time) after the repair and it worked incorrectly, she had to go back and undo the repair and run the program again in order to see the manifestation of the original bug.

One of the subjects did seem to have an effective top-down strategy with elements similar to that of our troubleshooting/debugging model. However, even he had difficulty because his CHARACTERIZE procedure was not well developed. Figure 10a is a complete commentary from this subject for the bugged program shown in Figure 10b. He reads the program first, but only to identify its structure. He then runs the program, but happens to choose inputs for which the program works

S1F

This program looks scary because it's so long. I'm going to try to analyze this program in groups that were delineated in the abstract. That is: (1) check to see if input is correct; (2) count the odd numbers; (3) print the odd numbers. Statement 30 I don't understand. I'll look it up when I'm done reading. As I read down, I see a lot of symbols I don't understand. That's very discouraging. I'll run and see what happens. (1)

The program worked very nicely. I asked Roger what's happening. He said to try more possibilities, so I'll try more disparate numbers. (2)

I found a problem. When I input some numbers it doesn't work. I'll try to see if there's a certain spread that is the line between working and not working. (3) I'll try distances of 2, 4, etc.

(4) I found that any distance past 2, i.e., juxtaposed odd numbers, doesn't work. I'll trace and hope I find something. I have very little idea of what I'm looking for. I just know some loop goes wrong because the machine said that's probably the problem.

Trace sent it into an infinite loop. I'll look at the numbers for a while and see if I can figure anything out of that. Well, P stayed the same, n kept changing. I'll look at the program to see what they mean. I just noticed that at 190 and 200, Y and X are inverted compared to lines 160 and 170. I'll try changing them back and see what happens when I run. (5)

That didn't work so I'll change them back and go into the manuel looking up symbols: < > INT.

Couldn't solve by 1315.

Figure 10a. Debugging commentary of an inexperienced programmer attempting to debug the program shown in Figure 10b.

The user inputs two unequal odd numbers (the program checks to make sure that this is the case and asks the user to try again if a mistake has been made). Odd numbers between his two numbers, inclusive, are counted. For example, there are 3 odd numbers between 5 and 9 — they are 5, 7, and 9. Finally the number of odd numbers between the user's two numbers is printed.

```
01/01/77 00:00:01
  27
10 PRINT "TYPE AN ODD NUMBER"
20 INPUT X
30 IF X/2 <> INT(X/2) THEN 60
40 PRINT "THAT IS NOT AN ODD NUMBER. TRY AGAIN."
50 GOTO 20
60 PRINT "TYPE ANOTHER ODD NUMBER"
70 INPUT Y
80 IF Y/2 <> INT(Y/2) THEN 110
90 PRINT "THAT IS NOT AN ODD NUMBER. TRY AGAIN."
100 GOTO 70
110 IF X <> Y THEN 150
120 PRINT "YOUR TWO NUMBERS ARE EQUAL. TRY AGAIN, THIS TIME"
130 PRINT "USING TWO ODD NUMBERS WHICH ARE NOT EQUAL."
140 GOTO 20
150 IF X < Y THEN 190
160 H = X
170 L = Y
180 GOTO 210
190 H = Y
200 L = X
210 N = 1
220 P = L + 2
230 N = N + 1
240 IF P = H THEN 260
250 GOTO 220
260 PRINT "THERE ARE "; N; " ODD NUMBERS BETWEEN "; L; " AND "; H
999 END
```

Figure 10b. Bugged solution to BIP's task ODDCOUNT, used to study debugging by inexperienced programmers. The bug is in Line 220 which should be P = P + 2. In addition, Line 210 must be P = L and Line 215 must be N = 1.

correctly and becomes "stuck." Only a prompt from the observer induces him to try other inputs and thereby detect the bug. He arrives at a correct ACTION-DESCRIPTION that the program works correctly only if the pair of numbers are consecutive. He does not debug the program within the time allowed, but this can be attributed his forgetting some of the BASIC language constructs needed to understand the function of parts of the program.

The effective strategy of the same subject can be seen in the following excerpt in which he was debugging the program shown in Figure 9. Note his careful initial characterization and testing following repairs, and how he resists jumping to conclusions until he has examined the program's output.

> I just read SID (the program). I just thought the problem may be there's a problem with end or stop statements. I'll run the program to have a look at it. My suspicion seemed correct. The calculator outputs all functions, so I've got to find a way to limit the calculator to its assigned function. I'll look in the glossary to find the right word. I couldn't find anything so I'll try GO TO statements after each function. They'll say: GO TO...end. I just typed a 125 GO TO 199 statement. I'll now run the addition and see if it stops. It worked. I was pretty confident it would. Now, I'll add these expressions to the other functions. I made a mistake in typing, so I'll look up the CTL button for offing a line. found it, I'll CTL X. Now, I'll run again, checking all the functions. It worked. I want to try TRACE now, just to make sure I understand it.

Our observations of debugging by inexperienced programmers support the hypothesis that some of them have difficulties not only because they are not well-versed in programming fundamentals and lack libraries of specific experiential knowledge, but because they have inadequate general debugging strategies. In particular, the are deficient in running a program to obtain information that can be used to deduce logically where a bug is located. In addition, they make repairs without good reasons and lose track of repairs they have attempted, thereby confounding their problem.

III. Teaching Troubleshooting/Debugging

Improving instruction in complex problem-solving

In the introduction, we described the indirect method by which troubleshooting/debugging and other types of complex problem-solving are currently taught. We mentioned two problems with this method. First, in domains where problem solving requires specialized facilities, such as electronic troubleshooting, costs limit the range and number of examples and exercises students may experience during formal instruction. Thus, students of average or above average aptitude may not have experience sufficient for them to acquire problem-solving competence. Second, students with lower aptitudes may have fundamental difficulties learning by the indirect method even when a relatively broad range of experiences can be provided.

One solution to the first problem, and perhaps the second, is to elaborate on the indirect approach in ways that can increase student exposure to problem-solving experiences and add structure to these experiences by providing more and better feedback to him. A landmark example of this type of solution is the SOPHIE system developed over a period of several years by Brown, Burton, and their colleagues (Brown & Burton, 1975; Brown, Rubenstein, and Burton, 1976), which provides instruction in electronic troubleshooting. Through the use of computer simulation and other AI techniques, SOPHIE creates an enriched environment in which students may acquire both a general troubleshooting strategy and domain-specific knowledge for understanding interactions between parts of circuits. SOPHIE does have its limitations-- most notably, that all its exercises and monitoring capabilities are limited to a single circuit-- but these are overshadowed by the advances it represents in teaching by the indirect method.

<u>SOPHIE</u>. The basic SOPHIE system is an interactive computer-based troubleshooting laboratory built around a simulation of a non-trivial power supply circuit. All student activities require only the simulated circuit and no real circuits or test equipment. In various operating modes, components in the simulated circuit can be faulted as specified by a human instructor, by the student, or randomly by SOPHIE itself. The student makes measurements on the faulted circuit simply by requesting them; they are determined by the simulation. Similarly, he specifies repairs by requesting SOPHIE to replace a component. These interactions are facilitated by SOPHIE's limited, but very habitable, natural language front-end, which relieves the student of learning a special language for communicating with the system.

In a basic operational mode, SOPHIE allows an individual student to troubleshoot an unknown fault or investigate the effects of a fault he himself has specified, much as he might in a normal circuit laboratory. However, it eliminates many of the peripheral problems involved in setting up and using real circuits and test equipment. Beyond this, SOPHIE constantly performs two powerful monitoring functions as the student works with the faulted circuit. First, before performing a measurement requested by the student, it determines whether the requested value is redundant-- i.e., whether it can be deduced logically from the measurements that have already been made-- and, if so, refuses to make the measurement. In this way, SOPHIE alerts the student that he has some misunderstanding of the structure and teleology of the circuit. Second, when the student asks that a part be replaced, SOPHIE determines whether that part being faulted is consistent with the measurements that have been made. This is accomplished by faulting that component in a copy of the simulated circuit, making the measurements the student had made, and comparing them with the values obtained from the version of the circuit the student is working with. If the specified repair is inconsistent, the student is told so. Again, this alerts the student to problems in his reasoning and understanding of the circuit.

In a second, more recently developed operating mode, SOPHIE provides the student with "real-time" examples of expert troubleshooting involving the power supply circuit. SOPHIE's "articulate expert" is programmed as blocks of human-authored text that are accessed dynamically in examples involving different faults. Each episode begins by the student selecting a fault he would like the Expert to find. The faults are specified with respect to their location within the circuit's top-level functional organization. A "demon", another entity within SOPHIE, inserts the fault in the circuit. The Expert proceeds with a commentary on isolating the fault which exemplifies an optimal top-down strategy similar to that in the troubleshooting/debugging model we described in the previous section of this report (Figure 11).

Rather than directly make measurements on the circuit at each step in the localization process, the Expert asks the student to tell him the qualitative value of the measure he wants to make (e.g., Is the voltage at point A high, low, or normal?). Thus, the student must understand the effects of the fault he chooses (a device for motivating the student to understand the circuit). The students answer is checked

How does the OUTPUT VOLTAGE compare with a working circuit?

DICK: LOW

EXPERT:
Every functional block has a failure mode which could cause the output to be low under these conditions, so I need some more information.

The Expert knows of faults in every block which cause low output, the only symptom so far.

Is the OUTPUT VOLTAGE essentially zero?

The Expert asks for a more refined, although still qualitative prediction about the output.

DICK: YES

EXPERT:
AH GOOD! That is important information for me because I don't suspect the Vref now. If the Vref were dead, that is, had no output whatsoever, then the output voltage would still be around one and one half volts. It would have this magnitude because the voltage limiter cannot even begin to function until its input voltage rises above the junction voltage drop of its input transistor. And, of course, if Vref had any voltage output greater than zero, the output voltage would be still greater. Having eliminated the Vref, I now wonder about the CL, VL, DARL, CCS and OPF. Any one of these could cause the output voltage to be zero so I am going to narrow the search.

An instance of teleological knowledge at a block level: failures in Vref cannot make the output essentially zero. This is a somewhat surprising fact which the student may want to verify experimentally in the SOPHIE lab.

How does the VOLTAGE AT N1 compare with a working circuit?

Although this test is aimed primarily at only one block, it is nonetheless desirable because the rest of the blocks cannot function unless this block is working.

Figure 11. Excerpt of an interaction with SOPHIE's "articulate expert." From Brown, Rubenstein, and Burton, 1976.

by the demon, who "invisibly" runs the simulated circuit to get the actual quantitative value. If the student's answer is wrong the demon interrupts and tells the student and the Expert. The Expert follows a strategy of choosing measurements that enable him to reduce progressively the part of the circuit's functional organization that remains to be considered. After obtaining each qualitative measure from the student, he explains how it enabled him to deduce that the fault could not be in certain subcircuits. The Expert never describes this localization strategy in general terms; instead, the student is left to induce the general principles from the specific examples of reasoning.

Brown, Rubenstein, and Burton (1976) report a study in which they evaluated the reactions of a small group of second-year electronics students from a technical school to the SOPHIE system. Each subject interacted with SOPHIE in several modes, including the two we have described here.[8] In questionnaires and interviews, the students in general indicated that SOPHIE was superior to their normal experiences in a circuit laboratory. They believed that the individual troubleshooting activity did teach them knowledge that would be useful in troubleshooting other types of circuits. Their criticism was that when they were told about their attempted redundant measurements or illogical repairs, they could not always understand why they were wrong and could obtain no further help from SOPHIE.

The interaction with the Expert was also rated favorably, but not as highly as the other conditions and with more variability among

---

[8] Other conditions included a competitive troubleshooting game between two-person teams, and an exercise in which the student had to specify a fault which when inserted in the circuit would cause another target component to fail as well.

students. Students who liked this condition reported that they believed that by observing the Expert they had learned a general problem-solving strategy of top-down decomposition and testing that they could apply in a range of problem-solving contexts. The students who rated the condition poorly tended to be those of lower aptitude who had trouble in individual debugging and in answering the Expert's questions. They said they found the expert too glib and were frustrated by not being able to question him. They had trouble just following his commentary on a specific problem, let alone being able to induce the underlying general strategy.

The results can in general be taken to indicate that the capabilities of the SOPHIE system can improve learning of both domain-specific knowledge and general strategies within the indirect method of leaning by example and exercise. However, these benefits accrue primarily for students of normal or higher than normal aptitude, and are diminished or lost on those of lower aptitude.

