

1-1-1988

A study of high school students' learning Logo : microanalysis of uses of variables.

Richard J. Horlick

University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_1

Recommended Citation

Horlick, Richard J., "A study of high school students' learning Logo : microanalysis of uses of variables." (1988). *Doctoral Dissertations 1896 - February 2014*. 4359.

https://scholarworks.umass.edu/dissertations_1/4359

This Open Access Dissertation is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations 1896 - February 2014 by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

UMASS/AMHERST



312066008667882

C

A STUDY OF HIGH SCHOOL STUDENTS' LEARNING LOGO:
MICROANALYSIS OF USES OF VARIABLES

A Dissertation Presented

by .

Richard J. Horlick

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF EDUCATION

February 1988

School of Education

© Copyright by Richard J. Horlick 1988

All Rights Reserved

A STUDY OF HIGH SCHOOL STUDENTS LEARNING LOGO:
MICROANALYSIS OF USES OF VARIABLES

A Dissertation Presented

by

Richard J. Horlick

Approved as to style and content by:

Howard A. Peelle

Howard A. Peelle, Chairperson of Committee

Allen R. Hanson

Allen Hanson, Member

John Clement

John Clement, Member

George E. Urch

George E. Urch, Acting Dean
School of Education

ABSTRACT

A STUDY OF HIGH SCHOOL STUDENTS' LEARNING LOGO: MICROANALYSIS OF USES OF VARIABLES

FEBRUARY 1988

RICHARD J. HORLICK, B.A., UNIVERSITY OF MASSACHUSETTS

M.A., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Howard A. Peelle

This study explores how high school students develop an understanding of certain programming language constructs, particularly focussing on concepts and misconceptions of variables in Logo. Using a case-study approach, three nonexpert subjects were asked to solve simple programming problems that require using variables in different ways. Selected protocols on one problem were subjected to a "cognitive microanalysis", which involves transcribing protocols, adding commentary, model-fitting, proposing mental constructs, diagramming probable interactions and constructing plausible overall hypotheses. Special attention was paid to the interaction among subjects' assumptions, action-plans and experimentation -- based on their ability to utilize available instructional resources during problem solving. A protocol diagramming technique was developed for expressly depicting subjects' cognitive activities, including decomposition of the problem from a general plan to domain-specific plans and the interaction of assumptions and experiments with these plans.

Additional student protocols -- some "near-novice" and some "near-expert" -- were obtained and superficially examined for related phenomena. Also, for comparison, three protocols of adult expert programmers solving the same problem were analyzed and contrasted with students' work. Many student-programmers exhibited misconceptions about variables that interfered with their solving simple problems. Many of these misconceptions were related to proper use of punctuation associated with variables. Such errors were characterized as generally reflecting the absence of strong, guiding principles of variable use rather than misapprehensions about the nature of variables or their notation. Two near-expert subjects demonstrated impressive instances of the transfer of variable knowledge from other programming languages.

Overall, students' problem-solving behavior appeared inflexible and distractible by superficial features of a problem. Expert behavior was determined to be qualitatively different. Experts consistently produced multiple alternative solution-plans and evaluated these plans based on their consideration of program aesthetics, solution-optimization and efficiency. Such facility seems to indicate presence of expert meta-programming knowledge, which could not be adequately explained by either programming plans or descriptive knowledge alone. This was hypothesized to be an integration of both types of knowledge. The implications of these results for teaching,

learning theory and cognitive science were discussed.
Complete transcripts of protocols were included in
appendices.

TABLE OF CONTENTS

	<u>Page</u>
ABSTRACT	iv
LIST OF TABLES	ix
LIST OF FIGURES	x
 Chapter	
1. Introduction	1
Historical Motivation	1
Focus	2
Conceptual Hypothesis	8
Technical Overview	9
Some New Variable Distinctions	14
Rationale/Importance of Study	16
2. Review of Literature	23
Cognitive Effects of Programming	23
Studies of Programmers' Conceptualization of Programming	33
Cognitive Models of Programming	37
A Model of Device Learning	40
Program "Bugs" and Programming Expertise	45
Programming Plans and Procedural Knowledge	52
Metacognition and Programming Knowledge	59
Studies of the Learning of Specific Language Features	63
Program Control and Expression Parsing	63
Variables	70
3. Description of the Study	85
Methodology	85
Limitations	88
Subjects	91
Method	95
Analysis	100
Schematic Diagrams	105
Further Analysis	109
4. Results and Analysis	113
Microanalysis	113
Expert Protocols on Problem A-2	113
Nonexpert Protocols on Problem A-2	131

Summary of Other Solutions	157
Summary of Remaining Solutions to Problem A-2	157
Summary of Solutions to Remaining Problems	176
5. Summary, Implications and Recommendations	202
Summary of Results	202
Implications	213
Implications for Education	213
Implications for Language Designers	220
Implications for Designers of Intelligent Tutoring Systems	221
Recommendations for Further Research	223
Conclusion	226
APPENDICES	231
A. Instructional Script	231
B. Problem Set	240
C. H, Problem A-2	250
D. P, Problem A-2	253
E. B, Problem A-2	260
F. R, Problem A-2	269
G. M, Problem A-2	274
H. A, Problem A-2	283
BIBLIOGRAPHY	295

LIST OF TABLES

	<u>Page</u>
Table 1, Summary of Subjects' Backgrounds	93
Table 2, Classification of Problems	98
Table 3, Schematic; H, Problem A-2	115
Table 4, Schematic; P, Problem A-2	120
Table 5, Schematic; B, Problem A-2	125
Table 6, Schematic; R, Problem A-2	132
Table 7, Schematic; M, Problem A-2	139
Table 8, Schematic; A, Problem A-2	146
Table 9, List of Concepts and Misconceptions from Six Selected Protocols	158
Table 10, Classification of Misconceptions from All Problems	200

LIST OF FIGURES

	<u>Page</u>
Figure 1, Symbols Used in Diagramming	106
Figure 2, H, Problem A-2	116
Figure 3, P, Problem A-2	122
Figure 4, B, Problem A-2	128
Figure 5, R, Problem A-2	134
Figure 6, M, Problem A-2	142
Figure 7, A, Problem A-2	152

C H A P T E R 1

INTRODUCTION

Historical Motivation

This study was inspired by the author's frustration as a teacher of an introductory Logo course.* In each of the four semesters the course was offered, a large number of students were observed to have difficulty understanding and using variables correctly and exhibited persistent errors in variable use. A smaller number of students seemed relatively immune to these errors and were, in general, more successful in the course and more likely to take other courses in the Computer Science curriculum.

* The author was one of three instructors of a one-semester course, offered in each of four consecutive semesters from Fall, 1984 through Spring, 1985 at Lincoln-Sudbury Regional High School in Sudbury, Massachusetts. It was intended as an entry-level course, challenging enough for students with previous programming experience but appropriate for students with little or no programming background. The course content ranged from an introduction to basic Logo commands and concepts to fairly complex Logo programming techniques, including units on recursive operations and procedures, fractal graphics, and mathematical modeling. These topics were taught with an extensive set of worksheets, developed by several individuals in the Computer Science Department over a number of years. This allowed an emphasis on hands-on learning and afforded instructors the opportunity for much individual observation as students progressed through the worksheets. The author had used such an individualized approach with good success in other programming classes, including classes in a computer camp, for elementary and secondary school students, for undergraduates studying instructional computing and with elementary and secondary school students during individual tutoring sessions.

Each semester the class became divided into these two groups. While other instructors also observed similar errors, none were fully able to help students overcome their difficulties or explain the misconceptions that underlie them. This was the dilemma that motivated this study.

Over the course of nine years teaching computer programming (primarily APL, BASIC and Logo) the author had formulated several working theories about how one develops an understanding of variables in the context of programming (see Technical Overview). These theories were for the most part unverified but had been useful in the past and seemed to have predictive and explanatory power. They were based in part on a task-analysis of the disparate uses of variables in programming, in part on observations of students and in part on introspection. The author's hope was that through a more careful and in-depth analysis of natural student dialogs he could either find ways to apply these existing theories or to develop new theories that could explain the type of behavior described above as well as to gain insight into the general development of programming skill.

Focus

There is little doubt that computers are among the most important technological innovations of this century

and will continue to be a topic worthy of study. In fact, this decade has come to be known as the "Information Age", and the computer is seen by many as the only means by which mankind may cope with the "information explosion." The proposal is often advanced that the study of computer technology become a formal requirement for elementary and high school students. Most often, this is expressed as a need for a new "literacy" (in the sense that reading and writing are considered minimal requisite skills), from which is derived the term, "computer literacy".

Even among proponents of computer literacy, however, there is little agreement as to precisely what skills or activities should be mandated. It is generally acknowledged that computer programmers are the most sophisticated class of computer users and that programming is the most general and empowering of computer skills. This leads some educators to conclude that programming should play a central role in computer literacy training; namely that all students should learn some amount of programming (Luehrmann, 1980). Others see programming skill as too complex for the average student. Such educators support instead the use of application packages by students and argue that it is no more necessary for a computer user to learn to program than it is for the driver of an automobile to learn how to repair it.

There may be other reasons to discourage mandatory computer programming in the public school curriculum, e.g., a challenge to the assumption that there exists any one set of general computer skills (see Harvey, 1985). Yet the "complexity" argument is somewhat disturbing. The development of two programming curricula, a serious course-of-study in programming for some students and a special, watered down approach for the "general" student could lead to institutionalized "tracking". Such a scenario would contribute to the schism between the "hard knowledge" of the technical and the "soft knowledge" of the less technical computer user that already troubles some educators (Turkle, 1984).

Whether or not programming is incorporated into the standard curriculum, teachers must find techniques to teach complex skills without trivializing them. Furthermore, programming may have distinct educational benefits and applications, beyond the notion of some sort of requirement for informed citizenship. For example, there is some evidence that programming helps algebra students avoid certain misconceptions (Clement, 1980). If such claims are borne out, programming could be used to form new bridges, not barriers, to knowledge. This makes it imperative that the means be found to teach programming to all who wish to learn it.

Logo is a programming language developed specifically as an environment for learning. It is an extensible and fully modular language in which programmers utilize recursion and functional composition as the primary control structures. The best known feature of Logo is the turtle, originally a mobile robot that could be used as a plotter but usually now, an internal object on a graphics screen that simulates the behavior of the turtle-robot. The turtle can be driven from a local point of view with simple commands such as FORWARD (abbreviated FD) and RIGHT (RT). Seymour Papert, designer of Logo, claims that these features make Logo powerful enough to do college-level Computer Science but simple enough for very young children -- that Logo has a "low threshold" and a "high ceiling". Logo is generally regarded as one of the two most popular computer languages for instructional use. It is purported to provide an excellent environment that may accelerate cognitive development and stimulate some social aspects of intellectual activity, something which Papert refers to as "computer culture" (Papert, 1980). These features made Logo both a practical and an interesting choice as the language in which this study was conducted.

In terms of content, the focus of this study is on utilization of variables in the context of Logo programming. There were several reasons for choosing variables as a focus. For one thing, variables are a critical concept to be mastered in learning Logo

programming. However, the notion of a variable is a very general concept, and so variables form a common link, not only between Logo and other computer languages, but with other knowledge domains as well. For example, there is a sizable body of literature on correct and incorrect conceptualization of variables in mathematics, where they are of pivotal importance (see Chapter 2). In addition to mathematics, variable is an important concept in all of the sciences, in linguistics and in the social sciences. At the same time, the concept of variable is more discernible as a conceptual entity, discrete from the syntax and semantics of a particular programming language, than comparable sub-concepts such as flow-of-control or data-structure. In this sense, it was a convenient and manageable area of concentration through which to gain insight into the larger questions, the acquisition of general programming and cognitive skills.

The students chosen as subjects for this study had all completed the equivalent of at least one semester's study of Logo. A number of earlier studies have looked at novice preconceptions (Anderson, 1984; Soloway, Ehrlich & Bonar, 1982; Bonar & Soloway, 1985; see also Chapter 2). Others have studied how expert programmers structure their thinking (Soloway, Bonar & Ehrlich, 1981; Adelson, 1981; see also Chapter 2). But very little is known about the transition between the two. The study of this transitional period can tell us a great deal about the learning

process. In that sense, this is an inquiry into the nature of learning.

The approach chosen was qualitative rather than quantitative. Nine high-school students and three Logo experts were videotaped during clinical interviews as they attempted to solve problems drawn from a set of nine. These tapes were later transcribed and analyzed. Six problem solutions were subjected to a cognitive microanalysis. The intent was to study a complex set of skills in a naturalistic setting.

Finally, this study attempted to collect anecdotal data for those interested in Logo-learning, especially for teachers of Computer Science at the high school level. Effective teaching requires ongoing refinement of one's instructional approach. Toward this end, teachers, like other practitioners, hold working theories about their students' learning (usually in the context of their own teaching) and conduct "experiments-in-action" as a means to form and reform working theories (Shon, 1984). Teachers seldom have the luxury to test their theories in greater detail or to compare their experiences and observations with others with similar interests. The author hopes that the raw data in extensive protocols, as much as the author's comments and analysis of subjects' misconceptions, will help educators as they work to refine their ideas and

instructional materials, and to develop more efficient and effective instruction.

Conceptual Hypothesis

This research addresses the following questions: 1. How do expert and nonexpert Logo programmers compare in their use of variables in the solution of programming problems? 2. What variable misconceptions do nonexperts hold, as compared to experts? 3. How do programming misconceptions affect the dynamic activity of both expert and nonexpert on a micro-level, including their utilization of available resources, planning, production of "bugs", debugging and experimental activity?

Certain variable misconceptions in algebra appear to be quite resilient to instruction (Rosnick & Clement, 1980). We want to ask whether misconceptions of programming variables are similarly resilient to instruction. If so, what factors might allow or prevent the efficient utilization of available resources, such as reference documents, objective facts or past experience, and how do misconceptions eventually give way to the more generalizable concepts exhibited by experts? Does mastery occur gradually and continuously, in stages, or as a one-step process?

Some recent research suggests that some programming misconceptions result when general problem solving

techniques derived from natural problem solving experience are misapplied (Bonar & Soloway, 1985). In a similar way, we ask if developing programmers harbor misconceptions that are adaptive in that they explain and make accessible some aspects of variable use that the student-programmer is not presently able to generalize. Or, can a developing programmer hold an accurate sub-concept of variable in one context while maintaining a misconception of variables in another?

Another critical question for those who endorse programming as a form of discovery learning is this: How do developing programmers actually gain insight into areas of misconception? If we are able to capture observable examples of intuitive learner insight, (sometimes referred to as the Aha! phenomenon), we can begin to understand what it is and how to promote it.

Technical Overview

A task-analysis shows that a Logo programmer must understand a number of things in order to use variables effectively. First, one must recognize the relative permanence of a variable; that a variable, once created, remains intact until it is explicitly removed, changed, or until its host environment terminates, (e.g., when a procedure to which a variable is local stops running). This is true of both global and local variables, although

the issue is potentially much more complicated and critical to a proper understanding of local variables, especially in variables local to recursive procedures or to procedures that call sub-procedures. One must see a variable as an attribute of its environment, i.e. a local variable is always associated with a procedure, and a global variable is always part of a workspace.

Second, one must recognize that by definition a variable associates a name with a value, where the value may be any member of a set of permissible values. The Logo programmer must clearly distinguish between a variable's name and its stored value. This distinction is often overlooked, but in certain cases it becomes critical. This is true in the case of a programming technique called "multiple indirection." For instance, if a Logo programmer is asked to interpret the expression:

THING :USER

where "USER is defined as the name of a variable holding the word, "RICK, as its value, and "RICK is itself defined as a variable name with the number 642 as its value, then the programmer must decode a chain of values from variable names in order to evaluate the expression as the number, 642. :USER must first be evaluated as "RICK before the number is extracted as the contents of the second variable. An experienced Logo programmer might read the above expression as, "the value of the value of the variable, USER." An equivalent expression to the above is:

THING THING "USER

It becomes clear in this example that the expressions, THING "USER and :USER are, in effect, identical; both output the value of the variable, USER. Indeed the colon is sometimes taught as an abbreviation for 'THING "' (Harvey, 1985).^{*} This can be a point of confusion, however, since the "dots" (:) that often precede a variable name are sometimes mistakenly thought of as a mandatory prefix to the variable name.

Thirdly, in the case of variables whose values are dynamically assigned with READWORD, READLIST or REQUEST, one must distinguish between the role of the programmer, who chooses the name of the variable, and the user who determines its value. (I like to think of this as a "temporal" distinction between "function definition" time and "run time").

Finally, studies of expert programmers suggest that they recognize certain programming idioms which may utilize variables (Soloway et al, 1982, and see Chapter 2). The

^{*} This "replacement" interpretation of the colon can lead to errors as well. It would not be surprising to see a student attempt to use two colons to create a third expression:

::USER

This, of course, is not a legitimate shorthand form of either of the first two expressions.

student programmer must learn to utilize variables in high-level plans when such use is suggested by the programming problem, and in doing so, simplify the programming task and concentrate their efforts on unique aspects of the problem. Common Logo variable idioms include variable-as-a-counter, flag, representation of a machine state, argument to a procedure, and user input.

Accompanying standard variable concepts is the concept of a "function", Logo's only means of representing covariation. Technically, a function is a procedure that takes one or more inputs and has an explicit result. Functional inputs are simply variables local to the function, but in order to make a procedure produce an explicit result, Logo provides a special command called OUTPUT. The expert Logo programmer recognizes that a function can have only one output (although a function may have any number of alternative paths to produce that one output). Functions and variables may be seen as conceptually very similar. (In fact, a global variable may be closely modeled by a procedure with an explicit result). Functions, like global variables, reside within a workspace.

A function relates to other procedures in a Logo command sequence in a fundamentally different way from procedures that do not produce results. A function can be seen as replacing itself with a value while a

non-outputting procedure is normally viewed as performing some task. In that sense, a function is more like a variable than a non-outputting procedure.

A variable is distinct from a function, however, in that a variable may contain only one value at any time (although that value may be a composite in the form of a list) while a function represents a covariate relationship, i.e., it defines the mapping of one or more inputs to the functional result. Also, functions may be defined recursively while a variable may not. Both of these features require one to view a function dynamically (see also, Kuchemann, 1978, in Chapter 2).

Parameter passing in functions is also somewhat more abstract than explicit variable assignment, in that the programmer provides the variable name while defining a function and the value is given at a later time, when the function is invoked. This makes it convenient to talk about the "inside" and the "outside" of a function when referring respectively to a parameter variable's name and its value. In terms of parameters this is analogous to the name-value distinction in a variable.

Functions, like variables, can become associated with certain common programming idioms, and here too the expert must learn and be able to apply idioms appropriately. Common functional idioms include simple functions, recursive functions and "predicates" (functions that return

the value "TRUE or "FALSE). Of course, local variables play a critical role within such functional idioms.

Some New Variable Distinctions

The above task-analysis is represented in terms of three conceptual sub-categories, drawn from developmental psychology. First, the concept of a variable retaining its value within certain well-defined limits is referred to as variable permanence, and is viewed as the most basic variable sub-concept. Second, the clear distinction between variable name and value -- when assignment of the name occurs during procedural definition and the value during "run-time", is referred to as a temporal distinction. Third, is the clear sense of "where one is" in the Logo environment, including the distinction between procedure definition mode (sometimes referred to as "being in the editor") and "toplevel" (sometimes called "being in Logo), or between "being in..." one workspace vs. another. The common use of prepositions in such examples leads the author to refer to them as a positional distinction.

An ordering of mathematical variables resulting from a developmental study (Kuchemann, 1978) and the above technical overview suggested the following classification of variable use for this study:

1. MAKE-ing a global variable is the simplest variable usage, i.e. it is "time-constant" and

"dimensionally constant".

2. Both the use of local variables and asking the user for a value with RQ (in the context of a MAKE "FOO RQ construct) are less straight forward. They seem to be akin to Kuchemann's category "Letter as Specific Unknown", (see also Chapter 2), because they involve a gap in time between the selection of a variable name and its value; i.e. they are "temporally variable".

3. Understanding variables in the context of a recursive operation or in a problem involving "multiple indirection" (i.e., one in which one or more variable names are stored as data within another, to be decoded by the programmer) were the most difficult, because they involved both a "temporal offset" (especially in the case of recursive operations) and a "dimensional offset" (i.e., added complexity in terms of the "level" of recursive calls).

4. A function is a procedure with both inputs and an output, and may be viewed as representing a covariate relationship. The concept of covariation was seen by Murray & Clement (1986) as more difficult than simple variation, and one may assume that this makes a function more difficult than simple variation in the context of programming as well, though it is not clear how

this distinction might interact with the abstract concepts of temporal and positional factors.

While the design of this study, involving microanalysis of a limited number of subjects, precluded an exhaustive examination of the question of stage-ordering, several aspects of the above classification are informally explored in Chapters 3, 4 and 5.

Rationale/Importance of Study

Computer programming is an activity that has aroused much interest in the educational community. Many respected educators have voiced their beliefs that computers will revolutionize education (Luehrmann, 1980). Some see computers as a more effective means to familiar and more or less traditional educational ends (Elliott, 1978). Others have viewed the new technology as a radicalizing force (Dwyer, 1980). For example, some educators have proposed teaching students how to think via computer programming (Papert, 1972).

Recently this latter movement has come under attack (Pea, 1984). Specifically being challenged is the notion that programming teaches thinking. At the core of this criticism is the implication that a large scale educational commitment is premature or even dangerous due to the lack of solid evidence both of the nature of programming knowledge and of its relationship to other thinking. While

this objection seems to lay an excessive burden of proof upon the proponents of computer programming (few if any traditional subjects could be justified in this manner), it would be very helpful to know more about the cognitive nature of programming so that educators could make intelligent decisions about the appropriate educational role for this relatively new and, for many students and teachers, alien activity.

Computer programming is a new area of study in cognitive psychology. Only in the past ten years or so, with the development of microcomputer technology and the resulting growth of the personal computing industry, has programming become generally accessible to the average student. As a new area of learning, it provides us with a rare opportunity to address some fundamental questions about how people learn. A number of subjects of this study, for example, exhibited misconceptions that were surprising and unique (see Chapter 4). Some programming texts rely on the use of metaphors to teach programming (e.g., Harvey, 1985), and some researchers suggest that difficulty in finding ready metaphors underlies many of the learning problems of novice programmers (Mayer, 1979). Instead, a rash of new terms to describe the workings of computer software and hardware have had to be coined, and computer glossaries to explain these new terms in common language have proliferated. This scarcity of ready

metaphors suggests that computer programming is, in many ways, quite unlike any other common activity.

The specifics of development of programming knowledge also have important implications in the field of artificial intelligence for the development of expert programming instruction systems (Sleeman & Brown, 1982). Such systems require a thorough knowledge-base of both concepts utilized by expert programmers and common misconceptions encountered by computer programming students. This same knowledge could aid the thoughtful teacher of programming in formulating instructional material and in understanding difficulties experienced by students.

While there is an growing body of practical research on the teaching and learning of mathematical variables, the study of variables in the context of programming has lagged behind, and is largely theoretical. Many basic questions, such as the relationship between the understanding of variables in programming and in mathematics, are yet to be addressed. This exploratory study is a first attempt to address both deficiencies. As an early attempt at a general understanding of a complex and important sub-skill of programming, it seeks to identify the important issues for more specific follow-up study. For example, one issue which arose and was addressed during microanalysis was the important way in which concepts derived from subjects' experience with variables in other computer languages

(BASIC and Pascal, in particular) both enhanced and interfered with the learning of Logo. Several specific questions for further research are identified in Chapter 5.

Finally, the clinical approach of this study was chosen, in part, to help close the gap between the cognitive researcher and the teacher, an extension of the more general and long recognized problem of integrating theory with practice. There is a scarcity of practical information to help the teacher who, confronted by the variable misconceptions of his students, must try to analyze and correct these problems "on the fly". In the past, cognitive science has often addressed the problems of teachers, including many practical studies of memory, cognitive development and learning theory. More recently, fueled by an interest in expert systems, many cognitive studies have focused upon isolating the specific world-knowledge associated with expert behavior (Sleeman & Brown, 1982). Recent work on intelligent tutoring systems includes the development of an expert system to teach computer programming (M. Miller, 1982; Anderson, 1984). These new systems place a strong emphasis on analyzing student errors, and such studies hold the eventual promise to the teacher of well refined prescriptions for particular programming errors. At present, however, intelligent tutoring systems tend to be too deterministic and domain-specific to be of much practical value to the

teacher (Sleeman & Brown, 1985). By choosing a descriptive approach, but one informed by these recent detailed studies and deterministic models, this study describes learning in a more natural setting, making these observations more accessible to the teacher.

From a learning theory point of view, this approach allows a more integrated and general view of programming variables in the context of other human experience. Chapter 4 includes a discussion, informed by protocols, of how and when specific learning of isolated cognitive skills become integrated with a general, natural knowledge, especially in terms of specific and general concepts of variables. Such general studies as this one can serve as an invaluable means to critique, verify and expand upon the theory that underlies necessarily deterministic intelligent tutoring systems (Lin, 1979).

Footnotes

Chapter 1

Luehrmann, A. "Should the Computer Teach the Student or Vice-versa?", in Computers in the Schools, Tutor, Tool, Tutee, (R. Taylor, Editor), Teachers College Press, 1980.

Harvey, B. Computer Science Logo Style: Intermediate Programming, MIT Press, 1985.

Anderson, J., Farrell, R., Sauers, R. "Learning to Program in Lisp", Cognitive Science, 8, 87-129, 1984.

Turkle, S. The Second Self, Simon & Schuster, 1984.

Clement, J. "Cognitive Microanalysis: An Approach to Analyzing Intuitive Mathematical Reasoning Processes", Technical Report, Cognitive Process Research Group, University of Massachusetts, 1980.

Papert, S. Mindstorms, Basic Books, 1980.

Anderson, J. "Learning to Program in Lisp", Cognitive Science, 8, 87-129, 1984.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. "What do novices know about programming?", Technical Report, Yale University, Department of Computer Science, 1982.

Bonar, J. & Soloway, E. "Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers", Human-Computer Interaction, Fall, 1985.

Soloway, E., Bonar, J. & Ehrlich, K. "Cognitive Strategies and Looping Constructs: An Empirical Study", Technical Report, Yale University, Department of Computer Science, 1981.

Adelson, B. "Problem solving and the development of abstract categories in programming languages", Memory and Cognition, 9, 422-433, 1981.

Schon, D. The Reflective Practitioner: How Professionals Think in Action, Basic Books, 1983.

Elliott, P. "Computer 'glass-boxes' as advance organizers in mathematics instruction", International Journal of Mathematics in Science and Technology 9, (1), 79-87, 1978.

Dwyer, T. "Significance of Solo-mode Computing for Curriculum Design", in Computers in the Schools, Tutor, Tool, Tutee, (R. Taylor, Editor), Teachers College Press, 1980.

Papert, S. "Teaching Children to be Mathematicians vs. Teaching Children About Mathematics", International Journal of Mathematics Education in Science & Technology, 3, 1972.

Pea, R. "Logo Programming and Problem Solving", Technical Report, Bank Street College, 1983.

Mayer, R.E. "The Psychology of Learning BASIC", Communications of the Association of Computing Machinery, 22, 589-594, 1979.

Sleeman, D. & Brown J.S. Intelligent Tutoring Systems, Academic Press, 1982.

Miller, M.L. "A Structured Programming and Debugging Environment for Elementary Programming", Intelligent Tutoring Systems, Academic Press, 1982.

Lin, H. "Approaches to Clinical Research in Cognitive Process Instruction", Cognitive Process Instruction: Research on Teaching Thinking Skills, (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.

C H A P T E R 2

REVIEW OF LITERATURE

The review of literature begins with an examination of literature on the general cognitive effects and potential educational benefits of teaching computer programming. Following that is a discussion of several interesting models of programming knowledge and learning. The chapter ends by looking at some studies of specific features of computer languages -- first at aspects of the flow of control (which is of peripheral interest to this study) and then at variables, both in mathematics and in programming.

Cognitive Effects of Programming

Several authors suggest that computer programming can enhance a student's skill in other knowledge domains while others remain skeptical about the possibility of such a transfer. Conflicting studies can be cited to support either point of view.

Among the first group of authors, some emphasize those specific skills or concepts utilized both in programming and in the recipient domain, suggesting that one may expect to see a transfer through the reinforcement of these sub-concepts. Some believe that, through the expressive power of particular computer languages, one may expect students to gain insight into concepts being modeled in the language in question. Others see affective aspects of

programming as providing the greatest instructional potential.

Elliott (1978) has proposed that short, concise computer programs written in APL ("glass boxes") be used as advanced organizers in mathematics instruction "...to bring clarity and structural integration to mathematical, instructional and cognitive structures important to the mastery of mathematical concepts". In her example of a function that outputs the greatest common divisor of its inputs, she believes that the glass box shows the relationship of "subordinate concepts" from set theory and number theory to the "...subsuming concept of GCD" (Elliott, 1978). Peelle (1980) proposes that mathematics teachers explore alternative algorithms for common skills such as addition, as a means to appreciate the possible alternatives to conventional algorithms, to improve their own comprehension of mathematics and as a pedagogical model for their own teaching. He also proposes that by using recursion in APL programs, students can develop "...a habit of mathematicians and some computer scientists" (Peelle, 1977). Lewis (1980) reports favorably on the teaching of recursive Logo programs in a high school pre-calculus course and mentions, in addition to familiarity with recursion, two other benefits of including a programming component in mathematics instruction at this level: (1) a greater emphasis on the "axiomatic structure of mathematics" and (2) the notational power of Logo's list

structure. In all of the above cases, the authors noted the expressive power of the particular language chosen as an important factor of its pedagogical impact.

Some researchers see programming as a means of teaching logical thinking. Mathematician George Polya (1958) has suggested that "heuristics", high-level problem-solving techniques gleaned from interviews with expert mathematicians, be explicitly taught in schools. Papert (1980) argues that programming computer graphics in the Logo language with a graphic object known as a "turtle" is the ideal forum in which to do this. "I believe that Turtle geometry lends itself so well to Polya's principles that the best way to explain Polya to students is to let them learn Turtle geometry". This is less a formal suggestion by Papert for a computer-based course on Polya's problem-solving techniques than a casual afterthought, yet it reflects Papert's belief that by introducing the right sort of programming experiences, teachers can shift their emphasis from rote learning of mathematical algorithms to the process of mathematical problem-solving (Papert, 1980). Unfortunately, this becomes translated into a popular belief that any computer programming experience will automatically make students more logical. Indeed, the assumption of transfer of programming knowledge to other domains, specifically the belief that "programming teaches thinking", has recently begun to come under attack.

Pea and Kurland (1984) have argued that there is no objective evidence that adopting what they call Papert's "radical child-centered" approach will increase problem-solving ability. In three studies, they have sought to disprove what they see as a popular misconception. In all three, they worked with students who had just completed a year of learning Logo in what they describe as an environment based on Papert's ideas. Students averaged 45 minutes per week of programming time. In the first study, they tested students for LOGO command understanding, program writing ability and error correction ability. They found some interesting things: children found semantic errors more difficult to locate and correct than syntactic errors; older children understood far more commands even though they received the same amount of instruction as younger ones. However their overall conclusion was that subject performance in all three areas was poor (Pea & Kurland, 1984). In a second study, Pea (1983) found no increase in planning skills in task-scheduling among young programmers as compared to a control group. Finally, Kurland (1984) found that these children understood tail-recursion, but not embedded-recursion. He further found that his subjects believed that recursion was a looping rather than a procedure-calling construct. In a study of the same children, Mawbry et al (1983) found children's ideas of what computers are and how they perform to be naive.

I see several problems with Pea and Kurland's criticisms. For one thing, it seems to me that the subjects in these studies simply had not learned enough programming to evidence much transfer. The group reportedly had difficulty understanding conditional statements and inputs to procedures (Pea, 1983). Rather, this result may emphasize one of the conclusions of the current study: that it may take a significant amount of time before mastery of some fundamental but critical Logo concepts are reflected in low-level and high-level planning. Furthermore, it seems that the program in which Pea & Kurland's subjects were trained was not time-intensive enough. forty-five minutes a week, whether distributed as nine minutes a day for five days or as one session per week seems, by my experience, to be too little time to allow for reasonable progress in acquiring new programming skills, insufficient time to practice skills already internalized, and (even if we assume a single weekly time-block) too brief a period for the intensive activity of true problem solving. More importantly, I think that Pea & Kurland miss the point of Papert's writing when they focus on specific claims of cognitive transfer. It seems to me that the main claim of Papert when he advocates the use of computers to teach "powerful ideas" is that programming and the development in the classroom of a "computer culture" can help the student to understand the

nature of inquiry and the attitudes that foster it (Papert, 1980).

There is a good deal of similarity in the observed general attitudes and work habits of those who have demonstrated problem-solving skill. Good problem-solvers show a greater ability to frame a problem in terms of rival theories and to design experiments to test them against one another (Schon, 1983; Driver, 1983; Smith & Inhelder, 1975). They are more likely to pose their own questions without prompting (Kamii, 1973; Duckworth, 1973). They will actively and confidently attack a problem (Polya, 1957; Schoenfeld, 1979; Kamii, 1973); and they will routinely use heuristics to approach a difficult problem (Polya, 1957; Schoenfeld, 1979). The development of such attitudes toward problems seems to be of critical importance to the development of problem solving expertise.

Papert and others make this point in a number of anecdotal reports. Papert (1980) describes the experience of a fifth grader who had a "bug" in the way he added numbers. (He would add 35 to 35 and get a sum of 610, for instance). Papert analyzes Ken's problem to be, in large part, a failure to recognize the adding procedure (or any procedure) as an entity, which may be altered. He says that he has seen many children get over such misconceptions after writing and manipulating their own procedures in LOGO

(Papert, 1980). He relates the experiences of another fifth grader who could remember most things but had a block against remembering and doing mathematics, and whose problems were dramatically reduced once he began programming (Papert, 1980). Turkle talks about a black fifth grade student, alienated from her white instructors and classmates, whose experiences writing programs to generate poems (as well as her experience with word processing) helped her to overcome cultural isolation and become a productive young poet (Turkle, 1984). Clearly, these programming experiences cannot be seen as singular attempts to teach logic or specific problem-solving skills, but must be seen as fostering the ongoing development of attitudes and self-concept as well as cognitive structures.

Other stories told by researchers suggest that programming may help to break down the institutionalized isolation of most mathematics classrooms from true mathematics reported by Confrey (1984). Papert (1972) speaks of "syntonic thinking" (feeling an emotional connection with the thing being studied), and "learning to be a mathematician vs. learning to do mathematics". He describes Deborah, who was immobilized by her frustration with programming until she developed a means of constraining the LOGO turtle to 30 degree turns. Turkle (1984) talks about Ronnie, who forms a link with mathematics by writing a computer program that makes

figures "dance in exactly the way he had envisioned", and about Anne , who invents original programming techniques so that she can think of the colors in her computer graphics as colors on a painter's palette rather than as "variables". Turkle sees learning breakthroughs such as these as idiosyncratic. She sees a potential for fostering, with computer programming, an active intellectual community, a "computer culture" that shares ideas, problems and their solutions, and she favors such an approach over a focus on teaching specific concepts or facts (Turkle, 1984).

One justification for Turkle's approach can be seen in Steven Louie's work (1985.) He has reported a modest shift in locus of control (LOC) toward more internal LOC from pretest to post test in his study of children who learned both word processing and Logo programming in a computer camp. An internal shift in LOC, thought to be a measure of greater self-confidence and self-reliance, has been associated with improved learner attributes in the affective domain and with success in schools. Louie found no correlation between internal LOC and subjects' regard for planning as a useful problem-solving skill. He notes that this agrees with Pea's findings, and suggests that a sense of "empowerment" is engendered by certain computer activities, as suggested by Papert, while planning skills are not. Louie sees no contradiction between his findings

and Papert's claims of the potential of Logo programming to "teach thinking" (Louie, 1985; Papert, 1980).

On the other hand, Clements and Gullo found that Logo instruction can lead to improvement in young children in some measures of cognition that are associated with planning. They found that after 16 weeks of instruction in Logo, first graders showed a significant improvement in measures of metacognition, creativity and reflectivity (the latter being the tendency to pause before solving a problem to consider strategy). A second group that used a sequenced set of CAI programs showed no improvement. While this seems to be in contradiction with Pea & Kurland's findings, Clements & Gullo's instruction in Logo was significantly different from Pea & Kurland's. Clements & Gullo provided a sequenced set of Logo lessons in which instructors suggested to their students certain planning strategies, while Pea & Kurland taught only the basics of the language. Clements and Gullo's findings can be interpreted as showing that cognitive skills can be taught in a learning environment centered upon Logo instruction, while Pea & Kurland showed that the inclusion of the Logo language (as opposed to the constructivist philosophy associated with Logo) does not assure the improvement of cognitive skills, or, at least, not of planning skills. Another interesting finding of Clements & Gullo is that they found no significant difference between their two groups in general measures of cognitive development,

including Piagetian task measures and the McCarthy Screening Test (Clements & Gullo, 1984).

Whatever the case, in terms of transfer of overall cognitive skills, the prospects for transfer of programming knowledge to one particular area of education, mathematics, may be fairly good. Howe, O'Shea & Plane (1979) compared learning in a mathematics class that included a computer programming component with a class that did not. They found significantly better performance in the class that programmed (cited in Clement Lochhead & Soloway, 1980).

Clement et al (1980) have shown a relationship between the use of variables in BASIC and the correct solution of algebra word problems. They studied both college students of low to moderate mathematical ability and experienced engineers. Both groups were asked to translate a mathematical word problem into an algebraic equation (in two unknowns) and also to translate a similar problem into a computer program in the BASIC language. Clement, et al, found that significantly more subjects performed the programming task correctly than the algebraic one. They attribute this, in part, to the very explicit syntax and semantics of a computer language. They also note that, in a programming language, an equation takes the form of a function, an active object that accepts inputs and returns a result, and that this active form emphasizes the dynamic nature of a bivariate relation. These results suggest that

programming languages provide a powerful alternate notational system for mathematics instruction (Clement et al, 1980).

The author takes the position that programming can have educational benefits in the affective domain, as Papert, Turkle Louie all suggest. The power of a computer language as a highly formal, dynamic representational system also highlights its potential to enhance instruction in traditional subjects, including mathematics, as reported by Clement, Lochhead & Soloway and by Howe, O'Shea & Plane. Furthermore, as Clements & Gullo have shown, programming in a language such as Logo is a natural vehicle for teaching some general cognitive skills, including metacognition, creativity and introspection, although Pea & Kurland's studies emphasize that this learning is a function of the overall instructional environment, not of the decision to teach programming or of language selection alone.

Studies of Programmers' Conceptualization of Programming

Two types of studies strongly suggest a picture of expert programming knowledge as complex in nature, characterized by the bundling of information based on functional attributes. In both novice and expert studies, learning is seen as quite idiosyncratic, suggesting that programming skill represents a set of internal concepts,

associations and/or sub-skills, constructed by the learner over an extended period of time.

Some researchers have studied the cognition of expert and novice programmers through performance on memory-recall tasks. This work is generally based on Chase and Simon's study of chess expert-knowledge, in which they discovered not only that expert chess players can memorize much more of a complex board position, but that they remember board positions in a different way from novices -- essentially, by structuring their memory of positions functionally, as offensive or defensive configurations (cited in Adelson, 1981). Beth Adelson, using techniques pioneered by Tulving, took expert programmers (her graduate teaching assistants) and students from an introductory programming class in PPL, (Prototypical Programming Language) and studied performance on a recall task. She took three complete PPL programs, scrambled the lines of each and presented them, one at a time to subjects in a "free recall" format. This was done through nine trials, and the recall-performance of the subjects after each administration was graded, summed and analyzed, using a technique expected to uncover subjects' subjective organization. She found that both novices and experts showed subjective organization based on conceptual categories, but that experts organized by semantic categories, while novices grouped by command type (syntactically) in their recall performance. The semantic

organization of the experts apparently explained their ability to eventually reconstruct the original programs. It would have been helpful if Adelson had provided raw data or analysis of each trial separately rather than summed performance alone, to see whether or not the expert's performance was strongly weighted by later trials, when the experts had completed their reconstruction of the scrambled programs or were due to a more unconscious organization by semantic chunks. In any case, this study strongly suggests that an expert's association of programming code is significantly different from that of a novice, with experts grouping code into functional chunks (Adelson, 1981).

Another study viewed expert programming as associated with the functional "chunking" of programming code, and provided some insight into the mechanism by which this grouping may take place. Soloway, Ehrlich, Bonar and Greenspan (1982) compared the performance of two groups (one that had completed a first semester course in Pascal and another that had just completed a second semester course) on some simple programming problems in Pascal. The researchers found that while the more advanced group (termed "intermediates") might use an inefficient looping construct for a given problem, their choice of an overall looping strategy was usually superior to that of novices and acted as a good predictor of programming success. The authors have constructed a partial frame model of looping plans with which to explain this phenomenon. In it,

specific blocks of working Pascal code ("tactical plans") are seen as descendants of a smaller number of computer language specific "implementation plans", which are in turn seen as the descendants of an even more abstract and presumably language independent "strategic plan", such as that which acted as such a good predictor of success (Soloway, et al, 1982). For example, when asked to write a Pascal program to allow the input of integers until a "flag value" (9999) is read, then average the non-flag integers, students might choose a "Process-i/Read Next-i" strategy or a "Read-i/Process-i" strategy. Choosing the latter strategic plan proved to be the only good predictor of success with this problem. Expert Pascal programmers generally suggested use of the WHILE looping construct as the most efficient choice, but there was no significant relationship between the choice of that implementation plan and success unless the preferred strategic plan was also chosen. Intermediate programmers did slightly worse than novices on this problem. It is not clear why this is so, although in an earlier study these same researchers established that the "Process/Read" strategy was preferred by novices and seemed to be a more natural construct; intermediate performance may have decreased as a result of their internalization of the less natural but more efficient strategic plan favored by experts. (Soloway et al, 1981).

Cognitive Models of Programming

In addition to the functional chunking of programming knowledge, expert programmers are also said to have the ability to construct "detailed mental models of how the computer is functioning" (Pea & Kurland, 1984). These models are described as dynamic in nature. Supposedly, experts can build these "...runnable mental models, and can simulate computer operations in response to specific problem inputs" (Collins & Gentner, 1981, cited in Pea & Kurland, 1984).

Mayer (1979) hypothesized a cognitive architecture for programming knowledge that is an isomorph of machine and language architecture. He suggested a hierarchical structure composed of eight "levels of knowledge" of the BASIC language, each level providing the constituent "atoms" of knowledge for the level above it. They were as follows:

1. The physical machine.
2. Transactions (sounding very much like assembly language, but really an abstract formation). These commands consist of operations, objects and memory addresses; e.g., "Create (some numeric value) in memory space A1".
3. Pre-statements. When one attempts to construct all BASIC commands using only

transactions, one finds that BASIC statements may represent more than one set of transactions. Each set is defined as a pre-statement.

4. Statements; legal BASIC commands.
5. Mandatory chunks; FOR/NEXT, READ/DATA, IF/THEN and other primitively paired statements.
6. Basic non-mandatory chunks; statement combinations that are often found together as idioms, related to Soloway's "tactical plans".
7. Higher chunks, apparently large functional blocks of code, or modules.
8. Program.

Mayer proposed that students learn at all of these levels, except for #1. He suggested, for example, that students learn to distinguish

```
LET A=1
```

```
LET A=A/2 (Mayer, 1979).
```

In a later study, Mayer (1981) claimed some success in improving BASIC language learning by exposing students to machine-language like constructs in this framework.

However the learning of some programming concepts seem to be retarded by this exposure, a phenomenon that he does not sufficiently explain (Mayer, 1981). Mayer seems to be

looking unrealistically for an exact correspondence between physical, machine architecture and conceptual structure.

Mayer's notion, at least of the importance of machine-level concepts in learning high-level languages, is strongly supported, however, in the work of Wyer & Cannara at Stanford University, who taught students both Logo and SIMPER, a simulation of assembly language. Three groups were established, one that learned Logo first, then SIMPER, a second that learned SIMPER first, and a third that studied both simultaneously. Surprisingly, Wyer & Cannara found that the third group performed the best of the three. They found "...some confusion between the languages, but each illuminated aspects of the other. (This) outweighed the effects of the interference between them" (cited in duBoulay, O'Shea & Monk, 1981).

DuBoulay et al (1981) used the concept of the "notational machine", which they defined as "an idealized conceptual computer whose properties are implied by the constructs in the programming language employed". They promoted a language design that provides feedback on the current state of the notational machine. Such features were said to provide "commentary" on the notational device, which they believe makes the notational device accessible to the user and so the language easier to learn (duBoulay et al, 1981). Thus, they believe that error messages should be consistent with and relate to features of the

notational machine, as should, ultimately, screen displays, keyboard design, even sound and graphics. (One can see such features implemented on some of the window-and-mouse operating systems now commercially available). In terms of general language features for a first language, duBoulay et al recommended logical simplicity (the use of simple and effective logical constructs), syntactic simplicity (a simple syntax with few rules and few exceptions) and functional simplicity. They defined functional simplicity in terms of Mayer's "transactions". To be functionally simple, a language must have relatively few primitive conceptual "transactions", and instructions in the language must be each be composed of relatively few of them (duBoulay & O'shea, 1981). While it is helpful to have a concrete definition of "language simplicity", there is insufficient evidence to accept Mayer's "transactions" as the conceptual basis for it.

A Model of Device-Learning

Both the ideas of experts' functional bundling of knowledge and through building runnable mental models can be justified with a developmental model such as Hoc's notion of "machine learning" (1977), which will herein be utilized as a conceptual framework of concept-learning. Hoc proposed a model based on a machine-like construct which he called a "device". He defined device as "a means of relating the conduct of both processes and behavior of

the subject to his environment". (By this definition, one's own body can be thought of as a device). Such a device must have operations which it performs, and the set of rules for these operations constitute what Hoc calls its "device language". One never directly interacts with a "device", only with a "device language". This is a critical aspect of this model; the constructivist notion that the reality (of the "device") is only understood indirectly through interaction (with the "device language") (Hoc, 1977).

In the context of computer languages, the device language might be the low-level operation codes of a machine-language or the more symbolic commands in a high-level computer language. In the former case, it seems that the "device" in question would be the computer itself. However the latter case is less obvious. If one treats the Logo language, for example, as a "device language", then what is the "device" underlying it? The physical computer hardware? One could interpret the situation in that way, as does Meyer in his analysis of BASIC. However Hoc's approach was to talk about the device as an idealized representation of the computer language (like duBoulay's "notational machine"). The "device" of Logo is a characterization of the entire language, above and apart from its grammar and syntax.

The human subject in Hoc's model acts upon the device language, and "catches information" through the responses of the language to his actions. Simultaneously, the subject is receiving feedback from his personal environment, which eventually, through a process Hoc calls "interiorisation", comes to codify this dialog as a mental representation. "As soon as the subject is able to operate a device mentally and predict the outcome", says Hoc, "even if incorrect, we say that he's constructed a representation with which he can make calculations". Hoc refers to this representation as a "Systeme de Representation et de Traitement" or SRT, (translated as "Representation and Processing System"), which seems to correspond closely to the "runnable mental model" of Collins & Gentner.

Any experimental subject may be thought of as having a vocabulary of SRTs in place. Given a new device, Hoc believes that a subject will adopt an analogous SRT and attempt to use it, making superficial alterations as necessary to adapt it to the new application. If this strategy leads to unrepairable errors, (i.e., the SRT is basically inappropriate), then he must adopt a more generic SRT, which will require more editing than a close match would have. Hoc sees this as a costly approach, to be avoided where possible. These processes, which can be likened to Piaget's processes of assimilation and accommodation, Hoc referred to as "representative activity" ("talk" between SRTs), and are a part of the

"interiorisation" of the representation.

Once an SRT is constructed, it may be used, according to Hoc, to solve problems. A situation may be seen as a problem, "when the subject represents to himself a pair of states ...the initial state and the final state... and a procedure which leads from one to the other in one or several of the SRTs he has at his disposal, providing such a procedure is not translatable, word-for-word, into the device language." He sees a problem solution as "the organization of the actions that the device is capable of performing from an initial state to obtain the objective". Notice that the aspect of this model that deals with problem-solving "...emphasizes the construction of a procedure in the device language and the definition of a representation of initial and final states compatible with that language" (Hoc, 1977). This is thoroughly compatible with the notion of bundling knowledge into functional classes, presented earlier. Again, a device and a device language are seen as inseparable constructs, and the mental representation (SRT) a subjective model of some aspect of the device, whether accurate or not. SRTs are functional, runnable bundles of device language, which may themselves be bundled together in order to solve a given problem. Presumably, procedures for individual problem solutions themselves then become SRTs.

This does not mean that expert programmers always generalize from their problem solutions, however. Hoc makes a distinction between a general and a specific solution to a problem. "It is one thing to ask a subject what is to be done in each case", Hoc states, "and another to ask him to construct a general solution for all cases" (Hoc, 1977). Vermersch (1972) defined the latter as "algorithmic behavior", and found that subjects often had trouble developing a general algorithm, even when they are familiar with many specific ones (in Hoc, 1977). Here Hoc's model falls short, failing to close his theoretical system by describing the means by which one generalizes classes of problem solutions. Using a Piagetian framework, one may assume that this involves a reorganization of knowledge, some sort of internal mental activity resulting in greater abstraction and more generalized mental representation of an entire class of problems. The question remains whether such reorganization takes place in the development of programming expertise.

Hoc's model is flexible and elegant; it makes a clear distinction between the formal and psychological aspects of a language, and it explains their interaction. It distinguishes between a language grammar and the conceptual device that the language grammar suggests. It models concept learning as a mental representation of the conceptual device (rather than of the device language). It provides an explanation for the idiomatic bundling of

programming knowledge observed independently by others and offers a framework for describing the means by which "runnable mental models" may be built and then linked together to form an expert knowledge-base composed of such models. It is also strongly reminiscent of Piaget's familiar description of cognitive structures, and this increases its utility as tool for analyzing the divergent work in this general field.

Program "Bugs" and Programming Expertise

The analysis of program errors and debugging behavior will be an important technique used in this study. By analyzing programming bugs and debugging activity, other researchers have gained important insights into how both experts and novices think about programming and how they solve programming problems

Debugging falls into the category of the "pragmatics" of programming, recommended as a focus in computer instruction by Minsky (1970). Papert (1980) believes that debugging is an important cognitive skill, which he feels can be learned through programming in a language such as Logo. This suggests that debugging strategies and associated skills may develop independently of other programming concepts. Surely, some students of computer programming come to develop very domain-specific and clever debugging strategies that undoubtedly help account for

programming success. But debugging ability must at some point become closely linked with other programming conceptualizations to produce the unified knowledge typical of experts. In Hoc's model, errors simply provide data about the programming language; they are an aspect of the device language. It is difficult to describe any distinction at all between internalization of general concepts and debugging skills using that representation. While other models of programming knowledge would allow one to treat debugging skill as independent, existing studies provide little support for that view.

In order to confirm and expand his model of "device learning" described earlier, Hoc (1977) constructed an experiment using an analysis of bugs of different sorts to establish the deep conceptual functioning of programmers of different levels of expertise. He assigned 20 COBOL programmers (from a commercial software house), with abilities ranging from near novice (but having an knowledge of COBOL syntax) to expert programmers. He asked all of them to create a flowchart of a computer program to control the ticket dispensing and coin-changing functions of a theoretical automated ticket dispenser, with only the communication protocols between parts of the change machine and the controlling machine specified. By Hoc's analysis, this task required the development and interiorisation of new subjective representations (for the change-making and ticket dispensing devices) and the active use of the

programmer's mental model of the COBOL computer language. He studied programmers' generation of errors, (classified with a taxonomy designed to show the relation of an error to the subject's SRT) as well as their activities in creating the flowchart, and gave a structural analysis of their final or "terminal" flowchart. By design, the study was focused only on "representative activity", i.e., the subject's manipulation of his internal model of the devices in question, not interaction with the device language. His assumption, one which will be adopted in this study, was that a subject's physical activity (creating flow charts, diagrams and programming code) can be taken to represent internal activity and can indicate a great deal about a subject's conceptual processes, because their work is sharply focused on those aspects of the problem that they find troublesome.

Hoc analyzed his results using his model of device learning, to find three stages of interiorization of COBOL SRTs. The first corresponded with the performance of novice programmers, who show little of Vermersch's "algorithmic thinking"; Hoc believed programmers in this first group had constructed only minimal SRTs. They did not calculate in an SRT associated with COBOL, made little association between similar classes of "test-condition" tasks, but rather constructed a horizontally wide "tree structure" in their flowcharts, indicating an attempt to deal with many similar processes as special cases. They

wrote little COBOL code in early passes at the flowchart, and when they did attempt to write code, they tended to commit errors as well.

The second group were seen to have partially internalized cognitive models of COBOL. Their terminal flowcharts were more vertical, utilizing conditional branching rather than looping, they did write in programming code in places that they found familiar problems, but abbreviated other, difficult parts of the problem.

The third group, who Hoc believed had thoroughly interiorized both an accurate mental representation and a large library of usable algorithms, tended to return to writing abbreviations rather than COBOL code, but moved quickly to define the "best representation of the data, to detail only the difficult parts of the problem for which there are no obvious algorithms". It is not clear how Hoc distinguished between "difficult" parts or "best representations" and their opposites, and he apparently made no attempt to associate group membership and other measures, though his anecdotal descriptions of the meanings of different factors was informative.

Jefferies (1982) provided some support for a model such as Hoc's. He found that experts debug programs in a fundamentally different way than do novice programmers. He believed that experts use the "runnable mental models",

mentioned earlier, in their debugging activity. He also found that "Experts read for flow of control as expert readers do, rather than line-by-line as does the novice" (cited in Pea & Kurland, 1984).

It is not clear to what extent computer language learning and expertise differs between computer languages. In a examination of bugs in learning the APL language, Eisenberg & Peelle (1983) found programming bugs that seemed to have no ready analogy in reported bugs in other programming languages.

APL combines an extensible language design (i.e., programs, once defined, follow the same syntactic rules as primitive functions), with a rich set of functions and operators, which may be functionally composed or organized into programs. It has an extremely simple rule for evaluating expressions with no hierarchy of functions, and utilizes its own keyboard and symbol set. Eisenberg & Peelle's classification of bugs was an informal one, and was designed in part to uncover the specific idiosyncrasies of APL programming. They found some bugs to persist well into intermediate stages of programming, while others passed out of use as novices gained a modest exposure to the language. For example, in their "Naive Bugs" and "Babel Bugs", found especially in novices with previous experience in other programming languages, neophyte APL programmers showed a tendency to ignore new, more efficient

functions in favor of more familiar ones. Their "Logical Bugs" indicate failures to understand the semantics of APL's primitive logical functions or to construct correct Boolean expressions, (not a fault in program logic or flow-of-control, as others have used the term) (Mayer, 1981; Soloway et al, 1982). "Dummy Bugs" come when novices try to write APL procedures. One they report as very persistent, and seems to result from the combination of a misconception about replacement of a function call with its result and the way APL handles a result used as input to another function. "Inventive Bugs" of various types seem to indicate an early tendency of students to develop a model of APL's grammatical structure and that novices interact with the language experimentally to a degree that researchers of other languages have not reported (Eisenberg & Peelle, 1983).

Debugging may be useful as an instructional device as well. Lemos (1979), building on the concept of "team debugging" developed by the IBM Chief Programmer Team, implemented a program of what he calls "Structured Walk-Throughs" in teaching COBOL programming. Using one section of a beginning and one of an intermediate COBOL programming class, he had his students critique each others' un-debugged programs. After critiquing, students had an opportunity to correct their own programs and run them one time only. This was continued over a 10 week period. Control groups, composed of other sections of the

same classes, received more detailed instruction about the language but did no critiquing of the sort described above.

At the end of this time, Lemos administered a one hour test of language grammar, program reading/debugging and program writing. He found no significant differences between the groups in understanding language grammar or in their program reading and debugging skills. But the Structured Walk-Through group showed a significant improvement in their ability to successfully complete a program writing task. They also showed a significant decrease in the number of test-runs required to complete programming assignments during the remainder of the semester (Lemos, 1979).

With these results, Lemos concluded that the Structured Walk-Through more efficiently teaches good programming practice. The results are confusing, though. After so much more emphasis on critiquing in the groups doing Structured Walk-Through, why did they fail to demonstrate an improvement in program debugging skills? One would expect to see such improvement, simply as a learned behavior, especially if one views debugging as an independent skill. If the Structured Walk-Through students did not show improvement in understanding language grammar or in programming reading, then what accounts for the decrease in program writing time? One explanation is that

Lemos's approach helped habituate students to proofreading before a test-run. Turkle (1984) has identified several personal styles, and has seen them reflected in programming habits; careful proofreading is one of the traits identified with what she calls an "obsessive-compulsive" style personality. Lemos may really be teaching some aspect of behavior related to a personal style. Turkle is critical of the tendency of many teachers to favor the obsessive-compulsive style, and notes that teachers often confuse programming style with demonstrated programming ability, something that Turkle believes will discourage students with the more creative and less organized ("hysterical") learning style more typical of young girls (Turkle, 1984). In any case, Lemos's study diminishes the view that debugging is primarily an independent cognitive skill.

Programming Plans and Procedural Knowledge

In an attempt to explain bugs in the addition algorithms of young children, Brown & VanLehn proposed a procedural model for knowledge and misconceptualization (1979). They likened a correct addition algorithm to a procedure in a computer programming language which includes all the steps necessary to carry out the process of addition--in effect, a plan for doing addition. They postulated that errant algorithms are generated when a line or part of a line is somehow lost from a correct plan, and

the missing parts are then "patched" by a set of repair rules and evaluation heuristics called "critics", the metaphorical equivalent of a "bug" in a computer procedure. This results in a procedure which will "run" but produce faulty results. Brown & VanLehn's "critics" seem to function as a mechanism to resolve ambiguity and provide a kind of closure. They have developed a computer program called BUGGY to simulate "bug-patching" in the algorithm for addition. They have also generated all known student math bugs for this type of problem, and only a few bugs generated by their model have never been seen to arise in student work (Brown & VanLehn, 1979). The predictive power of this model argues for its validity.

Brown & VanLehn's research suggests that children are constantly altering personal computational algorithms in unconscious, somewhat automatic and sometimes maladaptive ways. If the structure of BUGGY has some real correspondence in human information processing, then the mandate for teachers is to learn better how to understand the logic behind their students' incorrect answers and provide them with the insight to correct their own "buggy" plans. This implies not only listening to one's students but also learning to recognize when a student's wrong assumptions, though not apparent, have led to the maladaptive plans evidenced in student errors.

While the author is suspicious of any notion of problem-solving that de-emphasizes human choice, protocols utilized in the present study include long episodes of plan-dominated activity. Some programming errors seem as persistent as the difficulties with addition algorithms studied by Brown & VanLehn, and both of these things suggest that some of the of high-level notions of programming may be internalized as procedural structures, similar to Brown & VanLehn's addition algorithms.

Anderson (1982) generalized the notion of procedural knowledge to all cognitive structures and proposed a mechanism for the learning of any cognitive skill. He theorized that any skill is learned in two stages. In the first stage, the "declarative" stage, the skill is encoded as a set of facts which can be utilized by general, interpretive procedures (previously internalized). In the second, "procedural" stage, knowledge has been formulated into a domain-specific "production", composed of a condition specifying when to use this production and a sequence of actions given in terms of existing productions (Anderson, 1982). It should be noted that the productions function as goal-setting mechanisms. Anderson's assumption is that only one goal may be actively pursued at a time, and so productions can be thought of as plans for organizing which aspects of a problem the solver will attend to.

In a prolonged study of the first 30 hours of learning to program in Lisp, the parent-language of Logo, Anderson, Farrel and Sauers (1984) observed a large increase in speed when students solved a novel problem and then a second, analogous problem. They interpreted this increase in speed as indicative of the proceduralization of the problem as a complex cognitive skill, a process that they call "knowledge compilation". In mapping the productions associated with a nontrivial problem, they produced a hierarchical tree structure, composed of frames with a variable to represent all potential objects of each action. They observed a common, and what they believe to be a natural, progression through the frames that make up the tree as top-to bottom and left-to-right. Besides the limitation to one active goal, they perceived a limit of what they call "working memory" to be a major constraint on this production system, and they interpreted several observed difficulties during the declarative stage as a function of this limit (Anderson, et al, 1984).

M. L. Miller's (1980) extensive analysis of high school students' thinking aloud while doing graphic-oriented Logo programming proposed a different hierarchical tree structure for programmer planning. Miller felt that the three stages of planning activity were, in order, identification (identify the problem as a previously solved problem), decomposition (decompose into sub-problems) and reformulation (reform into an alternative

problem). He implemented this idea into a structured planning and debugging environment called SPADE-O. Initially SPADE-O was programmed to guide students through the identification, decomposition and reformulation processes, in that order, carrying each to completion in a top-to-bottom, left-to-right pattern. Complaints and suggestions from users led to a set of "preference rules" that override that order in the present implementation. New considerations for ordering, integrated into the preference rules included: (1) solve main steps before interfaces (e.g. build both a "square" and a "triangle" before creating a super-procedure to combine them into a "house" (2) prefer direct neighbors (in the tree) and (3) prefer simpler to more complex goals. Miller observed that all of these "preference rules" served to minimize future modification. He called this the "least-scope principle", and he saw it as the motivation in the "top-down" approach sometimes taught as a preferred planning structure in programming courses (M.L. Miller, 1982)

Bonar & Soloway (1983) have found that "novice programmers have deep and interesting misunderstandings" derived from general experience and knowledge. They suggest that student-programmers utilize those aspects of general knowledge that most closely (but not exactly) match the programming problems that they encounter. But such an association often matches the problem only in superficial ways and may eventually lead to programming errors. Bonar

& Soloway use the term "natural language strategies" for these potentially distracting strategies. They uncovered, through close examination of the "loud thinking" of novices (a technique in which problem solvers are asked to verbalize their reasoning as they work on a problem) and through a very careful look at novice bugs, some of the natural language strategies developed by novices as they attempt to understand looping structures as they learn to program in Pascal. In specifying a procedure for repetitively processing a string of information in a non-programming context (the problem was to determine average salary for factory workers as they leave work through the gate, trailed by their supervisor), the natural strategy, produced by all subjects, was to read the salary from each worker, then add it to the sum of the salary of all previous workers (keeping track of the number of worker so far encountered) and finally, upon meeting the supervisor, to compute the average. They referred to this as a "Read-Process" strategy. In a similar Pascal problem, however, that strategy tended to lead to errors (Soloway et al, 1982).

In a later study that drew heavily on Brown & VanLehn's theory of addition errors (mentioned previously), Bonar & Soloway (1985) reported other sources for programming errors and proposed a general mechanism for the generation of programming bugs. They believe that most bugs begin with gaps in programming knowledge (PK), which

they see as distinct from natural knowledge. In groping for a means to fill this gap, novices usually draw upon natural knowledge, thus generating a bug. Repetitious techniques found in common problem solving experience, such as seen in the factory gate problem, above, are referred to as "step-by-step knowledge (SSK)", and a bug can occur when "SSK confounds PK". However, other bugs seemed to be related to natural language interpretations of programming concepts. Bonar & Soloway classified these as a type of SSK, but I like to think of them in a separate category, and refer to them as "natural language confounds". Still other bugs can result from confusion of programming constructs with one another (Intra-PK confounds), or with knowledge from other domains, such as mathematics (Other Domain confounds) or with aspects of the current operating system (OS confounds). Both Brown & VanLehn and Bonar & Soloway refer to these "confounds" in general, as "bug generators". In a broader view, it is possible that "bug patching", in programming, mathematics or other domains, may serve as a mechanism to integrate new knowledge into existing mental constructs. Only when it becomes "compiled" into an algorithm and used in an unconscious and maladaptive way should it be thought of in negative terms.

To summarize the research cited on planning, Anderson et al (1984) believe that raw novices begin with descriptive knowledge utilized by general problem solving plans and progress to domain-specific, procedural knowledge

through the encoding of problem solving plans called compilation. Soloway et al (1981), Bonar & Soloway (1985) and Hoc (1977) all studied plans, which are close analogs to procedural knowledge, and all saw procedural knowledge as developing first as low-level implementation plans and progressing to high-level strategic plans. One sees a picture emerge of the learning of a complex cognitive skill as a movement from general knowledge to domain specific knowledge which slowly accumulates (in a building-block fashion) back to general knowledge. The studies cited above provide some detail as to the first two stages, but what of the third? The author proposes that the final stage of learning programming, or of any similar cognitive skill, must entail the integration of domain-specific knowledge into general knowledge structures. Only at this point can one fully reflect on one's own knowledge. I propose the term "meta-programming knowledge" for this stage of true mastery, to emphasize the important role of metacognition.

Metacognition and Programming Knowledge

Silver, Branca & Adams (1984) summarized general research on metacognition in terms of two themes: "(1) with development, individuals adopt an active, self-directive role in certain areas, and (2) individuals develop the ability to monitor and evaluate their own cognitive processes." While there is some disagreement on this point,

development in both of these areas seems to progress both with age and with experience in a particular domain. They saw this development as corresponding to an awareness of and attention to structure, i.e., the sort of "functional chunking" previously discussed, though they note that metacognition may decrease when cognitive skills become completely automatic.

Smith & Inhelder (1975) used a block-balancing task (which was already known to progress through several stages, linked to general cognitive development) to study the "micro-formation of physical knowledge." Several blocks of varying shape and construction, including some built with hidden weights, were presented to children ranging in age from 4.5 to 9.5 years. The researchers distinguished between actions related to theory-testing and goal oriented activity. As expected, subjects' behavior fell into age-linked stages. In a pre-study, children eighteen to thirty-nine months old were observed placing blocks on the fulcrum in a single location and pressing down hard above the fulcrum. If the block fell, they would replace a block in the same position and press down once more. The youngest children in the formal study seemed to know that movement could lead to success, but had little ability to predict the results of their action. They began by balancing blocks at a random location, sometimes pressing down from above the fulcrum like the toddlers. However they would progress to making small adjustments based on

proprioceptive information, until a block was balanced. These children were able to balance the more complicated blocks (i.e., ones with asymmetrical structure, some requiring counterweights or especially ones with hidden weights), in much the same way as they balanced the simplest ones, i.e. using proprioceptive information. Some began to shift their attention to an exploration of the properties of blocks, a shift that Smith & Inhelder saw as indicating the development and testing of predictive theories.

More advanced subjects began near the geometric center of the base of a block, then made small shifts to quickly balance the simple blocks (i.e., the unweighted, symmetrical ones), and eventually to balance all blocks but the ones requiring counterweights. Some children at this stage became distracted by the weighted and asymmetrical blocks, and began to explore these, apparently trying to develop new and more predictively powerful theories. Often, these subjects would begin to experience difficulty balancing the asymmetrical and hidden-weight blocks that they had previously balanced successfully, although they could achieve success if they closed their eyes. Smith and Inhelder believed that these children were also shifting their attention toward the theory that has been shown to develop at around seven years of age, that of balance based on weight. Older children were seen to pause BEFORE attempting to balance a block and placing it close to its

balance point on the first try. This pause before the problem was seen as indicative of an internal process of theoretical prediction. The notion of experimentation as being either theory-responsive or goal-responsive, the idea of retrograde learning as indicative of a new theory-in-development and the attention to pauses as an indication of internal processing are several features of Smith & Inhelder's study which will be important mechanisms for analyzing experimental activity in the present study.

Papert (1980) observed that children often are unable to recognize mathematical algorithms as correctable entities, which supports the notion of a procedure-like bundle of algorithmic knowledge. His claim was that programming experience helped some of these children to identify and correct math bugs by making them more conscious of bugs and debugging behavior (Papert, 1980). This suggests that procedural knowledge can be "edited" through self-conscious activity (metacognition), that the right kind of activity can result in a shift to metacognition, and that computer programming can provide such activity. If metacognition is an important component to mathematical problem solving, as Silver et al (1980) believe, then it is likely to be important in computer problem solving as well. Experimentation, according to Smith & Inhelder, can at some stages induce metacognition and shift focus from goal achievement to theoretical reformulation. All of this leads the author to see

metacognition as the final stage in learning programming, following first the use of natural problem solving concepts and strategies that lead to many programming bugs and then the development of domain specific programming plans.

This concludes the review of literature on general conceptualization as it relates to programming skill. The next section goes on to examine some particular features of programming languages as they effect learning, most notably program control and the use of variables

Studies of the Learning of Specific Language Features

All of the studies mentioned up to this point have been top-down, attempting to describe the nature of expert programming knowledge. The following studies, however, are bottom-up. They examine in detail the conceptualization of particular constructs, specifically the concepts of program control, expression parsing and variables.

Program Control and Expression Parsing

Sime, Green & Guest (1977) saw two sorts of important information implicit in a computer program, which were commonly used by experienced programmers. The first, sequential information, was generally recognized as the sequence of commands that are developed when an experienced programmer translates a problem into a program. The

second, which they call "taxon information", was not so commonly recognized but was equally important to the expert programmer. Taxon information refers to the taxonomy of the underlying problem, and can be thought of as the residual information left after code is written that can aid the programmer, for example during debugging, in recreating the original problem.