The Why system. The Why system (Collins, 1976; Stevens & Collins, 1977) is another CAI system that may be viewed as an enriched context for teaching problem-solving by the indirect method. The stated purpose of the system is to teach domain-specific declarative knowledge about geography by engaging the student in a natural language dialog with a simulated Socratic tutor (Figure 12). However, as we noted in the introduction, tutoring can readily be viewed as debugging of the student's knowledge system. A tutor's behavior therefore exemplifies the application of a debugging strategy, which the student may induce and subsequently use to debug himself. An interesting feature of Why is that, unlike SOPHIE's articulate expert, the Why tutor generates his

DO YOU THINK THE AMAZON JUNGLE HAS HEAVY RAINFALL OR LITTLE RAINFALL? (Rule 1: Ask for a prediction about a particular case)

HEAVY RAINFALL

WHY DOES THE AMAZON JUNGLE HAVE HEAVY RAINFALL? (Rule 2: Ask for prior causes)

BECAUSE IT'S NEAR THE ATLANTIC

YES, THE AMAZON JUNGLE IS NEAR THE EQUATORIAL ATLANTIC. HOW DOES THAT AFFECT THE RAINFALL IN THE AMAZON JUNGLE? (Rule 3: Ask for intermediate causes)

THE WATER IN CURRENT IS WARM

(section omitted)

WHAT HAPPENS AFTER THE MOISTURE FILLED AIR REACHES THE LAND? (Rule 4: Ask for subsequent causes)

I DON'T KNOW

DO YOU THINK THE ANDES MOUNTAIN RANGE HAS ANY EFFECT ON THE RAINFALL IN THE AMAZON JUNGLE? (Rule 5: Suggest a factor and ask if it is a cause)

YES

HOW? (Rule 3: Ask for intermediate causes)

THE PREVAILING WIND HITS THE ANDES AND THE AIR COOLS

HOW DOES THE FACT THAT PREVAILING WIND HITS THE ANDES, LEAD TO AIR COOLING? (Rule 3 again)

(section omitted)

DO YOU THINK THAT ANY PLACE WITH MOUNTAINS HAS HEAVY RAINFALL? (Rule 6: form a general rule for an insufficient cause)

YES

SOUTHERN CALIFORNIA HAS MOUNTAINS. WHY DOESN'T SOUTHERN CALIFORNIA HAVE HEAVY RAINFALL? (Rule 7: Pick a counter-example for an insufficient cause, and Rule 2: Ask for prior causes)

Figure 12. Annotated excerpt of a dialog with the Why system. From Stevens and Collins, 1977.

questions and comments dynamically using an explicitly represented process model of a general tutoring strategy (i.e., one that is independent of the content of the subject being taught). The model is expressed as a production system of about two dozen rules (Figure 13), which were derived by analyzing dialogs between students and human tutors. While this model underlies the tutor's behavior, it is not articulated directly to the student and is actually communicated more indirectly than the strategy underlying SOPHIE'S expert's "canned" explanations.

<u>More direct methods for teaching strategies</u>. A second approach to teaching complex problem-solving, which might help those students who have the most difficulty learning by the indirect approach, is to provide explicit descriptions of the procedures for solving problems that can serve as prescriptions for the student. As noted in the introduction, an impediment to this approach previously has been the lack of a suitable language for conceiving and talking about problems and problem-solving processes. The development in AI and information-processing psychology of formalisms for representing knowledge has caused researchers concerned with learning and instruction to reexamine the need and potential for more direct and explicit instruction in problem-solving.

Why do we not attempt to teach some basic cognitive skills such as how to organize one's knowledge, how to learn, how to solve problems, how to correct errors in understanding: these strike us as basic components which ought to be taught along with the content matter.

Norman, Gentner, and Stevens, 1976, p. 194.

RULE 2: Ask for any factors

If 1) a student asserts that a case has a particular value on the dependent variable,

then 2) ask the student why.

EXAMPLE:

If a student says they grow rice in China, ask why.

REASON FOR USE:

This determines what causal factors or chains the student knows about.

RULE 3: Ask for intermediate factors

If 1) the student gives as an explanation a factor that is not an immediate cause in the causal chain,

then 2) ask for the intermediate steps.

EXAMPLE:

If the student mentions monsoons in China, as a reason for rice growing, ask "Why do monsoons make it possible to grow rice in China?"

REASON FOR USE:

This insures that the student understands the steps in the causal chain, for example that rice needs to be flooded.

RULE 4: Ask for prior factors

If 1) the student gives as an explanation a factor on causal chain where there are also prior factors,

then 2) ask the student for the prior factors.

Figure 13. Several of the production rules used in the Why system as a computational model of a tutoring strategy. From Stevens and Collins, 1977.

...as information-processing analyses succeed in identifying the processes underlying problem solution, these processes-- at least some of them-- can be directly taught, and that individuals will then be able to apply them to solving relatively large classes of problems. ... ways can be found to make individuals more conscious of the role of environmental cues in problem solving and to teach strategies of feature scanning and analysis.

Resnick, 1976, pp. 79-80.

Papert (1971) at MIT has played a prominent role in articulating the position that by teaching general problem-solving strategies more directly, students can become better learners. His argument is that learning to do things is facilitated by giving the learner a procedural representation of his task and having him debug his attempted execution of that procedure. Papert feel that this methodology applies to tasks as diverse as computational mathematics and juggling. Much of his work has involved teaching computational mathematics (primarily geometry) to children by teaching them to write programs in the LOGO language. The students learn the mathematics by discovery (i.e., inductively), but they are taught strategies for design and debugging explicitly. The strategies, however, are not presented <u>in toto</u>. Instead, the method adopted is to present them in parts as separate <u>heuristics</u> in reaction to events that transpire as the student designs and debugs his programs.

In this context, a heuristic may be defined as a rule-of-thumb, a piece of a larger procedure that enables a correct or more efficient solution under a set of conditions. The effectiveness of using a heuristic depends both on being able to identify the contexts where it applies or is more effective than other heuristics and on access to other knowledge needed to execute it. Heuristics may embody either general or domain-specific procedural knowledge. The following are both heuristics for troubleshooting; however, the first is limited to a very specific context, while the second is part of the general strategy we presented in the previous section of the report.

> If the car is idling unevenly, the first thing to do is to strike the body of the carburetor with several crisp (but non-damaging) blows.

> If you have decided to make an observation of the system's behavior, choose the observation that has the potential to eliminate the greatest part of the system as a possible fault location whether the observation proves to be normal or abnormal.

In Papert's research studies, the student's problem solving is continually monitored by an instructor. When the student has difficulty or uses less than optimal strategies for designing and debugging his programs, the instructor interrupts and describes an applicable heuristic to him. The heuristics are explicit, but couched in informal speech. For example, if a program intended to draw some figure fails because the student's design does not take into account an interaction between two procedures, he might be told "look carefully at the position and orientation of the pen between the procedures that draw the parts of the figure that are incorrect." There are several comments to be made about this method. Clearly, it is not a cost-effective approach to large-scale instruction; however, Papert has been concerned with gaining initial acceptance for its principles with the idea that implementation problems can be resolved subsequently. Second, although the students learn an explicit formalism (LOGO) for representing procedural knowledge, the heuristics themselves are expressed in natural language. Finally, the interrelations among the individual heuristics within an encompassing design and debugging strategy are not explicitly described to the student.

Carr and Goldstein (1977) at MIT have described a computer-based system called WUSOR-II that refines Papert's method of reactive teaching of heuristics and exemplifies how it can be made more cost-effective by automating the monitoring of the student. WUSOR-II is built around a game called Wumpus, a version of Theseus and the Minotaur, which requires a fundamental deductive problem-solving strategy for optimal play. The player is placed somewhere in a maze of caves, told the names of the neighboring caves, and warned if certain dangers are present in those caves, although the exact location of the danger is not specified. He then selects a cave to move to. His goal is to find and slay the Wumpus by shooting an arrow into the cave where it is lurking before it slays him. The reasoning involved is fairly simple; for example, if a cave has a warning and all but one of its neighbors are known to be safe, then the danger is in the remaining neighbor. Note that this type of reasoning resembles that required in troubleshooting/debugging to localize a fault given a set of observations. The optimal strategy for selecting a move is to determine the safest neighbor as deduced from the history of warnings.

WUSOR-II incorporates an expert monitoring procedure. Because the problem is well-structured, it was possible to implement a computational model for playing the optimal strategy. The monitor uses this model to evaluate the student's move. WUSOR-II incorporates a sophisticated pedagogical strategy to determine when it is appropriate for the monitor to interrupt play and describe a heuristic that generates a better move than the student had just selected. One of the principles is to interrupt only when the student has consistently failed to make moves that could be improved on by a particular heuristic; that is, do not interrupt if the student fails to use a heuristic once when it is appropriate, when you have seen him use appropriately before. Another principle is based on a representation of the interrelationships among the heuristics which Carr and Goldstein call a syllabus: A heuristic is not mentioned unless the heuristics prior to it (e.g., use of double evidence depends on use of single evidence) in the syllabus are inferred to be learned from the moves the student has made. The teaching method itself (Figure 14) consists of articulating the faulty logic of the student's move, the detailed logic for generating a better move, and finally a general description of the heuristic used to generate that move.

WUSOR-II is a noteworthy elaboration on Papert's method for explicitly presenting problem-solving heuristics. However, its capabilities are highly dependent on the simplicity of the problem domain in which the heuristics are taught. The heuristics themselves are part of a general, deductive problem-solving strategy that is applicable in many problem domains, including troubleshooting/debugging. An unanswered question is whether students who learn general heuristics in such a "toy" domain can in fact transfer them to a "real-world" domain and incorporate them in more comprehensive strategies. A factor that might affect their success is whether they have interrelated the heuristics they have had described to them separately across different problem-solving episodes into an overall strategy.

The alternative to teaching heuristics reactively one-by-one is to introduce them to the student according to a prior plan so that they

Ira, it isn't necessary to take such large risks with pits.

Cave 4 must be next to a pit because we felt a draft there. Hence, one of caves 15, 2 and 14 contains a pit, but we have safely visited cave 15. This means that one of caves 2 and 14 contains a pit.

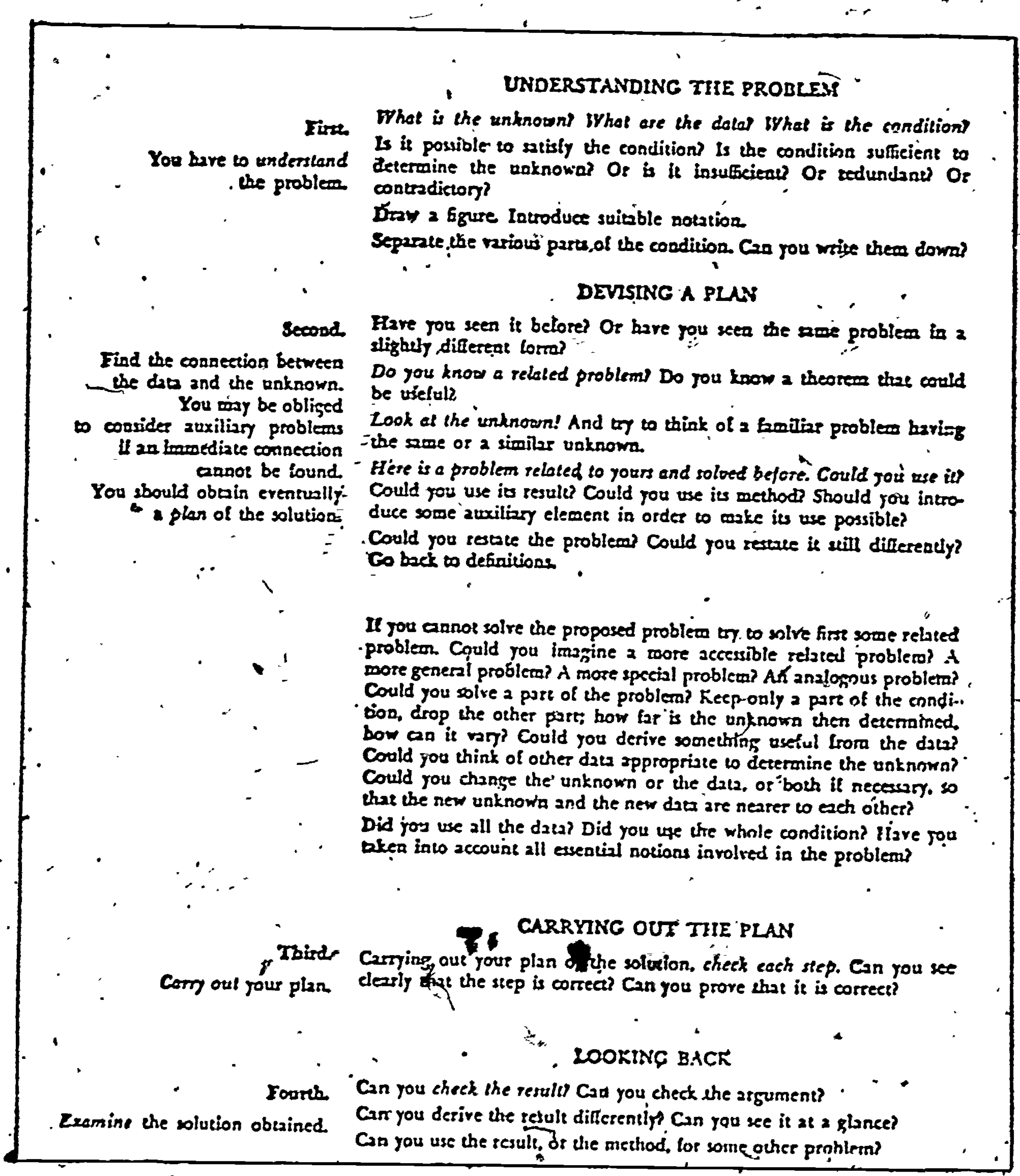
Likewise cave 15 must be next to a pit because we felt a draft there. Hence, one of caves 0, 4 and 14 contains a pit, but we have safely visited cave 4. This means that one of caves 0 and 14 contains a pit.

This is multiple evidence of a pit in cave 14 which makes it probable that cave 14 contains a pit. It is less likely that cave 0 contains a pit. Hence, Ira, we might want to explore cave 0 instead.