To test this hypothesis, they created three pseudo-languages. One allowed only for a JUMP command as a control structure. Dijkstra (1980) has criticized this control structure as destructive to the smooth and simple translation by the programmer from the dynamic process that the programmer has in mind as a goal to the text of the actual program (i.e., it is weak in sequential information) and back again (i.e. weak in taxon information). This has been the underlying argument in favor of the nesting constructs favored by the proponents of structured programming.

A second pseudo-language had logical structures allowing nesting in a fashion similar to ALGOL, i.e. of the form

```
IF (condition) THEN
    BEGIN
        STMT 1
        STMT 2
    END
```

```
ELSE  
  
    BEGIN  
  
        STMT 3  
  
        STMT 4  
  
    END
```

The third language allowed structuring but also required a redundant statement of the condition controlling the nesting structure. In constructing it, Sime, et al (1975) decided to forego the use of BEGIN and END to mark the scope of the control structures, but chose instead a scheme such as;

```
IF (condition) STMT1 STMT2  
NOT (condition) STMT3 STMT4  
END (condition)
```

They felt that this hybrid structure was richer in taxon information.

In testing the ease with which these languages were used by beginners and then by expert programmers, they found that the simple nest structure was most susceptible to syntax errors, mostly caused by forgetting the last of a BEGIN/END pair. It also performed the worst in terms of perfect execution of the first programming pass. The JUMP language was most susceptible to logical errors, (i.e. errors in regard to the program logic). Their hybrid language, which required the statement of the condition before the THEN clause and also before the ELSE clause, was

10 times faster to decode! By Sime et al's analysis, this performance was due to the clearer taxon information in that language. They were critical of the performance of the structured style, and saw much of the justification for it by researchers such as Dijkstra to be lacking in experimental evidence (Sime, et al, 1977). This criticism may account for the lack of reasonable performance observed by some researchers in solving fairly simple programs in PASCAL, even by intermediate-level programmers (Soloway et al, 1982).

L. A. Miller found that the conditional construct was itself at odds with natural problem specification. He found that non-programmers preferred a "qualificational" ("put all the red things in box 1") over a conditional ("if thing is red, then put it in box 1") IF statement in their natural statement of a problem or in a first computer language (cited in duBoulay & O'Shea, 1981). This seems to support Sime et al's emphasis in taxon information, in that a qualificational construction seems to point more strongly to the taxa being acted upon than does the conditional form.

In another study of Miller's (1974), he found that novice programmers were far more successful using the AND construct than using OR. He administered tests of program generation with a pseudo-language of his own design, featuring only a conditional branching command for altering

sequential flow of control. The tests were administered under computer control, and Miller was able to report not only on the successful completion of the sorting problems given to subjects but also on the amount of time spent making selections by command type in AND problems as opposed to OR problems. He found that OR problems required almost half-again as much overall time to complete, but more than twice as much time making modifications and viewing displays and 10 times as much total editing time.

Miller found that thirty-three of sixty-seven programming errors generated were in conditional statements, indicating the importance of an efficient conditional structure in a computer language. OR problems also resulted in more errors than AND problems overall by two to four times. Problems that required the negation of a conditional also gave the novice programmers trouble. The worst performance of all was for OR problems requiring one but not both conditionals to be inverted. Most surprising of all, Miller found that as many as 50% of his subjects would reconstruct an OR problem to force it into an AND program structure. This strategy led to more errors in conditional statements, but fewer implementation errors, indicating that once the conversion was made, programmers understood their programs better and had less trouble manipulating them (Miller, 1974).

All three of these studies of program control structures support Bonar & Soloway's notion that novices possess strong, natural inclinations toward some structures over others (Bonar & Soloway, 1983).

Program control in Logo is not the same as in Pascal, or in the proto-languages used by Sime et al, or in Miller's two studies. Sheil (1981) has offered, in a data-analysis language for Social Scientists called IDL, what he believes to be a more natural language structure than iteration, and one which is also closer to the control structure of Logo. IDL offers a small set of highly specialized operators and uses only functional composition, which he believes is easier for novices to use, as a control structure. (Functional composition is the main control structure in Logo, as well.) Sheil found that users did find IDL to be simple and natural to use in most applications, but that more complicated uses required too deep a nesting of functions and of parentheses for the novice user. He also found IDL inappropriate for many computer applications which were essentially procedural, (e.g., automating office procedures) (Sheil, 1981).

In a functional language such as Logo, the main rule of precedence is that, barring parenthesization, functions are ordered in precedence from left to right (this is identical to functional composition in mathematics). The exception in Logo are the infix operators, (+, -, *, /), which

bind most tightly and are themselves ordered as per algebraic convention. This is a variation from the simple grammatical structure suggested by duBoulay et al, and may be a source of confusion for novice programmers. The author's assumption is that in order to attain expertise as a Logo programmer one must learn to compose functions, even deeply nested functions, and to master the rules of precedence for exceptions such as the infix operators mentioned above.

Allen & Davis (1984) support the use of functional composition as an underlying language structure. In their discourse on the state of computer programming languages today, they offer three categories: procedural, functional and relational. In the first category they lump BASIC, PASCAL, FORTRAN and most versions of LOGO. In the second, they place APL, LISP and TLC-LOGO. "In a purely functional language the notation only describes relationships between components and makes no demands on how these relationships are computed". Allen & Davis argue that functional languages are more mathematical, that they "...can be looked upon as abstract descriptions of phenomena independently of how (or even if) they are executed on a machine. This abstraction means they have notational/expressive power that may be reasoned with and about." Their third category is the relational family, including "logic programming languages" such as PROLOG. Such a language "expresses problems as a collection of

logical assertions -- typically assertions about individual objects and relationships between objects. Though these assertions appear to be purely descriptive, such notations are executable." Relational languages also may be abstractly manipulated, but are "descriptive" rather than "prescriptive", i.e., they describe a problem rather than a solution. (Allen & Davis, 1984)

While program control and expression parsing are beyond the scope of this study, both functional composition and the primitive control-structures of Logo must be considered in the light of the above discussion. In particular, this study will bear in mind the effects of control structure in a programmers' use of the REPEAT statement, of sub-procedures, operations, recursion and of workspace organization, (workspaces being collections of the "abstract descriptions of phenomena" that Allen & Davis describe).

Variables

An understanding of variables is sometimes seen as an important measure of programming ability. DuBoulay et al recommended a programming environment that included a visual display of currently defined variables as well as other features (duBoulay et al, 1981). Soloway et al found that some novice-programmers see loops with internal counters and loops that affect external variables as

distinct cases, to which they ascribe unique knowledge-frames. Kurland & Pea (1983) suggested that understanding the role of the local variables is landmark in understanding recursion.

We can learn something about how variables are perceived by examining the metaphors that are commonly used to understand them. Harvey (1985) claimed that there are several metaphors for variables commonly used by Logo experts. Some experts think of a variable as a mailbox. The name of the variable corresponds to a person's name on the mailbox, and its value can be likened to a letter inside. Others think of a variable as a frame with a slot, like the frame in a taxi cab that announces the name of the driver, with the driver's nameplate corresponding to the value of the variable. Still others think of variable names as labels for something and their values as the things being labeled. What all of these metaphors have in common is: (1) a distinction between variable name value, (2) a functional distinction between the outside of a variable (its name) and the inside (its value) and (3) the idea a one-way reference of a variable's name to its value but not back again (e.g., one looks in a mailbox to find what's inside, but never examines a letter to determine what mailbox it was placed in) (Harvey, 1985).

A simple developmental model for variables is suggested by Rodgers (1980). She makes an attempt to

correlate the stages of cognitive development of students with the cognitive demand of various concepts of variables. She notes, first, the assertion of developmental psychologists that children move from a focus on concrete objects, to begin to classify objects into groups and finally to classify processes, ideas and even problems into groups. Concomitant with this is the movement from physical to more abstract representations, and of a simultaneous change of logical analysis from a less formal to a more formal style (Inhelder & Piaget, 1958).

Rodgers suggests four "hierarchical levels of data structures" in BASIC: (1) simple data, (2) simple variables, (3) "structures through which the data are addressed less directly", such as arrays or records and, finally, (4) programmer-defined data structures. While Rodgers' approach of linking computer concepts with developmental theory is appealing, the author finds some problems in her her categorization of data structures. As an example of "simple variables", she offers both

```
10 INPUT A
20 PRINT A
```

and

```
10 LET A=1
20 LET B=A+1
```

```
30 PRINT B
```

 However, the author has found

through experience that the INPUT statement of the first program is far more difficult for novice BASIC programmers to understand than are the PRINT or LET statements. Perhaps this can be explained in terms of a dynamic interaction between the programming environment and the data structure.

The variables in the second program are "temporally constant", in that their values were introduced at the time that the program was written; all values can be easily decoded by simply looking at the program listing. This is not to say that they were necessarily written at the same time; line 10 might have been defined 2 hours before line 20, but even were this the case a novice programmer would have only slightly more trouble understanding the program than if both lines were written at one 2 minute sitting. The point is that A and B are both members of "the program", a single conceptual unit that can be viewed by the LIST or executed with the RUN command. Contrast this with the first example. A novice might experience difficulty in recognizing the value and understanding the meaning of "A", because the variable here is created at what can be called "programming time", while its value is assigned at a distinct conceptual time-frame, known to programmers as "run time". Most BASIC instructors and most BASIC textbooks fail to recognize the importance of this feature of the programming environment, and Rodgers seems to make the same mistake. (She also fails to make a

distinction between lines of a program and statements executed outside of a program in "immediate execution" mode, another "temporal" distinction).

Based on experience, the author tends to agree with some aspects of Rodgers' hierarchy; for example it seems that students do have more trouble understanding variables than simple data, and that user defined records are more confusing to students than primitive data structures. This may be explained in terms of levels of indirection. A variable is a name for data but a data-type definition is a name for a type of name. To understand the latter, it would seem that one must rely on knowledge of simple naming conventions of the former. It is not clear, however, that there is anything inherently more difficult about arrays than simple variables.

The problems chosen by Clement et al (1980) in the previously cited study that showed that college students and engineers had better success translating a word problem into a program than into a mathematical equation, required an understanding of co-variation to be solved correctly. Students were asked, in part of the study, to talk about the problem and to explain their solution as they were solving it, a technique known as "Loud Thinking". For example, when explaining his answer $(6S=1P)$ to the task of writing an equation to express the relationship of six students for every professor, one student commented;

"There's six times as many students, which means it's six students to one professor and this (points to 6S) is six times as many students as there are professors (points to 1P)."

The authors comment, "The correct equation, $S=6P$ does not describe sizes of the groups in a literal or direct manner. Rather, it describes an equivalence relation that would occur if one were to make the group of professors 6 times larger". Their results show that during programming, subjects are better able to see the relational role of variables. They did not study the question of whether programming experience has a long term effect on students' ability to solve such problems (Clement, Lochhead & Soloway, 1980).

The body of literature on variables in programming is very sparse, but variables have been studied extensively in mathematics. While one cannot assume that identical skills are involved in using variables in computer programming and variables in mathematics, a general concept of variable should integrate the idea from both domains. For this reason, several studies of concept development and misconceptions of variables in mathematics form the basis of a generalized concept of variable.

In a Piagetian study, Kuchemann (1978) presented a set of fifty-one questions to 3000 high school students in Great Britain. Based on the results, Kuchemann has

identified six distinct categories of variables which he believes are mastered at different stages of development. His statistics both differentiate some problems as more difficult than others and show direct relationship between age and success rate in each category.

Kuchemann's categories range from more concrete uses of variables to more abstract. They include the following (in the order of Kuchemann hierarchy): "Letter Evaluated" (e.g., $a+5=8$; $a=?$), "Letter Ignored" (e.g., $a+b=43$; $a+b+2=?$) and "Letter as Object" (e.g., write an equation for the perimeter of a geometric shape illustrated with four sides labeled "h" and one labeled "t"), "Letter as Specific Unknown" (e.g., write an equation for an incompletely drawn figure with n sides of 2 units each), "Letter as Generalized Number" (e.g., $c+d=10$, $c<d$, $c=?$) and "Letter as Variable" (e.g., "Which is larger, $2n$ or $n+2$?").

Kuchemann interprets these results as showing that the understanding of the concept of variability is linked to a kind of "closure"; the more indefinite the value represented by a letter, the more difficult it is to understand. In other words, the concept of variable is seen to correspond to a Piagetian ordering from concrete to abstract. A true understanding of mathematical variability lies at the end of this scale, with a recognition that a

symbol can stand for any given instance in an infinite series of acceptable values.

Applying this to programming, one would expect a command in Logo like

```
MAKE "L 7
```

to correspond with the category, "Letter as Object", because the meaning of the letter can be immediately associated with a specific thing. The command,

```
MAKE "X RQ
```

corresponds nicely to the category, "Letter as Specific Unknown"; the letter stands for that specific value which the user will enter at a later time. An input to a procedure could also be thought of as similar to the category of a "Specific Unknown"; (whatever number the user supplies when he uses this procedure), but the integral relationship between a procedure name and its parameter inputs suggests a closer association with the category of "Letter as Generalized Number" (i.e., the letter stands for any permissible number). The output of a function corresponds to the "Letter as Variable" classification, since the programmer must specify the relationship of the functional result to its input in the form of an algorithm (Kuchemann, 1978). If Kuchemann results are correct, and if the association we suggest is accurate, then one would expect that the each class of variable use mentioned here in sequence would be more difficult to master. While this question is beyond the scope of this study, Kuchemann's

extension of the ideas of a concrete/abstract continuum and a need for closure to the examination of variability have influenced both the design and the conclusions of the present study.

In a study of subconcepts and misconceptions of covariation in algebra word problems, Murray & Clement (1986) found three independent subconcepts -- single variables, functions and equations; and three skill levels -- basic, static/discrete and dynamic/continuous. As with Kuchemann's model, an unadorned MAKE statement would appear to parallel the simplest sort of variable while functional output would seem to parallel the most complicated, while variables with their values interactively assigned by the user and procedural inputs would seem to lie somewhere in between (Murray & Clement, 1986).

Finally a particular algebra misconception, the reversal error in the Student and Professor Problem, discussed earlier (Clement et al, 1980) was found to be very resilient to explicit instruction (Rosnick & Clement, 1980). While several instructional approaches were utilized, including identifying errors, suggesting conceptual models, graphing, plugging numbers into the reversed equation and demonstrating the correct solution, none were found to be very effective. The authors reported that other misconceptions showed signs of similar resistance to instruction. This seems to enhance the

importance of the positive effect of programming found in the previous study (Clement et al, 1980). Apparently programming was not used as an instructional technique by Rosnick & Clement (1980), and that study shed no further light on the question of whether programming experience somehow taught against the reversal error in the earlier study (Clement et al, 1980) or whether the effect was due to a greater linguistic simplicity of the functional form of the equation.

While much of the research cited here is intriguing and emphasizes the complex nature of variability, it does not address many of the basic questions about variables in programming. One question is the role that mastery of the concept of variability plays in programming skill development. What is needed is a careful charting of the subconcepts and misconceptions of variables as they occur over the course of learning to program. Another useful observation would be of the images and metaphors utilized by programmers at different stages of development to determine, for example, if nonexpert metaphors have the same consistency that Harvey finds in those of experts (Harvey, 1985). The relationship between variables in mathematics and programming variables needs to be further explored as well. In what ways are programming variables truly a subclass of mathematical variables and in what ways are they a unique and independent class of their own? More extensive pedagogical models of variable learning in the

context of programming must be developed to integrate some of the features discussed here and to produce specific suggestions for instruction.

Footnotes
Chapter 2

Elliott, P. "Computer 'glass-boxes' as advance organizers in mathematics instruction", International Journal of Mathematics in Science and Technology, 9, (1), 79-87, 1978.

Peelle, H.A. "Alternative Algorithms in APL: Implications for Education", Proceedings, APL '80, Association for Computing Machinery, 1980.

Peelle, H.A. "Learning Mathematics with Recursive Computer Programs", Journal of Computer-Based Instruction, 3, (3), 97-102, 1977.

Polya, G., How to Solve It: A New Aspect of Mathematical Method, Princeton University Press, 1957.

Papert, S. Mindstorms Basic Books, Inc., New York, 1980.

Pea, R. & Kurland, M. "On the Cognitive and Educational Benefits of Teaching Children Programming: a Critical Look", Technical Report, Bank Street College, 1984.

Pea, R. "LOGO Programming and Problem Solving. Technical Report, Bank Street College, 1983.

Kurland, M. & Pea, R. "Children's Mental Model of their Own Recursive LOGO Programs", Technical Report, Bank Street College, 1984.

Mawbry, R., Clement, C., Pea, R. & Hawkins, J. "Structured Interviews on Children's Conceptions of Computers", Technical Report, Bank Street College, 1983.

Schon, D. The Reflective Practitioner: How Professionals Think in Action, Basic Books, 1983.

Driver, R. The Pupil as Scientist, Open University Press, 1983.

Smith, A. & Inhelder, B., "If You Want to Get Ahead, Get a Theory", Cognition, 3, 195-212, 1975.

Kamii, C. "Pedagogical Principles Derived from Piaget's Theory: Relevance for Educational Practice", in Piaget in the Classroom (Schwebel, M. & Raph, J., Editors), Basic Books, 1973.

Duckworth, E., "The Having of Wonderful Ideas", in Piaget in the Classroom (Schwebel, M. & Raph, J., Editors), Basic Books, 1973.

Schoenfeld, A., "Can Heuristics be Taught?", in Cognitive Process Instruction: Research on Teaching Thinking Skills (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.

Turkle, S. The Second Self Simon & Schuster, New York, 1984.

Confrey, J. "An Examination of the Conceptions of Mathematics of Young Women in High School", unpublished paper, 1984.

Papert, S. "Teaching Children to be Mathematicians vs. Teaching Children About Mathematics. International Journal of Mathematics Education in Science & Technology 1972.

Louie, S. "A Report of a Pilot Study", Tucson Learning Center, Tucson, Arizona, 1985.

Clements, D., Gullo, F. "Effects of Computer Programming on Young Children's Cognition", Journal of Educational Psychology, 76:6, Pp 1051-1058, 1984.

Clement, J., Lochhead, J. & Soloway, E. "Positive effects of computer programming on the students understanding of variables and equations", Proceedings of the Association for Computing Machinery, National Conference, 1980.

Adelson, B. "Problem solving and the development of abstract categories in programming languages", Memory and Cognition 9, 422-433, 1981.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. "What do novices know about programming?", Technical Report, Yale University Department of Computer Science, 1982.

Soloway, E., Bonar, J. & Ehrlich, K. "Cognitive Strategies and Looping Constructs: An Empirical Study", Technical Report, Yale University Department of Computer Science, 1981.

Mayer, R.E. "The Psychology of How Novices Learn Computer Programming. Computing Surveys, 13, (1), 1981.

duBoulay, B., O'Shea, T. & Monk, J. "The black box inside the glass box: presenting computing concepts to novices", International Journal of Man-Machine Studies, 14, 237-249, 1981.

Hoc, J.M. "Developmental stages in learning to program" International Journal of Man-Machine Studies, 9, 87-105, 1977.

Minsky, M. "Form and Content in Computer Science", Communications of the Association for Computing Machinery, 17, (2), 197-215, 1970.

Eisenberg, M. & Peelle, H.A. "APL learning bugs", APL Quote Quad 13, (3), 11-16, 1983.

Lemos, R. "An Implementation of Structured Walk-throughs in Teaching COBOL Programming", Communications of the Association for Computing Machinery, 22, 6, 1979.

Brown, J.S. & VanLehn, K. "Towards a Generative Theory of 'Bugs'", Cognitive and Instructional Series # 2, Xerox Palo Alto Research Center, 1979.

Anderson, J. "Acquisition of Cognitive Skill", Psychological Review 89, 4:369-406, 1982.

Anderson, J., Farrell, R., Sauers, R. "Learning to Program in Lisp", Cognitive Science, 8:87-129, 1984.

Miller, M.A. "A Structured Planning and Debugging Environment for Elementary Programming", in Intelligent Tutoring Systems, (edited by Sleeman, D. & Brown, J.S.), Academic Press, 1982.

Bonar, J. & Soloway, E., "The Bridge From Non-Programmer to Programmer", University of Massachusetts, Amherst; Department of Computer and Information Science 1983.

Bonar, J. & Soloway, E., "Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers", Human-Computer Interaction, Fall, 1985.

Silver, E., Branca, N., Adams, V. "Metacognition: The Missing Link in Problem Solving?" Proceedings of the 4th International Conference for the Psychology of Mathematics Education, 1980.

Sime, M.E., Green, T.R. & Guest, D.J. "Scope Marking in Computer Conditionals- A Psychological Evaluation", International Journal of Man-Machine Studies, 5, 105-113, 1977.

Dijkstra, E.W. "Goto statement considered harmful", Communications of the Association for Computing Machinery 11, 3, 147-148, 1968.

Miller, L.A. "Programming by Non-programmers" International Journal of Man-Machine Studies 6, 237-260, 1974.

Sheil, B.A. "Coping with Complexity" in Cognitive and Instructional Sciences Series, Xerox Palo Alto Research Center, 1981.

Allen, J. & Davis, R., "In praise of fingertips", unpublished paper, 1984.

Harvey, B., Computer Science Logo Style: Intermediate Programming MIT Press, 1985.

Rodgers, J. "Teaching Beginners to Program: Some Cognitive Considerations", unpublished paper, University of Oregon, 1980.

Inhelder, B. & Piaget, J. The Growth of Logical Thinking from Childhood to Adolescence, Basic Books, 1958.

Clement, J., Lochhead, J. & Soloway, E. "Positive effects of computer programming on the students understanding of variables and equations", Proceedings of the Association for Computing Machinery, National Conference, 1980.

Kuchemann, D. "Children's Understanding of Numerical Variables", Math in School, September, 1978.

Murray, T. & Clement, J. "Progress Report: Evidence for Building Blocks Contributing to a Robust Concept of Variation and Covariation in Two-Variable Algebra Word Problems", Cognitive Process Research Group, 1986.

Rosnick, P. & Clement, J. "Learning Without Understanding: The Effect of Tutoring Strategies on Algebra Misconceptions", Journal of Mathematical Behavior, 3, (1), 3-27, 1980.

C H A P T E R 3

DESCRIPTION OF THE STUDY

Methodology

This study focused on the learning of and impediments to understanding variables in computer programming in the Logo computer language. Subjects included high school students who had a modest amount of training and experience with Logo and adult expert Logo programmers. After subjects were administered a concentrated instructional presentation on relevant Logo commands, each was asked to solve between four and seven programming problems in a microcomputer environment. The researcher was present during all problem-solving and encouraged subjects to discuss their thinking and the reasons for their actions. These interviews were videotaped and later analyzed using a cognitive microanalysis technique to uncover subjects' understanding of Logo variables within the context of their planning, factual knowledge and experimental activities.

The cognitive microanalysis approach basically involved a detailed analysis of subject-protocols in an attempt to plausibly reconstruct subjects' cognitive processes. While one cannot directly study cognition, one can make assertions about cognitive processes that explain observable behavior. Actual transcripts of protocols are included as appendices, to allow readers to evaluate this analysis independently or to utilize the raw data to

analysis independently or to utilize the raw data to formulate their own hypotheses. Analysis is provided in the form of a commentary on the protocols plus a collection of first order models concerning the cognitive processes operating in the subject. Diagrams are included as a means of representing this commentary.

The fundamental assumption in this study is the constructivist notion that the individual builds his own internal model of the world which influences his perceptions, actions and learning. Such a viewpoint places a greater burden upon the teacher to meet the student on his own terms, since learning is seen as a necessarily personal construction. Several studies have guided this approach. Smith & Inhelder's (1975) study of how experimental activity influences and is influenced by both theories and goals suggested the fundamental classification of protocols into three categories (i.e. theories, goals and experimental activity). The assumption that a programmer maintains only one active goal at a time was borrowed from Anderson et al (1984), along with his convenient classification of knowledge as either "procedural" or "descriptive" (although the term "plan" is preferred here for the former and "theory", "concept" or "assumption" for the later). Also loosely adapted from Anderson was the notion of a hierarchical tree structure for procedural knowledge, which forms the skeleton of the researcher's model of experimental

programming/problem-solving and the basis of the diagramming method employed here. Clement's (1977) study of mathematical reasoning provided insight into the complex interaction of plans with active, semi-active and latent theories and misconceptions. Soloway, Bonar and Ehrlich (1981) provided a frame model for plans and the notion of misconceptions arising from natural problem-solving strategies. The follow-up work of Bonar & Soloway (1985) (informed by a much earlier theory by Brown & VanLehn (1980)) brings with it the notion that "buggy" conceptions may be generated by gaps in knowledge and includes specific observations of programming misconceptions such as the notion of a "language confound" (a misconception resulting from the misapplication of linguistic knowledge). We borrow from Hoc's (1977) study of device learning the assumption that physical activity during problem-solving (drawing flow charts or code on scratch paper, speech or keyboarding) generally indicates concepts that are under active development rather than those already internalized, and also an alternative conceptual framework for understanding how programming experience leads eventually to the building of "runnable mental models."

Constraints on hypotheses generated in a cognitive microanalysis lie in their internal simplicity, consistency and ability to believably explain the protocol (Easley, 1979). Assertions made in any descriptive study must explain the data in question, though they need not be

approach (Lin, 1979). This approach is clearly inferential and cannot provide absolute proof of any aspect of our hypothesized model of cognitive activity. Rather, the reader must bring to this study a willingness to hypothesize with the writer on possible explanations for subjects' behavior. The reader ultimately decides whether the analysis is worthwhile, based on its overall plausibility in explaining the data. This seems a reasonable first approach to the study of internal processes which have not been well documented in the past, and which are in any case far from transparent, even to the problem solver herself. The approach is "naturalistic" in that it attempts to minimize the effects of observing on the behavior of subjects.

Limitations

One general limitation of this study has already been mentioned: that the nature of the analysis forbids any claim of objective proof of cognitive structures inferred from protocols. This is in part a result of the nature of the activity: one cannot study thought processes directly, but only through observable behavior. However, it must be noted that the researcher's choice of a descriptive rather than a quantitative approach necessarily complicates this issue. A controlled study, while more limited in scope,

assures a level of objective proof impossible with a more subjective design.

Another limitation is the lack of generalizability to other subjects. The presence of conceptions or misconceptions in one subject cannot be construed to indicate their presence elsewhere. The goal of this study is akin to the goals of field studies in anthropology and ethology: to construct plausible hypotheses concerning cognitive structures -- hypotheses which are informed by careful observations from case studies. What the researcher hopes to establish here is an initial set of observations and hypotheses that may be used as a basis for more quantitative follow-up research. Chapter 5, a discussion of the results of this study, suggests a number of areas that merit further study.

As mentioned earlier, a premium was placed on maintaining a natural and non-obtrusive environment during interviews. Although it is the assumption of this study that the observed behavior is closely akin to behavior that would have occurred had the experimenter not been present, it is important to acknowledge that this assumption is open to question. Any observation has the potential to introduce a bias. The generalizations and conclusions drawn from these data may be subject to the same bias; i.e., the behavior observed in these protocols may differ significantly from unobserved behavior.

Subject selection must also be acknowledged as a limiting factor. Subjects, outside of the three adult experts, were all of High School age, were generally of a modest programming background and were drawn from three different educational programs. Results may not generalize to other age groups, other computer languages, or to other stages of Logo learning. The instructional approaches of these three educational programs were not coordinated in any way, although they did share some similarities. In any case, instructional approach must be thought of as an uncontrolled variable in this study.

Two different computers, the Acorn and the Apple (both Apple IIe and Apple II+ models) were used in interviews. Two versions of Logo (Terrapin and Apple Logo) were used on the Apples, and the Acorn had its own version of Logo (but which was very similar to Apple Logo). Two versions of each problem were prepared to partially compensate for these differences, but in some cases the differences could not be avoided. These cases are noted in discussions of the interviews in Chapter 5.

Finally, certain environmental limitations must be noted. Video taping was momentarily interrupted on some occasions by technical problems or by the other people entering the room where interviews were held. Also, most interviews were conducted in two sessions, separated by between four hours and three days.

Subjects

Subjects for this study were drawn from two distinct groups: adult experts and high school students of varying abilities.

The first group consisted primarily of seven high school age students having over 50 but less than 120 classroom hours in an introductory Logo programming course, accrued within six months of the study. The majority of this group was composed of five female high school students selected at random from among the twenty volunteers, all first year participants in a six week Summer mathematics and computer programming experience called "Summermath", held at Mt. Holyoke College in South Hadley, Massachusetts during June and July, 1986. Two additional students had completed a one semester Logo programming class conducted at Lincoln-Sudbury Regional High School, an upper-middle class high school located in a suburb of Boston, Massachusetts, during September and October, 1986. These two were randomly selected from among six students who volunteered for this study. None of these students had any previous experience with Logo. This combined group of seven students will be identified throughout this study as near-novice programmers.

Two high school students who were junior instructors of Logo at New England Computer Camp, a computer camp

composed of about 125 mostly upper-middle class children ranging in age from eight to eighteen, were identified as having advanced Logo programming ability. Both of these individuals had approximately thirty hours of formal Logo instruction, but also had over 100 hours of experience in other programming courses, and claimed over 200 hours of independent programming experience, mostly in BASIC and Pascal. They clearly constituted a special category, since much of their demonstrated Logo skill seemed to derive from their experience with other languages. This second group will be designated as near-expert programmers, and these interviews were analyzed with special attention to the role that their knowledge of other languages played in their conception of variables in Logo.

Three adult experts were chosen from the Logo programming and teaching community based on a recognized mastery of the Logo language and of Logo programming techniques. All three have authored published articles about Logo learning or programming practice. The adult experts were studied to provide a model of expert Logo problem-solving and variable conceptualization.

Background information was collected for all nonexpert subjects. This included chronological age, school grade, prior computer experience and instruction, (including instruction in Logo prior to their current class). This data is summarized in Table 1.

Table 1
Summary of Subjects' Backgrounds

	<u>Age</u> <u>Yrs/Mos</u>	<u>Previous</u> <u>Logo</u> <u>Experience</u>	<u>Other</u> <u>Programming</u> <u>Experience</u>	<u>Other</u> <u>Computer</u> <u>Experience</u>
Near-novices:				
Summermath:				
E	15/2	None	None	Spelling & Educational games: 3 hrs.
N	15/6	None	BASIC: 41 hrs. instruction, 41 hrs. programming	Appleworks 41 hrs. instruction , 41 hrs. keyboard
M	17/0	None	None	Drawing pictures with MacPaint.
A	17/4	Approx. 15 hrs.; used occasionally in Geometry class;	BASIC: 52 hrs. instruction 52 hrs. programming Pascal: 75 hrs. Instr. 50 hrs. programming	Accounting software SAT preparation Games Print Shop Word Processing
L	16/11	None	BASIC: 20 hrs. Instr. 10 hrs. programming	"A lot of computer games" (No specific data given)
Lincoln-Sudbury:				
O	15/4	5 hrs. In 8th grade; 30 hrs. in 6th grade	BASIC: 120 hrs. instruction, 60 hrs. programming	
J	No Data Available			
Near-experts:				
L	16/9	9 hrs. In- struction, 6 hrs. programming	Assembly: 54 hrs. BASIC: 36 hrs. Instr. 100 hrs. programming C: 18 hrs. FORTRAN: 40 hrs. Forth: 5 hrs. Lisp: 5 hrs. Pascal: 198 hrs. Instr. 18 hrs. programming	Varied and extensive- no specific data collected
R	17/2	9 hrs. In- struction, 6 hrs. programming	Assembly: 18 hrs. BASIC: 109 hrs. Instr. 300 hrs. programming Pascal: 228 hrs. Instr. 520 hrs. programming	Varied and extensive- no specific data collected

For the near-novices, information was informally gathered about the instructional approach used in their current Logo programming classes.

Subjects from the SummerMath had completed their fifth week of a Logo program that emphasized mathematical discovery with graphics. Each subject spent ten hours per week in combined Logo instruction and programming for a total of about fifty hours. Instruction was done with printed worksheets supplemented by class presentations and individual help. Variables were introduced as local variables in procedures to draw geometric shapes. Text commands, including PRINT, FIRST and LAST, were briefly introduced in the worksheets. REPEAT was introduced in the context of the repetition of graphic commands. Global variables were presented briefly in an exercise in the worksheets but not emphasized. Cartesian commands (SETPOS, XCOR, YCOR) were introduced in a worksheet exercise as well.

The course at Lincoln-Sudbury had lasted twelve weeks, meeting fifty minutes per day, five days a week for a total of fifty hours. This course emphasized text commands and list processing, including PRINT, FIRST, LAST, BUTFIRST, BUTLAST, LIST, SENTENCE, FPUT and LPUT. The OUTPUT command was introduced in the first two weeks of instruction and emphasized throughout the class, leading to the idea of a function that recursively traverses a word or list.

Graphic commands were taught briefly, as were commands relating to Cartesian coordinates, the latter in a worksheet on graphing.

Method

The focus of this study was on how conceptions and misconceptions of variables influence the action-plans and experiments of high school students at an intermediate stage of learning the Logo computer language. The two operational goals were: (1) generalize from observations of the interaction between concepts and misconceptions and subjects' action-plans and experimental activities; (2) identify Logo programming errors involving variables and uncover those misconceptions that can plausibly be seen as causing them.

Programming problems were chosen that required each of four common classes of variable use:

- global variables: these are simple variables, accessible inside or outside of a procedure. Global variables are associated with the workspace as a whole and can be reported with a Logo system command such as PO NAMES.
- local variables: These are variables used as labels for function parameters.

- variables whose values are explicitly requested at run-time: These are global variables initiated in the midst of a procedure with a command line such as:

MAKE "NAME READLIST,

where READLIST pauses to accept input from the program user. (In other versions of Logo, substitute REQUEST for READLIST).

- functions: A function is the only way to represent a covariate relation in a non-relational language such as Logo. For example, a relation like $Y=2X$ might be converted into a function, F, taking as input an instantaneous value for X and outputting the Y-value for that input. Creating such a function in Logo requires not only the use of a local variable but also an OUTPUT command to provide an explicit result to the function.

Nine problems were chosen to represent each of these classes at varying levels of complexity (See Figure 2; problems are listed in their entirety in Appendix B).

Initially, subjects were shown an 18 minute instructional videotape, which explicitly presented all of the commands and concepts needed to solve these problems.

At its conclusion, nonexpert subjects were invited to ask the interviewer any questions about the instructional videotape (experts were asked for critical comments), and an instructional script containing the text of that presentation was made available to each problem-solver to use at will for the remainder of the session (see Appendix A). * Subjects were then videotaped while attempting to solve the problems, which were presented to them in a random order to control for any effects arising from the order of presentation.