Figure 14. Dialog with the WUSOR-II system illustrating an attempt to teach the user a heuristic for applying multiple evidence in deduction. From Carr and Goldstein, 1977.

are available to him whenever he is ready to use them. George Polya's book, "How to Solve It" (1957) is most often cited as the first attempt to teach a problem-solving strategy directly by a text. A mathematician and teacher, he had observed basic similarities in the methods used by expert problem solvers to solve mathematical proof problems. If these methods could be described, he concluded, they could be taught to students, thereby saving the students the years it would take them to discover the methods on their own. Indeed, he felt some students never discovered these principles simply by working on exercises by themselves. Figure 15 summarizes the four stages of Polya's strategy and the heuristics applicable at each stage. Polya's work though it is now recognized as a precursor to information-processing analyses of problem solving, has never had an impact on practical instruction in mathematics (Schoenfeld, 1977a). The difficulty seems to be that people reading the text may understand the strategy and heuristics, but, when faced with a particular problem, have difficulty determining the particular heuristic that "unlocks" that problem; that is, while Polya's descriptions are perhaps accurate, the way in which they are presented in his book does not enable most readers to adopt them as prescriptions.

Wayne Wickelgren, an information-processing psychologist, has authored a more recent book, "How to Solve Problems" (1974), which is similar to Polya's, but incorporates information-processing formalisms for describing problem structures and problem-solving processes and a presentation intended to teach the reader how to recognize when particular strategies and heuristics are applicable. Wickelgren also does not restrict himself to mathematics problems but addresses a more general taxonomy of problem types. The problem-solving methods he



UNDERSTANDING THE PROBLEM

First.

You have to *understand* the problem.

What is the unknown? What are the data? What is the condition?

Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?

Draw a figure. Introduce suitable notation.

Separate the various parts of the condition. Can you write them down?

DEVISING A PLAN

Second.

Find the connection between the data and the unknown. You may be obliged to consider auxiliary problems if an immediate connection cannot be found. You should obtain eventually a *plan* of the solution.

Have you seen it before? Or have you seen the same problem in a slightly different form?

Do you know a related problem? Do you know a theorem that could be useful?

Look at the unknown! And try to think of a familiar problem having the same or a similar unknown.

Here is a problem related to yours and solved before. Could you use it? Could you use its result? Could you use its method? Should you introduce some auxiliary element in order to make its use possible?

Could you restate the problem? Could you restate it still differently? Go back to definitions.

If you cannot solve the proposed problem try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? Keep only a part of the condition, drop the other part; how far is the unknown then determined, how can it vary? Could you derive something useful from the data? Could you think of other data appropriate to determine the unknown? Could you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other?

Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

CARRYING OUT THE PLAN

Third.

Carry out your plan.

Carrying out your plan of the solution, *check each step.* Can you see clearly that the step is correct? Can you prove that it is correct?

LOOKING BACK

Fourth.

Examine the solution obtained.

Can you *check the result?* Can you check the argument?

Can you derive the result differently? Can you see it at a glance?

Can you use the result, or the method, for some other problem?

Figure 15. Polya's heuristics for problem solving. From Polya, 1957.

considers include drawing inferences, classificatory trial and error, using an evaluation function to choose an action (hill climbing), defining subgoals, deriving a contradiction, working backwards from the goal, and recognizing the relations between problems. His approach is to describe in general terms, using formal representations when these exist, a problem type and the applicable problem-solving method and then give a series of examples which illustrate that method. The examples are most often puzzles or mathematical problems that require a minimal background. The solution to each example is presented in steps and the reader is instructed to attempt the solution according to the method he has just studied before he reads each step. The text for each step describes a heuristic to be applied at that point, allowing the reader to assess the heuristic he used or to continue on if he is stuck.

Wickelgren presents as comprehensive catalogue of problem types and methods as one could hope for given the present understanding of problem solving. Two comments about the learnability of this information can be made. First, it is exemplified largely with toy problems, a feature necessitated by the fact that the book is intended for a general audience and not as a text for student entering a specific discipline. Second, although each method is described very thoroughly, they are not explicitly interrelated. Thus, it still could be difficult to determine which method applies when a problem is presented outside the context of a chapter describing an applicable method.

Alan Schoenfeld, a mathematics instructor, has described in an unpublished report (1977a) a method for teaching problem-solving heuristics for mathematical proof that builds upon the work of Polya and Wickelgren and that he has evaluated in a real instructional setting.

He states that for a student to use a heuristic he must not only understand the procedure it specifies, but also understand the subject in which he is to use it and recognize the situations in which it can be used. The innovation in Schoenfeld's method is the explicit articulation of what he calls a managerial strategy, a prescriptive model of the relationships between individual heuristics. The managerial strategy is taught to the student as a device for monitoring his progress through the solution to a problem and thereby focusing his attention on the subset of the heuristics he knows that may be relevant at each point. Figures 16 and 17 are the heuristics and managerial strategy Schoenfeld used in a small course he taught. His tentative conclusion based on an informal study of the solutions generated by students on examination problems was that the students did develop a better ability to select appropriate proof methods relative to students in standard courses of mathematics instruction.

Schoenfeld, in a second unpublished report 1977b, describes another small, but more formal study evaluating his method for teaching heuristics, this time for calculus problems involving indefinite integration. Fewer heuristics and a more limited managerial strategy are involved for this domain. Schoenfeld developed a brief text describing these and illustrating their application. The text was given to half of a calculus class four days prior to an examination. The examination involved nine problems, seven of which could be solved by the methods covered in Schoenfeld's text. The students who received the text outscored those who did not on six of the seven problems, while the two groups did not differ on the other two problems. Furthermore, the students were asked to record the time they spent studying for the

ANALYSIS

1) DRAW A DIAGRAM if at all possible.

2) EXAMINE SPECIAL CASES:

   a) Choose special values to exemplify the problem and get a "feel" for it.

   b) Examine limiting cases to explore the range of possibilities.

   c) Set any integer parameters equal to 1, 2, 3,..., in sequence, and look for an inductive pattern.

3) TRY TO SIMPLIFY THE PROBLEM by

   a) exploiting symmetry, or

   b) "Without Loss of Generality" arguments (including scaling)

EXPLORATION

1) CONSIDER ESSENTIALLY EQUIVALENT PROBLEMS:

   a) Replacing conditions by equivalent ones.

   b) Re-combining the elements of the problem in different ways.

   c) Introduce auxiliary elements.

   d) Re-formulate the problem by

      i) change of perspective or notation

      ii) considering argument by contradiction or contrapositive

      iii) assuming you have a solution, and determining its properties

2) CONSIDER SLIGHTLY MODIFIED PROBLEMS:

   a) Choose subgoals (obtain partial fulfillment of the conditions)

   b) Relax a condition and then try to re-impose it.

   c) Decompose the domain of the problem and work on it case by case.

Figure 16. Schoenfeld's heuristics for solving mathematical proof problems. From Schoenfeld, 1977a.

(Figure 16 continued)

EXPLORATION (continued)

3) CONSIDER BROADLY MODIFIED PROBLEMS:

a) Construct an analogous problem with fewer variables.

b) Hold all but one variable fixed to determine that variable's impact.

c) Try to exploit any related problem which have similar

i) form

ii) "givens"

iii) conclusions.

Remember: when dealing with easier related problems, you should try to exploit both the RESULT and the METHOD OF SOLUTION on the given problem.

VERIFYING YOUR SOLUTION

1) DOES YOUR SOLUTION PASS THESE SPECIFIC TESTS:

a) Does it use all the pertinent data?

b) Does it conform to reasonable estimates or predictions?

c) Does it withstand tests of symmetry, dimension analysis, or scaling?

2) DOES IT PASS THESE GENERAL TESTS?

a) Can it be obtained differently?

b) Can it be substantiated by special cases?

c) Can it be reduced to known results?

d) Can it be used to generate something you know?

Schematic Outline of the Problem-Solving Strategy

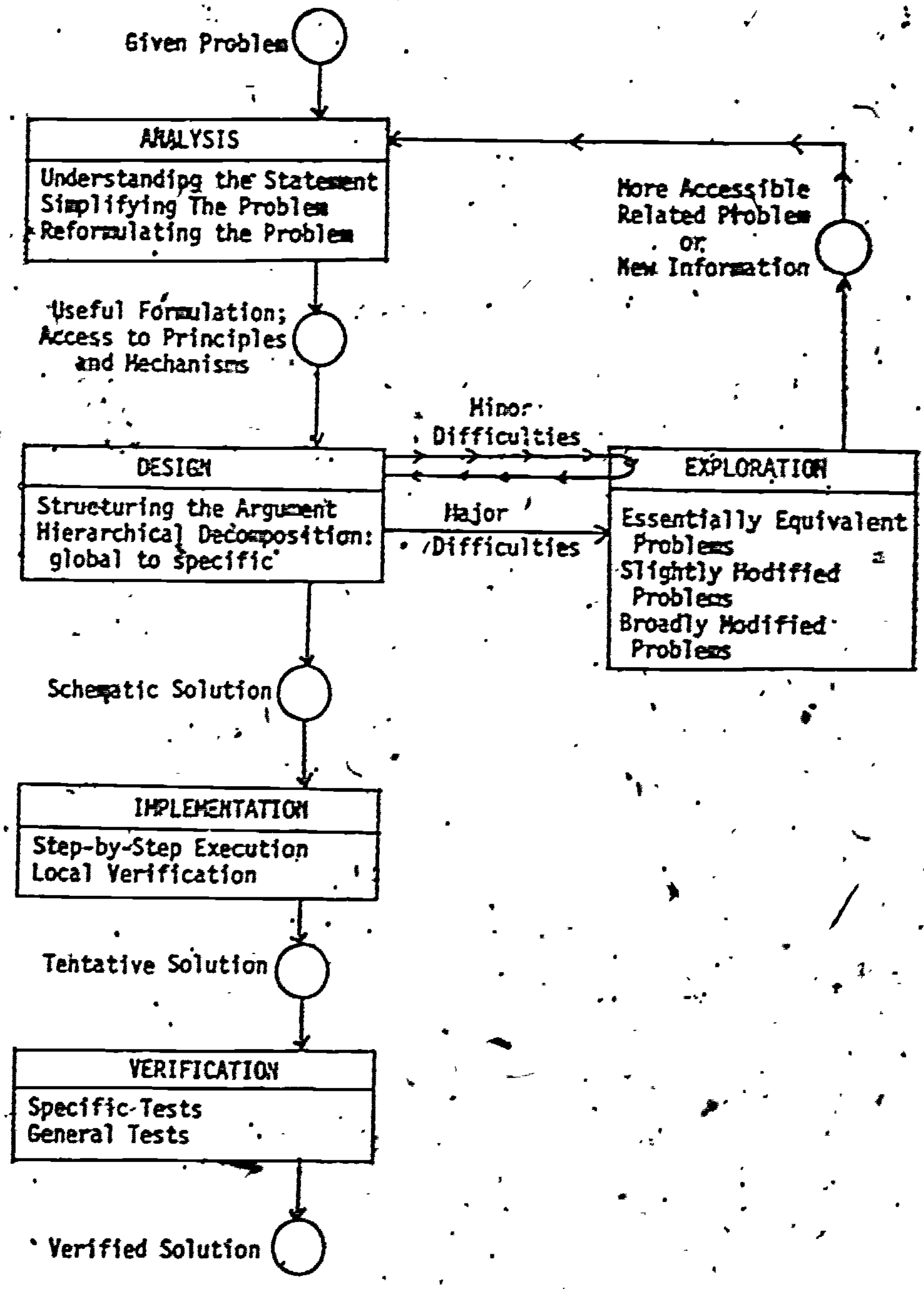

Given Problem
ANALYSIS
Understanding the Statement
Simplifying The Problem
Reformulating the Problem
Useful Formulation; Access to Principles and Mechanisms
DESIGN
Structuring the Argument
Hierarchical Decomposition: global to specific
Minor Difficulties
Major Difficulties
EXPLORATION
Essentially Equivalent Problems
Slightly Modified Problems
Broadly Modified Problems
More Accessible Related Problem or New Information
Schematic Solution
IMPLEMENTATION
Step-by-Step Execution
Local Verification
Tentative Solution
VERIFICATION
Specific Tests
General Tests
Verified Solution

Figure 17. Schematic representation of Schoenfeld's managerial strategy for mathematical problem solving. From Schoenfeld, 1977a.

examination and those who studied with the text spent less time on the average.

Schoenfeld's results, though based on a limited sample, do suggest that heuristics can be taught directly to advantage, provided they are taught in the context of the domain in which they will be used subsequently and they are explicitly interrelated within a larger strategy that predicates when each is applicable. In the next section, we present a study that investigated whether a direct presentation of heuristics can be used to teach inexperienced programmers how to debug.