In preliminary sessions, subjects were given all nine problems, but five of the complex problems were found to be inappropriate for intermediate programmers and were given only to experts in the later stages of data-collection. This left a core of four simple problems, Problems E-2, D, A-2 and B-2, one for each class of variable-use (see Table 2).

 * With this design, a subject who fully utilized the initial instruction had the essential building blocks with which to construct a problem solution. In this sense, this was a study of the efficient utilization of available information. Our prediction was that the subjects' need to mentally refer to the instructional presentation or consult the script at appropriate times reflected in large part a lack of "readiness" to integrate the concept in question; i.e., that the concept was for some reason inaccessible to the subject.

Table 2
Classification of Problems

	<u>Simple:</u> The problem requires only the concept explicitly named.	<u>Complex:</u> The problem requires more planning and/or the additional concepts.
Global variable:	Problem E-2 ("Create a variable called NUMBER, such that "PRINT :NUMBER" prints out the number, 7...").	Problem E ("Write a procedure called COUNTER that prints out how many times it has been used..."). {local/global distinction}
		Problem F ("...Write a procedure called WAGE, that takes one input... (and) print(s) out that person's salary..."). {misdirection}
Local variable:	Problem D ("Write a procedure called MOVE, that takes...as Input...an X and a Y coordinate").	
Explicit Input:	Problem A-2 ("Write a procedure that first prints the message, 'GIVE ME A NUMBER' ...").	Problem A ("Write a procedure that...reads in integers until it reads... 99999"). {iteration or recursion}
Function:	Problem B-2 ("Write a procedure called R100 that outputs a random number from 1 to 100 ...").	Problem B ("Write a procedure that computes the factorial of a number..."). {recursion}
		Problem C ("Write a procedure that...points the turtle to a new heading, one half of (its) starting heading"). {system "reporters" and the idea of heading}

During problem-solving, the interviewer acted as a listener in the context of clinical interviewing (see Whimby & Lochhead, 1981), asking the subjects to vocalize their reasoning and helping them check their work for superficial oversights and mistakes in the execution of a stated plan. For example, the interviewer would offer suggestions for the correction of typing errors, as long as these suggestions did not unduly influence subjects' overall problem-solving strategy. At times counter-examples would be posed, in order to probe into the programmer's conceptualization of the problem.

When, in the judgement of the interviewer, a subject reached an impasse, that subject was offered explicit instruction on one of the more superficial aspects of a complex problem. This will be referred to as a "teaching probe". *

 *For example, in an early interview, a subject's confusion over the use of the colon led her to attempt to type

RANDOM :100

In Logo, when a colon prefixes a name, the expression refers to the value named, a concept sometimes referred to as misdirection. While misdirection is an important aspect of the notion of a variable, it was not the focus of this particular problem. After asking the subject to explain what she had typed in order to document the error and to gain some insight into what caused it, a teaching probe was used to refocus the subject's attention to the problem at hand. The subject was simply told that the colon was not appropriate in this situation, a statement which she accepted at face value. This allowed her to continue work on more critical aspects of the problem at hand. In this case, the subject's difficulty seemed to be more one of correct notation and the meaning of the colon than a conceptual difficulty, such as difficulty distinguishing a variable from its value.

If such a probe was helpful, it would tend to suggest that the subject's initial difficulty was due only to a lack of factual information, while a misconception which shows resilience to explicit instruction suggests a deeper problem, either deficiency in prerequisite skills or concepts or a general deep-seated misconception of the problem. It should be noted that these teaching probes were not hints designed to suggest the "right answer", but rather means to discover possible underlying causes of an impasse, after it had been sufficiently documented.

Analysis

The analysis stage included two general tasks:

- (1) Identify and classify conceptions and misconceptions of variables. This required first the location of Logo programming errors or examples of correct variable use and then the examination of such cases in the overall context of each subject's problem solving in order to identify those conceptions and misconceptions which can plausibly be said to dominate problem-solving.
- (2) Propose an overall model of the interaction of plans, concepts and experimental activity.

As a preliminary step in this analysis, the researcher made transcriptions of selected protocols and noted the following kinds of subject behavior:

Nonverbal BehaviorVerbalizations

- | | |
|--|-----------------------|
| -keyboard activity | -statements of belief |
| -reading from the script | -answers to questions |
| -pointing/hand gestures | -questions |
| -eye gaze | -exclamations |
| -facial expression | -"stalling" sounds |
| -experiments (use of the computer
to obtain an observable result) | -talking to oneself |
| -silence/pauses in activity | -irrelevant chatter |

The task then was to propose first order models of the high level cognitive structures that most plausibly explain subjects' physical behavior and verbalizations. For the next stage of analysis, one particular problem (A-2) was selected for detailed examination based on the interestingness of protocols associated with it. All three expert solutions of this problem, along with three particularly interesting student protocols, were selected and subjected to a cognitive microanalysis.

Several procedural recommendations by Hoc (1977) facilitated this process. Hoc's assumption that the majority of internal activity and observable behavior are dedicated to aspects of a task that are not completely assimilated was incorporated into this study. Keyboard activity, pointing, eye gaze and questions for the

interviewer were thus viewed as indicating subjects' focus and areas which they had not yet internalized. Also, pauses in activity and gaps in verbalization were viewed, after the observations of Hoc (1977) and Smith & Inhelder (1975), as indications of more intense mental activity.

Verbal exclamations and facial expressions were carefully noted, in an attempt to isolate potentially significant aspects of the affective domain. One of the purposes of this was to identify intuitive leaps or insights. (Such insights, dubbed the Aha! phenomenon by Martin Gardner, are seen by him as the most critical aspect of the problem solving process (Gardner, 1978)). Another important feature of affect is a subject's confidence and conviction or conversely her frustration or level of fatigue.

Early attempts at analysis verified the usefulness of the following two-way classification of high-level cognitive structure, derived from both Anderson (1984) and Smith & Inhelder (1975):

- Plans correspond to Anderson's "procedural knowledge" and to Smith & Inhelder's "goals".
- These are the high-level procedures which guide subjects' goal-setting and so drive both physical and mental activity. Plans are action-oriented. At their simplest level, they detail behavior to be carried out in steps by an individual; each of

these steps can be thought of as an action-goal. On a more complex level, plans can invoke other plans, a process analogous to one procedure calling a subprocedure. During early attempts at analysis it was determined that most actions can be linked to a single goal derived from a higher level plan, which is in keeping with the findings of Smith & Inhelder (1975) and Anderson (1984). The term current goal will be used to designate a plan-element that seems to be the primary focus of subject behavior at a particular point in time.

-Concepts (or misconceptions) are reflected in action as predictive theories or assumptions about a given domain. While plans are procedural in nature and can be recursively decomposed into sub-goals, concepts and theories are descriptive, better thought of as statements of fact or belief than as action-goals or procedures, and seem to be simple and indivisible (Anderson, 1984). They can be likened to the mental models described by Collins & Gentner (1981) and the conceptual devices of Hoc (1977).

Each of these cognitive structures can be reduced to a single instantiation. Some plans or concepts are verbalized directly by subjects, but others can only be

induced from subjects' actions. For example, much problem-solving activity on the computer is based upon experimental action, as in the classic study of children's block balancing (Smith & Inhelder, 1975). Early attempts at analysis indicated that experiments are usually directed either toward the current goal or toward a particular theory. Any keyboard activity will be viewed as such an experiment in this study. While programming errors are sometimes viewed as "failed" experiments, this is not always the case. Some failed experiments are misinterpreted as successful by inexperienced programmers concentrating on the current goal. At other times, unexpected experimental results may cause subjects to dramatically shift their attention from achieving the current goal to exploring a theoretical prediction that had previously been taken for granted (Smith & Inhelder, 1975). Utilizing the Smith & Inhelder methodology, this study classifies all experiments as either goal-oriented or theory-oriented, and makes note of any shift from goal to theory.

Theories that are clearly false, i.e., inconsistent with what they intend to model, can be termed "misconceptions". In some cases, subjects verbalized a mistaken theory that directly resulted in a programming error. Such misconceptions were, of course, easy to identify. Other errors seemed to be the result of subjects choosing an inappropriate plan, either because they

misclassified the problem or because the solution required a mastery of concepts that the subject did not possess. In the latter case, the researcher identified that misconception inferred to be most responsible for the error. During the attempt to simplify and successively refine the analysis of the selected protocols, misconceptions that were not initially obvious often emerged as likely causes for observed programming errors (Clement, 1977).

The plans, theories and activities of these six target protocols were first diagrammed, and simultaneously a "schematic" was created as a summary of key parts of each interview (after Anderson, 1984).

Schematic Diagrams

All symbols used in schematic diagrams, as discussed below, are shown in Figure 1.

The basic skeleton of these diagrams was a hierarchical tree structure of plans, diagrammed in rectangular boxes, placed along a horizontal time-scale that identified the point at which each plan seemed to become active. Plans are procedural in nature, and the normal progression was from a general plan to a more specific one. This refinement is referred to as a mapping of the general plan to the more specific one. Such a mapping was normally diagrammed with a plain line

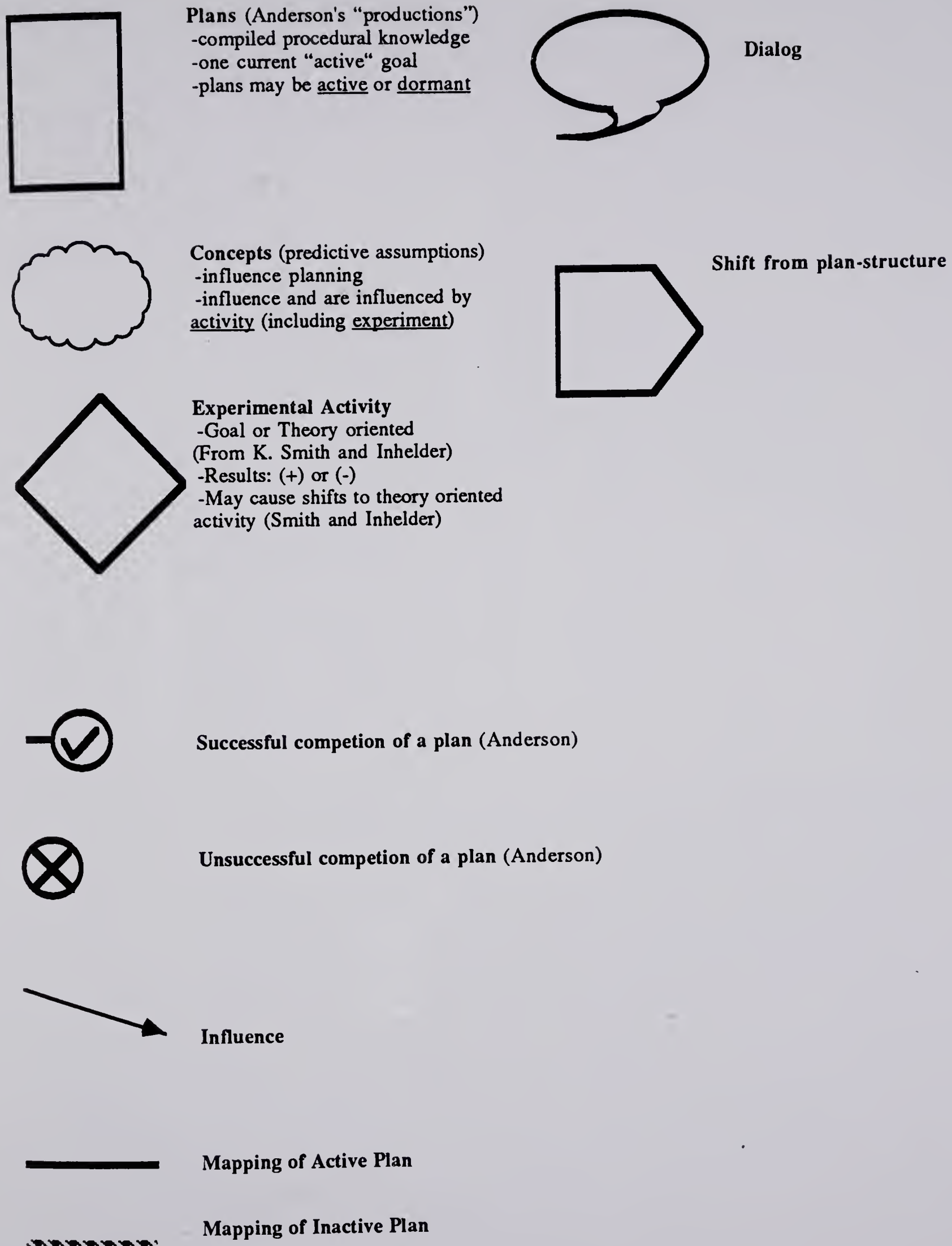


Figure 1
Symbols Used in Diagramming

connecting two plans. While these mappings should be thought of as one-way, the lines that represent them had no indication of direction; the reader may determine direction from the orientation of the chart, with mapping proceeding from earlier, more general plans to later, more specific plans; from left to right on the page. Thus plans and the connections between them form a hierarchical tree-structure. When a specific plan is successfully executed by a subject, a branch of this tree ends, and attention normally shifts to the next step in a more general plan (beginning the mapping of a new branch. The successful mapping of a plan to action is shown with a check mark. When the mapping fails, planning will shift unpredictably. An unsuccessful termination of a plan is shown with an "X". Everything described to this point is derived largely from Anderson's technique for diagramming what he calls "procedural knowledge".

Sometimes, a plan ceased to exert active influence over a subject's behavior. This would then be described as an inactive plan, and the lines showing mapping from that plan shifted to dotted lines to graphically portray this (sometimes temporary) desertion of plan.

Occasionally a plan may exert influence upon an independent branch of the plan-tree. Influence exerted upon any construct from another is shown as an arrow (i.e.,

a line terminating in an arrowhead), the arrow showing the direction of influence.

An important addition to Anderson's technique is the "thought balloon" which represents a concept, a predictive assumption about some aspect of the problem domain. A concept may be discussed directly by the subject, or strongly implied by her action, or implied by other aspects of subject behavior. Concepts can influence plans (or other structures) and plans can influence concepts. As before, such influence is shown with arrows that show the direction of influence. However, conceptual knowledge is seen as different in nature from the procedural knowledge represented as plans and is normally shown outside of the actual plan tree; i.e., lines of mapping will normally not lead to or from concept balloons.

Three other symbols are commonly used. A diamond represents an experiment. Experiments following the normal course of planning, i.e. from left-to-right and top-to-bottom, were diagrammed as part of the plan tree. Following Smith & Inhelder, experiments were classified as either goal-response or theory-response. In the first case, an antecedent plan normally served as the goal being tested. In the second, the concept showing the strongest influence over the experiment was usually named as the relevant theory.

Smith & Inhelder have observed in their subjects an occasional shift in experimental activity from goal-oriented to theory-oriented behavior. Such a shift, when detected, was represented as "Home-plate" symbol. This symbol, used in flow-charting to denote the start of a process, sits at the head of the new planning-tree, generated by the shift. Such shifts were sometimes associated with "ahas", suggesting a connection between these two ideas.

At times it was convenient to include bits of dialog on the diagram. Such dialog was shown with a dialog-bubble. Dialog can influence and be influenced by any other construct, though it is not normally a part of the planning tree-structure itself.

Further Analysis

After a concentrated analysis of these six solutions to Problem A-2, the remaining student-solutions to the same problem were examined. The plan-structures of these protocols were informally analyzed, with special attention paid to variable usage and related misconceptions.

Finally, protocols for all other problems were surveyed for examples of variable misconceptions. All variable misconceptions were catalogued into types. In classifying variable misconceptions, one must necessarily speculate on the probable causes of observed errors. For

example, a given error may be related to some previous instruction or to other previous experience such as other programming experience or natural language problem-solving. On the other hand, misconceptions may reflect inconsistencies or confusing aspects of the computer language or an incomplete understanding of some of its complexities, or may reflect a subject's general conceptual difficulties. Additionally, several causes may interact with one another in complex ways. Only rarely could subjects' direct reference to the causes of their errors be found, making classification a nontrivial judgement. Accordingly, a loose classification was developed for what the researcher judged to be the dominant cause of errors, leading to further speculation on what caused misconceptions and how they might have been avoided.

Other interesting comments or behaviors by subjects that yielded important insights into the learning process were reported and discussed in an attempt to discover what may be fruitful areas for further study. Included were observations that tend to verify, repudiate or elucidate theories or suggested approaches from the literature cited in Chapter 2, with the aim of developing guidelines to help teachers and researchers identify theories and representations of programming and variable conceptualization that show the greatest promise for practical application. The hope is that these

comments may help to illuminate thinking on the more difficult problems of understanding why misconceptions occur and of finding ways to help students overcome them.

Footnotes
Chapter 3

Smith, A. & Inhelder, B. "If You Want to Get Ahead, Get a Theory", Cognition, 3, 195-212, 1975.

Anderson, J., Farrell, R., Sauers, R. "Learning to Program in Lisp", Cognitive Science, 8, 87-129, 1984.

Clement, J. "Quantitative Problem Solving Processes in Children", Doctoral dissertation, University of Massachusetts, 1977.

Soloway, E., Bonar, J. & Ehrlich, K. "Cognitive Strategies and Looping Constructs: An Empirical Study", Technical Report, Yale University, 1981.

Bonar, J. & Soloway, E. "Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers", Human-Computer Interaction, Fall, 1985.

Brown, J.S. & VanLehn, K. "Towards a Generative Theory of 'Bugs'", Cognitive and Instructional Series # 2, Xerox Palo Alto Research Center, 1979.

Hoc, J.M. "Developmental stages in learning to program" International Journal of Man-Machine Studies, 9, 87-105, 1977.

Easley, J.A. "The Structured Paradigm in Protocol Analysis", Cognitive Process Instruction: Research on Teaching Thinking Skills, (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.

Lin, H., "Approaches to Clinical Research in Cognitive Process Instruction", Cognitive Process Instruction: Research on Teaching Thinking Skills, (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.

Whimby, A. & Lochhead, J. Problem Solving and Comprehension: A Short Course in Analytical Reasoning, Franklin Institute Press, 1981.

Gardner, M. Aha! Insight Scientific American Inc., 1978.

C H A P T E R 4

RESULTS AND ANALYSIS

This chapter reports on results obtained in the following areas:

1. The microanalysis of the work of three adult experts and three selected High-school age subjects on Problem A-2, the problem chosen for detailed analysis. This included a full transcript for each protocol as appendices. Analysis for each protocol is composed of a summary of the more important plan-elements and concepts, called a "schematic" (after Anderson et al, 1984), and a related diagram for each protocol.
2. A collection of the concepts and misconceptions from the above microanalysis, in list form.
3. A summary of the remaining solutions to Problem A-2.
4. Highlights of other interesting protocols, including examples of both interesting variable misconceptions and moments of insight.

Microanalysis

Expert Protocols on Problem A-2

Protocol 1: H; Problem A-2: (Note: In order to protect their anonymity, abbreviations are used to identify subjects. As a matter of style, such abbreviations will not be punctuated by a period).

This protocol is summarized in Table 3. Figure 2 gives the microanalysis of H's solution, in diagramatic form. (The full transcript of this protocol is included as Appendix C.) Several aspects of the protocol merit further comment.

Based on the work of Anderson et al (1984), expert programmers were expected to begin by mapping a problem to a general plan and then mapping, in order, each step of the general plan to more specific plans, eventually coding each plan into Logo code. However, beginning at 00:05 and continuing throughout the protocol, H's overall approach showed a different pattern. He did not begin with an overarching, general plan. Rather, his approach was to carefully read through the problem from start to finish, stopping to code each problem element in order, and carefully proofreading each block of code before moving on to the next step of the problem. While the use of such superficial features as the wording of a problem have been observed as a strong factor in the work of novice programmers, it has not been previously reported in expert programmers (Adelson, 1981; Soloway et al, 1982). The assumption was that H was using a general-knowledge plan to Map the problem to a sequence of steps. The proposed plan contains three parts, applied to each element of the written problem in turn: I. Read a problem element, II. Code the problem element III. Proofread that code. A fourth part, IV. Test the entire procedure, is applied once

Table 3
Schematic; H, Problem A-2

- 00:05 - Map the written problem onto a plan, MAP THE PROBLEM AS A SEQUENCE OF STEPS, a two-step plan: I. READ, CODE & PROOFREAD EACH PROBLEM-ELEMENT, II. TEST ENTIRE PROCEDURE
- 00:27 - Maps "Write a procedure that first prints the message, GIVE ME A NUMBER..." from the written problem onto a WRITE A PROCEDURE THAT PRINTS A LIST plan and codes that plan directly as:
 TO GLUB
 PRINT [GIVE ME A NUMBER]
 Upon proofreading, Ph proceeds to next problem-element
- 00:47 - Maps "...and prints, THE NUMBER SQUARED IS...", followed by the square of the number supplied by the person using the program", from the written problem, onto a two step plan: 1. READ USER INPUT INTO A VARIABLE, (0:54: "...I want to pick it off the keyboard"), and 2. REPORT THE SQUARE OF THE VARIABLE.
 - Immediately maps #1 onto the template, "MAKE (NAME) RL" and codes that plan directly.
- 01:04 - Refines 2. REPORT THE SQUARE OF THE VARIABLE into a plan, PRINT A LIST AND THE SQUARE OF THE VARIABLE BY MERGING THEM INTO A SINGLE LIST.
- 01:12 - Maps ...A LIST AND THE SQUARE OF THE VARIABLE... (above) to ...A LIST AND AN OPERATION...
- 01:36 - Maps ...AN OPERATION... to a plan to WRITE-AN-OPERATION, and directly codes as:
 SQ :N
 OP :N * :N
 END
- 01:56 - Proofreading-lines-2-and-3 shifts attention to a previous problem-element, (MAKE (name) RL).
- 02:06 - Refines "READ USER INPUT INTO A VARIABLE to READ USER INPUT INTO A VARIABLE, AS A WORD, and immediately codes as:
 MAKE "NUM FIRST RL
- 02:15 - Begins to PROOFREAD WHOLE PROCEDURE.
- 02:25 - Experiment #1. (Goal:CHECK WHOLE PROCEDURE), with positive result (GOAL +).
- 02:31 - Experiment (GOAL:CHECK WHOLE PROCEDURE), with positive result (GOAL +).
- 02:38 - Task completed, with success.

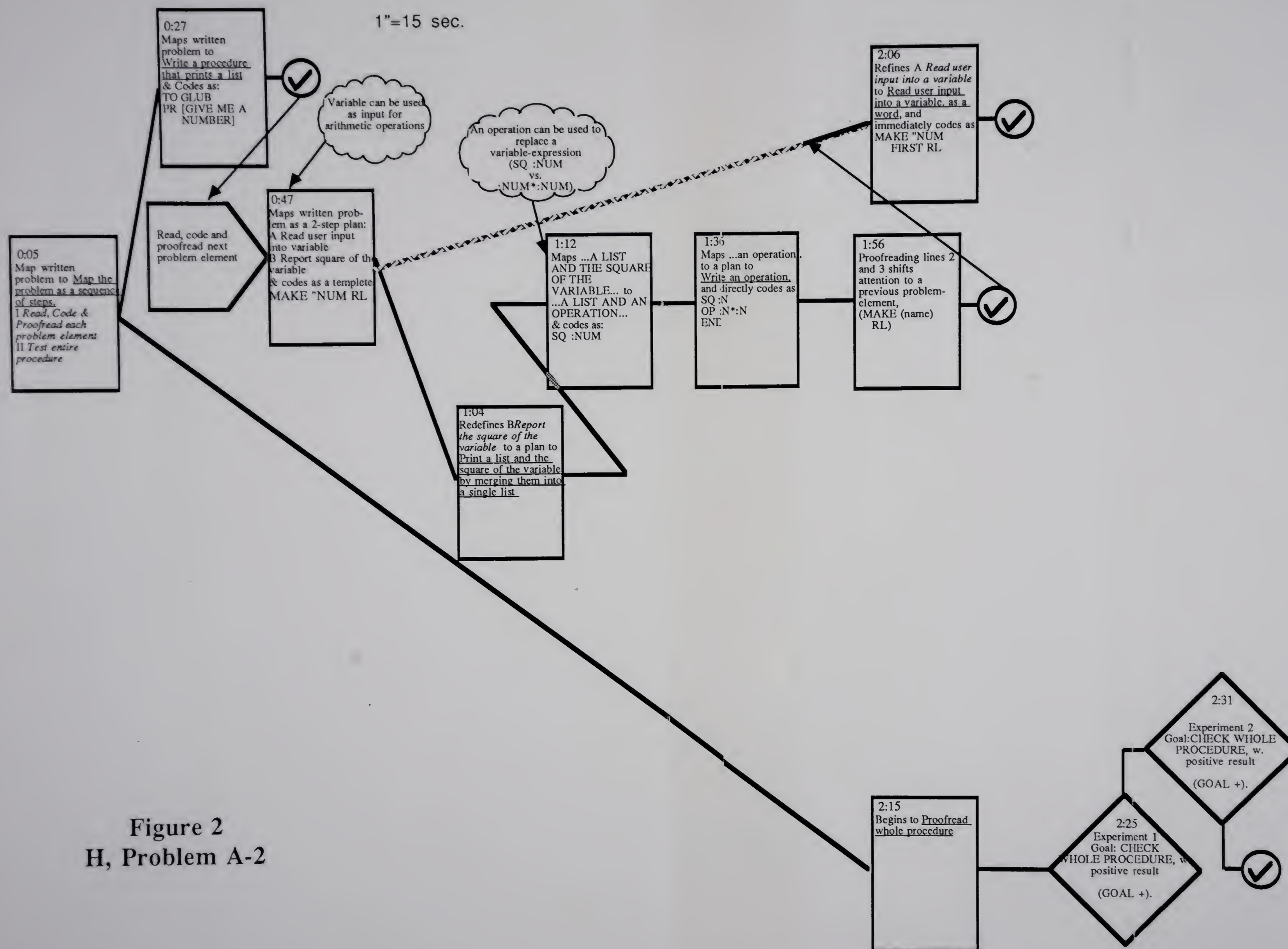


Figure 2
H, Problem A-2

the last problem element is coded. The author assumes that, in his rapid, initial reading, H has determined that this plan is appropriate for this problem, i.e., that problem steps are sufficiently independent to be coded individually. Such a plan and the ability to use it appropriately represents a more pragmatic aspect of programming knowledge. An approach such as this may have functional advantages for the expert. It may serve as a means to minimize errors (mapping problem-elements onto lines of code in an organized way, so as to avoid overlooking any problem-element) and/or minimizing the time required to code a solution by combining a proofreading pass with a coding pass. (Notice that H appears to place a strong emphasis on assuring the accuracy of each step before going on to the next).

Once H settled on this strategy, he proceeded to efficiently map each part of the problem to an appropriate plan and to quickly code each plan into a procedure called "GLUB". He made only one error in coding (at 0:51). Otherwise H seemed to have no difficulty recovering appropriate plans or accessing facts needed to correctly code these plans into Logo commands. In fact, plans and the resulting code seemed to come without much conscious effort, exactly as one using Anderson's model of compiled procedural knowledge would predict.

The most complex part of the procedure is line 3. Notice that H preceded [THE NUMBER SQUARED IS] with SE, showing that he had already determined at that point (1:04) that the PRINT procedure on this line should utilize a list composed of two elements, and that H has mapped this to a plan to use an operation as the second element (at 1:12). H wrote this operation, SQ, and managed to interface the two procedures, GLUB and SQ without difficulty. Notice also that rules of syntax are well internalized for H. He appropriately used quotation marks, spaces and colons without confusion or difficulty. This was one of the most striking distinctions between expert and near-novice protocols.

H's ability to catch a minor error through proofreading alone, as he inserted the previously omitted FIRST before RL in line 2 (t 1:47), provokes speculation as to the nature of expert proofreading. H's careful reading of the problem at this point can be seen as an attempt to stimulate his memory of dormant facts or plans, but the efficiency of H's behavior, i.e., his ability to predict the mis-performance of this segment of code through reading alone, encourages a different reading; that H was, literally, "running" these lines of code through his own, internalized model of Logo. That is, H was stepping through each line of code, predicting the computer's parsing and execution for each step along the way. Such evidence of an expert "playing computer" in this way lends

support to Hoc's idea of "machine learning" as a highly-predictive, functional organization of expert programming knowledge.

H's comments on the name he had chosen for his procedure, GLUB, is a short meta-discussion of procedure-naming. His selection of this particular name was related both to his thinking about procedures and his aesthetic judgement about the problem as well. Besides demonstrating H's understanding that the programmer has the flexibility of choosing a descriptive name, his comments suggest that H makes it a practice to choose titles for procedures that reflect something about their function. The fact that he commented on his choice of name at all suggests that he attends to this aspect of programming in a way that is somewhat surprising. Such examples of metacognition were quite common in expert protocols in comparison to those of either near-novices or near-experts.

Protocol 2: P; Problem A-2: Table 4 is the schematic P's solution to Problem A-2; Figure 3 is a diagram of the microanalysis of that solution. (See Appendix D for a full transcript). P began by keying in on the phrase, "Write a procedure..." (at 0:20) and quickly coded this as the header-line of a procedure, SQUARE. Here, P appeared to focus on one part of the written problem-statement, much as did H. Unlike H, however, he did not maintain a

Table 4
Schematic; P, Problem A-2

- 00:20 - Maps "Write a procedure..." to a general plan to Write-a-procedure. This plan is composed of steps to I. Code-header-line II. Code-remainder-of-procedure and III. Check-the-procedure. (This approach seems related to the Map-the-problem-as-a-sequence-of-steps plan of P.)
- 00:34 - Maps I. (from general plan to Write-a-procedure) onto the template:
TO (proc. name) {Input(s)}
and codes, as:
TO SQUARE (ret.).
- 00:42 - Maps II. Code-the-remainder and the fact that "Its going to ask for a number"
(00:38) to a 3 step Poll-user-for-Input plan, Including: 1. Prompt-user 2. Accept-user-input, 3. Use-inputted-value
- 00:53 - Maps #1 to a plan to Print-a-list, and codes as PRINT [GIVE ME A NUMBER]
- 01:11 - Maps #2 and #3 to two alternative plans, A Four-Line-Program and a Three-Line-Program. The original version of the Three-line-program is never coded, but is later described as operating without creating a name for user input. One possibility is as follows:
TO SQUARE
PRINT [GIVE ME A NUMBER]
PRINT SE [THE NUMBER SQUARED IS] PRINTNUMBER
END

TO PRINTNUMBER
OP RQ (* Itself)- (perhaps OP SQ RQ, where SQ outputs the square of Its Input).
END

The above would be in keeping w. P's comments at 05:08, 05:27 and 05:58.
- 01:32 - He selects the Four line program for this solution.
- 01:38 - Notes RQ as a needed tool in the mapping the four line program
- 01:52 - Maps middle two lines of this four line program (line 1 is already coded as PRINT [GIVE ME A NUMBER], to a line-by-line plan to a. Receive-user-input, b. Print-a-message c. Use-Inputted-value (Note that the fourth line (c. Use-Inputted-value) corresponds exactly to the previous plan for II. Coding-the-remainder, 3. Use-Inputted-value (0:42)).
- 02:01 - Maps a. to two alternative plans: a1. Store-RQ-with-MAKE and a2. Use-a-sub-procedure-on-RQ.
- 02:23 - After extensive consideration beginning 02:01, including some sort of internal debate (2:23), P settles on a2., Use-a-sub-procedure... In retrospect, one consideration is reported as a desire for the greater "robustness" afforded by this choice (07:23, 07:42). (We treat this as a piece of meta-knowledge, growing out of 1. the previous internalization of both plans and 2. Pragmatic concepts of "robustness", "optimization" & "efficiency").
- 02:32 - Maps a2. as I. Call-procedure-with-RQ-as-input and II. Define-procedure-to-use-requested-value, and codes I. as PRINTANSWER RQ.
- 02:49 - Maps II. as a Procedure-taking-number-In-list, a 3-step plan consis-

ting of 1. Code-header 2. Print-something 3. Report-result-using-requested value, and codes 1. Code-header, as:

TO PRINTANSWER :NUMBERINLIST

03:10 - Codes 2. Print-line as PRINT [THE NUMBER SQUARED IS...]

03:17 - Begins to code 3. Report-result-using-requested-value as PRINT..., but pauses in the midst of his coding.

03:39 - After some thought, beginning 3:29, P decides on a revision of steps 1. and 11. of a2. Use-a-sub-procedure-to-get-RQ, specifically to alter 1. to 1.2 Call-procedure-to-get-FIRST-of-RQ and to alter 11. to 11.2 Define-procedure-to-use-FIRST-of-RQ. P later reports this as balancing greater efficiency (in not unnecessarily repeating an operation (08:12, 08:25) against the greater "robustness" of the earlier plan for a2. (07:06-07:59)

03:47 - Codes 1.2 by inserting FIRST before RQ in line 2 of SQUARE.

04:00 - Codes 11.2 by changing :NUMBERINLIST to :NUMBER.

04:12 - Returns to the coding of 3. Report-result-using-requested-value (from interruption at 03:17), producing the line:

PRINT :NUMBER * :NUMBER

04:28 - Executes 11. Check-the-procedure by calling SQUARE and entering 2 as input.

04:39 - Task completed, with success.

1"=10 sec.

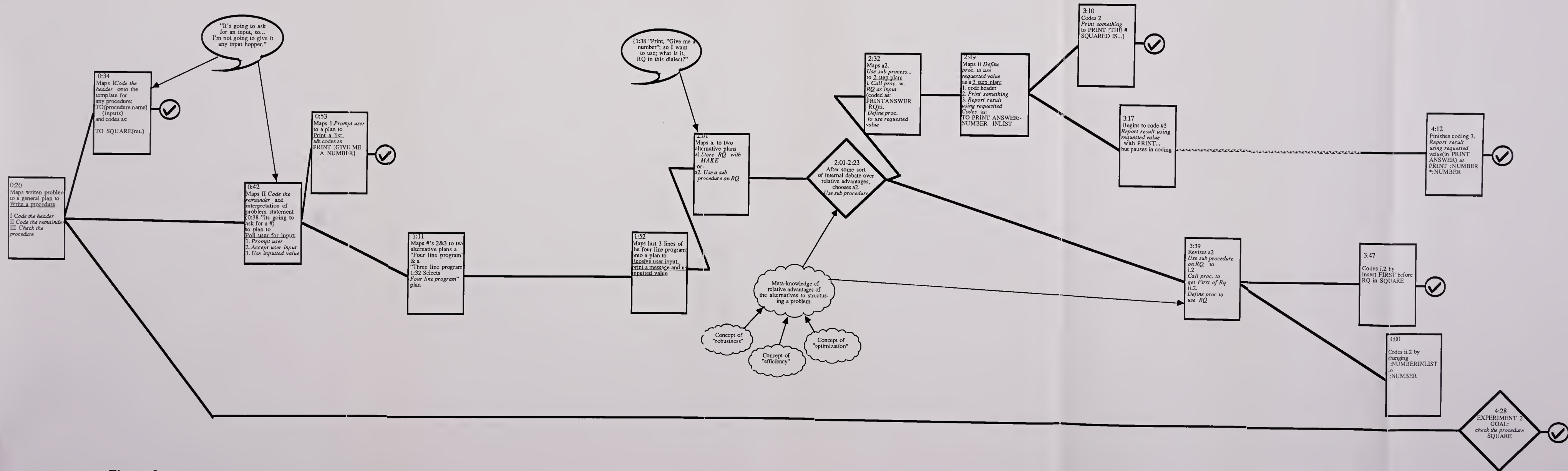


Figure 3
P, Problem A-2

step-by-step approach but proceeded to refine his general plan to a more problem-specific one, shifting his attention (at 0:42) to functional aspects of the problem.