IV. Directly teaching debugging heuristics: an experimental study

Rationale

In examining chronologies of debugging behavior we found that the difficulties of inexperienced programmers are due as much to their lack of a rational general strategy as to their unfamiliarity with the declarative and procedural knowledge needed to understand programs and to operate in a specific programming environment. In this section, we discuss an experiment we conducted, in which we attempted to teach directly to inexperienced programmers a few heuristics that are part of a useful (though possibly conservative) debugging strategy. The experiment was intended more to be an exploration of methodology, than a definitive test of whether it is worthwhile to teach representations of procedural knowledge directly. At the outset, limitations on our access to subjects over an extended period precluded any attempt to teach a complete debugging strategy, or even to teach part of a strategy thoroughly in a natural instructional situation. Instead, a brief tutorial text was developed to present a few relevant heuristics and subjects studied it only briefly in an experimental setting prior to attempting a few test problems. Thus, we knew that whatever the results of the instructional treatment, the adequacy of the pedagogy used to communicate the heuristics could be questioned. Nonetheless, for a first attempt to teach debugging heuristics it was not unreasonable to test a minimal instructional method. Possibly, the results would indicate that mere identification of general debugging heuristics is sufficient to modify the behavior of inexperienced programmers (e.g., if they already "knew" the heuristics, but needed an external cue to make them more readily accessible when needed). In this case, the costs of developing more substantial, but unnecessary, instructional methodology could be avoided.[9]

The overall plan of the experiment was to compare the behavior of two groups of inexperienced programmers on debugging problems, one of the groups studying and referring to the tutorial and the other receiving only some unassisted practice in debugging. Data analysis was to be exploratory, with a goal of identifying measures that could indicate the role of the debugging heuristics in subjects' problem solving.

Debugging tutorial

The debugging tutorial we created presents eight "guidelines" that are part of a general debugging strategy. Following the guidelines will not always lead to the most efficient debugging but for an inexperienced programmer without much specific debugging knowledge they will tend to reduce false starts and to help determine a course of action when he is "stuck." The guidelines can be seen as elements of three encompassing heuristics for (1) testing a program sufficiently to detect errors, (2) generating a thorough characterization of an error's

---

[9]Schoenfeld's ~~results~~ on teaching heuristics for mathematical proof problems (described in the previous section) became available only after the experiment described here was underway. In any case, there is a basic difference between heuristics for proof and integration problems and those for debugging. In the proof problems, a single applicable heuristic must be selected; the subject's main problem is recognizing the features of a problem that make a specific heuristic applicable. In debugging, the use of several heuristics must be coordinated at several points in every problem; the main problem in debugging is remembering to use all of the heuristics. Of course, in both cases the heuristics must be used appropriately.

manifestation, and (3) backtracking from unsuccessful repairs. A summary of the guidelines from the tutorial is shown in Figure 18. The number next to each guideline indicates which of the three heuristics it is part of. The heuristics were decomposed into separate guidelines to facilitate their comprehension. The guidelines are shown in Figure 18 in the order in which the tutorial introduces them. This order reflects that in which the guidelines are applicable during each iteration (or recursion) of the general debugging strategy that was described in Section II.

The eight guidelines were formulated to correct the most frequently observed shortcomings we had previously identified in the debugging behavior of inexperienced programmers. All of these guidelines, except perhaps for those concerned with backtracking, have straightforward mappings onto other troubleshooting situations, like electronic and mechanical maintenance and repair. For example, varying a program's inputs is analogous to varying the inputs and external controls of electronic and mechanical devices.

The tutorial (Appendix A) is a rather minimal piece of pedagogy. In a linear narrative mode, it introduces each guideline, giving a rationale for its use and a specific debugging scenario that illustrates its successful application. The examples are intended to demonstrate <u>when</u> it is appropriate to apply the guidelines: having problem-solving heuristics available is of little use if one does not know the circumstances under which they should be applied. The example programs were taken with slight modification from the programming chronologies we had examined earlier. The narrative for the examples was developed in part from the written commentaries we had collected from the

TESTING THE PROGRAM

(1) TEST THE PROGRAM WITH ALL POSSIBLE TYPES OF INPUT FOR WHICH IT IS DESIGNED

(1) TEST THE PROGRAM WITH THE EXTREME VALUES THAT THE INPUT CAN HAVE

CHARACTERIZING THE ERROR

(2) CHARACTERIZE THE WAY THE ERROR(S) SHOWS UP IN TERMS OF THE INPUT AND OUTPUT

(2) EVEN IF A PROGRAM IS SHORT AND EASY TO TRACE BY HAND, YOU SHOULD FIRST RUN THE PROGRAM. (ERROR MESSAGES, AS WELL AS A CHARACTERIZATION OF THE ERROR IN TERMS OF INPUT AND OUTPUT, CAN BE VERY HELPFUL IN FINDING AN ERROR.)

(2) SOMETIMES A PROGRAM GIVES THE CORRECT OUTPUT FOR SOME INPUTS BUT NOT FOR OTHERS. WHEN THIS HAPPENS YOU SHOULD EXAMINE THE DIFFERENCE(S) BETWEEN THE INPUTS FOR WHICH THE PROGRAM WORKS AND THE ONES FOR WHICH IT FAILS.

(1) AFTER A CHANGE, RETEST THE PROGRAM USING ALL POSSIBLE TYPES OF INPUT FOR WHICH THE PROGRAM WAS DESIGNED.

(3) IF YOU MAKE A CHANGE TO A PROGRAM, AND IT STILL GIVES THE SAME ERRONEOUS OUTPUT, RESTORE THE PROGRAM TO ITS STATE BEFORE THE CHANGE. YOU HAVEN'T FOUND THE ERROR(S) IN THE PROGRAM, AND YOU MAY HAVE INTRODUCED A NEW ERROR.

(3) IF YOU MAKE A CHANGE TO A PROGRAM, AND THE OUTPUT IS STILL WRONG: IF THE CHANGE CORRECTS ONE PART OF THE PROGRAM (e.g., one part of the output), THEN LEAVE THE CHANGE IN THE PROGRAM.

Figure 18. List of the debugging guidelines presented in the debugging tutorial. Numbers in parentheses indicate grouping of guidelines into three encompassing heuristics for testing, characterizing, and backtracking.

inexperienced programmers who had tried to debug them. The commentaries contained instances of both productive and non-productive reasoning useful for illustrating how the guidelines could help during debugging. By using examples with errors and problem-solving introspections actually produced by inexperienced programmers, we hoped to create a text consistent with the experience of subjects we would employ. Other than indentation and emphasis, the tutorial makes no use of text engineering techniques like hierarchical outlining or systematic review which might improve comprehension. In fact, the tutorial assumes a high level of literacy and motivation. These were characteristics we expected of the undergraduates who would participate as subjects and we chose to make the text consistent with their aptitudes. We did create a brief openbook test (Appendix B) to accompany the tutorial so that these subjects could monitor their comprehension and determine their own review strategy.

Procedure

Subjects. The subjects were twelve paid volunteers from a group of 28 students who completed fifteen hours of curriculum in the BIP course in the weeks prior to the experiment. Thus, at the time of the experiment, each subject had written several dozen short programs within BIP, but had no other programming and debugging experience. The subjects were recruited approximately halfway through their participation in BIP.

Prior to beginning BIP, each subject had been pretested with the Computer Programmer Aptitude Battery (Palormo, 1964). On the basis of their scores, the twelve subjects were divided into two "matched" groups of six subjects each. This was done in an attempt to control for pre-existing differences that might interact with the experimental instructional treatment.[10]

Experimental environment. All experimental sessions were conducted in the same setting in which subjects had worked with BIP. All experimental test exercises were conducted using BIP's programming facilities (of course, those facilities specific to the BIP curriculum-- e.g., HINT-- were inoperative for the test exercises). Two CRT terminals were available, allowing either one or two subjects (always from the same experimental condition) to be scheduled for experimental sessions.

An experimenter was available throughout the sessions to help with procedural problems (e.g., loading exercises into the subject's program space in BIP and recovering from system crashes) and to list hardcopy of test programs for subjects upon their request.

Method. Each subject participated in three sessions. Session 1 was the (different) treatment/testing session for the experimental (TUTORIAL) and control (NO-TUTORIAL) groups. Sessions 2 and 3 were test sessions identical for both groups.

Session 1 for the TUTORIAL subjects began with a text introducing the general logic of troubleshooting/debugging (Appendix C). The subject was then given the tutorial text presenting the eight

---

[10] Scores reflecting ability after BIP would have been preferable, but could not be used because time constraints required that each subject begin the experiment as soon as he completed his 15 hours in BIP. Thus, assignment to treatment groups had to be made while subjects were still in BIP. Subjects did take a programming posttest after completing BIP and before participating in the experiment. Subsequent analysis (see Results below) indicate that the two groups of subjects differed markedly with respect to the posttest scores.

guidelines to study for one-half hour. After study, the open-book quiz was given and the subject reviewed the text as necessary to complete the quiz. The subject then moved to the terminal to work a debugging problem. He was given (in his program space) a program and a written description of its intended function. He was told that their was something wrong with the program and that his task was to change it so that it worked according to the description. The instructions emphasized that the necessary changes were minor and that he was not to write his own program to satisfy the description. A time limit of one-half hour was imposed for this debugging exercise.

The test program was an atypical solution to a task in the BIP curriculum called CHANGER. All of the subjects had worked on this task, but had used algorithms different from the one in the exercise. CHANGER is supposed to ask the user for a purchase price less than one dollar and print the amount of change and the list of coins needed to make that amount of change most efficiently. The bug in the test exercise manifested itself as an incorrect list of coins whenever two dimes were required as part of the answer (Appendix D).

Session 1 for the NO-TUTORIAL subjects began with a brief description of their task. The subject then was given one-half hour to work at the terminal on the first of two debugging exercises. Again, the exercise involved a malfunctioning program, a description of its intended function, and instructions to seek a minimal repair. The program was one of those used as an example in the tutorial to illustrate the use of the guidelines for debugging. It was chosen as an exercise for the NO-TUTORIAL group in order to minimize differences in knowledge about specific programs and bugs. The second debugging exercise, given during the second one-half hour of Session 1, was the CHANGER program given to the TUTORIAL subjects in the second half of their first session.

Testing in Sessions 2 and 3 was identical for both groups, except that subjects in the TUTORIAL group could refer to the tutorial text as they pleased. The exercise given in Session 2 was to write a program DRILL (Appendix E). DRILL, a program to provide drill-and-practice in addition and subtraction, was longer [illegible] a more extensive control structure than any program that [illegible]jects were required to write in BIP. The necessary control structure was such that the guidelines for program testing given in the tutorial could reasonably be expected to facilitate its successful implementation. Two hours were allowed to write the program. If a subject completed the program within one and one-half hours, the experimenter tested it and informed the subject simply whether it did or did not satisfy the specifications. If it did not, the subject was allowed the remaining time to complete (or debug) his program. This was done to provoke debugging in cases where a subject had not been able to detect a bug in his own program.

The exercise given in Session 3 was to debug (under instructions identical to those for the debugging exercises of Session 1) a program ARITH-CALC which had been written and "bugged" by one of the research team (Appendix E). ARITH-CALC was designed to evaluate in strict left-to-right order strings representing numeric expressions input by a user. (BIP's dialect of BASIC has no automatic type-conversion mechanism.) Again, the program was longer and more complicated than those required by the BIP curriculum or used in the tutorial and

Session 1. In particular, the algorithm almost certainly was unfamiliar to all the subjects and difficult for them to trace mentally, although no specialized background knowledge is required to gain an understanding of it. ARITH-CALC and its bug were generated so that the tutorial guidelines for characterizing an error in terms of input-output relationships would be relevant for efficient solution of the exercise. The bug does not manifest itself for every input and the discrepancy in the output value varies as a function of the arithmetic operations and their order in the input string. Subjects were given one and one-half hours to complete the exercise.

In all sessions, data on each subject's programming and debugging behavior was collected automatically (and invisibly to subjects) by BIP's chronology facility. In addition, subjects in the experimental group were given a written questionnaire at the conclusion of Session 3 designed to elicit their reactions to the tutorial and the experiment (Appendix G).

Results

Our efforts at earlier stages of the research to derive debugging grammars to describe BIP chronology data had been unsuccessful. Therefore, we did not have available any comprehensive mechanism for analyzing the chronologies collected in the experiment in order to describe differences between subjects' strategies. The type of analysis we conducted was thus much more limited than we desired. The present experiment was concerned specifically with (1) whether the behavior of subjects in the TUTORIAL group would reflect their attempted use of the guidelines given in the tutorial text and, (2) if so, the extent to which the guidelines were in fact acquired from the text rather than inferred from prior experience (as determined by comparison with the NO-TUTORIAL group). The chronologies were analyzed to assign values to five relevant "measures" for each exercise.

(a) adequacy of solution
(b) detection of bugs via program execution (as opposed to mental analysis of the code)
(c) characterization of bugs via extended program execution revealing input-output relationships
(d) extended testing of attempted program repairs
(e) backtracking from unsuccessful repairs

The measures represent the success of the attempted solution and the extent and success with which the heuristics encompassing the guidelines were applied.

Each measure was assigned a value "+" meaning "done successfully" or "-" meaning either "not successful" for a or "not attempted" for b-e. Measures b-e could also be scored as "0" meaning "attempted, but with unsuccessful results." For instance, a "0" value would be assigned to measure b if a subject ran the program several times with different inputs, but failed to find inputs that caused the bug to manifest itself. In addition, some measures could be scored "NA" meaning "not applicable"; for example, if the subject never attempted repairs, no score could be assigned to measures d and e on that exercise.

Determination of scores for measures b-e from the chronologies proved to be a rather complex judgement process. There are no singular events in the chronology for an exercise that determine unambiguously the values of these four measures. For example, whether or not a subject actually characterized an error in terms of the input-output relationships obtained by his execution of a program can be determined only by examining his subsequent repairs and the other events leading up to them. In a sense, the scorer had to try to simulate the reasoning underlying the subject's actions and see if it was consistent with a hypothesis that the error had been characterized in terms of input-output relationships. A further complication is that an attempted solution may involve more than one debugging cycle, or episode. In these cases, scores for the measures were determined by judgement of the predominant behavior across the episodes.