Another notable feature in P's approach was his attention to alternative solutions. First, in refining a loose plan for an interactive input from the user, P developed plans for both a three-line and a four-line version of the solution. After choosing one of these alternatives, P proposed two alternative approaches to storing user-input data. His ability to conceive of and determine the respective advantages of more than one potential solution to a programming problem illustrates that P has deeply internalized each alternative. He seemed able to visualize each solution in some detail, as if the code were already written, behavior markedly different than that of any of the student-programmers. In this analysis, such ability to reflect on virtually any aspect of the problem is viewed as more than descriptive or procedural knowledge, but a higher level of understanding encompassed by the term "meta-programming knowledge".

Much of time expended by P in solving this problem was devoted to several "internal debates", which he described in some detail in retrospect. Hoc (1977) might describe him as running some sort of mental model at this point. However, I see P's interest in the optimization of his solution as key factor. The recognition and manipulation

of alternative plans and discussion of optimization typifies the meta-programming knowledge exhibited by all adult experts of this study. Other features of the dialog that can best be described as a sort of meta-knowledge are P's initial attempt to avoid the "extra baggage" of naming the REQUEST with a MAKE statement (at 5:08), his ongoing consideration of a second, "three-line version" of the procedure and his descriptive naming of the of the sub-procedure PRINTANSWER and of its input, first as :NUMBERINLIST and later as :NUMBER.

Protocol 3: B; Problem A-2: Table 5 is the schematic for this protocol, and Figure 4 is a diagram of the solution. (For a full transcript, see Appendix E).

Notice that B started with a general Write a procedure plan, as did all of the experts who attempted this problem. However, unlike the others she shifted to a more specific plan to Write an interactive procedure before creating the header line of her procedure. While this more closely matches the behavior predicted by Anderson for a programmer with already compiled procedural knowledge, her ordering was less regular, digressing from the consistent top-to-bottom, left-to-right order of plan-execution that Anderson predicts on two occasions (at 1:11 and at 6:25). These "detours" from Anderson's order are interpreted as a means by which B checks critical aspects of her code. This phenomenon appears to be closely related to the "preference

Table 5
Schematic; B, Problem A-2

00:50 - Initially maps the problem onto a general plan to Write-a-procedure:
I. Write-the-procedure, II. Check-the-procedure

00:57 - Influenced by the example, B maps I. Write-the-procedure onto a plan to Write-an-interactive-procedure ("...square, or something..."), which will "...basically request them (the users) to give me an input, which will be the number two". Based on the visual structure of the example (4 lines: computer prompt, user enters number, computer embellishment, square of user input), we assume this to be a 3 step plan: A. Ask-user-for-input, B. Accept-user-input, [C. Use the Input]. (Brackets here denote a part of the plan which is unsupported by direct evidence).

01:05 - Maps the problem to one or more (unspecified) alternate plans (see the extensive discussion starting at 08:49), but continues with the the original plan.

NOTE: B's quick mapping of the problem to alternative plan(s) is diagrammed as an extra branch from the starting node of the diagram. This representation is somewhat inelegant and arbitrary, but allows a reference to the interesting discussion of alternative plans at the end of the protocol and emphasizes the formulation of alternative plans as an important aspect of this protocol.

01:11 - Maps [C. Use-the-input] onto a two-step plan to Use-a-tool-on-the-input: 1. Write-the-tool, 2. Write-the-line-that-uses-the-tool.

NOTE: This differs from Anderson's "natural" top-down, left-to-right order for utilizing procedural knowledge, with which we would expect her to begin work on A. Ask-user-for-input, followed by B. Accept-user-input.

01:22 - Maps 1. Write-the-tool to a frame-type plan to write-an-operation:

```
a. Write-an-operation, (frame)
  TO (name) (variable)
  OUTPUT (expression-with-variable)
  END
and b. Check-the-operation,
...Immediately CODES a. Write-an-operation, as:
  TO SQUARE :NUM
  OUTPUT :NUM * :NUM
  END
```

01:44 - b. Check-the-operation maps to Experiment #1.

```
Goal: Check-the-procedure
  SQUARE 4 (ret.)
  -> RESULT: 16
Result: Goal (+)
```

NOTE: B. Interprets RESULT: 16 as a positive result; she seems in no way distracted by "RESULT:" as an error message, which it technically is, suggesting that she either expected this response to her experiment or treats the "RESULT:" error message as a special case.

Table 5, cont.

01:51 - With the positive result of Experiment 1, B turns to the inactive plan A. Ask-user-for-Input, maps it to a plan to Print-a-list and immediately codes as:

TO SQ.NUM
PRINT [GIVE ME A NUMBER]...

02:32 - Maps B. Accept-user-Input and the example to a plan to Echo-user-input. Presumably the "two" in here question ("...you want me to print the number two there?") serves only as a reference to the "2" in the example, not as a constant).

02:56 - Following a dialog with I, B drops plan to Echo-user-Input.

03:11 - Maps B. Accept-user-Input and 2. Write-the-line-that-uses-the-tool and the example as plan to Print-the-sentence-of-a-phrase-and-the-square-of-the-Input.

03:55 - Codes the plan to Print-the-sentence-of-a-phrase-and-the-square-of-the-Input as:

PRINT SENTENCE [THE NUMBER SQUARED IS:] SQ FIRST RQ

.
.
.

04:08 - (Discussion of an alternative plan for an Interactive-procedure-using-MAKE, informed by the concept of a Distinction-between-user-and-programmer)

.
.
.

06:09 - Experiment #2:

Goal: 11. Check-the-procedure
SQ.NUM

->"THERE IS NO PROCEDURE NAMED SQ IN SQ.NUM...",

Result: Goal (-).

06:12 - Previous experiment quickly stimulates a general diagnostic plan to Find-the-error and Patch-errant-code

06:14 - Find-the-error plan, the error message ("THERE IS NO PROCEDURE SQ..."), and a presumed Set-of-Interpretive-Diagnostic-Principles (including the concept of a variable-as-an-element-of-a-workspace) leads B to Interpret the error as a Call-of-a-non-existent-procedure.

(8:00 - "I'm trying to call a procedure by that name and there is no name...").

The Interpretation is almost instantaneous and relatively automatic for B,...

(7:50 - "...its sort of rote at this point...")

...suggesting some sort of compilation process.

This Interpretation (as the Call-of-a-non-existent-procedure and the plan to Patch-errant-code, maps (immediately) to a matched plan to

Table 5, cont.

Patch-a-call-of-a-non-existing-procedure:

1. Find-errant-procedural-call,

2. Find-Intended-procedure-name

(8:00 - "...so by saying POTS...),

3. Correct-spelling.

(1. Find-errant-procedural-call is achieved Instantaneously by an Immediate examination of the error message).

(2. Find-Intended-procedure-name is nearly Instantaneous, the correct name (SQUARE) being recalled from memory).

06:18 - 3. Correct-spelling maps to a change in procedure SQ.NUM ("SQ" is changed to "SQUARE")

06:25 - Experiment #3

Goal: Verify previous mapping of 2. Find-Intended-procedure-name to SQUARE

POTS

-> a list of titles, including SQUARE, appears on the screen

Result: Goal (+)

06:30 - Experiment #4

Goal: 11.Check-the-procedure

SQ.NUM

->GIVE ME A NUMBER

12

->THE NUMBER SQUARED IS 144

Result: Goal (+)

06:39 - Problem completed w. success

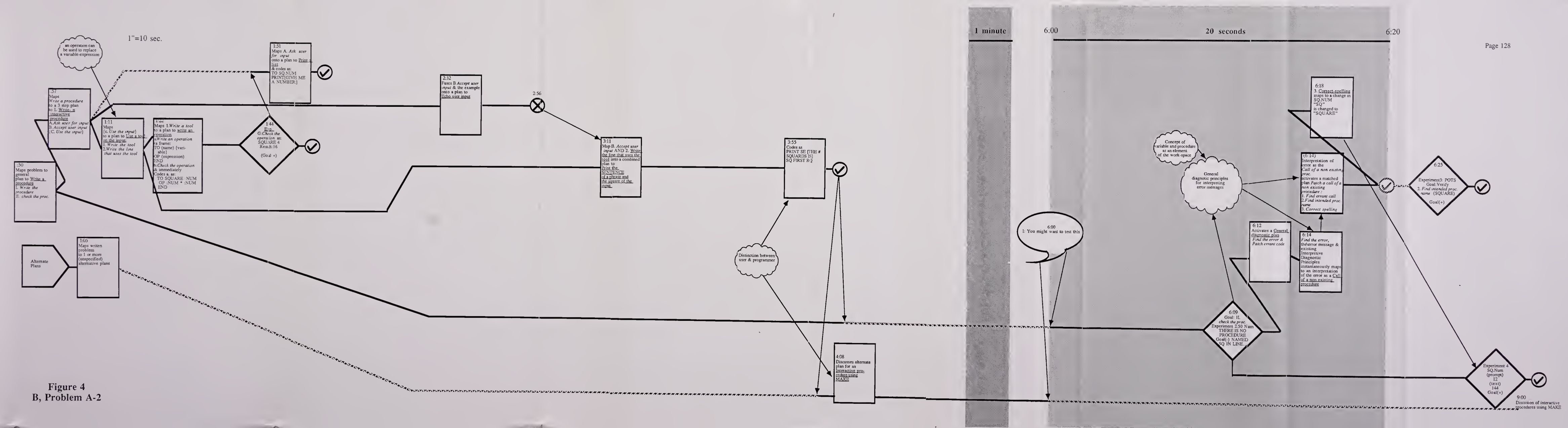


Figure 4
B, Problem A-2

rules" that M. Miller (1982) said were demanded by subjects working in the strict top-to-bottom and left-to-right order initially designed into his Logo planning and debugging environment. Miller believed that preference rules were an attempt by the programmer to optimize her efforts by preventing mistakes in the first programming pass, thus reducing the likelihood of needing a later debugging pass. This is exactly how B's ordering of her work on Problem A-2 is interpreted in this study, as an attempt to save time and effort by selectively expanding upon those aspects of her plan deemed critical. This is classified as another example of expert meta-programming knowledge, in which an expert programmer evaluates her own problem solving.

B's metaphor of the "branches" formed by alternative plans (at 03:19) provide a wonderful and powerful insight into her conceptualization of the problem. For one thing, they demonstrate B's complex value judgements based on subtle aspects of the problem (see comment at 9:00) rather than a static and deterministic mapping of the problem onto internalized procedural knowledge. For another, they indicate that the mapping of a problem to an initial plan, and of goals to sub-goals can be not only one-to-one but may also be a one-to-many mapping. This ability to acknowledge and manipulate multiple alternative solutions to a problem, exhibited both by B and by P, seems to be another aspect of the meta-programming knowledge of experienced and sophisticated programmers, and related to

evaluative concepts that they exhibit, such as optimization, efficiency and elegance in programming.

The instantaneous debugging in response to an experiment (at 6:09) and B's comments about it shed light on the development of the pragmatics of programming, specifically of debugging skill. Two aspects of B's debugging activity seem notable. First, error messages have an explicit meaning for her, evidenced by her clear interpretation of the error message at 8:09 and by her ability to interpret another error message, the outcome of a successful experiment (at 1:44). Second, B's debugging seems both like and unlike simple procedural knowledge. Debugging knowledge does seem to be subject to a compilation-like process (note B's near-instantaneous interpretation at 7:50). However, the process of understanding an error message involves a highly interpretive reading of the error followed by the selection of a plan to patch the error, from the myriad of all possible error patches. Complicating the issue is the fact that error-fixing is by nature an operation, acting upon existing code rather than creative/productive process like the generation of computer code from scratch. In the diagramming of this protocol, this complex debugging process is represented simply, as an arrow, showing the influence of a failed experiment on an inactive plan. This is, necessarily, an oversimplification of the process.

Nonexpert Protocols on Problem A-2

Protocol 4: R; Problem A-2: The schematic and diagram for this protocol can be seen as Table 6 and Figure 5, respectively. (The full transcript is included in Appendix F).

R's discussion of the "...crucial few lines" of the problem (at 1:24), by which he referred to that part of the problem that is eventually coded into line 4 (see 3:43) is a very interesting one in that it shows an aspect of R's problem solving that more closely resembles that of experts than of most high-school aged subjects. Such a comment indicated a sort of evaluatory thinking that seemed to guide the development of his plan to Write an interactive procedure and influence the way R carried out the remainder of the solution.

One striking aspect of this comment about the crucial part of the problem is that it came so early on in the protocol. R seemed to have temporarily deferred consideration of the earlier aspects of the problem until after he identified this critical part, indicating some sort of a pre-processing of the problem to locate clues for efficient solution or "heuristics". The assumption made about this first pass is that R progressed through the problem in a step-wise fashion, considering functional aspects of each problem-element until he found one that

Table 6
Schematic; R, Problem A-2

- 00:50 - Maps the problem onto a 2-part plan to Write-a-procedure:
I Write-the-procedure, I Check-the-procedure.
- 00:52 - Immediately maps I Write-a-procedure to a 4-step plan to Write-an-interactive-procedure:
A. Print-something,
B. Input-something,
C. Print-something,
D. Print-the-square-of-the-stored-input.
Immediately codes I. Write-a-procedure as:
TO SQUARE...
Notice that D. Print-the-square-of-the-stored-input is more detailed and better refined than A, B or C.

In the section that immediately follows (1:01 - 2:45), R refers to each of these steps, leading to the conclusion that, rapid as it is, R's coding reflects this entire plan (presumably in the form of "compiled" procedural knowledge (Anderson's term, 1982)).

He also (at 1:24) describes his thinking in some detail. Of particular interest is his focus on the "...crucial several lines..." that required more careful attention. In discussion, R described a kind of evaluatory thinking which guided the development of his plan to Write-an-interactive-procedure. The author hypothesizes that R is progressing through the problem from start to finish, breaking it into functional steps as a sort of first pass at the problem. Such a process appears quite similar to the diagnostic knowledge of some experts. The main reason for selecting this explanation over a number of alternatives (see below) was R's reference to specific Logo commands that can be associated with the proper coding of the problem, made in the order of the problem itself. This suggested that R was somehow "walking through" the problem.

The author assumes that R is matching each part of the problem with a previously internalized plan (i.e., compiled procedural knowledge). and that a failure to find a match indicates a "harder" or more "crucial" aspect of the problem. Such a first pass at the problem would serve at least two functions: 1. It would allow R to determine which parts of the problem most acutely need his attention and 2. it could serve as a "critic" for alternative approaches to the problem, and help to determine which choice was most efficient. In the diagram, this is shown as a Stepwise-heuristic-evaluation-of-the-problem.

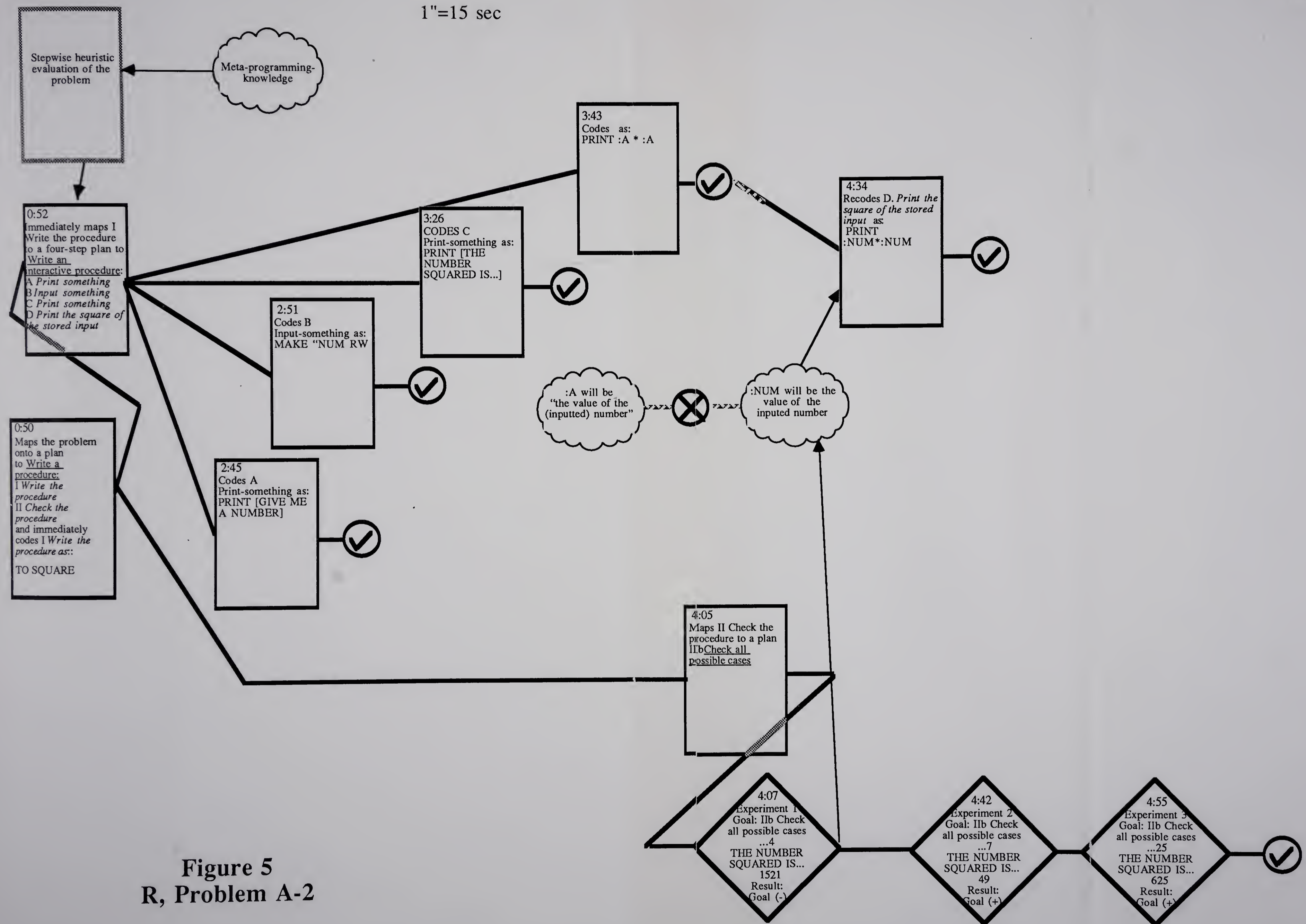
It would be possible to interpret this activity in a number of alternative ways. For example, one could interpret the subject's differentiation between certain parts of the problem as information compiled along with the procedural knowledge of the Write-an-inter-

Table 6, cont.

active-procedure plan. Although it seems unlikely that he has encountered exactly this problem in the past, it might closely enough resemble a previously solved problem to allow such a transfer of knowledge. The author regards this as slightly less plausible than the first explanation, however.

Another alternative would be to treat evaluatory information as a collection of learned facts (i.e. descriptive knowledge). A computational analogy would be a numeric evaluation factor associated with each command, that designates that command as more or less important or difficult or crucial. However, this seems too static a representation for this process describe here.

- 02:45 - R resumes his keyboard activity by quickly coding A. Print-something (with this skeletal plan "fleshed out" with information taken directly from the written problem, (see 2:43 & 2:48)), as:
 PRINT [GIVE ME A NUMBER](return)
- 02:51 - R codes B. Input-something (and the written problem, see 2:51) as:
 MAKE "NUM RW
- 03:26 - Codes C. Print-something as:
 PRINT [THE NUMBER SQUARED IS...]
- 03:43 - Codes D. Print-the-square-of-the-stored-input (which has been refined at an earlier point in the protocol) as:
 PRINT :A*:A
- 04:07 - Maps 11. Check-a-procedure to a plan 11b. Check-all-possible-cases and immediately codes into Experiment #1-
 "SQUARE"
 ->GIVE ME A NUMBER
 "4"
 ->THE NUMBER SQUARED IS...
 ->1521
 (Goal: 11b. Check-all-possible-cases)
 Result: Goal (-)
 R seems to use this result to quickly diagnose his error, (the use of an incorrect variable name in line 4).
- 04:34 - With the understanding of his error derived from Experiment #1, R recodes line 4, PRINT :A*:A to:
 PRINT :NUM * :NUM(return)
- 04:42 - Maps 11b. Check-all-possible-cases to Experiment #2-
 "SQUARE"
 ->GIVE ME A NUMBER
 "7"
 ->THE NUMBER SQUARED IS...
 ->49
 (Goal: 11b. Check-all-possible-cases)
 Result: Goal (0) (i.e. neutral)
- 04:55 - Maps 11b. Check-all-possible-cases to Experiment #3-
 "SQUARE"
 ->GIVE ME A NUMBER
 "25"
 ->THE NUMBER SQUARED IS...



seemed more important. One indication of this is the fact that R failed to refer back to the written problem until 2:43, when he needed more detailed information in order to code line 1 during a second, coding pass through the problem.

It would seem that R's first pass entailed an attempt to match each part of the problem with a previously internalized plan (i.e., compiled procedural knowledge). (Compare this with H's step-wise method in his expert solution). Failure to find a match during such a process would indicate a "harder" or more "crucial" aspect of the problem. Such a first pass at the problem would serve at least two functions: (1) it would allow MA to determine which parts of the problem most acutely needed his attention and (2) it could serve as a "critic" for alternative approaches to the problem, helping to determine which choice was most efficient. In the diagram, this is shown as a Stepwise heuristic evaluation of the problem.

It would be possible to interpret this activity in a number of alternative ways. For example, one could interpret the subject's differentiation between certain parts of the problem as information incorporated in the previously compiled procedural knowledge represented by the Write an interactive procedure plan. Although it seems unlikely that he has encountered exactly this problem in the past, it might closely enough resemble a previously

solved problem to allow such a transfer of knowledge. The main reason for selecting the given explanation over this alternative is that R referred to the coding of noncrucial parts of the problem (at 2:03), in the order of the problem itself, suggesting an active rather than a passive process.

Another alternative would be to treat evaluatory information as a collection of learned facts (i.e., descriptive knowledge). A computational analogy to this would be a numerical evaluation factor associated with each command, designating the relative importance of that command. However, this seems too static a representation to explain the above protocol.

Another interesting feature of this protocol is R's attitude on procedure testing, reflected in his comments at 5:10 and 5:19. These comments suggest that he has internalized certain aspects of program-testing behavior, making it a natural and semi-automatic process. This reminds one strongly of Anderson's notion of knowledge compilation, and would seem to lend support to that researcher's assertion that a cognitive skill (here procedure-verification in the successful problem-solving of a near-expert) can be explained by the compilation of procedural knowledge.

R readily utilizes the error message (at 4:26), resulting from the failure of Experiment 1, to isolate a

programming error and immediately repair it. This suggests, as it did with expert programmers, that R has internalized more than descriptive facts and rote procedures but an understanding of the underlying process of programming, i.e., meta-knowledge of the programming process. Again, other explanations are possible. For example R may have internalized a body of procedural knowledge that directed his search for the cause of the failed experiment. If R had previously committed a similar error, using an incorrect reference to a particular variable, this might predispose him to check variable names for consistency on encountering odd results from a line that uses the variable. However the absence of ready examples of such behavior in near-novice protocols, even when those protocols showed evidence of knowledge compilation, suggests that such knowledge comes later, after much procedural knowledge has already been internalized. This also suggests that debugging may be linked to other meta-programming knowledge.

It should be noted, however, that neither R nor the other near-expert, K, have spent protracted amounts of time either studying or programming in Logo (see Table 1). This suggests to the author that meta-programming-knowledge developed by these two subjects during their extensive programming experience with other languages may in part have been transferred to their work in Logo. In any case,

the assumption is that this is the case with this particular protocol.

Protocol 5: M; Problem A-2 Table 7 is the schematic for M's work on this problem. Figure 6 is a diagram of the microanalysis. (For full transcript, see Appendix G).

One of the most striking features of this protocol is the presence of so much discussion of outputting and the OUTPUT command. This may be a carry over from Problem B-2, M's first problem and the one immediately preceding this one. There, as here, M exhibited contradictory conceptions of OUTPUT that seemed to be competing for dominance.

On the one hand, M held a misconception that allowed her to associate the OUTPUT command with the idea of "screen output". This seems to be a good example of what Bonar calls a "language confound", i.e., a "bug" (a buggy concept or misconception) generated by the over-generalization of one's natural language knowledge (Bonar, 1985). Alternatively, this association may be thought of as reflecting a natural or "correct" approach to the idea of procedural output. In at least one other computer language, APL, the default action for the explicit result of a function is to print it on the current output device. In APL, therefore, there is no equivalent to Logo's PRINT command, only commands to direct functional output and "formatting" commands to reorganize output as a

Table 7
Schematic; M, Problem A-2

- 01:07 - Maps problem onto a two-step plan to Write-a-procedure: I. Write-the-procedure, II. Check-the-procedure, and immediately codes I. Write-the-procedure as: TO NUM(ret.)
- 02:02 - Maps I. Write-the-procedure and the problem statement to a three-step plan to Write-an-interactive-procedure: A. Print-a-prompt, B. Accept-&-report-user-input, C. Report-square-of-user-input, and immediately codes A. Print-a-prompt
- 02:46 - Maps B. Accept-&-report-user-input and two misconceptions (a "language confound": OUTPUT-means-"screen-output" (see also, 3:23 and 3:47) and a misreading of the problem as requiring the procedure to Echo-the-input) to a plan to Simultaneously-accept-&-report-user-input. Her initial attempt is to code this plan with a single command line utilizing the OUTPUT command; she will eventually give up on this approach (see 4:03 & 4:13).

It should be noted that in her work on a previous problem (Problem B-2) M. sometimes exhibited the same language confound as she does here (OUTPUT-means-"screen-output"), while at other times she showed a deep, detailed and correct understanding of the concept of output, sometimes explaining and using the OUTPUT command in a manner that seems contradictory to the way she uses it here. Our belief is that the concept of procedural output is actively under development in this subject, and that part of this concept-refinement process involves resolving ambiguities and contradictory assumptions, leading M. to fluctuate between two rival concepts of OUTPUT.

- 03:23 - Deserts the plan to Simultaneously-accept-&-report-user-input and maps it to a Two-step-plan-to-accept-&-report-user-input:
1. Accept-user-input-into-a-variable and 2. Echo-user-input-using-OUTPUT. M. begins to consider using MAKE to code 1. Accept-user-input-into-a-variable, but does not yet begin actual coding.

In describing her present goals, M. says, "I'm going to be given a number here; i want to put that number in a variable..." This line gives evidence of two important concepts. First, M. makes a Distinction-between-user-and-programmer. "I" is really a reference to the M. as the programmer, distinct from the person who is "...going to (give) a number" (see also, 4:42, 4:47, 4:57). Second, she has internalized the metaphor of a Variable-as-a-container; both her use of the word, "variable", in this context and the use of the preposition, "in", indicates that M. understands this important concept.

- 04:00 - After M. considers coding 2. Echo-user-input-using-OUTPUT command (3:47), she suddenly decides that it is unnecessary for her procedure to echo the inputted value, and abandons 2. Echo-user-input-using-OUTPUT. Note, though, that M. has not necessarily abandoned the belief that OUTPUT-means-"screen-output", and that she later utilizes OP in a fashion almost identical to that which she proposes here (14:32), an error attributable to both the OUTPUT-means-"screen-

Table 7, cont.

- output" misconception and to a degree of "cross-talk" from the Echo-user-input-with-OUTPUT plan developed here.
- 04:13 - M. returns to the plan 1. Accept-user-input-into-a-variable, mapping this plan to an Implementation plan based (apparently) on the example given in the instructional demonstration (MAKE "PLAYER1 REQUEST), Frame-plan-for-an-interactive-variable (see also, 5:44, 6:25, 6:49) and begins coding with: MAKE "X...
- 04:42 - Another concept is revealed in M.'s question, "...how do I make MAKE dots x that number that they just typed in?". The concept is of Variable-as-an-alias (the variable names rather than contains the value in question), and through the rest of this protocol it replaces the Variable-as-container concept. Both are generally recognized as valid and useful ways to represent a variable.
- 04:57 - M. expresses another concept in comparing the still missing section of this idiom to the idea of a parameter input; that the unknown command or commands Transfer-data-from-outside-to-inside-the-procedure (see also 5:09, 5:50).
- 07:35 - After I. supplies a command, REQUEST, that will meet M.'s specifications, M. uses RQ to complete her coding of the Frame-model-for-an-interactive-variable (see also 09:52).
- 10:08 - Following a suggestion by I., M. begins to implement the plan 11. Check-the-procedure by exiting edit mode.
- 10:29 - Experiment #1 - "NUM"
 ->GIVE NUMBER
 "2(ret.)"
 ->?
 (Goal: 11. Check-the-procedure)
 Result: Goal (0; i.e., neutral) (10:37: ...I can't tell from what I've done so far, so I'd better just go on with the program").
- 11:12 - In what the researcher interprets as a shift to the theoretical question and apparently influenced by a notion that Dots-belong-with-a-variable, M. considers re-coding the Frame-plan-for-an-interactive-variable as:
 MAKE :X REQUEST
- 11:32 - M. abandons this re-coding, apparently based on a frame-related notion that Quotes-belong-in-this-frame. M. seems so to have become so attached to the Frame-plan-for-an-interactive-variable that we treat it here as a predictive assumption. This may be related to her concepts of Variable-as-container and Variable-as-alias, though there is no direct evidence of this.
- 12:12 - Influenced by the concepts of Variable-as-container and Variable-as-alias, M. maps C. Report-square-of-user-input to a plan to Report-square-of-user-input-using-a-variable, and codes this as:
 :X * :X
- 12:48 - In a new attempt to resolve the theoretical conflict between the ideas that a) Quotes-belong-in-this-frame and b) Dots-belong-with-a-variable, M. re-codes the Frame-plan-for-an-interactive-variable as:
 MAKE ":X REQUEST
 (discussion only)

Table 7, cont.

13:01 - I.'s comments lead M. to abandon this coding.

13:46 - As part of coding of Report-square-of-Input-using-a-variable, M. Inserts the line:

PR [NUMBER]

as an abbreviation of the commentary line, "The number squared is".

14:32 - Influenced by her misconception that OUTPUT-means-"screen-output", M. re-codes Report-square-of-user-Input as:

OP :X * :X

This is not a correct use of OUTPUT (see commentary within the transcript).

14:43 - In what we interpret as another shift to a punctuation, M. asks whether :X * :X will be interpreted literally or symbolically by the OUTPUT command.

15:13 - Experiment #2: NUM

GIVE NUMBER

2

* DOESN'T LIKE [2] AS INPUT IN LINE

OP :X * :X

AT LEVEL 1 OF NUM

Theory: :X * :X will be interpreted symbolically rather than literally

Response: Theory (+)

(I. offers a "fix" for this error using FIRST just before REQUEST on line 2; see commentary in transcript for a discussion of this procedural compromise).

17:33 - Experiment #3: NUM

GIVE NUMBER

2

RESULT: 4

Goal: Check-the-procedure

Response: Goal (+)

1"=30sec

Page 142

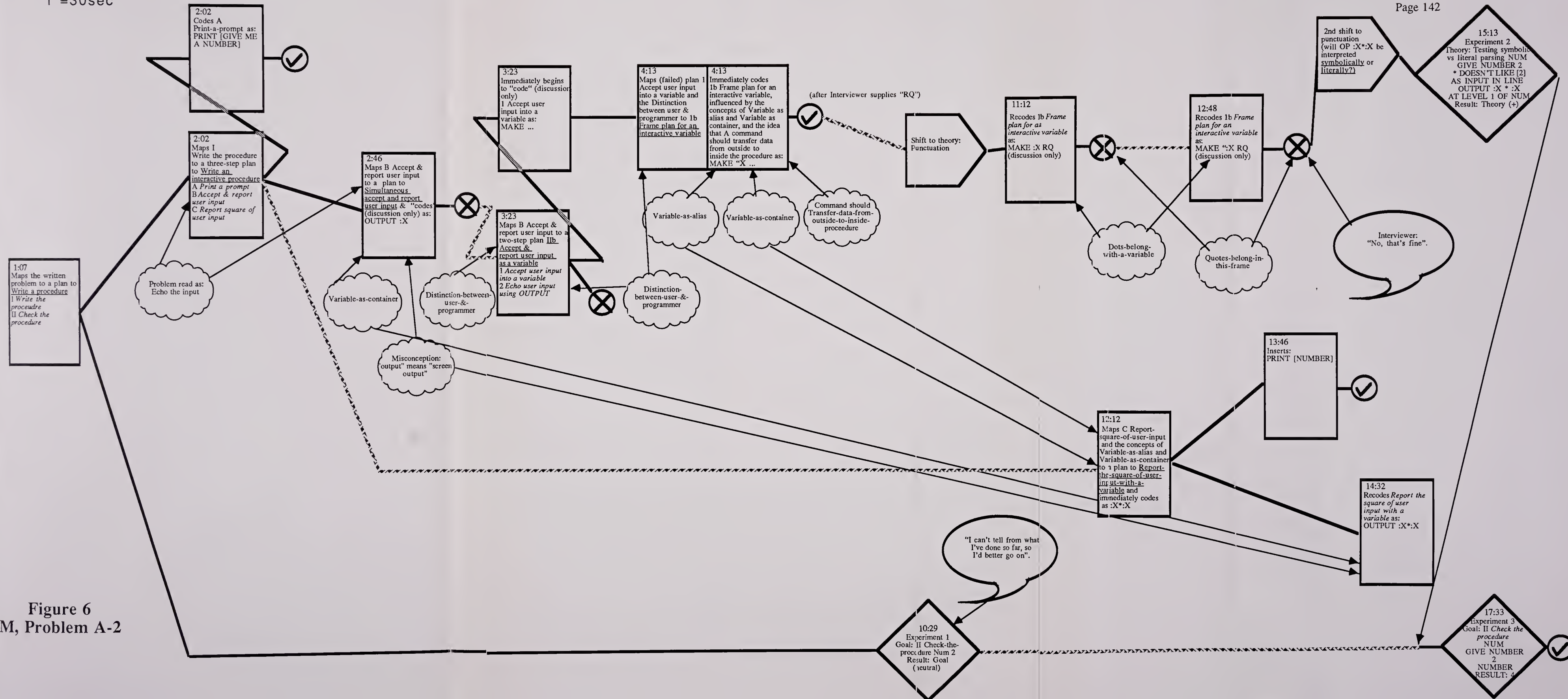


Figure 6
M, Problem A-2

character matrix. With this view, M's misconception could be attributed to a weakness of Logo's design. While this might carry import when considering the design of future versions of Logo, our view of programming-learning requires the learner to sometimes adjust to and find ways to internalize arbitrary constructs, and leads us to an explanation for this misconception that focus on the users' rather than the designers' failings.