In order to reduce potential bias in this subjective scoring process, chronologies (which contain repeated information identifying subjects) were scored primarily by a member of the research team not familiar with the assignment of subjects to groups. However, no data was on the intra- and inter-judge reliability of scoring for the results to be presented.

Results will be presented here for the debugging exercises CHANGER and ARITH-CALC attempted by both groups in the second half of Session 1 and in Session 3 respectively. Behavior in the programming exercise DRILL given in Session 2 proved impossible to score with any degree of confidence because of the great variability with which subjects approached it. Some subjects, in fact, never implemented enough of the program to execute it and examine any output. Others produced executable pieces of a solution program, but showed widely varying debugging behavior in different episodes within the exercise. Given these difficulties and the fact that there were no differences in the number of correct (or almost correct solutions)— measure a— between the TUTORIAL and NO-TUTORIAL groups, it seemed pointless to score the chronologies for DRILL with respect to measures b-e.

Tables 2a and 2b present the five scores on CHANGER and ARITH-CALC and also the BIP pretest and posttest scores for each subject in the TUTORIAL and NO-TUTORIAL groups. Most striking is the poor performance of subjects in both groups as indicated in column a. Three members of the TUTORIAL group and two members of the NO-TUTORIAL group solved neither of the problems. In each group, exactly five exercises were completed successfully. Thus, the instructional treatment for the TUTORIAL group does not seem to have improved their debugging ability as measured on two test exercises specifically formulated to be sensitive to that instruction. Unfortunately, however, even if there was an effect of the treatment, it may have been obliterated by a difference between the ability of the groups at the time they began the experiment. Recall that the groups were matched using the BIP pretest scores. Inspection of the posttest scores, available to us only after some subjects had begun the experiment, shows that a large difference in programming ability existed for the two groups. By chance, the subjects assigned to the NO-TUTORIAL group had become much better programmers on the average.[11] Thus, if the tutorial did improve debugging ability, the only effect may have been to cancel the initial difference between the experimental and control groups.[12]

[11] The small sample size precludes a meaningful statistical evaluation of the difference; however, in our experience, such a large difference in posttest scores does have practical significance and correlates with subjective impressions of programming sophistication.

[12] An attempt was made to obtain "difference" scores for each subject in order to see if the TUTORIAL group showed a larger improvement in debugging ability relative to their ability before studying the tutorial. BIP chronologies for the final few BIP tasks worked by each subject were examined. However, the variability in these chronologies resembles that found in the transcripts for the experimental DRILL exercise. Thus, it was not possible to score the pre-experimental debugging episodes with any confidence and thereby to obtain the desired difference scores.

Table 2a

BIP Test Scores and Debugging Measures for TUTORIAL Group

Subject	BIP pretest	BIP posttest	Task	Debugging Measures* a	b	c	d	e
316	116	153	Changer	-	+	+	NA	+
			Arith-Calc	-	+	-	-	-
317	129	195	Changer	+	+	+	+	NA
			Arith-Calc	+	+	+	+	+
322	131	231	Changer	+	+	+	+	+
			Arith-Calc	+	+	+	+	+
324	102	111	Changer	-	0	0	NA	NA
			Arith-Calc	-	+	0	0	+
333	79	89	Changer	-	+	0	NA	NA
			Arith-Calc	-	+	0	0	+
340	98	181	Changer	+	+	+	+	NA
			Arith-Calc	-	+	+	NA	0

*Key: Measure definitions

a	solved problem
b	detection of bugs via program execution
c	characterization of bugs via extended program execution
d	extended testing of attempted repairs
e	backtracking from unsuccessful repairs

Measure scores

+	successful
-	not successful (a) or not attempted (b-e)
0	attempted, unsuccessfully (b-e)
NA	not applicable in solution context

Table 2b

BIP Test Scores and Debugging Measures for NO-TUTORIAL Group

Subject	BIP pretest	BIP posttest		Debugging Measures* a	b	c	d	e
311	113	111	Changer	-	-	-	-	0
			Arith-Calc	-	+	0	0	0
319	96	225	Changer	-	+	+	-	-
			Arith-Calc	-	+	-	0	+
326	130	237	Changer	+	+	+	+	NA
			Arith-Calc	-	+	+	+	+
332	115	186	Changer	+	+	-	+	NA
			Arith-Calc	-	+	0	+	+
337	104	234	Changer	-	-	-	0	0
			Arith-Calc	+	+	+	+	+
339	111	242	Changer	+	+	-	+	NA
			Arith-Calc	+	-	-	+	+

*Key: Measure definitions

a	solved problem
b	detection of bugs via program execution
c	characterization of bugs via extended program execution
d	extended testing of attempted repairs
e	backtracking from unsuccessful repairs

Measure scores

+	successful
-	not successful (a) or not attempted (b-e)
0	attempted, unsuccessfully (b-e)
NA	not applicable in solution context

The completion times for the correct solutions to the CHANGER and ARITH-CALC exercises were also examined to evaluate the hypothesis that the TUTORIAL group would debug more rapidly than the NO-TUTORIAL group. The observed mean completion time, however, was shorter for the NO-TUTORIAL group, primarily because of Subject 339. As indicated by his posttest score in Table 1, this subject was the most proficient programmer at the time of the experiment. (He was also unusually motivated, being one of the few students in BIP who had generated his own programming exercises to supplement BIP's curriculum.) He correctly debugged both CHANGER and ARITH-CALC in short order, characterizing, locating, and repairing the errors apparently by analysis of the program code with little attention to the data provided by program execution. Thus, the debugging exercises, which were difficult for the majority of subjects, seem to have been too easy to tax the ability of Subject 339. Consequently, the data do not indicate that the TUTORIAL group debugged more rapidly.

Returning to the measures in Table 1 for apparent use of the heuristics given in the tutorial, there is marginal evidence that even if the the TUTORIAL group did not solve more problems (or solve them more rapidly) than the NO-TUTORIAL group, they did attempt to apply the guidelines for testing and debugging. The columns labeled b-e correspond to the measures described earlier. Column b indicates whether program execution was attempted and successfully caused error manifestation before the subject engaged in other debugging activities. For TUTORIAL subjects, such detection was successful in every case, except one where the program was executed several times, but the inputs used did not cause error manifestation. While NO-TUTORIAL subjects also did so frequently, in 3 of the 12 cases they did not.

Column c indicates characterization of errors by program execution sufficient to elaborate a description of the malfunction. TUTORIAL subjects attempted to do so in 11 of 12 cases, although in 4 of those cases the attempts were judged to be inadequate; the corresponding results for the NO-TUTORIAL group are 6 of 12, with 2 inadequate attempts.

Columns d and e are the measures of repair testing and backtracking from unsuccessful repairs. Both groups show equivalent evidence for such behaviors.

Examining measures b-e just for the exercises that were completed successfully (measure a), it is interesting to note that for the TUTORIAL group all of the guidelines were applied in each of the five cases. For the NO-TUTORIAL group, in 3 of the 5 correct solutions, the behavior prescribed by one or more of the guidelines was not observed. On the whole, it seems that the subjects who studied the tutorial did try use the guidelines. However, the data from the NO-TUTORIAL group does suggest that a majority of student programmers with the experience level of our subjects have already induced most of the guidelines (or similar heuristics). The differences between the groups are small and allow no strong conclusions. The tutorial text may simply have served to amplify and organize parts of a strategy already known to the subjects who studied it.

The written comments obtained from the TUTORIAL subjects at the conclusion of Session 3 provide some help in determining the effects of the text on their behavior. Figure 19 lists the more informative remarks that subjects made to items 4-7 shown in Appendix G. The comments about the tutorial are positive for the most part. With the

Do you have any suggestions (criticisms), in general, regarding the manner of presentation of the guidelines?

324-- Should have been more time to study them.

Would it have been better if the guidelines had been given to you before you finished the BIP course?

316-- Perhaps better in the long run. Actually ended up doing the things in the guidelines as time went on. Of course, having them given to you right away is less time consuming since you don't have to grope around trying to decide what to do next.

317-- It may have helped, but none of the programs in the course were that complicated that it was necessary, and most if it was fairly obvious.

322-- Didn't really need it in BIP itself except for complex programs.

324-- Yes, I could have studied them at my leisure and really learned them well.

333-- Doesn't make that much difference-- for BIP we didn't have so much as to debug programs. It was pretty much follow the examples.

340-- Not necessarily, these guidelines are pretty basic things to do and self-discovery is probably as useful.

Do you think it would be useful to have BIP introduce this material as part of the course?

316-- Yes.

317-- Yes, it's good to know.

322-- Yes, before presentation of complex problems.

324-- Yes.

333-- Yes, it does help a bit and might relieve the frustration of not having a program work and not knowing how to go about finding what was wrong.

340-- Perhaps.

Other comments.

322-- The last 3 sessions made debugging seem a much more orderly process, i.e., more manageable.

Figure 19. Replies of subjects in the TUTORIAL group to questions in the post-experimental interview (Appendix G).

exception of Subject 340, subjects thought that the guidelines were valuable knowledge, although they were not in agreement about how useful they could be for completing tasks in the BIP curriculum. Several of the subjects recognized that the guidelines are knowledge that they had or would have acquired indirectly through experience, but thought that the idea of teaching such knowledge explicitly could be more efficient.

The debriefing data does point to the inadequacy of minimal instruction, such as our tutorial, for insuring that heuristics will be learned and used by students who need them. The ratings given by subjects on items 1 and 2 of the debriefing questionnaire suggest that (1) they did not find the tutorial especially useful for the test exercises they worked in the experiment (five "3"'s and one "2"), and (2) they thought they were following the guidelines most, but not all, of the time (four "3"'s and two "2"'s). It is very interesting to note that the two "2'"s on item 2 came from Subjects 324 and 333, who had the lowest posttest scores in the TUTORIAL group (Table 2a). This again suggests that the students who had the most to gain from the guidelines could not or would not use them consistently. These two subjects were the only ones who reported referring back to the tutorial while they worked, and 324 was the subject who remarked that he did not have enough time to learn the guidelines. The other TUTORIAL subjects seemed to know the guidelines, but failed either to use all of them as regularly as they might have or to use them appropriately for the test exercises.

Discussion and Conclusions.

The results of the experiment serve to illuminate methodological issues more than to answer the question of whether it is worthwhile to teach debugging heuristics directly. Both the chronology data and subject's comments hint that TUTORIAL subjects recognized the value of the guidelines and tried to use them, but provided no evidence that they became better at debugging programs. The comments are most encouraging, but should be weighed cautiously, since the conditions of the experiment may well have prompted the subjects to tell us what they thought we wanted to hear.

As noted earlier, we were aware of some methodological problems at the outset of the experiment, and our subsequent experience has highlighted these and some other problems that must be solved before a substantial evaluation of teaching troubleshooting/debugging strategies directly can be conducted.

One problem is developing a pedagogy for teaching heuristics--for teaching procedural rather than declarative knowledge. Although we could rationalize a first attempt involving minimal instruction, we anticipated that the limited study of the tutorial, isolated from other instruction in programming, would be insufficient for precisely those students who most needed to improve their debugging--the students who had as yet not induced a viable strategy on their own. It is to be expected that meaningful learning of complex knowledge requires considerable time relative to the learning that takes place in laboratory studies of learning. Our situation of having limited access to student's time is, of course, the rule rather than the exception in a basic research setting. There is a "Catch-22" of sorts in effect: it is difficult to persuade and possibly unethical to compel tuition-paying students to participate in an unvalidated, innovative instructional program, but one cannot provide the needed validation without testing a sufficiently large and representative first group of students.

It is usually possible (as we did) to gain the cooperation of a small group of volunteers who tend to be either students having difficulty and seeking any means to improve themselves or students who are unusually bright and motivated. These individuals are not representative of the student population. Furthermore, small groups of volunteers do not allow for statistical tests of hypotheses which are needed to validate an instructional treatment.

In some cases, it is possible to gain access to a large student population; for example, if the researcher or sympathetic colleagues teach a course into which the new material can be integrated. However, there are ethical issues that surround the compulsory participation of tuition-paying students in experimental courses that are extensions of sponsored research programs rather than products of an instructor's initiative. If the effectiveness of the instruction is very tentative, then students should not be compelled to participate. If the effectiveness is highly probable (and the experiment being conducted only to collect supporting data), then how can a control group that receives less than the best instruction for their time and tuition be justified?

A second methodological problem we encountered is to determine test exercises that will be sensitive to differences that might result from the instructional treatments. The solutions to debugging exercises like those we used require general knowledge of a programming language (e.g., BASIC) and of a supporting computer system (e.g., BIP). In addition, idiosyncratic knowledge acquired from prior debugging may be applicable to a solution. Therefore, test exercises intended to indicate the role of general debugging heuristics can neither be too elementary nor too advanced. If they are too elementary (and hence familiar), idiosyncratic knowledge may enable an immediate solution solely by recognition. If the exercises are too advanced, then the student subject's limited competence with the language and programming system may prevent him from using heuristics successfully.