On the other hand, in her earlier work M expressed some very strong, descriptive ideas about functional output. In that solution, M functionally decomposed a long Logo command-line; i.e., she accurately described how the outputs of some procedures were simultaneously the inputs to others. There, as here, M's difficulty seemed not to be with the general concept of explicit result but isolated to the mechanics of constructing a user-defined procedure with an explicit result.

One can see a similar dichotomy between two well known variable metaphors that M has assimilated, Variable as container and Variable as alias (Harvey, 1985) and her difficulty deciding on the punctuation needed to implement this knowledge in the context of creating an interactive variable using MAKE and REQUEST (at 11:12). While REQUEST was new to her, M seemed quite comfortable with variables as parameters in the header line of user defined functions, and had previously used MAKE to store preliminary

calculations in a variable (see 8:42). One could simply claim that M had become disoriented by this new context, but the author sees M's lack of a strong concept of the meaning of quotation marks and colons as the primary cause for her trouble. M neither recognized that the quotation marks in the MAKE statement prevented the proposed variable name from being executed as a procedure (i.e., identified it as raw data) nor that a colon preceding variable name referred to the contents of that variable. (Davidson (1985) suggests that both of these ideas are important enough to be explicitly taught to students as part of a unit on Logo syntax). M did seem to recognize a distinction between variable name and value but rather than using the colon in a principled way, M appeared to have developed an arbitrary rule that associates the colon and a variable without a clear rationale. In both her use of variables and of procedural output, M appeared to hold a high level concept while struggling with the problems of implementation. If one treats M's high-level concepts as descriptive knowledge, this could be interpreted as supporting Anderson's ordering of descriptive before procedural knowledge (Anderson et al, 1984). The author, however, views M's high-level concepts as an early development of meta-programming knowledge, not as the collection of simple facts that Anderson describes.

Another interesting part of this protocol is M's development and coding of a Frame plan for an interactive

variable. Even though she has never coded such a frame, M has gleaned from the instructional videotape a very strong understanding of both its function (see 4:47) and its form, except that she had forgotten the last command in the frame, REQUEST (see 6:49). In contrast to her recall performance on other aspects of the instructional presentation (for example, note her difficulties with OUTPUT command) this seems remarkably good. In fact the frame plan seemed to immediately become much more than a simple plan, something more akin to an accepted fact that was strong enough to dissuade her from the rule that Dots belong with variables (see 11:32 and 13:01), which she probably internalized well before this interview. What leads to such ready assimilation of this particular frame into the conceptual framework of this learner while others seem to go by the wayside? There seems to be a sort of "readiness factor" operating in one case and absent in the others. The Distinction between "user" and "programmer", which M has apparently brought with her to this interview, seems a good candidate for such a critical factor. The recognition of this distinction seems an inescapable prerequisite to understanding the notion of an interactive variable and it appears to be strongly in-place at the onset of the protocol.

Protocol 6: A; Problem A-2: Table 8 is the schematic for this protocol and Figure 7 is the related diagram.

Table 8
Schematic; A, Problem A-2

- 00:39 - Maps problem onto a plan to Write-a-procedure (see 4:03). We assume that this includes the steps 1. Code-a-procedure, 11. Check-the-procedure. A. Immediately maps 1. Code-a-procedure to a plan to Write-an-Interactive procedure, with steps A. Ask-user-for-Input, B. Accept-Input (see 03:00), C. Use-the-Input.
- 00:45 - Codes A. Ask-user-for-Input as a print statement.
- 01:35 - Maps B. Accept-Input and a knowledge of the INPUT command in BASIC to a plan to 1. Code-with-an-INPUT-primitive. A. poses this plan as a question to the 1., as an immediate attempt to 2. Test-coding-with-an-INPUT-primitive. We treat this as Experiment #1 - "Is there an INPUT?" (Goal: 2. Test-coding-with-an-INPUT-primitive). Response: Goal (-)
- 02:07 (The plan to Code-with-an-INPUT-primitive is expanded upon in discussion at this time. A. explains that such a command-line would 1. Include the word, "INPUT" (1:35, 2:11), 2. use a variable to hold the Inputted value (3:00, 3:17) (though note A.'s incorrect statement, "It (the computer) gets a variable (sic) from the person typing it in")(see COMMENTS in transcript)). Two concepts that one would expect to be associated with the BASIC INPUT statement seem to be in evidence here. First, the concept of a distinction between programmer and user (see 3:00) and second the what I call a "temporal" concept of programming, specifically the notion that the designation of a variable name and the assignment of its value will come at different times (see 3:17).
- 04:10 - Through a definitive statement, (1: "There is no INPUT statement...") 1. establishes a result (Goal (-)) to 2. Test-coding-with-an-INPUT-primitive.
- 04:19 - With the failure of the above experiment, A. returns to the plan B. Accept-Input, adopting a two step plan 1. Accept-Input 2. Report-Input.
- 04:30 - A. begins reading the Script, in a search for the means to code B. Accept-user-input. We see this as a Search-for-keyword plan, really a general technique to exhaustively search some domain for an instance with certain attributes. It is composed of two steps, a. Find-a-keyword, b. Test-the-keyword (to see whether it is appropriate). In the event of a failed test, these steps can be repeated until an appropriate keyword is found. Note that such a plan can be randomly applied to every element within the domain, as A. begins doing here (see 4:50), or the search can be optimized through the use of some sort of heuristic.
- 04:50 - In carrying out a. Find-a-keyword, A. finds the keyword, PRINT, in the Script, leading to a plan to Use-PRINT. She immediately codes this plan as PRINT :S. While we treat the selection of PRINT as the candidate keyword, A.'s choice may have been in part based on her recognition that PRINT may be a useful tool for this particular problem (see 11:40).
- 05:10 - Codes C. Use-the-Input as PRINT [THE NUMBER SQUARED IS] and PRINT :S*2
- 06:08 - Executes Experiment #2 - "A"
->THERE IS NO NAME S

Table 8, cont.

IN LINE PRINT :S

(Goal: b. Test-the-keyword and simultaneously 11. Test-the-procedure).

Result: The error message is interpreted as a failure of PRINT as the desired keyword (Goal (-) for b. Test-the-keyword). However the influence of 11. Test-the-procedure seems so strong that one tends to predict that A. would have interpreted a positive result to the experiment as verifying this theory, and immediately ended her work here. This dual goal seems a bit of a hedge; A. does not seem confident in the present approach of using PRINT to accept input (4:50 : "...This probably won't work, but I can't think of any other way to do it"), but if it does, she is prepared to declare the problem solved. Upon failure, we would now expect A. to renew her Search-for-keyword plan and return to step a. Find-a-keyword (with some new candidate for the desired keyword, selected randomly from the script).

- 06:21 - (Rather than immediately trying a new keyword, in response to the failure of the above experiment, A. begins a lengthy (2 min. 20 sec.) discussion of the problem. We view this as a temporary abandonment of the exhaustive Find-a-keyword plan to review descriptive knowledge about the problem. As we see it, descriptive knowledge here function as HEURISTICS to direct the search, though this may not be the subject's conscious goal. Specifically, A. describes the needed keyword as similar to a parameter in the header line in that it will store user input in a variable (6:41, 7:27) but dissimilar in that the input needs to occur at a later point in time, (i.e., during program execution rather than at the time of the procedural call) (7:50)).
- 08:41 - A. returns to searching the Script for an appropriate keyword (1. Find-a-keyword).
- 09:16 - I. refers A. to a section of the Instructional Script that demonstrates the use of MAKE and REQUEST to store user input in a variable. This terminates her search for an appropriate keyword.
- 10:06 - After reading this section of the Script, A. maps B. Accept-Input to a frame-type plan to Use-MAKE-to-accept-Input. Her notion of the frame seems to be composed of 4 parts, as follows:
 (1)MAKE (2)"(3)(variable name) (4)REQUEST
 A. Immediately codes (1) & (2).
- 10:45 - After deleting what had been line 1 (PRINT :S), and coding:
 MAKE "... ,
 A. seems to invent a misconception. Based on what we believe to be a deeper misconception, or more correctly, a non-conception (the Absence-of-a-theoretical-model-for-punctuation-with-variables), A. asserts that Colon-punctuation-is-as-a-integral-part-of-the-associated-variable-name (It really is just a shorthand means to designate "value-of") leads to a coding error:
 MAKE ":S REQUEST
 Strictly speaking, this is a legal and acceptable coding in this particular version of Logo, (create a variable, :S, to hold the user's input), but in terms of A.'s obvious intent (to create a variable, S) it must be considered an error.
- 11:40 - A. Codes 2. Report-Input as line 2: PRINT :S . This is identical to

Table 8, cont.

the previous coding of line 1, but we interpret it as schematically (i.e., functionally) different.

11:53 - Experiment #3 - "A" (Goal: II. Test-the-procedure)

-> THERE IS NO NAME S IN LINE

PRINT :S

Result: Goal (-)

The experiment fails with same error message, but for a different reason. Earlier, as A. realized, this message indicated her failure to find the appropriate keyword to allow the procedure to take user input. This time, her error amounts to a misspelling of the variable, the result of her misconceptions about variable punctuation. Interestingly, it is very likely that A. has already made this distinction. She must have great confidence in MAKE as a tool to illicit user input (for the very good reason that it was offered to her as a given), and she never again alters the line in question, the second line of this procedure.

12:20 - A., observing the state of the computer screen after the previous error message, determines that b. Report-Input is superfluous, a mis-goal (see 12:31), and develops and executes a plan to Correct-the-error-by-removing-the-offending-line. The line is superfluous, but this has nothing to do with A.'s current difficulties with variable 'S'. A. is still, prematurely, focused on the goal II. Check-the-procedure, and by blaming a suspicious looking line she seems to be trying with some desperation to carry out that goal. Her failure to develop a quick theoretical understanding of error messages is a sharp contrast to the expert's ready ability to recognize and act upon error messages.

12:40 - Experiment #4 - "A"

-> THERE IS NO NAME S IN LINE

PRINT :S*2

(Goal: II. Test-the-procedure)

Result: Goal (-)

A. seems surprised by the new error, and disappointed that her Correct-error-by-removing-the-offending-line plan did not successfully end her work on the problem. Note that this error message might have been interpreted as a partial success in terms of her present goal, to complete testing the procedure, since more of the procedure ran successfully before an error message was encountered.

13:28 - A. says, "I just want to give this quotes, and see if it makes any difference". This comes after a long pause, it seems that A. has begun to SHIFT attention from her original goal, (Check-the-procedure) to a theory-testing plan meant to aid A. in understanding Logo punctuation.

It is our view that A. has developed a plan to Experiment-with-punctuation, a two part plan, composed of of: 1. Try-punctuation and 2. generalize-about-punctuation. This plan is probably influenced by A.'s previous goal (Check-the-procedure), and by the Absence-of-a-theoretical-model-for-Logo-punctuation.

Table 8, cont.

With line 2 gone, the object of A.'s attention is the fourth line of the procedure, and so, in our analysis, the experimental plan must also be influenced by the earlier plan, C. Use-the-input, but we see the new Experiment-with-punctuation plan as primary, and 1. Try-punctuation maps to a more specialized plan to Use-quotes-to-suppress-the-error-message. This is coded immediately as: PRINT ":S*2 . The protocol from this point on reveals a good deal of impatience on A.'s part to be done with the procedure; we would have expected her to choose a more careful and controlled plan for exploring punctuation. We believe this is an artifact of the semi-active goal-oriented plan 11. Check-the-procedure, a sort of "crosstalk" between a "goal response" and a "theory response" activity (as observed by K.S. & Inhelder). This seems to be true for the remainder of the protocol, and you may note it reflected in our diagram from here on.

13:58 - Experiment #5 - "A"

-> GIVE ME A NUMBER PLEASE

"5"

-> THE NUMBER SQUARED IS

-> :S*2

(Goal: 11. Check-the-procedure)

Result: Goal (-) (14:00 "No, no, no, no!")

14:36 - Following the failure of Experiment #5, A. immediately states that

"...when you have some'm with quotes around it, it'll have what's inside". This appears to be a transfer of the concept of quotes from BASIC, a reasonable but not exact isomorph to quotes in Logo. A. goes on to theorize that the cause of the failure of the last experiment was due to the quotes before the colon, and develops a plan to Fix-error-by-removing-quote, and codes this plan at 15:06.

15:14 - A. develops a second plan, "...for no particular reason"

to Fix-error-by-adding-spaces (on either side of the '*'), and codes the plan. Note that this is a legal configuration for the PRINT command, and would work if line 2 read MAKE "S RQ instead of MAKE ":S RQ.

16:00 - After a long examination of the script, A. develops a plan to Fix-

error-by-switching-MAKE-for-PRINT and codes the plan, though she has little confidence in any of these plans ("...I don't know if this is gonna work, either...").

16:18 - Experiment #6 - Goal: Fix-error-by-switching-MAKE-for-PRINT

"A"

-> GIVE ME A NUMBER PLEASE

"3"

-> THE NUMBER SQUARE IS

-> THERE IS NO NAME S IN LINE

MAKE :S * 2

Result: Goal (-)

Although the predominant high-level plan at this point is the loosely experimental plan to experiment with punctuation, the immediate goal to Fix-the-procedure... (reinforced by 11. Check-the-procedure) strongly influences A.'s experiment, as indicated by the fact that she follows its failure with new plans to fix the procedure, but not by

Table 8, cont.

rejecting the overall strategy of using MAKE to replace PRINT. Our assumption is that A. is capable of careful experimental activity, but is not purely operating in that mode.

- 16:57 - A. develops an original theory: that Variables-are-distinct-from-names". A. associates "name" with the MAKE statement and "variable" with parameter inputs, but she is not simply making a distinction between global and local. A. claims that a "name" can be composed of any number of characters, while a "variable" must be made up of only one letter (17:26). A.'s idea seems related to at least two preconceptions. First, she has undoubtedly been exposed to variables in Algebra, where they are usually limited to one character (One-letter-variables-from-Algebra). Second, while she has written procedures with inputs before, A. apparently has only seen and used one letter names, and has apparently generalized from this a limiting rule for legal "names" (Logo-inputs-may-have-only-one-character)
- 18:45 - Based on her reading of the script, and questions associated with her theory that Variables-are-distinct-from-names, A. develops a plan to Fix-error-by-removing-colon and corrects line 2 to read:
- MAKE "S RQ
- This corrects the error which been causing her trouble since 10:45.
- 18:56 - Continuing her plan to Fix-error-by-removing-colons, A. changes line 4 to read:
- MAKE S * 2
- While this strategy did fix line 2, it does not make sense out of line 4.
- 19:04 - Experiment #7-
- Goal: Fix-the-error-by-removing-the-colon.
- "A"
- > GIVE ME A NUMBER PLEASE
- "2"
- > THE NUMBER SQUARE IS
- > THERE IS NO PROCEDURE NAMED S IN LINE
- MAKE S * 2
- Result: Goal (-)
- 20:07 - After examining the script and the screen for over 10 seconds (A. seems very confused after the last experiment repudiated her last Fix-the-error plan), A. formulates a plan to Fix-the-error-by-using-the-example-as-a-frame, and codes line 4 as:
- MAKE "S * 2 RQ
- 20:50 - Experiment #8- Goal: Fix-the-error-using-the-example-as-a-frame
- "A"
- > GIVE ME A NUMBER PLEASE
- "6"
- > THE NUMBER SQUARED IS
- > * DOESN'T LIKE S AS INPUT IN LINE
- > MAKE "S * 2 RQ
- Result: Goal (-)
- 21:19 - A. quickly develops a new plan to Fix-the-error-by-removing-spaces-around-the-asterisk. We assume that this is influenced by the concept that Spaces-may-affect-arithmetic-operators, derived from previous

Table 8, cont.

experience or a piece of learned descriptive knowledge. A. codes the plan by changing line 4 to:

MAKE "S*2 RQ

21:30 - Experiment #9- Goal: Fix-the-error-by-removing-spaces-around-the-asterisk.

"A"

-> GIVE ME A NUMBER PLEASE

"3"

-> THE NUMBER SQUARED IS

-> (at this point, the procedure pauses, waiting for keyboard input).

Result: Goal (-)

While the goal is not reached, this experiment helps A. to gain insight into her problems. After observing the behavior of line 4 (see 21:34 and 21:44), she aborts the rest of the experiment, laughs and expresses a new gained insight into why line 4 should not use a MAKE statement. (For example at 22:37: "This line is supposed to take the input...of the first MAKE statement, and...multiply it by 2 and print out the answer". This seems markedly different from the rambling style of A.'s recent activity. We regard it as an "aha!", related to some realizations about MAKE and RQ, and it marks the end of her experimentation with punctuation

22:57 - A. interprets the result of the last experiment as an indication that Line-4-should-not-use-REQUEST, and she removes that primitive, but she is confused as to what command should take its place, despite her clear analysis at 22:37. The reason for her inability to translate "...take the input of the first MAKE...multiply it by 2 and print out the answer" into a line that PRINTs 2 times a variable is not clear. It may be due to fatigue, or the interference of her recently invented misconception that names and variables are distinct entities, or a need for closure related to the goal II. Check-the-procedure, or a combination thereof.

23:57 - A. returns to her Search-for-keyword plan. After examining the script, A. maps a. Find-a-keyword to a plan to Use-OUTPUT which she immediately codes on line 4 as OP S*2

24:20 - b. Experiment #10-

Goal: b. Test-the-keyword

"A"

-> GIVE ME A NUMBER PLEASE

"3"

-> THE NUMBER SQUARED IS

-> THERE IS NO PROCEDURE S IN LINE

-> OP S*2

-> AT LEVEL 1 OF A

Result: Goal (-)

24:44 - Maps a. Find-a-keyword to a plan to Use-RQ and codes line 4 as RQ*2

24:56 - Experiment #11-

Goal: b. Test-the-keyword

"A"

-> GIVE ME A NUMBER

"5"

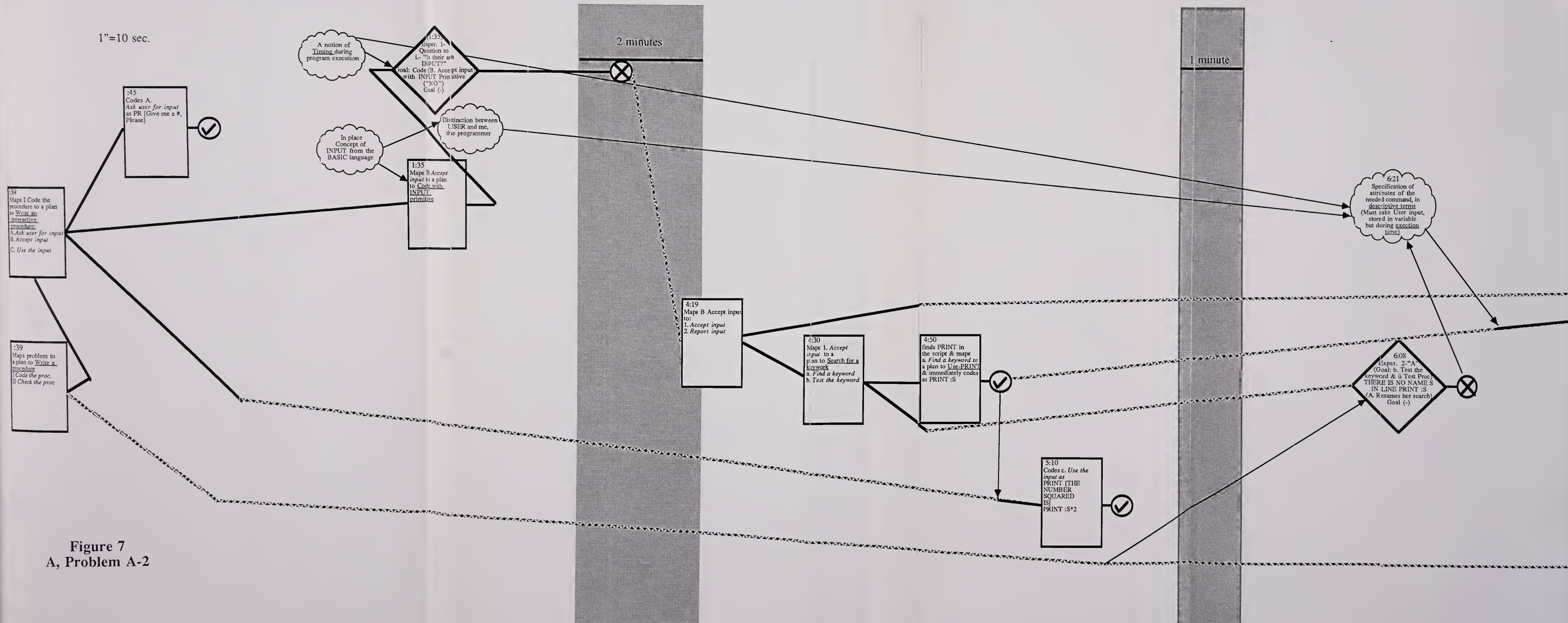
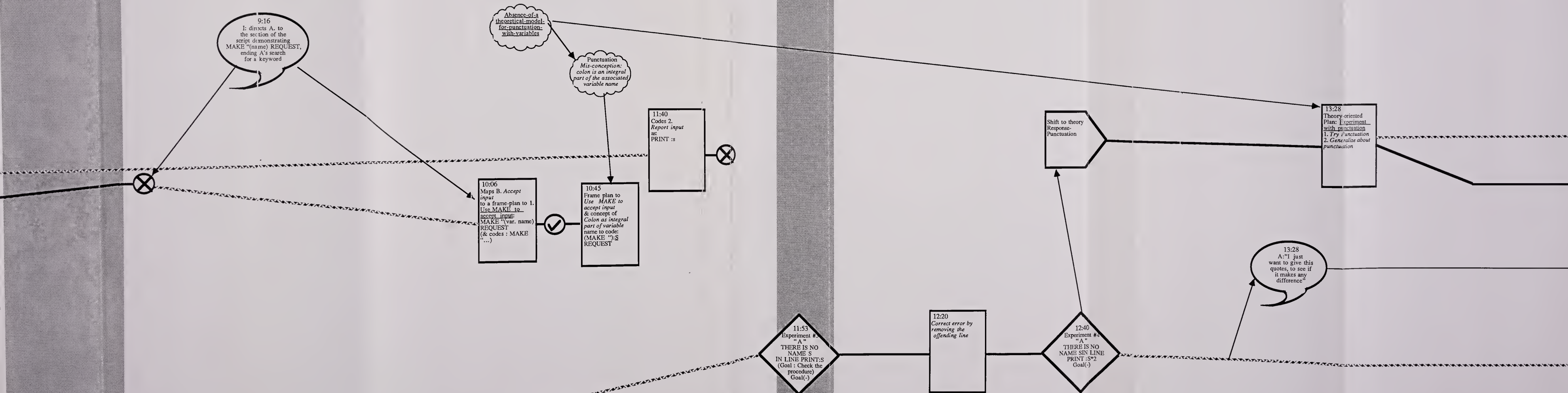


Figure 7
A, Problem A-2

30 Seconds



14:00

1 minute, 30 seconds

30 seconds

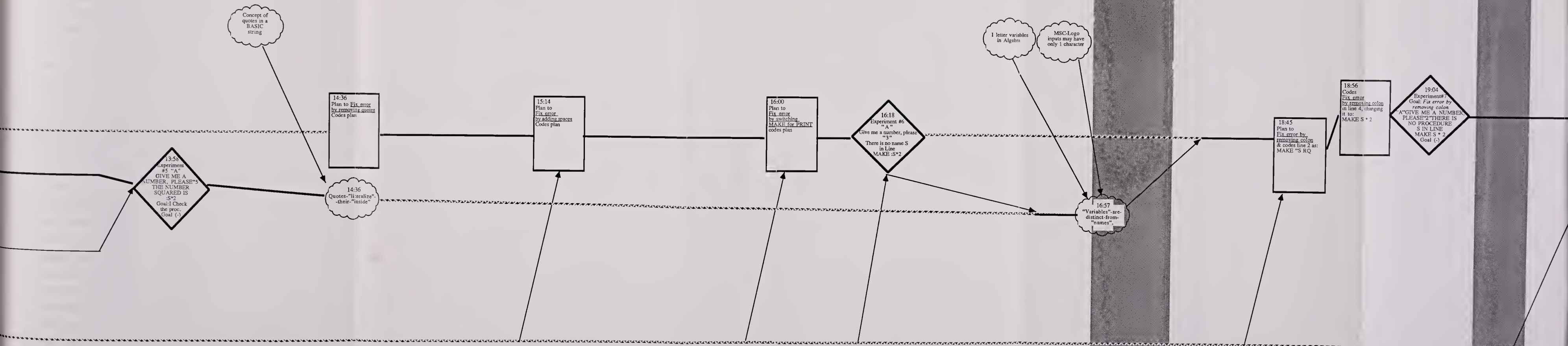
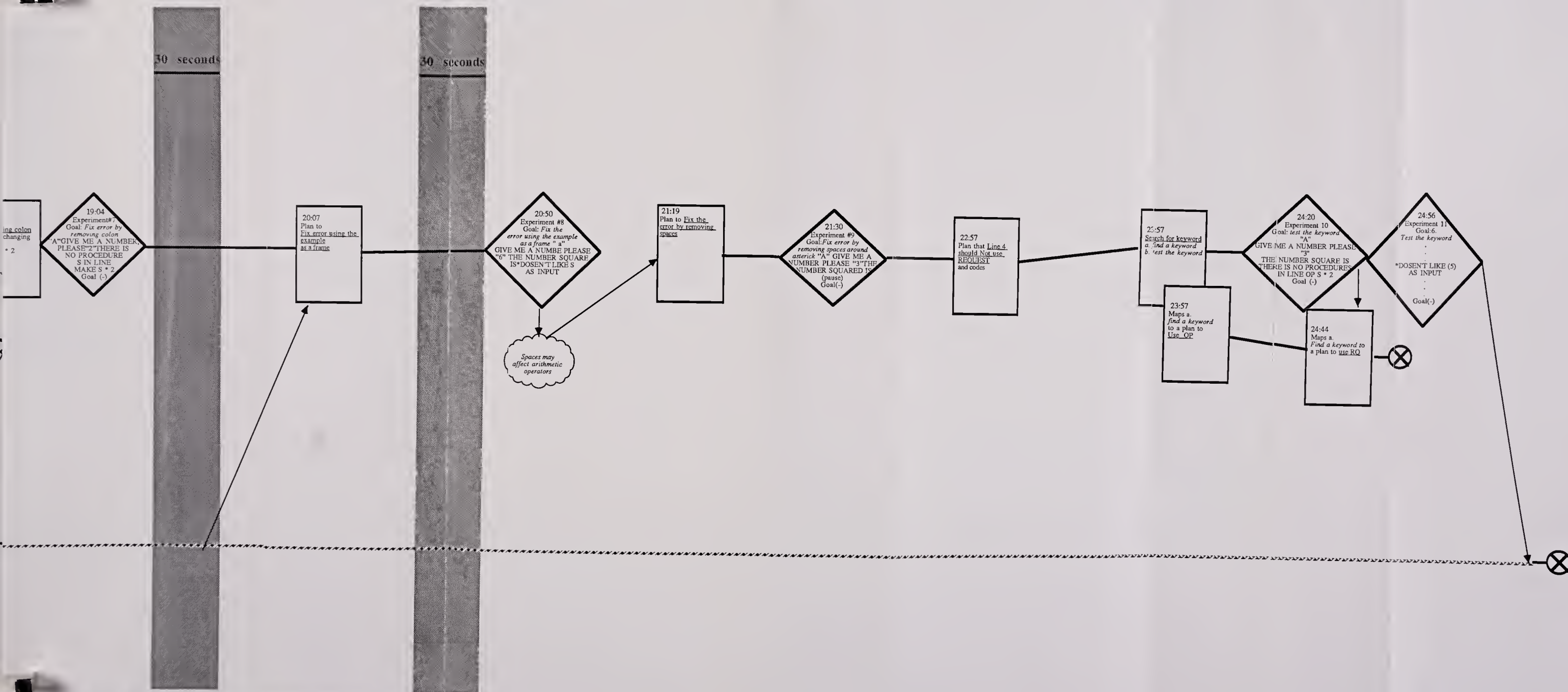


Figure 7 (Cont.)



Several features of this protocol are worth mentioning. At 1:35, A asked the interviewer whether there is an INPUT statement in Logo. This direct question was treated as an experiment to test a plan to code the problem using the proposed INPUT command. It was assumed that this represented, in part, a transfer from A's previously internalized knowledge of the BASIC language (which has an INPUT command that can be used in exactly this type of situation) to a plan to use the same command in Logo. (A may also have been influenced by the presence of the word "input" in the problem statement, (see Appendix B)). There is little research on the transfer of knowledge between computer languages but this observation (along with others, including R's work on Problem A-2, mentioned above) suggests that this question merits more careful attention. A couches the question in terms of a command that might form a complement to the OUTPUT command (which does exist in Logo but not in BASIC). This suggests an alternative explanation for her proposing INPUT. However, since the command A proposed (to allow the user to supply information to a running procedure) and Logo's OUTPUT procedure (which designates the explicit output of a function) are not exact parallels, this explanation is regarded as less likely than the transfer of knowledge from BASIC.

At 4:30, A began an exhaustive search of the instructional script for a keyword that will accept user input. Then, beginning at 6:21 and lasting until 7:50 A

shifted to a review of known facts about variables, perhaps as heuristics to help her with the search. This suggests that descriptive knowledge may play a complementary role with procedural knowledge, and that the problem solver requires both factual knowledge and experience to eventually derive true meaning from both.

A produced an error (MAKE ":S REQUEST) at 11:40 which was identical to one considered (but then abandoned) by M (see above) and by others as well. This is seen to occur due to the lack of a clear conception of syntax and punctuation in a MAKE statement, and of the meaning of quotes and colons in Logo. The presence of this error in more than one protocol supports Davidson's assertion that Logo syntax and punctuation should be carefully included in instruction (Davidson, 1985).

Starting at 13:28, A shifted her attention from the current goal to the meaning of quotation marks, ("I just want to give this quotes, and see if it makes any difference at all"). The result of this, Experiment #5, leads her to an insight on the function of the quotes, as expressed at 14:00. Many of A's attempts to check her procedures lead to theoretical questions, many of them about punctuation. The shift mentioned, above, and the "cross talk" between goal-oriented procedure checking and theory-oriented experimentation on punctuation indicate the

sort of shift from goal to theory observed by Smith & Inhelder (1975).

At 16:57, A generated a new and original misconception; that parameter inputs are distinct objects from variables created with MAKE. She called the former a "variable", but referred to the latter as a "name". (Normally in Logo, these two terms are considered synonymous). While the term, "name", was probably inspired by the previous error message, ("THERE IS NO NAME S..."), A's unique, dual classification is assumed to be the result of an earlier misconception that Logo variables have only one character, which prevented her from integrating the two concepts of variables into one more general conception.

At 21:57, as A viewed failed Experiment #9, she gained some insight into a long standing errant plan that led her to use MAKE in line 4 where she should have PRINTed something. As procedure A was executing, she observed a pause in that line, and after 2 seconds she laughed and aborted the experiment and corrected the error. While other errors in the procedure were never completely corrected, A seemed to have temporarily risen out of the confusion exemplified by a long series of poorly controlled experiments. Even though she never corrected the remaining errors or completed writing the procedure, A's spontaneous insight, coupled with an exclamatory verbalization (in this

case a laugh), meets our qualifications for an insight, or "Aha!".

This concludes the summary of the microanalysis of the six selected protocols. A list of concepts and misconceptions implied in these six interviews are collected as Table 9.

Summary of Other Solutions

Summary of Remaining Solutions to Problem A-2

E; Problem A-2:

In general, E probably had the greatest difficulty of any subject in this study. In the work on Problem A-2, the second problem she worked on, E was able to enter the text editor and to write two lines that correctly printed the messages called for in the problem ("GIVE ME A NUMBER" and "THE NUMBER SQUARED IS :"). At this point she stoped editing and tested the procedure. E gave no clear indication whether or not she recognized that important aspects of the problem remained to be solved. The assumption was that she did, and that the next section of her work began as a test of the code she had already written.

Table 9
List of Concepts and Misconceptions from Six Selected Protocols

Concepts:

1. An operation can be used to replace a variable.
2. A variable can be used as input for arithmetic operators.
3. Meta-knowledge about the relative advantages of alternative approaches to a problem.
4. Concept of robustness in programming.
5. Concept of efficiency in programming.
6. Concept of optimization in programming.
7. General diagnostic principles for interpreting error messages.
8. Concept of variable and procedure as elements of the workspace.
9. Concept of variable as container.
10. Concept of variable as an alias.
11. Distinction between user and programmer.
12. REQUEST transfers data from outside to inside a procedure.
13. The template: MAKE "(variable name) (value)"
to assign a value to a variable name.
14. Notion of timing during program execution.
15. Knowledge of INPUT command from the BASIC language.
16. A theoretical model for punctuation with variables.
17. Concept of a string, from BASIC.
18. Variables from mathematics.

Misconceptions:

1. Punctuation misconception: Dots must always prefix a variable.
2. Misapplication of knowledge of INPUT command from the BASIC language.
3. Lack of a theoretical model for punctuation with variables.
4. Logo inputs may have only one character.
5. "Variables" are distinct from "names" (Parameter inputs vs. global variables).
6. Spaces may affect arithmetic operations.

The remainder of her problem solving was driven by the immediate results of E's recent activity, first by the results of this test and then by results of each successive bit of activity. E's behavior accentuates the importance of some very basic assumptions about programming, and the degree to which programmers take them for granted. Without the overarching theories that guide activity and attention and give meaning to the responses of the environment, the problem solver is left to drift, responding to every meaningless stimulus with the same attention as that granted critical junctures in problem solving. In this sense, E's behavior can be viewed as the antithesis of R's approach to this problem.