Another related problem is when, relative to instruction in a programming language, to introduce instruction on general debugging heuristics and test for its effects. If the instruction on heuristics and testing are too early, then students will not understand how to apply the heuristics and test exercises will be too difficult for heuristics to have an effect. If the instruction and testing are delayed too long, then there will be significant differences between students' knowledge of the heuristics induced from their prior experience. In addition, test exercises difficult enough to require use of the heuristics (and not merely pertinent idiosyncratic experiential knowledge) will be so complex that analysis of subjects' behavior will be made more troublesome. The appropriate time to introduce the heuristic instruction is when the students have a minimally sufficient background that allows them to understand and use the heuristics, but not to have realized them spontaneously. Discovering the features that identify that point in time is the problem of course.

In our experiment, presentation of the tutorial and testing of it effects were probably too late for the few general heuristics we wanted students to learn. The behavior of the NO-TUTORIAL group and the comments of the TUTORIAL group indicate that many of the subjects had already inferred some of the heuristics included in the tutorial from their fifteen hours of programming experience in BIP. For students learning BASIC in BIP, presentation of the tutorial (or other instruction on debugging) probably should commence from 7 to 10 hours into the course. At that point, most students have worked with all the major constructs of BASIC and are familiar with the facilities of the BIP system, but have worked on only a few programs complex enough for a general strategy to be useful.

A most fundamental problem for studies of the effects of teaching general debugging heuristics remains the analysis of problem-solving data. In attempting to evaluate the role and effects of general heuristics in debugging, one is in fact trying to characterize not just the result of the problem-solving process, but the process itself. In analyzing the chronologies for the test exercises in the experiment, we found that simple tabulations of behaviors such as listing or running a program are not reliable indicators of the strategy being applied by the subject. Only by examining the structure and content of actions comprising larger episodes were we able to judge whether particular heuristics were applied and their contribution to ultimate solutions. The role of content, or semantics in the scoring process virtually precludes automated chronology analysis.

In our experiment, the collection of "thinking aloud" protocols from subjects as they worked test exercises might have provided data that would have increased the reliability with which chronologies were scored. However, this would have increased the already substantial cost of data analysis. For experiments with sample sizes great enough to allow statistical evaluation of measures abstracted from chronologies, the cost of collecting and examining thinking-aloud protocols would seem prohibitive. Furthermore, for a large-scale study integrated into a real-life instructional system, the collection of thinking-aloud protocols would destroy the advantage of inobtrusiveness obtained by the "invisible" recording of programming chronologies.

Further small-scale studies like that described here could provide a relatively informal and subjective evaluation of materials and methods for teaching debugging knowledge in an explicit manner. The main problems remaining to be solved are how to determine sensitive test materials and how to analyze complex problem-solving data comprehensively and reliably. Although we were unsuccessful in our efforts, one goal that should be pursued is the development of process models for describing debugging behavior in specific domains. Such models could be employed to represent changes in an individual's behavior as the result of instruction, and to contrast the behavior of individuals in different instructional treatments.

As for a large-scale, formal statistical evaluation of whether teaching debugging directly is worthwhile, there are additional problems. Since the constraints of academic research make it difficult to gain access to a large, representative student sample, instructional developments should probably be evaluated outside the research environment. Once an informally validated method for teaching debugging is available, it should be integrated into a real instructional program. Because of the methodological and ethical difficulties of conducting multi-group studies in an actual educational setting, evaluation of student performance would best be made relative to previous groups of students. Even if these problems can be overcome, the data analysis problem remains. It is unlikely that intensive methods suitable to small-scale studies (e.g., process models) will be feasible for large experiments. This will limit the analyses in large studies to gross measures of learning, such as total scores on in-class examinations. Our judgement for the present is that the state-of-the-art is still remote from a definitive large-scale evaluation of how direct instruction in debugging, or other complex problem-solving, will affect the abilities of students.

References

Barr, A., Beard, M., & Atkinson, R.C. The computer as a tutorial laboratory: The Stanford BIP Project. International Journal of Man-Machine Studies, 1976, 8, 567-596.

Brown, J.S. & Burton, R.R. Multiple representations of knowledge for tutorial reasoning. In D.G. Bobrow and A. Collins (Eds.), Representation and understanding: Studies in cognitive science. New York: Academic Press, 1975.

Brown, J.S., Burton, R.R., Hausmann, C., Goldstein, I., Huggins, B., & Miller, M. Aspects of a theory for automated student modelling. BBN Report No. 3549, Bolt Beranek and Newman, Inc., Cambridge, Mass., May, 1977.

Brown, J.S., Rubenstein, R., & Burton, R.R. Reactive learning environment for computer assisted electronics instruction. BBN Report No. 3314, Bolt Beranek and Newman, Inc., Cambridge, Mass., October, 1976.

Carr, B., & Goldstein, I.P. Overlays: A theory of modelling for computer aided instruction. MIT AI Memo 406, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., February, 1977.

Collins, A.M. Processes in acquiring knowledge. In R.C. Anderson, R.J. Spiro, & W.E. Montague (Eds.), Schooling and the acquisition of knowledge. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1977.

Dahl, O.J., Dijkstra, E.W., & Hoare, C.A.R. (Eds.), Structured programming. New York: Academic Press, 1972.

Finch, C.R. Troubleshooting instruction in vocational-technical education via dynamic simulation. Research Report, Dept. of Vocational Education, The Pennsylvania State University, August, 1971.

Goldstein, I. Summary of MYCROFT: A system for understanding simple picture programs. Artificial Intelligence, 1975, 6, 249-288.

Miller, M.L., & Goldstein, I.P. Overview of a linguistic theory of design. AI Memo 383, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., December, 1976a.

Miller, M.L., & Goldstein, I.P. SPADE: A grammar based editor for planning and debugging programs. AI Memo 386, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., December, 1976b.

Newell, A. Production systems: Models of control structures. In W.G.

Chase (Ed.), Visual Information Processing. New York: Academic Press, 1975.

Newell, A. & Simon, H.A. Human problem solving. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

Nilsson, N. Problem solving methods in artificial intelligence. New York: McGraw-Hill, 1971.

Norman, D.A., Gentner, D.R., and Stevens, A.L. Comments on learning schemata and memory representation. In D. Klahr (Ed.), Cognition and instruction: Tenth annual Carnegie Symposium on cognition. Hillsdale, N.J.: Erlbaum Associates, 1976.

Palormo, J.M. Computer programmer aptitude battery. Chicago: SRA, 1964.

Papert, S.A. Teaching children thinking. AI Memo 247, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., 1971.

Pirsig, R.M. Zen and the art of motorcycle maintenance. New York, Bantam Books, 1974.

Polya, G. How to solve it. Garden City, N.Y.: Doubleday, 1957. (Originally published in 1945.)

Potter, N.R., & Thomas, D.L. Evaluation of three types of technical data for troubleshooting: Results and project summary. Report AFHRL-TR-76-74(I), Air Force Human Resources Laboratory, Brooks Air Force Base, Texas, September, 1976.

Quillian, M.R. The teachable language comprehender: A simulation program and a theory of language. Communications of the Association for Computing Machinery, 1969, 12, 459-476.

Resnick, L.B. Task analysis in instructional design: Some cases from mathematics. In D. Klahr (Ed.), Cognition and instruction: Tenth annual Carnegie Symposium on cognition. Hillsdale, N.J.: Erlbaum Associates, 1976.

Ruth, G. Analysis of algorithm implementations. MAC TR-130, Massachusetts Institute of Technology, Cambridge, Mass., May, 1974.

Schoenfeld, A.H. Can heuristics be taught? Unpublished report, Group in Science and Mathematics Education, University of California, Berkeley, Calif., 1977a.

Schoenfeld, A.H. Presenting a strategy for indefinite integration. Unpublished report, Group in Science and Mathematics Education, University of California, Berkeley, Calif., 1977a.

Sacerdoti, E.D. A structure for plans and behavior. Technical Note 109, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Calif., August, 1975.

Stevens, A.L., & Collins, A.M. *The goal structure of a Socratic tutor.* BBN Report No. 3518, Bolt Beranek and Newman, Inc., Cambridge, Mass., March, 1977.

Sussman, G.J. *A computational model of skill acquisition.* AI-TR-297, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., August, 1973.

Wickelgren, W.A. *How to solve problems: Elements of a theory of problems and problem solving.* San Francisco: Freeman, 1974.

Woods, W.A. Transition network grammars for natural language analysis. *Communications of the Association for Computing Machinery*, 1970, *31*, 591-606.

Woods, W.A. What's in a link: Foundations for semantic networks. In D.G. Bobrow and A. Collins (Eds.) *Representation and understanding: Studies in cognitive science.* New York: Academic Press, 1975.

Appendix A. Tutorial Debugging Text

TESTING THE PROGRAM

After you have written a program, you need to test it to make sure there are no errors, or "bugs", in it. Many programs are designed to be run more than once. For example, some programs are written to compute payrolls and must be run at the end of every pay period; other programs are written to tabulate students' grades and are run at the end of each grading period.

> Since the conditions under which a program is run will not be EXACTLY the same each time the program is run, it is important to realize that just because a program works correctly for one set of conditions, you cannot assume that it will work correctly under all other conditions.

For example, in some programs different kinds of input cause different parts of the program to be executed; thus to check a program you need to run it using all possible types of input for which the program was designed. You must test every possible pathway through the program.

> TEST THE PROGRAM WITH ALL POSSIBLE TYPES OF INPUT FOR WHICH IT IS DESIGNED.

The following program demonstrates how different inputs cause different parts of the program to be executed.

```
10 X = INT(RND * 1001)
20 PRINT "I AM THINKING OF A NUMBER BETWEEN 0 AND 1000."
30 L = 0
40 H = 0
50 PRINT "WHAT DO YOU THINK MY NUMBER IS? "
60 INPUT G
70 IF G = X THEN 230
80 IF G > X THEN 160
90 IF L = 1 THEN 140
100 PRINT "TOO LOW; GUESS AGAIN"
110 L = 1
120 H = 0
130 GOTO 60
140 PRINT "IT'S STILL TOO LOW.  GUESS AGAIN"
150 GOTO 60
160 IF H = 1 THEN 210
170 PRINT "TOO HIGH; GUESS AGAIN"
180 H = 1
190 L = 0
200 GOTO 60
210 PRINT "YOU'RE STILL TOO HIGH SO GUESS AGAIN"
220 GOTO 60
230 PRINT "RIGHT! MY NUMBER IS ";X
240 END
```

This program generates a random integer (X) between 0 and 1000. The user then tries to guess the number (line 60, INPUT G). If the user guesses the number correctly (line 70), then line 230 is executed, and the program prints "RIGHT...". Otherwise, if the user guesses a number that is too high, then line 160 is executed. If the preceding guess was also too high (which is the case if H = 1), then line 210 is executed, "YOU'RE STILL TOO HIGH SO GUESS AGAIN" is printed, and line 220 causes a jump back to line 60. If the preceding guess was not too high (if H is not 1), then line 170 is executed and "TOO HIGH; GUESS AGAIN" is printed. AND SO ON.

Checking the "GUESS MY NUMBER" program requires that every possible class of input be tested, i.e., an input (guess) that is lower than the number generated (X), another consecutive input that is still lower than X, an input that is higher than X, another consecutive input that is still higher than X, and an input that is equal to X.

TEST THE PROGRAM WITH THE EXTREME VALUES THAT THE INPUT CAN HAVE.

Initially, it is a good idea to test a program with the extreme values that the input can have. It is usually not hard to think of the extreme types of input which your program must handle, and this test may reveal errors in your program. In the "GUESS MY NUMBER" program, the two extreme input values (guesses) are "0" and "1000".

If, during the testing of your program with different inputs, the output is ever wrong, then there is something wrong in your program. You must then try to characterize what is wrong.

CHARACTERIZING THE ERROR

CHARACTERIZE THE WAY THE ERROR(S) SHOWS UP IN TERMS OF THE INPUT AND OUTPUT.

Before you try to determine which part of the program is working incorrectly (unless it's immediately obvious), you should describe what is wrong with the output. For example, in the last program, if you input a guess of 0 and the program prints "TOO HIGH", your description would include the fact that the output is backwards for a too-low guess. If, in addition, the program said "TOO LOW" in response to an input of 1000, then you could characterize the erroneous behavior as being wrong for both too-low and too-high guesses. Describing the "symptom" carefully is very helpful in leading you to locate its cause (the bug in the program); the process is similar to a doctor asking questions about the exact location and nature of your pain before s/he begins to choose the appropriate treatment.

Since the output is the result of following the steps of the program, if you can characterize how the output varies from what it should be, given a particular input, then that may indicate which part of the program isn't doing what it was intended to do. In order to characterize the error(s) in a program, you should test it with different types of input in order to see how different kinds of input affect the output. For example, perhaps the output is correct or closer to the correct answer for certain inputs than it is for other inputs. If so, then it is important to ask how the inputs that give correct or "more correct" answers differ from the inputs that give "less correct" answers. If these two inputs require different parts of the program to be run, then that could guide you to the part of the program that is not working as it was intended.

SOMETIMES A PROGRAM GIVES THE CORRECT OUTPUT FOR SOME INPUTS BUT NOT FOR OTHERS. WHEN THIS HAPPENS YOU SHOULD EXAMINE THE DIFFERENCE(S) BETWEEN THE INPUTS FOR WHICH THE PROGRAM WORKS AND THE ONES FOR WHICH IT FAILS.