In E's case, the overarching theory that is conspicuous in its absence is a strong concept of procedure and a clear distinction between the Logo environment and a defined procedure. E started off quite properly, testing the code she had already defined under the title, NUMBER, but as that incomplete procedure finished executing (the procedure simple printed two lines of text on the screen), E failed to recognize that it had stopped. She behaved as if she had already imbued the program with the ability to take input from the user (the part of the problem that she had not yet addressed), and typed in the number that she wanted squared, 12. With RESULT: 12, (really an error message from the top level of Logo), E began to experiment with the computational ability of Logo in "calculator"

mode. The author assumes that E noted that 12 was not the square of itself, but became completely absorbed with the error message that had appeared on the screen. She next typed in two sixes, followed by carriage returns, and seemed surprised to see two identical lines, "RESULT: 6", on the screen. Apparently E intended to enter two numbers and see their product as the only result, as if she was still working under the control of a procedure that already had the ability to take input and compute with it. Next, E began to experiment with the phenomenon of "calculator mode". She typed, "6+6(return)", and saw RESULT: 12 (she was probably still trying to square the number 6, but was confused about the process of squaring and did not realize that it entails double multiplication, not addition). E remarks, at this point, that "Its just like a calculator...". Considering the her overall performance, this seems like quite an insight for E. "But its weird because you have to write the answer, kind of", acknowledged that E was having trouble explaining what had occurred with her present state of knowledge. However, she seemed very curious and excited (and surprised) by calculator mode, and tried two more simple addition problems, 3+3 and 2+2.

At this point, E claimed that her procedure, NUMBER, was responsible for these results, which were really the response of Logo's operating system. When asked to explain how her procedure worked, E entered the editor and

attempted to explain this calculator-like activity in terms of the simple and incomplete procedure that she had really written, and of course she could not. She seemed to lose confidence this explanation, after first suggesting that the colon, the last item in the second list that she has printed, allowed the procedure to accept user input.

("Dots..., I guess, are where its leaving room for the person to write an answer in. And the computer figures the answer, 'cause that's the procedure that I typed in"). E then claimed that the procedure was done, but she seemed unsure of herself, and in the remaining problems given during another 45 minutes of her interview she demonstrated few episodes of more or less free experimentation such as she did on the above problem.

Upon review, this seems a wonderful teaching opportunity missed. E's seemed to truly have been enjoying her discoveries and insights as she used the computer as a calculator. It is true that she was operating under the mistaken assumption that she was using her own procedure during this period. However, the content of her conceptual insight seems less important than several affective aspects of E's behavior in this section. While during other parts of her work on this problem E seemed slightly frustrated and confused, in this section she seemed to feel in control and to be gleaning information from the results of each experiment; i.e., each time she hit the carriage return she appeared to be engaged in an active learning process.

while the goal of this study was understanding subjects' programming and not actively instructing them, this interview does point out the lengths to which a teacher of programming may have to go at times to understand the assumptions of their students.

N; Problem A-2:

N's work on this problem was in several ways similar to that of A, reported earlier. N, like A, had studied BASIC programming and her work toward solution of this problem, like A's, was in some ways helped and in some ways hindered by this knowledge. Helped in that N, like A, showed sophistication in some of the high-level variable concepts related to this problem. Hindered in that N, like A, sometimes became confused in the implementation of these plans by elements of the BASIC language. In N's case, this sort of other-language distraction was more pronounced. Both N and A generated long protocols somewhat rambling in nature and which ended without success.

Quite early in the protocol, before attempting to actually type in any code, N verbally reviewed the problem and discussed her developing plan to solve the problem. In addition to demonstrating an understanding of procedure definition and naming (with TO NUMBER) and the syntax of a Logo PRINT statement, N considered how to make her procedure square a number, and decided on the use of a

variable. She stated, "You have to have a variable in there", (although, through much of the rest of the protocol, she mistook a semicolon for a colon as correct punctuation for a variable). N made a clear distinction between a variable name and the number-value that might be bound to it. "It represents a number you might type in and it doesn't have a certain value, it can be any value." This showed that N had a strong general concept of a variable. N went on describing the use of an interactive variable: "When you've pushed (;B), when you're putting in the number, which is 2, it would be the same thing...the number is B."

N further demonstrated a basic understanding of variables when she decided to change a variable referred to several times in her procedure. She quickly and easily changed the variable everywhere it occurred. Furthermore, her reason for changing the variable was reminiscent of the work of experts, with their attention to aesthetics and optimization. N changed a variable name when she felt that the new name would better represent its function in the procedure.. She changed variable names twice, first changing B to P ("P is for 'Product'"), than changing P to F (for "factor").

With such an informed discussion of the high-level aspects of variable use, it is surprising that N was unable to complete the problem. N's lack of progress was largely

attributable to difficulties not in planning but in implementation. As with other nonexperts, this was in general characterized by a lack of flexibility and an inability to interpret and utilize information in the form of error messages, references sources and previously learned facts. In particular, N had problems finding ways to elicit interactive input and with punctuation.

Early in the protocol, N formulated the following line:

PR "GIVE ME A NUMBER" ;P

(at a later point in the session, this became:

PR [TYPE IN A NUMBER] :P

and finally:

PR [TYPE IN A NUMBER :P] .

The original quotation marks were a carry over from BASIC (this will be discussed shortly), but the line may also be related to an INPUT statement in BASIC. Compare it with this line in BASIC:

INPUT "GIVE ME A NUMBER"; P

The input statement shown here prompts the user, pauses for user input and stores the inputted value in the variable, P. Obviously, the only difference between this and the line that N wrote was in the first word. This speculation was supported by the observation that N seemed surprised that her procedure did not take input from the user, even though she understood the concept of an interactive variable. Interestingly, when N attempted the problem in

BASIC, toward the end of the interview, she used the original Logo configuration exactly and attributed to it the ability to elicit user input ("If you type in '6' (the variable is 6)"). At a later point, N introduced a variable in the header line and from that point on she supplied a parameter when using this procedure.

N, like all near-novices, had considerable difficulty with punctuation. As mentioned previously, one of the first lines of code that she wrote,

```
PR "GIVE ME A NUMBER" ;P
```

contained quotation marks as used in a BASIC "string". (When she initially coded this line, N referred to the punctuation as "parentheses"). N consistently used quotation marks in this way until the interviewer instructed her in the correct punctuation of a list with square-brackets.

N prefixed all variables with a semicolon (rather than the correct punctuation mark, a colon) for much of the interview. About half way through the protocol she remembered that the correct punctuation was a colon. This correction was not prompted by any experimental results or comments by the interviewer but by N's recollection alone, and so it is treated as the activation of a dormant memory. This seems similar but not identical to an insight, which we normally think of as a working-out of some theoretical question. It is more difficult to

understand why an individual remembers some facts easily but not others, and how and why relevant facts are suddenly recalled. In N's case, there are indications that she was initially distracted by the juxtaposition of a semicolon and a variable name in BASIC's INPUT statement, as mentioned earlier. The visual proximity of the semicolon to a variable and the close relationship between the semicolon and the full colon punctuation probably led N to suppress any memory of correct punctuation (probably rote memory as opposed to functional encoding) and her adoption of the former in place of the later. Once N did recall that a colon was the correct punctuation to use in association with a variable, she used it consistently from then on.

L; Problem A-2:

L began in much the same way as N. She quickly made her first attempt at coding the problem:

```
TO NO. :N
PRINT [GIVE ME A NUMBER]
PRINT :N * :N
PRINT [THE NUMBER SQUARED IS]
PRINT "
```

This was similar to N's work in two ways: (1) The only provision for user input was as a parameter variable on the header line, and (2) the line to print the squared value (PRINT :N * :N) preceded a line meant to introduce it;

(PRINT "THE NUMBER SQUARED IS"). However, upon visually reviewing the procedure L recognized that the order of execution of lines 2 and 3 was incorrect. Her first modification was to line 2, which she changed to:

```
MAKE "N N * N
```

This was her description of the procedure:

"This says, 'Give me a number', you put in the number and then, whatever the number is, it will make that number to double itself, so instead of printing it, it will make it that."

Notice that L described the inputting of a user-supplied value in the midst of program execution, as specified by the problem, not as a parameter during the procedural call. When asked to specify the point at which the procedure takes in the number, L recognized this contradiction ("(otherwise) it'll say, 'NO needs more input' "). Sensibly enough, she began to search for an alternative way to accept user input, first by asking a direct question (which the interviewer refused to answer) and then by inspecting the instructional script. Within a minute she recoded line 2 to:

```
MAKE "NUMBER RQ
```

and deleted the parameter variable in the header line. She vacillated for a few minutes on the use of MAKE in line 2, but upon a careful reading of the instructional script she returned to this configuration, with the following comments:

"Now I understand, I need to REQUEST it, because its asking a question; I want it to ask

them...for a number, and then they'll type the number, and then it'll say, "THE NUMBER SQUARED IS", and then I'll have to write a procedure to square the number".

A few seconds later, L changed the 4th line to:

```
PR :NUMBER1 * :NUMBER1
```

but almost immediately changed PR to RESULT ("Because that's what the result of the square would be"). Again, L had come up with (essentially) a correct coding and then abandoned it. L then tested this version of the procedure, and Logo complained that "THERE IS NO PROCEDURE NAMED RESULT", so she tried MAKE as an alternative to PR, and made several other alterations. Within 6 minutes, however, L returned to her original (and correct) coding of line 4, resulting in an almost perfect coding of the procedure:

```
TO NO.
```

```
PR [GIVE ME A NUMBER]
```

```
MAKE "NUMBER1 RQ
```

```
PR [THE NUMBER SQUARED IS]
```

```
PR :NUMBER1 * :NUMBER1
```

```
END
```

The only error in this procedure is that RQ always outputs a list, and the multiplication operator accepts only simple numbers as input. When L tested this procedure, line 4 failed for this reason, and the interviewer quickly explained the problem to L and suggested the insertion of FIRST before RQ in line 2 as a way to fix it. (In Chapter 3, this problem was recognized as an irritating

distraction, resulting from differences in versions of Logo used by various subjects). Immediately following this explanation, 22 minutes into the interview, L inserted FIRST before RQ in line 2, but through an oversight, she deleted the colon in the second NUMBER1 in line 4. Once again, her procedure was very close to done, but before attempting to test it L began 10 minutes of alterations that lead her further and further from completion. First she changed line 4 to a MAKE statement. (This is the second time the lead command of this line had been changed from PRINT to MAKE). Rather than adding the missing colon to the second NUMBER1 variable, L then deleted the colon from the first NUMBER1, only to reinsert it a minute later, ending up with:

```
MAKE "NUMBER1 :NUMBER1 * NUMBER1
```

The error message (THERE IS NO PROCEDURE NAMED NUMBER1), produced when L then tested this procedure, led her only to replace the quotation marks with a colon and the colon with an equals sign. Further corrections were no more consistent with error messages received. L went on to introduce the FIRST command after the first occurrence of the variable, NUMBER1. Finally, in a postfix configuration, L introduced FIRST after each occurrence of the variable:

```
MAKE "NUMBER1 FIRST=NUMBER1 FIRST * NUMBER1 FIRST
```

And introduced a new line 5:

```
PR :NUMBER1
```

The equal sign that she used may have been a throwback to a LET statement from L's light background in BASIC. The apparent postfix configuration may have grown out of an attempt to imitate line 2:

```
MAKE "NUMBER1 FIRST REQUEST
```

In other words, L could have seen this structure as:

```
MAKE (variable name) FIRST (remainder of line)
```

Soon after this, L and the interviewer agreed to abandon the program. The protocol indicated that error messages and the instructional script influenced L's work less than her confusion about punctuation. Many of her changes to punctuation seemed arbitrary; some may have been experimental, but none (following the 22 minute mark) showed indication of a strong, high-level theory about punctuation and its relationship to variables.

O; Problem A-2:

O read the problem and immediately coded it, as follows:

```
TO SQ
PRINT [GIVE ME A NUMBER]
MAKE "N RL
PRINT [THE NUMBER SQUARED IS]
PRINT :N * :N
END
```

This coding was flawless except for a two minor errors:

(1) O used RL, a command to accept a user-supplied list in

the version of Logo with which O was familiar, rather than RQ, its equivalent in this dialect; (2) The multiplication operator (*) requires its inputs to be words, not lists, and so the user-supplied value must be converted into a word. This can be easily corrected by inserting the FIRST procedure before RQ on line 2. In this case, FIRST would extract the first word from the list output of RQ, essentially converting the user-input into a word.

On first typing line 4, O had used a quotation mark and quickly gone back and changed this to a colon. Before O tested this procedure, he was asked to explain these two types of Logo punctuation. At first O said he didn't know, but when pressed he explained, "Quotes means...making the variable; (the colons) mean using the variable...as far as I know." O's programming performance seemed strong; this first pass at coding revealed no conceptual problems other than the same lack of care regarding the data-type of the inputted value that many others, including experts, had shown. Aside from his tentativeness, his definition of the colon seemed acceptable. His description of the quotation mark, though not a general definition, properly identified it as used in the MAKE statement.

As O went on to test his procedure, he encountered a "* doesn't like [2] as input" error message. One would have expected him to correct the error without much difficulty, but O's inability to glean information from

this and later error messages led him to a long and rambling attempt to correct his procedure. The first thing O did was to change the spacing of the line in question, but the same error message occurred. When O could suggest no other means to repair this error, the interviewer drew O's attention to FIRST in the instruction script. O responded by inserting FIRST before the left-hand argument of "*", though not before its right-hand argument, and the error message recurred. In an attempt to suppress this particular error, O changed the fourth line to read as follows:

```
PR FIRST [ :N * :N
```

This produced a new error, "FIRST DOESN'T LIKE [)". O moved the close-bracket (]), to enclose the first ":N", which printed ":N" before the original error message (* doesn't like [2]...) recurred. At this point O asked the interviewer for help and he was advised to use FIRST before both occurrences of :N, changing line 4 to the following:

```
PR FIRST :N * FIRST :N
```

This repair left one problem for O to solve. The left-hand input to "*" (FIRST :N) needed to be parenthesized in order to prevent "*" from parsing :N before FIRST "converts" it to a word. However, O was frustrated by this problem as well. Even with the suggestion that parentheses might be useful in this situation, O had difficulty. At one point he used square-brackets in place of parentheses. At another point he generated a partially postfix expression,

reminiscent of one of L's productions:

```
PR :N FIRST * FIRST :N
```

In general, these examples demonstrate O's poor sense of how Logo expressions are parsed. O's lack of a strong conception of parsing was probably his most significant problem, underlying his generally poor performance. This nearly complete inability to bring meaning to his interpretation of each of Logo's responses to his experiments led O to express his sense of the futility of any attempt to understand and correct his errors. His response to each error message was haphazard, and aimed toward superficial aspects of each error. Rather than analyzing each error, O tended to exhaustively alter each element of the errant line, in hopes of stumbling upon the solution. As close as he was at the beginning of session, O was unable to independently complete his coding of the problem, and his final attempt was far from the mark. The difference between his initial proposal for coding and his performance when he encountered errors was striking.

K; Problem A-2:

K, like R, had extensive experience with several programming languages, including a great deal of independent programming in Pascal and especially BASIC. His performance was similar to R's in that he was able to

quickly and accurately code the problem, interpret error messages and correct any coding mistakes.

Before K began any coding, he thought about the problem for about 10 seconds and then immediately began typing. In coding the top line, he started to introduce a parameter-variable but changed his mind after a few seconds thought and went on to the rest of the procedure. He coded the rest of the procedure rapidly and with little difficulty. His only error was the use of RQ, a command from the Terrapin Logo dialect with which he was familiar, as a means to elicit interactive user-input rather than RL or RW from the Apple Logo dialect being used for this interview. Once the interviewer explained the two options available to him, K chose to use RW, recognizing that he wanted the users input in the form of a word, and he easily made this correction, leaving his procedure in the following form:

```
TO SQ
  PRINT [GIVE ME A NUMBER]
  MAKE "NUM RW
  PRINT :NUM * :NUM
END
```

In discussion, K suggested that, as an alternative, he could have created a procedure in this dialect called "RQ". He was not specific in the formulation of this procedure, other than that it would include the RW command

and that the original procedure would need to call this new procedure, which he calls "S", in some way. The consideration of alternative problem solutions has been seen only in the protocols of expert subjects and of the other near-expert, and the assumption made by the author is that K's attention to such alternatives grows largely out of his extensive experience in other programming languages rather than his modest training in Logo. At one other point, when coding the fourth line, K began to code the line with a MAKE statement. He, apparently, considered storing the square of the user-supplied value in a second variable, but decided on the above coding, with PRINT. This is a second example of K's recognition of alternative codings of the problem.

K had no trouble with punctuation. He used quotes and brackets correctly and without the need for any aid. The assumption made here is that in K's experience with other computer languages, specifically with BASIC and Pascal, he had developed both concepts and practical knowledge that informed his behavior. One similarity between these two languages and Logo is a distinction between commands and data, an important issue in the use of Logo punctuation such as the quotation mark and the bracket. Unlike BASIC or Pascal, Logo utilizes a special punctuation mark, (the colon) for a reference to the contents (as opposed to the name) of a variable. K showed no difficulty in use of colons, even though neither of the two languages with which

he was so familiar use special punctuation for a reference to a value, relying on syntax alone to distinguish a procedure call from variable-use.

Summary of Solutions to Remaining Problems

Much interesting data was collected on the eight problems other than Problem A-2. The method used during analysis was to successively refine the focus of this study, first by transcribing and examining the general performance of all subjects on the four "simple problems" (see Figure 2) which were attempted by all subjects and later by focusing on a particularly promising problem, Problem A-2. A brief summary of subjects' performance on the remaining problems is in order here (this data is available upon request from the author). To begin with, solutions of the three simple problems besides A-2, in alphabetical order, are discussed in some detail. The remaining five problems, those classified as "complex" (see Figure 2) were administered only where time permitted, and generally only to subjects who had demonstrated mastery of the prerequisite, simple problem. Each complex problem is mentioned, also in alphabetical order, with highlights of those solutions that seemed most intriguing. Finally, a classification of observed misconceptions is presented in outline form.

Problem B-2: Write a procedure called R100 that outputs a random number from 0 to 99, such that if you then type FD R100 the turtle will draw a line segment, but PRINT R100 prints a random number from 0 to 99.

This problem required a simple understanding of a function and the skills that enable a programmer to define a function in Logo. Fundamentally, this meant an understanding of the meaning, syntax and practical applications of the OUTPUT command. Secondly, solution of this problem required the correct use of RANDOM. Both of these ideas were covered in the Instructional Presentation.

All three experts solved this problem quickly and with efficiency, coding R100 as an operation. One of these, H, felt that the wording of the problem was deceptive, although he was delayed only briefly. In later discussion, H explained that, in his initial interpretation, the latter part of the problem statement ("...such that if you then type FD R100 the turtle will draw a line segment, but PRINT R100 prints a random number from 0 to 99") suggested the inclusion of an if clause as a sort of filter, to determine whether the user's input required that the procedure draw lines or print on the screen.

Two non-expert subjects (E and L) showed virtually no assimilation of either the concept of an explicit procedure-result or the mechanics of the OUTPUT command from the Instructional presentation. One of these two, E, used a PRINT command in place of OUTPUT. It was not clear

if this error was related to the "language confound" that led several subjects to use OUTPUT rather than PRINT on problem A-2 to mean "Output to the screen". L had a great deal of difficulty with the problem, and especially the syntax of RANDOM. Neither of the two were able to solve the problem, or to show much progress on the problem before they gave up on it.

Six subjects (J, K, R, O, M and A) seemed to have at least a vague sense of functional output as they began to work on the problem. One of the six (J) seemed familiar with the word, OUTPUT, in the context of Logo programming but worked in a disorganized and ineffective manner until finally giving up on the problem. Three of these six (O, M and A) misinterpreted the examples, which demonstrated how the output of R100 could be used by two primitive procedures, FD and PRINT, as the goal of the problem. Two of them (L and A) initially tried to code R100 as a procedure that drew a dashed line; the other (O) as a procedure that printed a random number rather than outputting one. O and A struggled with the problem, showing little clear direction until they had produced identical miscoding of R100, with PRINT in place of the correct, OUTPUT:

```
TO R100
```

```
  PRINT RANDOM 100
```

```
END
```

For both of these subjects, the error message (R100 DIDN'T

OUTPUT) led them to the correct coding as they appeared to express some insight into the meaning of the OUTPUT command. They both continued working on the problem but made little progress toward a solution before quitting. The two near-experts (K and R) searched the Instructional Script for a procedure that, in the words of one, declared an "explicit result". Both of them found and selected "OUTPUT" quickly and immediately and correctly implemented it.

M's work on this problem was very interesting. She was one of those who initially coded it as a procedure to draw a dashed lines. M claimed she had never seen the OUTPUT command before the administration of this problem, and no knowledge of the command was revealed in her code. However, as she worked on a procedure that imitated one of the examples, M began to discuss the passing of explicit results from a primitive operator (RANDOM 100) to a "destination" procedure (FD) in the line:

```
REPEAT 7 [PD FD RANDOM 100 PU FD 10]
```

She referred to her work as "Experimenting with outputting", and though she eventually became frustrated, M seemed to recognize the nature of her predicament, eventually summarizing her work as a difficulty "making some kind of connection between procedures". This seemed to indicate not only the concept of an output but a concept of the functional composition of operators, although M could not forge these concepts into a proper implementation

of the actual problem. The author interpreted the "crossover" conversation about functional composition as an example of a shift from goal to theory-oriented activity, as described by Smith & Inhelder (1975). This suggested that she had a partially developed concept of OUTPUT in place at the beginning of the protocol.

Before stumbling upon the OUTPUT command, A produced at least three interesting configurations:

1. A raised questions about whether or not to use ':' with the numeric input to RANDOM, considering "RANDOM :100 ."

2. She had difficulties with the mechanism for variable assignment, especially with its syntax. At one point she proposed using "RANDOM 100 = :S" as the coding of a plan to assign a random number to variable S.

3. A used MAKE in place of OUTPUT in one coding of R100, producing the following:

```
TO RND100 MAKE "S RANDOM 100 PU FD 10
PD END
```

Problem D: Write a procedure called MOVE, that takes two numbers as inputs, an X and a Y coordinate. The procedure should move the turtle to that point on the screen. For example:

```
MOVE 100 -5
```

should move the turtle to that point on the screen with an x-coordinate of 100 and a y-coordinate of negative 5.

Problem D tested subjects' ability to use accept parameter input to a procedure and input and to use that

input (as a variable) in the body of the procedure. SETXY was discussed in the instructional presentation in the context of a brief explanation of cartesian coordinates.

All three experts coded this problem quickly, as a procedure using SETXY. One (P) paused upon first reading the problem, then remarked that MOVE was really "just an ALIAS for SETXY", and immediately coded the problem.

Several non-expert subjects tried to write MOVE using only FD, LT and RT commands. E approached the problem in this way, but had difficulty using parameters, and was unable to conceive of a way that the procedure could be generalized to work regardless of the turtle's starting heading. O and L both succeeded using this approach. L had to overcome many implementation problems. O's solution was much less error-laden.

In his initial coding, K attempted to code the problem as a procedure, MOVE, which had two inputs, :X and :Y. In the body of the procedure, K used SETPOS, a primitive in the Logo dialect that he was familiar with. SETPOS takes a list-input composed of an x and a y coordinate. In order to convert :X and :Y into a single list he tried to simply enclose them within brackets:

```
SETPOS [:X :Y]
```

in the body of MOVE.

When he was unable to merge the two inputs in this way, he shifted to using two other Logo primitives, SETX and SETY,

each of which takes a single number as input. K quickly completed the coding using this strategy, and in a follow-up discussion had the interviewer explain the fault with his first approach. M solved the problem using this second strategy (SETX and SETY), and claimed it was "just like" other procedures she had already written. A also solved the problem quickly, using SETX and SETY. She initially coded the header line without variables, but added :X and :Y immediately after deciding on this approach. When asked about their function, she explained that, "...those are just variables. They just happen to be the x and y coordinate". Both these remarks and the speed with which she solved the problem suggested that A had a good grasp of both how to define local variables and how to pass parameters.

R initially misread the problem as having to REQUEST input from the user, but soon completed the problem once he caught this minor error.

Problem E-2: Create a variable called NUMBER, such that
 PRINT :NUMBER
 prints out the number 7.

This problem, which involved a straightforward use of MAKE to assign a value (7) to a global variable (:NUMBER) was solved quickly by all experts, who found the solution obvious.

The two near-experts, K and R, also solved the problem quickly, as did O. One interesting fact was that K, even though he used MAKE correctly and in general used punctuation with great understanding, could not explain the reason for quoting the first input of MAKE. His rote use of this punctuation this subject, whose work work in general showed something approaching meta-programming knowledge of the experts, suggests that this idiom, unsupported by conceptual understanding, may persist for some time without necessarily being reflected as an error in coding performance.

R, on the other hand, could explain the purpose of the colon in the line:

```
PRINT :NUMBER
```

"The colon tells Logo that what follows is not a constant; look to what it points to." In this passage, R described the role of the colon both as a data-type designator and in reference to its implementation at a machine language level, where the variable can be thought of as a "pointer", (i.e., referring only to the address of the memory cell where the value in question is stored, not to the value itself).

Most of the remaining non-experts were able to code this problem correctly. O had no difficulty describing the function of the colon as he coded the problem after a quick reading of it. He explained that, "The variable has '7'

inside of it, so when I do 'PRINT :NUMBER', it'll...print out the number." A could not remember which command to use, but returned to the Instructional Script, quickly found the MAKE command and used it correctly to solve the problem.

M started by creating a procedure, D, with a parameter variable, :N. In the body of the procedure, however, she coded the MAKE statement on the first line, and a PRINT statement on the second. M quoted the first input to MAKE ("N) and considered quoting the second input as well (the number 7), showing that she had no clear conceptual understanding of the quotation mark. Furthermore, M's initial performance indicated that she did not distinguish the parameter, :N, from the global variable. M called the procedure without supplying an input, and in response to the error message, D NEEDS MORE INPUT, M spontaneously decided to change the global variable name from :N to :NUMBER, then used procedure D (this time supplying a numeric input), observed its execution and declared the problem solved. What was interesting was that M's behavior suggested that she did not conceive of global assignment with MAKE, the goal suggested by a careful reading of the problem, as distinct from procedural definition.

L tried a similar approach. She referred to the task in the correct context; "I'm trying to create a variable...", but an early attempt at the problem produced the following code:

```
TO RICK :NUMBER  
MAKE "NUMBER :7  
PRINT :NUMBER  
END
```

This coding and L's behavior throughout the interview indicated that she, like M, did not carefully distinguish between local parameter and global assignment in this context. However the bulk of L's behavior for the remainder of the protocol was in reaction to the error message produced by ':7', as she struggled over correct punctuation for a number.

Problem A: Write a procedure or procedures that repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the correct average. That is, it should not count the final 99999. (Note: This problem was taken from a novice/expert study of Pascal (Soloway et al, 1981)).

One of the experts solved this problem with a single recursive procedure using an IF-THEN-ELSE clause to either accumulate the sum of inputs into a global variable and increment another counter-variable or, when the flag value (99999) was detected, print the average. The other two experts used a recursive operation as a sub-procedure. This sub-procedure monitored each input to determine whether the flag value had been entered, at which point it outputted the sum of inputted values. This sum was used by the super-procedure to print the average.

K, one of the near-experts, wrote two procedures, a main procedure, AVG, and SUMMER, a sub-procedure to collect numbers. Initially, K made an attempt to code SUMMER as an operation, but he encountered many difficulties with this approach, primarily related to the pragmatics of outputting the correct value in all situations. Finally, he abandoned this idea and coded SUMMER as a recursive procedure, using a global variable to accumulate sums and passing that value by referring to that global variable in AVG. In summarizing his solution, K attributed the failure of his initial plan to Logo's limitations, specifically an (imagined) inability to deal with explicit results, (this even though K had used OUTPUT successfully in Problem B-2).

R, the other near-expert, had an initial plan that was sound: write a recursive procedure, using an if clause to accumulate totals or print the average value, depending on whether or not the user inputs the flag value. But, a succession of minor errors resulted in error messages that distracted him from this plan, leading him in an outward spiralling of increasing insecurity as he questioned his more and more basic assumptions about Logo. R ended up with a slightly different plan that used two procedures, a super-procedure that initialized two variables to 0 (R had been confused by earlier results caused by left-over values in a global variable from previous problems) and then calls a recursive sub-procedure with four IF clauses (two to accumulate the total and recurse if the flag value was not

encountered and two to decrement the counter variable (which was incremented upon entering the procedure) and print the average when it did encounter the flag value. He, too, accused Logo of depriving the programmer of the ability to declare explicit results.

M encountered what I call "temporal difficulties" (difficulties related to the timing aspects of program execution) in her failed attempt at this problem. After her initial plan to write a recursive procedure that stops upon encountering the flag value, M shifted to the idea of collecting all values at one time, in a list, and began asking questions about the use of ITEM extract elements from a list. This second plan would require an organized plan to traverse the list, computing the total of it's elements. However, she was unable to implement either of these plans, and ended up with the following configuration:

```

TO D
  PR NUMBER
  MAKE "X FIRST RQ
  MAKE "Y ITEM 2 RQ
  MAKE "Z ITEM 3 RQ
  OP :X*:Y*:Z/2
END

```

Ignoring what appear to be more straightforward mistakes, the failure to quote NUMBER and the incorrect divisor in her averaging algorithm, notice that M used REQUEST in each

of lines 2, 3 and 4, as if it were a variable that held the desired input-list rather than a procedure to allow dynamic interaction with the user. However it would be an oversimplification to summarize M's misconception as simply mistaking RQ for a variable. The author's analysis is that a variety of conceptual weaknesses in both plans, especially a number of weak temporal concepts, led M to spliced together parts of both plans into the odd configuration seen above.

When she ran this procedure M was surprised and unable to explain its behavior and soon gave up on the problem. In a subsequent discussion of the code, M explained that line 1 "...should take the first number, second line, second number, third line, third number, because it says, 'ITEM 3 of the list'", but when asked, "What is the name of the list that ITEM 3 is taking the 'Z' from?", M answered, "Number", referring to the argument to the PRINT statement on the first line of the procedure, not RQ, as one would expect. During her later work on Problem E, M discussed her conception of REQUEST: "When you have it ask a question and you get an answer then you can use that answer by doing the command RQ." Here M describes RQ as an active agent in both eliciting a value and referring to it. Rather than saying that M believes that RQ is a variable, the author believes that M associates RQ with the variable in the frame:

MAKE "(varname) RQ

This frame, which was cited in the instruction presentation, has the function of saving user input for later reference. REQUEST plays the role of forging a communication passageway from the programmer (M) to the user (some as yet undesignated individual who will interact with this program at some point in the future). It is the variable that serves a storage function, not RQ. M has trouble making this distinction, probably because she is unclear about the temporal aspects of RQ in the above frame.

Conceptual aspects of ITEM in M's second, list-traversal plan are weak and ill-defined as well. Problem A calls for a procedure that generalizes to any number of user-input values. For those that solved the problem, this was done with some type of recursion and either a global variable (in a recursive procedure) or a local variable (in a recursive operation). The latter would appear to be more useful here, since M's intent was to collect all values at one time as a list, but she was unable to develop such a plan. The above configuration represents M's attempt to approximate the behavior of a general plan for the purpose of the interview, but instead of using FIRST, ITEM and the addition and division operators to compute the average, in effect mimicking the effect of a (non-general) operation, M chose to extract each value from the pseudo-list and store each into a variable before arithmetically manipulating them. M seems

to have seen this as a simpler approach although it clearly takes more steps than operating on the values directly. M probably felt that variables are more permanent than operational results. (An alternative explanation is that punctuation difficulties in expressions using any of the tightly-binding arithmetic operators may lead some programmers to favor more easily punctuated variables over function calls.

Problem B: Write a procedure or procedures that compute(s) the factorial of a number. Try to put it in as brief a form as possible.

The mention of brevity in the wording of this problem was designed to encourage its coding as a recursive operation. All three experts solved it rather quickly in this way. K's solution was also a recursive operation, and he described an alternative recursive procedure using global variables. R initially attempted to solve the problem with a recursive operation but was unable to complete it and shifted to a recursive procedure plan. This led to a quick solution. Two near-novices (L and A) tried this problem and failed. Both of their attempts showed some evidence of a high-level, recursive plan but both of these subjects encountered implementation problems. One (A) made initial attempts to solve it as a recursive operation, then shifted to recursive procedure plan using global variables but was unable to implement either plan.

Problem C: Write a procedure that, when run, finds the turtle's present compass-heading and points the turtle to a new heading, one-half of the the starting heading. The procedure should operate correctly, no matter what was the starting position or heading of the turtle.

This question was meant to examine subjects' understanding of a primitive operation (SETH). Results for this problem, however, were not very interesting, because most of the subjects who attempted it solved it without much difficulty. All three experts, both near-experts and two near-novices (L and A) wrote procedures that utilized both HEADING and SETH commands. All but one of these non-experts found this information after an efficient search of the script. One of the near-novices (A) correctly utilized the SETH command in a procedure that always set the turtle to a constant heading. This solution did not meet the constraints of the problem, even though A considered her solution to be complete.

Problem E: Write a procedure called COUNTER that takes no inputs, and that prints out how many times it has been used. For example, the first time you type COUNTER, it will print "1", the second time "2", etc.

The expectaton was that any subject who understood the permanent nature of a global variable would have been likely to use a global variable with the program, COUNTER, to solve this problem. All three expert subjects solved this problem quickly using this strategy, although one of them (H) briefly considered using DEFINE to dynamically

redefine COUNTER before he shifted to a global variable strategy and immediately solved the problem.

Both near-experts immediately recognized the same strategy to quickly solve the problem. Neither mentioned the idea of pointers in their solution, but considering R's discussion of Problem E-2 it seems likely that one or both of them had used pointers to create linked-lists in Pascal, a technique closely related to indirection in Logo.

Only two near-novices, M and A, attempted Problem E. Upon reading the problem, M immediately focused on the key question, "How do you make (a procedure) aware it's been used?" She recalled a technique to draw a figure known as a pursuit curve "...by remembering coordinates", an apparent reference to the MAKE command. However, M then considered using REQUEST, based on the following logic:

"Each time you typed COUNTER, you have to use the knowledge that you typed COUNTER before, again. So you have to use what has been typed in again. So that's what REQUEST does. The person types in something and REQUEST uses that again, for something else."