The following program was written to give change to a customer when the item being bought costs less than a dollar. The change can be in half dollars, quarters, dimes, nickels, and pennies. The program is designed to print the amount of change in cents and then give the fewest possible coins in change.

```
10 PRINT "TYPE THE PRICE OF YOUR ITEM.  IT SHOULD BE < $1.00"
20 INPUT X
30 LET C = 100 - X
40 PRINT "YOUR CHANGE FROM $1 IS " ; C ; "CENTS"
50 LET H = 0
60 LET Q = 0
70 LET D = 0
80 LET N = 0
90 IF C< 50 THEN 120
100 H = H+1
110 C= C-50
120 IF C< 25 THEN 140
130 Q=Q+1
140 IF C<10 THEN 180
150 D = D+1
160 C = C-10
170 GOTO 140
180 IF C < 5 THEN 210
190 N = N+1
200 C = C-5
210 PRINT "HERE IS YOUR CHANGE"
220 PRINT H ;" HALF DOLLARS"
230 PRINT Q ;" QUARTERS"
240 PRINT D ;" DIMES"
250 PRINT N ;" NICKELS"
260 PRINT C ;" PENNIES"
270 END
```

The programmer might decide that a good first test for this program would be the case in which one of each coin should be returned to the customer (1 half dollar, 1 quarter, 1 dime, 1 nickel, and 1 penny, for a total of 91 cents). So price of the item (the input number) must be 9 cents.

```
Input: 9
Output: YOUR CHANGE FROM $1 IS 91 CENTS
        HERE IS YOUR CHANGE
        1 HALF DOLLARS
        1 QUARTERS
        4 DIMES
        0 NICKELS
        1 PENNIES
```

It is immediately apparent that the wrong number of dimes and nickels has been returned. This might lead the programmer to test the program with an input which should return a dime and a nickel.

```
Input:  85
Output: YOUR CHANGE FROM $1 IS 15 CENTS
        HERE IS YOUR CHANGE
        0 HALF DOLLARS
        0 QUARTERS
        1 DIMES
        1 NICKELS
        0 PENNIES
```

The output is correct, so the problem certainly isn't with the dimes and nickels alone. Before the program is run again. the first test, the one with the incorrect output, should be re-examined. Evidence about the nature of the error might have been overlooked because of the obviously wrong number of nickels and dimes in the output. The programmer might add up the coins to see how much change in cents was actually returned in the first test and find the total to be 116 cents rather than 91 cents. The difference between these two sums is 25 cents, and this might suggest to the programmer that the error is related to the extra 25 cents. At this point the program should be examined for an error related to the '25 cents' calculations. While reading through that part of the program, the programmer should notice that another line is needed between 130 and 140 to subtract 25 from the total cents left at that point, or C. The absence of that line caused an extra 25 cents in the output (since when a quarter was given in change, 25 cents was not subtracted from the total cents still owed the customer). After this change, the testing of the program should be continued.

AFTER A CHANGE, RETEST THE PROGRAM USING ALL POSSIBLE TYPES OF INPUT FOR WHICH THE PROGRAM WAS DESIGNED.

After you've characterized the wrong output, located the section of code that you believe is responsible for the erroneous output, and changed that code to correct the error, the program must be tested once again for all possible types of input. You must retest your program thoroughly for several reasons: for example, you may have corrected the program so that it works for only one or two additional types of input; or the program may not work for some inputs that were handled correctly before your change, i.e., your change interacts with a portion of the program that was executing correctly before the change and now makes it give erroneous output. The program must be retested with all types of input, even those that were handled correctly before the change.

The following program, which demonstrates the importance of retesting after a change, asks the user to type in two numbers and tells him/her how many numbers lie between the two numbers (inclusive). For example, there are 3 numbers between 5 and 7, i.e., 5, 6, and 7.

```
10 PRINT "TYPE TWO NUMBERS, AND I WILL TELL YOU HOW MANY"
20 PRINT "NUMBERS ARE BETWEEN YOUR TWO NUMBERS (INCLUSIVE)."
30 INPUT X,Y
40 IF X < Y THEN 80
50 H = X
60 L = Y
70 GOTO 110
80 H = Y
90 L = X
100 P = L
110 N = 1
120 P = L + 1
130 N = N + 1
140 IF P < H THEN 120
150 PRINT "THERE ARE ";N; " NUMBERS BETWEEN "; L; " AND "; H
199 END
```

The user types two numbers, which are assigned to the variables X and Y. The variables H and L are used to hold the high and low numbers, respectively. So, if X is higher than Y, its value is assigned to H and the value of Y is assigned to L; if Y is higher than X, the H and L assignments are made in the opposite direction. The variable P is used to count from the low number up to the high number, and N is used to keep track of how many numbers are encountered along the way. Thus, if the user types 5 and 6 as the X and Y input, L becomes 5, H becomes 6, P counts from 5 to 6, and N ends up with 2.

When the program is run, it gives the correct output only when the two numbers are adjacent to each other, e.g., "5" and "6", or "6" and "5". The output is THERE ARE 2 NUMBERS BETWEEN 5 AND 6". Any pair of non-adjacent numbers causes an error message to be printed, which says that the program might be in an infinite loop. The programmer characterizes the error as occurring when any two non-adjacent numbers are given as input.

In the program above, where only adjacent numbers X and Y (both X < Y and X > Y) give the correct output, the programmer might go through the following reasoning process while looking for the error:
-AHA, P doesn't get set to L when X > Y, so line 70 should branch to line 100.
(The programmer changes line 70 to GOTO 100, and runs the program for X < Y and X > Y. S/He gets the same results as before, i.e., the program gives the correct output for adjacent pairs of numbers, otherwise it seems the program is in an infinite loop.)
-Well, same error, perhaps line 100 is superfluous, since line 120 assigns a value to P, so I'll delete line 100, undo the previous change so that line 70 is GOTO 110, and run the program again.
(The result of testing the program is the same as before: it works for adjacent number pairs, but every other pair gives infinite loop message.)
-AHA, line 120 should be P = P + 1, otherwise P is always reset to equal L, the lowest number, plus 1, and P can never reach H unless H is L+1! I'll change line 120 and run the program again.
(The program gives the error message "Line 120 VARIABLE WITHOUT A KNOWN VALUE--P" for both X < Y and X > Y.)
-Hmm. That's the first time I've gotten that message. Why does P suddenly not have a value? I know! P was L+1, and I changed it to P=P+1; so the line that I deleted, which set P equal to L, is necessary. I'll put line 100, P = L, back into the program and run it.
(S/He tries several pairs of input, e.g., 5 and 6, 5 and 8, 4 and 9; and they all work. Unfortunately, cases in which X > Y aren't tested.)
-Success! It finally works.

The program was fixed for one type of input, that is, for cases in which X is less than Y; but two other types of input were not tested, X greater than Y and X equal to Y. If examples of these two types of input had been tested, the error message "Line 120 VARIABLE WITHOUT A KNOWN VALUE--P" would have told the programmer that P still wasn't being assigned. Further examination of the program would have shown her/him that line 70 should, indeed, branch to line 100, so that P gets an initial value when X > Y and X = Y. Thus all types of input for which a program is designed must be retested after a change is made.

Sometimes you make a change to the program, and the output is still wrong. You have to make the choice between leaving the change in the program or returning the program to its state before the change.

Take the program, for example, which tells the user how many numbers are between two input values.

```
10 PRINT "TYPE TWO NUMBERS, AND I WILL TELL YOU HOW MANY"
20 PRINT "NUMBERS ARE BETWEEN YOUR TWO NUMBERS (INCLUSIVE)."
30 INPUT X,Y
40 IF X < Y THEN 80
50 H = X
60 L = Y
70 GOTO 110
80 H = Y
90 L = X
100 P = L
110 N = 1
120 P = P + 1
130 N = N + 1
140 IF P < H THEN 120
150 PRINT "THERE ARE "; N; " NUMBERS BETWEEN "; L; " AND "; H
199 END
```

Suppose that a beginning programmer is told that this program has an error and is asked to find and correct it. S/he might not have these guidelines for finding an error. Since the program is short, s/he might decide to examine the code before running the program. After doing this, the person might say "This equals business in lines 50 through 90 is confusing. Seems to me they're double assigning things. H and L are being given two values... I think maybe 50 and 60 can be deleted. I'll try it."
After deleting lines 50 and 60, the program is run. For inputs where X (the first input) is less than Y (the second input), the correct answer is given, and for all other inputs, the error message "Line 120 VARIABLE WITHOUT A KNOWN VALUE--P." is printed.

Since the program has not been corrected by the change, and even more errors may have been introduced into the program, the change should be undone and lines 50 and 60 restored to the program.

The reason given for deleting lines 50 and 60, i.e., that H and L are each being given two values, is true of course, but the person did not examine the program carefully enough, because s/he did not notice that the values given to H and L in lines 50 and 60 are used in one pathway through the program, and the values given in lines 80 and 90 are used in a different pathway through the program. Going through the step-by-step execution of a program (exactly as the computer would) is a very valuable way to find errors. However, after a superficial examination of a program, deleting a line is probably a bad idea. The person writing a program usually has a reason for putting in each line, and before you delete a line, you should understand the intended purpose of that line.

The programmer should have run this program before examining the code. The error message would have given her/him the information that P was not being defined when either X > Y or X = Y. This information points out which pathway through the program contains the error.

THUS, EVEN IF A PROGRAM IS SHORT AND EASY TO TRACE BY HAND, YOU SHOULD FIRST RUN THE PROGRAM. (ERROR MESSAGES, AS WELL AS A CHARACTERIZATION OF THE ERROR IN TERMS OF INPUT AND OUTPUT, CAN BE VERY HELPFUL IN FINDING AN ERROR.)

THEN

IF YOU MAKE A CHANGE TO A PROGRAM, AND IT STILL GIVES THE SAME ERRONEOUS OUTPUT, RESTORE THE PROGRAM TO ITS STATE BEFORE THE CHANGE. YOU HAVEN'T FOUND THE ERROR(S) IN THE PROGRAM, AND YOU MAY HAVE INTRODUCED A NEW ERROR.

Sometimes, when you make a change to correct a program, the output will still be wrong after the change, but you should leave the change in the program. (Obviously, if you see any typographical errors that you made while typing in the program, you should correct those.)

IF YOU MAKE A CHANGE TO A PROGRAM, AND THE OUTPUT IS STILL WRONG: IF THE CHANGE CORRECTS ONE PART OF THE PROGRAM (e.g., one part of the output), THEN LEAVE THE CHANGE IN THE PROGRAM.

It may be the case that there is more than one error in the program, and you have found one but not all of the errors. Take the following program as an example.

```
10 PRINT "THIS PROGRAM TALLIES THE VOTES OF 5 PEOPLE."
20 PRINT "TO VOTE YES, TYPE 1; TO VOTE NO, TYPE 0."
30 Y = 0
40 N = 0
50 FOR I = 1 TO 5
60 PRINT "VOTE NUMBER ";I
70 INPUT V
80 IF V = 1 THEN 100
90 N = N + 1
100 Y = Y + 1
110 NEXT I
120 IF Y <> N THEN 150
130 PRINT "TIE VOTE"
140 GOTO 190
150 IF Y < N THEN 180
160 PRINT "THE NO VOTE WINS"
170 GOTO 190
180 PRINT "THE YES VOTE WINS"
190 END
```

This program tallies the YES and NO votes of 5 people, then prints whether the 'YES's or 'NO's win. The user inputs the 5 votes. He types 1 for a YES vote and 0 for a NO vote.

The program is run. When all the votes are either YES or NO, then "TIE VOTE" is printed. When the number of YES votes input is greater than the number of NO votes, "THE NO VOTE WINS" is printed. When the number of NO votes is greater than the number of YES votes, "THE NO VOTE WINS" is printed. This program does the wrong thing for three of the four different kinds of input!

Since the program gives the correct output when the number of NO votes exceeds the number of YES votes. i.e., "THE NO VOTE WINS" (except in the extreme case where all the votes are NO); the programmer might check to see why line 180, "THE YES VOTE WINS", is not printed when it should be. S/He looks at line 180 and the line, itself, looks all right. S/He looks through the program to find the line that goes to line 180, which is line 150. S/He sees an error! In line 150 if Y, which tallies the YES votes, is LESS THAN N, which tallies the NO votes, then line 180 is executed, which prints "THE YES VOTE WINS". Line 150 should say "if Y is GREATER THAN N then execute line 180". This change is made to the program, and it is run.

After the correction, when the YES vote is greater than the NO vote, "THE YES VOTE WINS" is printed; but when the NO vote is greater than the YES vote, "THE YES VOTE WINS" is printed. It seems like the same wrong output as before the change, only switched around! (As before, when all 5 votes are either YES or NO, a "TIE VOTE" is printed.)

The programmer must decide whether to leave the change or not, i.e., 150 IF Y > N THEN 180; 180 PRINT "THE YES VOTE WINS". Since Y tallies the YES votes, and N counts the NO votes, if Y > N, then "THE YES VOTE WINS" SHOULD be printed. The programmer decides to leave the change and look for errors in other parts of the program.
In line 150 (which now says "IF Y > N..."), if N is greater than Y, then line 160 is executed, which prints "THE NO VOTE WINS", so that part of the program is correct.

This program illustrates the importance of testing the program with the extreme values that the input can have, in this case, 5 YES votes or 5 NO votes. Whenever the input is all YES votes or all NO votes, "TIE VOTE" is printed (line 130). The programmer looks for the line that must precede the execution of line 130. If line 130 was executed, then Y and N must have been equal in line 120. With an odd number of votes, this isn't possible. Because Y and N are both initialized to 0 (lines 30 and 40), something must be wrong with the counting procedure. The programmer examines the FOR loop, where the votes are counted. S/He notices that if the vote is NO, both N and Y are incremented! So, there should be a line 95 which says "GOTO 110". The change is made. The different possible types of input are retested. Success.

SUMMARY OF GUIDELINES

TESTING THE PROGRAM

TEST THE PROGRAM WITH ALL POSSIBLE TYPES OF INPUT FOR WHICH IT IS DESIGNED.

TEST THE PROGRAM WITH THE EXTREME VALUES THAT THE INPUT CAN HAVE.

CHARACTERIZING THE ERROR

CHARACTERIZE THE WAY THE ERROR(S) SHOWS UP IN TERMS OF THE INPUT AND OUTPUT.

EVEN IF A PROGRAM IS SHORT AND EASY TO TRACE BY HAND, YOU SHOULD FIRST RUN THE PROGRAM. (ERROR MESSAGES, AS WELL AS A CHARACTERIZATION OF THE ERROR IN TERMS OF INPUT AND OUTPUT, CAN BE VERY HELPFUL IN FINDING AN ERROR.)

SOMETIMES A PROGRAM GIVES THE CORRECT OUTPU[illegible]R SOME INPUTS BUT NOT FOR OTHERS. WHEN THIS HAPPENS YOU S[illegible]D EXAMINE THE DIFFERENCE(S) BETWEEN THE INPUTS FOR WHICH THE [illegible]ROGRAM WORKS AND THE ONES FOR WHICH IT FAILS.

AFTER A CHANGE, RETEST THE PROGRAM USING ALL POSSIBLE TYPES OF INPUT FOR WHICH THE PROGRAM WAS DESIGNED.

IF YOU MAKE A CHANGE TO A PROGRAM, AND IT STILL GIVES THE SAME ERRONEOUS OUTPUT, RESTORE THE PROGRAM TO ITS STATE BEFORE THE CHANGE. YOU HAVEN'T FOUND THE ERROR(S) IN THE PROGRAM, AND YOU MAY HAVE INTRODUCED A NEW ERROR.

IF YOU MAKE A CHANGE TO A PROGRAM, AND THE OUTPUT IS STILL WRONG: IF THE CHANGE CORRECTS ONE PART OF THE PROGRAM (e.g., one part of the output), THEN LEAVE THE CHANGE IN THE PROGRAM.

OPEN BOOK QUIZ

NAME:

1) After writing a program why should you test it with all the different types of input that it was designed to handle?

2) Testing a program gives the following results:

Input: 0 (number of days)
Expected Output: 0 dollars and 0 cents
Output: 0 dollars and 0 cents
Input: 1 (number of days)
Expected Output: 0 dollars and 1 cent
Output: 0 dollars and 2 cents

Input: 3 (number of days)
Expected Output: 0 dollars and 7 cents
Output: 0 dollars and 14 cents

Input: 10 (number of days)
Expected Output: 10 dollars and 23 cents
Output: 20 dollars and 46 cents

Characterize the error in this program.

3) If a program gives the correct output for some inputs but not for others, you should (a, b, or c)
(a) Scratch it and start over.
(b) Hope that a user will only use inputs for which the program gives the correct output.
(c) Examine the difference(s) between the inputs for which the program works and the ones for which it fails.

Why?

4) After making a change to a program why should you RETEST the program with all types of input for which it was designed?

5) If you make a change to a program in order to correct it, and it still gives the SAME erroneous output, you should (a or b)
(a) Leave the change in the program.
(b) Restore the program to its state before the change.

Why?

6) If you are told that a program has an error in it, and you are asked to find and correct that error, what is the first thing you should do after reading the program description? (a, b, or c)
(a) Go through the step-by-step execution of the program by hand (as the computer would) in order to find the error.
(b) Run the program with the different types of input for which it was designed in order to characterize the error.
(c) Read over the program, delete any suspicious looking lines, and run the program.

Appendix C: Introduction to the Tutorial Text

INTRODUCTION

This is an attempt to give you information that will help you to find the errors in a computer program more easily. This information will be presented in the form of rules which apply to certain situations, rules-of-thumb that have been formulated from the experience of programmers who have spent many hours in searching for errors, or "bugs", in computer programs.

Once you know there is an error in your program, the goal is to find it with a minimal amount of time and effort. Since it is very hard to formalize ALL the knowledge about finding bugs that an experienced programmer would have, the rules presented here will be general rules that provide the best way to go about finding the error(s) in a computer program most of the time. They provide a general framework, and as you gain experience, you will be able to add exceptions to these rules. If you follow these rules, the process of finding the error may seem to take longer than it could; however, it is much more likely that you will find the error or ALL of the errors in the program, and that can save quite a bit of time in the long run. As you gain experience the process of finding the error may go faster.

These rules lead to the desired result (which is finding the error with a minimal amount of time and effort) most of the time. Everyone employs this type of rule when trying to solve problems. When there is more than one possible course of action to reach a goal, a person may weigh the positive and negative effects of each action under consideration before s/he makes a decision. For example, suppose you are in a strange city, you need to get from where you are to a hotel in another part of the city, and a map of the city is all the information you have to help you plan your route. In that situation (going from one place to another in a strange city), a general rule-of-thumb you might have is to stay on main streets. If you have this rule, it is because of knowledge you have gained (e.g., from your own experience, or from talking to friends, etc.), for example this knowledge could be:
(1) street signs are more visible on main streets
(2) if you get lost, it is easier to ask directions on a main street
(3) main streets are safer, if that section of town is unknown to you
(4) a backstreet route may make crossing intersections more difficult.
Even if your general rule is to stay on main streets in a strange city, you may choose not to follow the rule in certain instances. Perhaps the most important consideration is getting to place X as quickly as possible, and you choose a backstreet route because it is shorter and will allow you to miss the rush hour traffic on the main streets. The circumstances under which you make a decision will vary (e.g., finding the "best" route, where "best" means one that fulfills certain requirements such as "requires least amount of time"), and general rules will not always give the best solution to a particular problem. If you are a beginning programmer who is trying to find the errors in your program, since you have no programming experience upon which to formulate general rules for finding the error, being given these general

rules should save you both time and effort. After you have more programming experience, you will be able to add exceptions to these rules.

Appendix D. Test exercise CHANGER

This program was written to give change to a customer when the item being bought costs less than a dollar. The change can be in half dollars, quarters, dimes, nickels, and pennies. The program should print both the amount of change in cents and then the FEWEST possible coins in change.

```
5/31/77 11:12:06
  19
10 PRINT "TYPE THE PRICE OF YOUR ITEM. IT SHOULD BE < $1"
20 PRINT " (THE PRICE SHOULD BE IN CENTS, E.G., 25, 49.) "
30 INPUT X
40 LET C = 100 - X
50 PRINT "YOUR CHANGE FROM $1 IS " ; C ; " CENTS."
60 DATA 50, "HALF-DOLLARS", 25, "QUARTERS", 10, "DIMES"
70 DATA 5, "NICKELS", 1, "PENNIES"
80 PRINT "HERE IS YOUR CHANGE"
90 N = 0
100 READ A
110 READ D$
120 IF A = 1 THEN 180
130 IF C < A THEN 160
140 N = N + 1
150 C = C - A
160 PRINT N; " ";D$
170 GOTO 90
180 PRINT C; " "; D$
199 END
```

143

Appendix E. Test Exercise DRILL

TASK DRILL

We want you to write a BASIC program that presents simple arithmetic problems -- your own computer-assisted instruction program: The required program will be longer and more complex than those you have previously completed in BIP, but you probably worked with all the BASIC statements you will need. You will have at most 1 and 1/2 hours to work on the task during a single sitting at the terminal. Do your best to complete a program that satisfies the specifications given below (use the DEMO to see a fancy model program in operation), but you will be paid even if you can't do so in the allotted time. (Given the time constraint, one possible approach is to design your program and then implement it in successive stages, adding more advanced features at each stage; however, you are free to tackle the problem in any manner you prefer.)

After 1 and 1/2 hours (or sooner, if you are confident your program worked correctly), we will examine your program and try it out. If your program isn't satisfactory, you will have at most another 1/2 hour to fix it.

To begin work, signon to BIP and type the command TASK DRILL. BIP will not print the text of the problem as it does normally: instead refer to the specifications given below in these instructions. During your work you can use any BIP commands except the following: MODEL, MORE, REP, DEMO TRACE. Use RUN to try out your program as many times as you like. Since you can't use MORE, you will have to be the judge of whether your program satisfies the specifications before you are ready to have us look at it. You may use the BIP manual. Run the DEMO as often as you like, but do not ask for the MODEL; _if you use the MODEL command, we cannot pay you for your work_. REP does not work for this task, but FLOW does. There is paper for you to do any scratch work you want to: please number any sheets you use and turn them in at the end of the session. Since the program you will write will be too long to LIST on the terminal screen at one time, we have set-up the teletypes in the room to provide hardcopy of your program (you may use the LIST command, but the output will go off the top of the screen-- use the 'HOLD' key on the terminal to stop-and-start the output). To obtain hardcopy, SAVE your program in BIP as a file and then, at the teletype, type (as requested) your student number and name of the file you SAVE'd. You may list your program on the teletype as many times as you like, and write on the listings, but we want you to turn in the listings at the end of the session.

Program specifications for TASK DRILL:

1) The user selects whether he wants to do addition or subtraction problems.

2) The user selects whether he wants problems that involve 1-digit integers (1-9) or 2-digit integers (10-99, not 1-99). The integers used in each problem are randomly generated.

3) The user specifies how many problems he will work, with a minimum of 1 and a maximum of 10 problems.

4) Subtraction problems must always have an answer that is equal to or greater than 0 (no negative answers).

5) The answer to each problem is checked and appropriate feedback is printed. Feedback on incorrect answers includes the correct answer.

6) When the user finishes the number of problems he specified, the program prints his score as number and percent correct.

7) Assume that the user of the program is naive and may type invalid responses to any question asked by the program. The program should not "blow up" in these cases. In general, it should also provide clear questions and print output suitable for naive users. Try to write a program you would want to show off to another programmer. It does not have to have all the fancy features of the DEMO program, but should satisfy the requirements listed here.

Appendix F. Test Exercise ARITH-CALC

PROGRAM ARITH-CALC

This program is supposed to act as a calculator for simple arithmetic expressions (e.g., 9*8, 43/5+11, 10^2*777) which have no parentheses to organize them. It is intended to perform the operations in an expression in a left to right order; for example, 10+2*6 first adds 10 and 2 to get 12 and then multiplies 12 by 6 to get 72. Note that this is different from the way BASIC evaluates such expressions (BIP manual II.12). The program is intended to handle only "well-formed" input from the user and is expected to behave unpredictably if the input contains bad characters. The following are examples of expressions for which the program is and is not expected to work.

SHOULD WORK FOR	NOT EXPECTED TO WORK FOR
4*5	4 + 5 (no spaces allowed)
334/667^23+8*3	4+(5*3) (no parentheses)
8/0 (gives error message)	4.5+13 (no decimals)
	4A7+13 (illegal character)

The program is complex. The main difficulty is that the expression input by the user is a string, and strings in BASIC (and parts of strings) cannot be multiplied, added, etc. The string must therefore be analyzed to find the strings of digits it contains (i.e., the numbers in the expression) and then these strings of digits must be "translated" into numeric values that can be manipulated with arithmetic operations. Part of this work is done by a subroutine in the program. BIP didn't give you any work with subroutines (BIP Manual II.22), and we don't expect you to understand the one in this program. The way in which it is used is explained by the REM statements in the program. <u>The error(s) in this</u>

program is(are) not in the subroutine or in the first lines of the program which set up an array of values used by the subroutine. The error(s) is(are) in the part of the program delimited by the REM statements containing stars (asterisks). The program can be fixed with only minor modifications (extensive re-writing is unnecessary).

To get the program into your program space, say GET ARITH-CALC after you signon to BIP. You may RUN, LIST, and TRACE the program as you please, but do not use FLOW. Make any changes you wish; if, at any point, you want to get the original program back, then just say GET ARITH-CALC again.

Appendix G. Post-experimental Questionnaire for TUTORIAL group

QUESTIONNAIRE

NAME:

RIP NO.:

(1) Do you feel like the material you read during the first session was useful to you in the subsequent tasks? (Circle the appropriate number.)

Not Useful At All				Extremely Useful
1	2	3	4	5

(2) As you were testing and debugging programs during the sessions, did you follow the guidelines presented in the material? (Circle the appropriate number.)

Never			Always
1	2	3	4

Did you find it difficult to remember the guidelines? (Yes or No)

If so, did you refer back to the lesson?

(3) Was any part of the lesson difficult to understand, or unclear, etc.?

If so, which part(s)?

(4) Do you have any suggestions (criticisms), in general, regarding the manner of presentation of the guidelines?

(5) Would it have been better if the guidelines had been given to you before you finished the BIP course? Please explain your answer.

(6) Do you think it would be useful to have BIP introduce this material as part of the course?

(7) Would you like to make any further comments on the three sessions you just completed or on the BIP course itself?

150