M's misconception of RQ (a confusion about timing aspects of the familiar frame,

MAKE "(varname) RQ

mentioned earlier) seemed to have resurfaced in her work on this problem. Again, this was reflected as a failure to distinguish between the function of RQ, a command to bridge

the partition between program time and run time, and the storage function of a variable. She went on to suggest an association between this problem and the technique of recursion used in association with a counter variable. "You need recursion, anyway", she said, "if it were used right...upping itself". Neither the association of the problem with RQ nor with recursion led M to fertile intellectual ground, and failing to make much progress she soon gave up on the problem.

A found MAKE in the Instructional Script and developed the following procedure:

```

TO COUNTER
MAKE "C C+1
MAKE FRED RQ
IF :FRED=[COUNTER] THEN MAKE "FRED :FRED+1

```

This procedure contains several punctuation errors, and when A began to test it they produced error messages that led A to give up on the problem very quickly. (It was the last of eight problems that she had attempted, and A may have been too fatigued to work seriously on debugging this procedure). Clearly the procedure was incomplete. A mentioned the possibility of adding a print statement. The first line (MAKE "C C+1) which she later deleted but considered reinserting just before she stopped work on the problem, was an attempt to implement a counter-variable, like one she had found in the script. (Notice, however, that she left out the colon punctuation in the second C).

In the second line, although she omitted a quotation mark, A seemed to have made an attempt to implement a frame-model for an interactive variable. Her plan might have involved making COUNTER recursive, and one could consider the attempt to increment FRED in the IF statement of the third line as a simple oversight in a plan to endlessly accept input from the user, incrementing the counter variable (C) whenever [COUNTER] was input. There is not sufficient data to verify this, however, since A mentioned neither the role she intended for FRED nor any plan to make the procedure recursive. A's inconsistent use of variables and variable punctuation, however, and the observation that the resulting error messages distracted her from making any further progress on the problem, support the authors contention that difficulty in using variables is a serious problem for some Logo programmers at this level.

Problem F: Type in the following commands:

```
MAKE "BILL "TEACHER
MAKE "GEORGE "PROGRAMMER
MAKE "SALLY "PROGRAMMER
MAKE "PROGRAMMER [$20 PER HOUR]
MAKE "TEACHER [$15 PER HOUR]
```

Now write a procedure called WAGE, that takes one input. If the input is a person's name (e.g., SALLY), the procedure should print out that person's salary. For example:

```
WAGE "SALLY
should print
$20 PER HOUR
(Sally being a programmer).
```

All three of the experts and one of the near-experts (R) recognized that this problem could be easily solved by using THING to interpret the contents of one variable as

the name of a second variable, a process called "indirection". All four of these subjects quickly developed a procedure that contained a key line of either the form:

```
PRINT THING :(varname)
```

or:

```
PRINT THING THING "(varname)
```

and completed coding with little difficulty. The second near-expert (K) started with a similar plan to solve the problem using THING, but began to have trouble with once he tried to code the key line. His early coding attempts used SALLY, the variable name used in the example, punctuated at different times with a colon, a quotation mark and with no punctuation at all. Since K had previously demonstrated an understanding of variable punctuation, this was interpreted as being primarily due to difficulties with the complexity of the dual-indirection of this line. K eventually developed a rule to algebraically manipulate variable names and punctuation. Essentially, K determined that a colon could be replaced by the word, THING, followed by a space and a quotation mark. Following this rule carefully allowed him to successfully finish the problem quickly thereafter.

All three of the near-novices who attempted this problem came up with interesting approaches. M, who would later use a MAKE statement to successfully solve Problem E-2 stated her conviction that a MAKE statement would only

be permanent if included within a procedure. She appeared to understand some high-level aspects of indirection and discussed the similarity of this problem of this to a nested-repeat statement. She appeared to understand the what a global variable was and the use of a colon to punctuate it, likening the problem to procedural output. M was able to recall THING as a useful tool. However, her critical problem was an apparent inability to utilize THING to formulate the key line. In her later work on this problem, M created a chain of values with MAKE statements, typing:

```
MAKE "A "B
```

```
MAKE "B "C
```

```
MAKE "C "D
```

but became distracted by this activity, as she stated, "O.K., Now I've got MAKE A B, MAKE B C, now I need MAKE C A." In her example, M created an isomorph of the problem showing that she recognized its chain-like nature, but the solution required that she repeatedly apply THING to unravel this chain, arriving at the desired value, not use the MAKE statement to assign a new value to "A.

A had various difficulties in this problem with punctuation and variable use. She inserted all of the MAKE statements called for in the problem statement within the procedure, WAGE. Like M, she seemed believe that the MAKE statement had to be included within a procedure to make them permanent, although she did eventually remove them

from the procedure. She wrote WAGE as a procedure that took no inputs, but she consistently supplied inputs when she ran the procedure. Eventually she included a line to REQUEST a value from the user, but to the end of her interview she ignored the error messages that were caused by irrelevant and unused input. A did not recognize the possibility of chaining values through indirection. Instead she used IF statements to deal with each user input as a special case. For example, she coded one line read:

```
IF :PL1=[BILL] THEN PRINT :TEACHER
```

Notice that A did use the colon here to represent the value stored as TEACHER. Though her coding was a slow process and she never inserted a parameter variable in the header line to accept the parameters she continued to supply at run-time, A was able to eventually complete a program that worked with the user-supplied input.

Like M, L put a parameter variable (:W) on the header line. In her initial attempts to write a PRINT statement, L typed:

```
PRINT :NAME
```

although she had made no attempt to assign NAME a value. This seemed to be an example of what Bonar and Soloway (1985) call a "language confound", in this case the attribution of the variable name with a natural-language meaning.

In her work, L referred to the instructional script, located and read information on MAKE and colon punctuation but after considering THING she dismissed that critical section of the script. "THING, quotes, Bill wouldn't help", she remarked, "(because) its just like the colon." L did not recognize that a strategy based on indirection, in which she could chain the value of SALLY to the value of PROGRAMMER by using THING twice, or in combination with the colon. Like A, she eventually solved the problem by using IF statements instead of THING.

Protocols of all subjects on all problems were viewed to locate programming errors and to determine the misconceptions that most plausibly explain the errors. Errors were classified as being caused by (1) variable misconceptions, (2) "pathfinding problems" and (3) mixed causes. This classification of misconceptions, with references to one or more representative examples, are included as Table 10.

FOOTNOTES
Chapter 4

Adelson, B. "Problem solving and the development of abstract categories in programming languages", Memory and Cognition 9, 422-433, 1981.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. "What do novices know about programming?", Technical Paper, Yale University, 1982.

Hoc, J.M. "Developmental stages in learning to program" International Journal of Man-Machine Studies, 9, 87-105, 1977.

Miller, M.A. "A Structured Planning and Debugging Environment for Elementary Programming", in Intelligent Tutoring Systems, (edited by Sleeman, D. & Brown, J.S.), Academic Press, 1982.

Bonar, J. & Soloway, E. "Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers", Human-Computer Interaction, Fall, 1985.

Davidson, L. "Logo Syntax: Another Story", (unpublished manuscript), 1986.

Smith, A. & Inhelder, B. "If You Want to Get Ahead, Get a Theory", Cognition, 3, 195-212, 1975.

Table 10
Classification of Misconceptions From All Problems

I. Variable Misconceptions

A. Problems with the general concept of VARIABLE

1. "TEMPORAL" misconceptions

-M, Prob. A;

confusion between the items of an input-list and the order of consecutive reads.

2. Questions about the permanence of a variable.

-M, during presentation of Logo

commands: after MAKE "C :C+1, PR :C

"will (another) PR :C still be same?"

-MAKE is not permanent unless it is inside a procedure (M, Prob. F), (A, Prob. F)

3. Viewing variables created with MAKE as distinct from variables created as input

-A, Prob. A-2

4. Confusing a variable with its value (in header)

-L, Prob. E-2 (attempts " :7 ")

5. Difficiencies in the concept of an explicit result

-(M, Prob. B-2) Fails to distinguish "operation" from "procedure" with protracted discussion of OP in different contexts (result-passing of primitive procedures; OUTPUT used for PRINT)

-(L, Prob. B-2) Sees no possibility of a procedure having an output

-Using MAKE for OUTPUT (R, Prob. A), (M, Prob B-2)

B. Difficulties in resolving ambiguous language

1. Natural Language confounds

a. Treating PRINT as synonymous w. OUTPUT ("Output" interpreted as "outputting to the screen").

(M, Prob. A-2, (A, Prob. A-2), (A, Prob. B-2)

b. A, Prob. D, uses MAKE SETX :X to "make" a dot.

2. Using an explicit command as a natural language concept

-M, Prob. B2, considers "using OUTPUT" to use a primitive operation that has an output

-A, Prob C, confusing the attribute of "heading" w. the HEADING command.

II. Errors attributable to "Pathfinding" problems

A. "Avoidance" theories

1. Blaming Logo after negative theory-response during debugging

-denial of Logo's ability to handle functions and their parameters. (K & R, Prob A)

2. Reading a problem as simpler than it is

-M, Prob. B-2 seeing an operation as a simple procedure

-(A, Prob. B-2), (E, Prob. B-2)

3. (A's) Inventing false rules:

"RANDOM can't have a variable for input.";

"Can I have variables in a MAKE statement?";

"You can't have a variable after FD, right?"

B. Incomplete Idioms

1. Mentally clumping procedures by idiom during recall.
 - M, Prob. A2 uses OP Inappropriately, as if she knows it "belongs" but not how to implement it
2. Recognizable Idioms in an incomplete form
 - A, Prob. B (almost completes recursive factorial, then abandons it for an iterative approach). (and see 3, Mark & Larry, Prob. A)
3. two strategic plans interfering w. one another
 - R, Prob. A, typing first MAKE var. value, then OUTPUT (in what he stated was an operation)
 - R & K, Prob. A, ending up with an inefficient strategic plan (AVG as recursive procedure vs. recursive operation)
 - CONFOUNDED by a PASCAL paradigm?
4. difficulties finding or selecting (once found) appropriate facts
 - Passing over information in SCRIPT (L, Prob. F, reads but passes by THING)
 - Tries, then deserts correct construct
 - A, Prob. A-2, uses OP, then deserts it

III. Errors of mixed interpretation

A. Difficulties with syntax

1. Failure to supply "destination" procedure
2. Irregular direction of operational flow
 - A, Prob. A-2, MAKE S*2 RQ for doubling the input
 - A, Prob. B-2, RANDOM 100 = :S ("trying to get the of the random pick"))
 - O, Prob. A-2, (e.g., ...FIRST :S * :S FIRST
3. Difficulties w. quotation marks
 - Quoting commands within brackets (Larry- [:X :Y])
 - K's inability to explain QUOTES in MAKE "N (value)
 - Quoting dots
 - M, Prob. A-2: MAKE ":X ...
 - A, Prob. A-2: PR ":S * 2
 - A, Prob. A-2: MAKE ":S RQ

B. Inconsistencies in use (of colons, OUTPUT, etc.)

- inconsistent use of OP in a recursive proc. (K, Prob. B & Prob. B-2
- OP for simple case only)
- R's (Prob A), Inconsistent use of : with variables

C. difficulty COMPUTING WITH an internal model (as though "keeping too many balls in the air")

- K, Prob. F, confusion about "direction of reference" of colon
- L, Prob. F, acknowledges parts of a solution (:Bill="Programmer, :Programmer=[\$20 PER HR.]) but not how to combine them.

D. simple slip

- R, variable names didn't agree (Prob. A, Prob. A-2)

CHAPTER 5

SUMMARY, IMPLICATIONS AND RECOMMENDATIONS

Summary of Results

As noted in Chapter 3 (see Limitations), the chosen methodology of this study precludes any claim to objective results in the sense of statistically proving or disproving a particular hypothesis. Rather, the goal of a study such as this must be to describe important aspects of subject behavior and to generate hypotheses that plausibly address the important and intriguing questions raised by such behavior. The results reported in Chapter 4 are primarily comprised of a set of observations of the behavior of expert and nonexpert programmers accompanied by the author's running commentary, which represents an attempt to reasonably interpret these observations. It is important now to shift from the specifics of microanalysis to a more general summary of the results cited in Chapter 4:

1. The protocols of nonexpert subjects, especially of those classified as "near-novice" (see Chapter 3) included many cases in which the misconceptions displayed by the subjects were local to the problem and inventive in nature. Very few resembled the sort of strongly held, resilient and high-level misconceptions described by Clement in his study of the "Students and Professors problem" (Clement, 1980). In fact, most of the observed errors seemed more

attributable to "missing conceptions" (the absence of strong, guiding, high-level predictive assumptions) than "misconceptions" (mistaken assumptions or wrong facts). This lends support to the conclusions of Brown & VanLehn (1979) and Bonar & Soloway (1985) that many observable errors are "inventive" in the sense that they are idiosyncratic and appear to be generated by the problem-solver rather than learned.

The author confirms the observations of these researchers that such errors can occur when a subject encounters a knowledge gap and reasonably attempts to fill that gap with knowledge drawn from general experience (what Bonar refers to as "bug-patches"). This pattern seems to reflect the problem solver's need for closure on a problem. It also demonstrates that the task of integrating descriptive knowledge into one's existing knowledge base is a complex process.

The implication of "bug-patching" is that descriptive knowledge must be forced into a packet of meaning in order to be useful and retrievable during problem-solving. This, presumably, is the what drives the need for closure. This supports a major conclusion of this study: that neither a procedural nor a descriptive model alone is sufficient to explain the acquisition of a complex cognitive skill such as programming; rather, descriptive knowledge and procedural knowledge must interact in complex ways and over

an extended period of time, leading eventually to the meta-programming knowledge seen in expert programmers (see also #6, below).

2. Experts' solutions on Problem A-2 were fundamentally similar to one another, and differed markedly from nonexpert solutions to this problem. I would concur with Adelson that novices seemed easily distracted by superficial information -- by any surface resemblance of this problem to a familiar problem (Adelson, 1981). Their most common difficulty seemed to be in integrating new knowledge into their existing knowledge-structure. One example was in A's solution to problem A-2, when A, failing to see a similarity between a new technique for defining variables (with a MAKE statement) and a familiar one (as a parameter in the header line) ended up inventing a new entity -- a "name" as distinct from a "variable".

Experts, on the other hand, have the ability to ignore irrelevant similarities. They appear to have, embedded within their image of a programming problem, heuristic information or meta-programming knowledge that enables them to focus only on certain branches of the tree of all possible associations with the problem (Newell & Simon, 1972). The assumption made here is that this knowledge is specific to the domain of programming rather than a general cognitive skill, such as a general ability to recognize the meaningful and ignore the trivial.

Furthermore, misconceptions interfered with the problem solving of many nonexperts, especially during the introduced of new material, and even after approximately 60 hours of formal instruction and hands-on experience. The author assumes that this holds true in general, and that misconceptions persist and can hamper programming ability well into intermediate stages of programming. This seems to favor the wide scope of time favored in a study such as Hoc's (1977), which recognized an extended period of intermediate programming ability, over the short time-frame of Anderson's model (1984). If a student has been given a clear introduction to new facts and has demonstrated knowledge on a limited sample-problem or two, teachers and researchers should not assume that the student is performing at expert level, even in this limited domain. Rather, it is the view of the author that programming expertise (meta-programming knowledge) is a gestalt, a wide-view of the whole that is more than the sum of its parts. This develops gradually as procedural and descriptive knowledge are reorganized and ultimately assimilated into the programmer's existing knowledge structure.

3. The most common variable misconceptions were related to Logo punctuation and syntax. For example, a number of subjects exhibited a misinterpretation of quotes in the first input to a MAKE statement. Three subjects, M, A and

L, came up with the following configuration in their work on Problem A-2:

```
MAKE ":N REQUEST
```

Similarly, A struggled over the question of whether or not to use quotes before a colon-prefixed variable in a PRINT statement, seriously considering the following configuration:

```
PRINT ":N
```

Similarly, M tried putting the variable inside brackets in the same problem:

```
PRINT [:N]
```

L seemed to confuse variable name and value as she attempted to create an input parameter and generated the following configuration:

```
TO FOO :7
```

M was unable to explain the meaning of the quotation marks in a correctly configured make statement, nor was K (who otherwise performed at close to expert level).

All of the above cases demonstrate a lack of understanding of the specific function of each element of a variable idiom. For example: the MAKE primitive, the quotation marks before its first input, the variable name that properly follows the quotes and the assigned value (MAKE's second input) all have distinct and particular functions, as does a colon that precedes a variable. Such punctuation errors related to variables occurred through a broad range of student protocols.

This difficulty with punctuation differs from what has been reported in studies of the learning of other computer languages: the syntax of a language is internalized before semantic features. For example, Soloway et al (1982) found that implementation plans, including syntactical knowledge, were in place before higher level tactical and strategic plans. This suggests that there are aspects of Logo syntax that cause unusual problems for new programmers. The observation of such an apparently general difficulty presents a challenge to Logo instructors, and challenges the notion of Logo's "low threshold". Variable punctuation may be the most awkward aspect of the Logo language; it is regarded by some as a language design issue that has not been satisfactorily resolved. One new version of Logo that Papert has been closely associated with, LogoWriter, has added a new command as an alternative to MAKE, with new syntax:

NAME (value) (word)

It remains to be seen whether this new syntax will be more easily assimilated, but since the quotation marks and colon have been retained unchanged we see little reason for optimism.

4. Some, but not all, near-novice subjects had difficulty with aspects of the REQUEST command, not unexpectedly. The issues of timing (programming vs. run time) and person (programmer vs. user) were seen as contributing to such

errors, but punctuation difficulties were deeply intermingled with these other occurrences and may have contributed to such errors (see #3, above). For example, one near-novice (A, Problem A-2) constructed the following line:

```
MAKE S*2 RQ
```

Although dialog revealed that A intended S to be a variable name, she used neither a colon to refer to its contents nor quotation marks to use S as a literal input to MAKE.

However, deeper misunderstandings about variable assignment were suggested by this line. From her comments, it was clear that A meant to take input from the user, process that input using multiplication and store the result in a variable. By trying to multiply the first input to MAKE, A shows a misunderstanding either of the way the inputs will be interpreted when this line is parsed or a complete misunderstanding of the timing aspects of interactive variable assignment. The dialog and other programming experiments associated with this protocol lent greater support to the latter rather than the former explanation.

L and N's work on the same problem more strongly indicated a confusion about the timing of this idiom. Both of these individuals created procedures in which the REQUEST line preceded a line that prompted the user for input. However, both described their procedures as executing in a normal manner. In both cases, the programmer explained the order of execution of the switched

lines in terms of the natural language meaning of the prompt, a phenomenon related to what Bonar & Soloway (1985) labeled a "language confound" (see also #5, below) In other words, both subjects ascribed computational power to literal data, although such data is, in formal terms, meaning free. Neither subject generated any example of such data-driven order-of-execution in other problems, and so the author interprets these errors as arising out of "temporal" confusion related to the use of the REQUEST command.

5. Several subjects (A & M) clearly demonstrated a language confound, namely the confusion of the command, OUTPUT with the idea of outputting a message to the screen. This confirms Bonar & Soloway's observation that natural language can interfere with learning to program in Pascal (1985). This observation is especially interesting in that the phenomenon seems to cross language boundaries. It also contributes to a view that the concept of procedural OUTPUT is a difficult one.

The OUTPUT command gave difficulty to the two near-experts as well. Both near-experts initially attempted to use OUTPUT to solve Problem A (average all user input numbers until a "flag" value is entered) as a recursive operation, but abandoned the strategy when they encountered difficulties with this approach. Eventually, both turned to the use of global variables as an

alternative to OUTPUT, and both completed a solution of Problem A (as a recursive procedure) using this strategy. Similarly, Problem B (write a procedure that gives the factorial of any number) suggested the use of the OUTPUT command in a recursive operation (this was the approach chosen by all three experts). Both of the near-experts abandoned initial attempts to code the problem this way, choosing an iterative model with global variables instead. In both of these problems, both near-experts blamed their difficulties on what they imagined to be Logo's inability to specify either procedural output or to allow recursion, although one of the two admitted difficulty with recursion in Pascal as well. In these cases, both near-experts demonstrated a rigidity in their approach and in their ability to integrate new information from the instructional script about OUTPUT.

Rigidity of this kind was not in evidence in the work of these near-experts in most other areas, however. Neither demonstrated any difficulty using or generating local variables, global variables, interactive procedures that poll the user for a value and store it in a variable or punctuation such as colons or quotation marks. These skills seem to have been directly transferred from their own "meta-knowledge" developed as a result of their experience programming with other languages. The one exception seems to have been in utilizing explicit results to compose recursive operations, as mentioned above.

6. Procedural knowledge acquisition is less "matter-of-fact" than one would think after reading Anderson's article (1984). Specifically, the following observations may challenge the simplicity of Anderson's model:

- A subject's plans are often directly reflected as behavior. This makes procedural knowledge (incorporated in the diagrams of this study as "plans") much more obvious to the observer than descriptive knowledge. So it is more noticeable and measurable. However, these two types of knowledge should not be thought of as isolated from one another. The detailed probes of this study have revealed a number of misconceptions that appear to be a strong causative factor of procedural errors. Some of these were explicitly described by subjects but most were not.

- The claim of Adelson (1981) that expert programmers are able to adapt to unique and unfamiliar situations is supported by the observations of experts in this study. In particular, experts consistently demonstrated an understanding of ambiguous solutions and questions of program optimization and aesthetics; but it is difficult to explain this solely with Anderson's notion of pre-packaged clusters of

compiled procedural knowledge. Rather, the experts in this study showed an expanded awareness of the implications of various elements of programming activity and of connections between the parts of a programming problem. With these observations, programming expertise seems more closely related to an ability to form analogies than to a rule-driven mapping of a problem to a solution.

-Anderson gives several examples of novice programmers, after being presented with factual information about a language-feature and one or two examples, "compiling" this information into an internalized cognitive skill ("procedural knowledge"). In this study it was observed that procedural knowledge sometimes develops with much more difficulty, stretched out over the "learning life" of a student, (i.e. more stretched out over the learning curve), and that descriptive knowledge appears to exert an important influence on planning. For example, consider the difficulties of several subjects internalizing correct punctuation for a MAKE statement, mentioned earlier. In none of the protocols subjected to microanalysis was consistent mastery of the use of colons and quotation marks demonstrated without an accompanying high-level

conceptualization of a variable (e.g., variable-as-container). We hypothesize that procedural knowledge as it might be observed by a teacher or researcher is built upon a conceptual framework; it requires the presence of an existing body of conceptual knowledge into which new knowledge must ultimately become assimilated.

Implications

Implications for Education

1. Implications for Computer Literacy

There seems to be a growing interest in creating a new, general, and mandatory "computer literacy" requirement in schools, i.e. a policy requiring that all students be exposed to certain aspects of computer technology. Some educators believe that such a policy should be provided primarily to counteract "computer anxiety." They believe that the thrust should be to give students enough familiarity with computer hardware and software to be comfortable when they use computers in the future. This seems a modest objective, although it does not clearly suggest which computer applications should be emphasized.

However, others believe that this is too shallow a goal, and that the mandate for the schools should be to teach students those skills that are likely to be useful to them in college or in the workplace. While there is no clear consensus as to exactly what these skills should be, some that have been proposed are: word processing, usage of data bases and business software, and computer programming.

The general performance of the nonexperts in this study seemed to indicate that some computer skills associated with computer programming are slow to develop. The development of programming ability (and, probably of other computer skills as well) requires not only the learning of facts and the development of procedural knowledge but also their incorporation into one's existing framework, occurring over a significant period of time. Such an investment of time and effort should not be taken lightly. This suggests that a full-scale commitment to the mandatory teaching of computer skills be approached with caution, at least until it becomes clear exactly which skills will truly become a requirement for informed citizens of the future.

2. Implications for a theory of teaching

It has been almost 10 years since the introduction of low cost microcomputers into the consumer market and,

subsequently, into many classrooms, but educational results appear to be quite uneven. Why do some students learn to master computers and others remain "computer illiterates", "computer anxious" or "computer phobic"? Certainly, affluent students have differential access to computers over those less fortunate, both at home and in public institutions such as schools in the more affluent communities where they are likely to live. But economic differences, while important to consider and to try to counteract, do not tell the whole story. For example, Turkle (1984) has expressed her concern that boys seem to dominate girls in access to computer resources.

Personal interest and motivation may be an important factor in explaining how some children attain computer mastery. Two high school age subjects (the two near-experts) each claimed hundreds of hours of programming and other computer experience. The results of this study suggest that an extended amount of hands-on experience such as this is necessary for true skill-mastery. While all educators recognize and support diversity in the interests of their students, there is something disturbing about the gap between computer "haves" and "have nots". What does one say to the student who expresses a sincere wish to learn how to program but is frustrated by the difficulties she encounters in the process? (Some of the subjects in this study fell into this category). The problem for the teacher becomes how to help such a student accumulate

enough computer experience to elevate her skills. In large part, this may mean focusing less on course content and more on helping students to avert frustration and gain access to those internal rewards of programming that have motivated other students to invest the amount of time necessary to become successful.

Another observation with implications for teachers in general is that misconceptions are often adaptive. In other words, they represent theories that often lead to programming success, although they are basically inappropriate and must ultimately fail. Yet, as a result of their adaptive nature, they often work for the student, i.e. produce correct code, and so may be elusive, veiled from the ready view of the teacher. All of the near-novices had used variables as parameters in most of their (admittedly brief) programming experience, and in most cases their teachers must have felt that they understood variables, at least in this context. What may not have been apparent was that their behavior was often dogmatically linked to familiar examples, rather than being based on a firm and accurate theory of variable use. With new types of Logo variables their stereotypes were challenged. Some of these misconceptions were exposed in microanalysis, but it is extremely difficult for a working teacher to interview individual students in this kind of detail. Rather, it is likely that, for students with deep misconceptions, the weakness of their predictive theories

will eventually affect their behavior while the misconceptions themselves may never be identified. Some students will probably recognize their oversights and eventually correct their own misconceptions, but some may not, experiencing only frustration and a vague sense that something is wrong.

All teachers are expected to evaluate the performance of their students; most teachers would like to go beyond this, to help students recognize misconceptions that cause mistakes and correct them. This study suggests the complexity of the latter task. Where time permits, a methodology such as that utilized in this study could provide teachers with a valuable insight into their students thinking.

3. Implications for teaching Logo

An inordinate number of the variable misconceptions observed in this study were related to problems with Logo syntax and punctuation. In the light of these results, it is difficult to imagine any student becoming proficient in the Logo language until they overcome these difficulties. This suggests a link between the understanding of syntax and the concept of a variable in programming. If teachers are aware of this problem, they may be able to help mitigate it. This research supports the recommendation of Davidson (1985) that teachers of Logo include an explicit

treatment of syntax, that they emphasize its importance and strive to reveal to their students its implications.

Rather than teaching syntax as a single unit, Logo teachers should help their students to focus upon syntax at all levels of instruction. For example, Friendly, in a new book on Logo, (1987) uses the metaphor of a "Genie" for the Logo parser (to accompany the better known metaphor of the "turtle" for Logo's graphic cursor) as a means of teaching aspects of Logo syntax. The adoption of such a metaphor and the teaching of syntax in general would also present an opportunity for Logo instructors to give a clear exposition to the concept of a data structure as they teach about brackets and quotation marks, and a careful discussion of variables as they discuss the meaning of the colon.

Another observation relevant to the Logo instructor is that variable misconceptions persist far beyond the novice programmer stage, and so Logo instructors need to extend the scope of time in which they view variable learning. In particular, it seems important to attach greater importance to the question of what their students learn about variables after they have been taught the basic facts. As a means to this end, the author believes that teachers should emphasize an extensive period in which students carry out programming projects. In a programming course utilizing this design, as much as fifty percent of class time would be devoted to writing original computer programs. During this extended utilization period, the

teacher would play a supporting role and would only present new material as it was needed by individual students.

One clear advantage of this approach is that it would give the Logo instructor a better opportunity to observe errors, discuss with students their thinking related to those errors and through such probing uncover elusive underlying misconceptions. The teacher's role is thus transformed; rather than providing information, he now plays a coaching role, providing help and inspiration. Dwyer (1980) uses the analogy of a flight instructor to characterize such an arrangement.

Another advantage is to strengthen students' concept of variables through experiential learning. While the question of how expert knowledge is developed is beyond the scope of this study, experience seems to play a vital role. All experts and near-experts in this study evidenced extended independent programming experience. If any received formal programming beyond their introduction to Logo, they also had extensive hands-on programming experience. While the concept of a variable seems simple and straightforward, this study demonstrates the degree to which near-novice programmers do not utilize information at their disposal. Extended programming experience may be the only way that some students come to terms with the subtleties of variable use.

The author also believes that Logo instructors should recognize the importance of the meta-programming knowledge of expert programmers. Their ultimate goal should be the development in the students of an ability to readily integrate and use new knowledge. This is an ambitious goal, as emphasized by the present study; but the author would argue that students would be better off to come away from any programming instruction with real programming facility, even if they had only been exposed to a limited subset of the language, than with broad factual knowledge but little demonstrated ability to solve programming problems. In any case, teachers must consider the long-term view of their discipline, and the observation of experts in this study shows that an ability to deal with ambiguous approaches to a problem, optimize programming effort and to appreciate the aesthetics of programming best characterize expert performance.

Implications for Language Designers

Recognizing that the concept of variable is a problem for many students, designers of future implementations of Logo should consider how the language could support the development of a correct concept of a variable. For example, at the University of Edinburgh, duBoulay et al (1981) have developed an experimental version of Logo called ELOGO that provides the programmer feedback on variables, program control and other factors to provide a

feature they call "visibility" for the programmer. In regards to variables, a "visible" environment might represent every assignment of a value to a variable name in some sort of iconic fashion. For example, whenever a variable is globally assigned, a box might appear in the corner of the computer monitor, labeled with the variable name and holding the assigned value. If the value changed, then the contents of the box would be replaced by the new value. Local variables could be similarly represented. For example, when a procedure with parameter inputs was executed, the local variables would "pop up" on the screen, next to existing global variables. When the procedure was concluded, the local variables would either disappear (certainly they would cease to have defined values) or change in some way that suggested that they were only names, awaiting the assignment of values passed as parameters when the procedure is called again.

Implications for Designers of Intelligent Tutoring Systems

Recent attention has been paid to the development of intelligent tutoring systems (ITS). Such systems grew out of an interest in the development of an active, discovery approach to automated tutoring. ITSs are based upon sophisticated models of student behavior, but in that field, according to Sleeman, "Much remains to be discovered and made explicit", and he calls for "...more precise theories of teaching and learning" (Sleeman & Brown,

1982). One generally accepted teaching technique is the Socratic method. With Sleeman & Brown's belief in the need for an ITS to incorporate discovery learning, it may be worth considering how one might model Socrates' teaching style.

At the heart of the Socratic method is the teacher's ability to pose relevant counter-examples that lead the student to question weaknesses and contradictions in his own theories. Another feature of a Socratic teaching-discourse is that students must generalize from their own working theories. An ITS utilizing the Socratic method would need the ability to diagnose student misconceptions in order to determine which counter-questions to pose, elucidate responses from the user and be able to evaluate these responses. The system would require the ability to analyze misconceptions from student solutions using general as well as domain-specific knowledge and an algorithm for generating counter-questions.

The present study attempts to shed light on the nature of conceptions and misconceptions, but in the process raises further questions for the ITS designer. Some misconceptions observed in this study appear to be more resilient than others. Difficulties with OUTPUT, for example, occurred even in near-expert protocols. How would an ITS recognize and handle such a resilient

misconception? Similarly, in this analysis, some programming errors were considered to be related to underlying misconceptions. Using this approach, an ITS would need to distinguish between a fundamental misconception and a superficial one. This might be a behavioral distinction -- misconceptions that showed resilience to instruction (that persist or often recur during the course of instruction), might be identified as deep misconceptions, requiring special treatment. On the other hand, the list of misconceptions in Chapter 4 of this study (see Table 9), compiled from analysis of student solutions, included several misconceptions that were observed in more than one subject. A detailed study such as this, but with a much larger number of subjects, could result in more complete map of the possible misinterpretations of a given problem.

Recommendations for Further Research

Some of the variable misconceptions observed in this study appeared to be quite widespread and somewhat resilient to instruction. For example, misconceptions as to the use and meaning of punctuation, specifically of colons and quotation marks, appeared in most near-novice solutions to Problem A-2. A controlled study of this particular phenomenon is suggested. Specifically, comparative studies to determine effective ways to correct

punctuation misconceptions (or alternatively, to teach correct concepts of punctuation) would seem to be in order. Elusive misconceptions, such as those inferred in microanalysis, need to be exposed to be corrected. Students may do this themselves, responding to contradictions between their internal model of the world and observed behavior by bringing into question assumptions that may be faulty. In order to further our knowledge of effective instruction, researchers need to find which approaches are effective in exposing such misconceptions either to the view of the student or the teacher.

Specifically, the author would recommend an examination of the role of independent programming as a means to uncover hidden student misconceptions and to increase a focus on the utilization of descriptive and procedural knowledge. One study might be a comparison of an introductory Logo course with an emphasis on independent programming, using an extended "utilization period" as described above to reinforce variable concepts with a fact-oriented course in which reinforcement is done with more traditional methods (quizzes, sample programs). The two groups could then be compared by determining the success or failure of students in each on a programming task involving variable use.

Another recommendation would be to further the time-scope of learning through a detailed study of

programmers with over 120 hours of combined instruction and programming time. Anderson studied the first 60 hours of LISP programming and uncovered some basic mechanisms to explain the initial performance of his subjects. This research was of subjects with over 50 but less than 120 hours of Logo experience, roughly the equivalent of a one-semester, introductory course, and revealed the difficulties in integrating individual units of knowledge in a problem-solving situation. It also included as subjects expert programmers, with hundreds of combined hours of formal instruction and programming experience, and revealed a striking contrast between experts' meta-programming-knowledge and the inflexibility of near-novices. A detailed study of the period of learning bracketed by the near-novices and the experts, such as a detailed study of students in a second semester Logo programming course, would follow the development of students from approximately 120 hours of instruction and experience to about 240 hours. The two near-experts in this study, for example, reported about 200 hours of programming experience (albeit in languages other than Logo), and demonstrated much of the meta-programming knowledge typical of experts. Such a study would have a good chance of catching the development of meta-programming-knowledge, and of uncovering the ways that near-novice knowledge is refined into the more easy and natural thinking of expert programmers. Such information

might help us to find ways to facilitate the transition in other students.

The possibility of a transfer of knowledge from other languages seems worthy of further examination as well. This might take the form of a quantitative study of pure-novices vs. those with other programming language experience. If differences can be found, further study to isolate the specific areas in which other-language knowledge aids in Logo-learning (and where it does not) would help to reveal more about the anatomy of learning in general and learning to program in particular.

Another area worthy of further study is the idea of meta-programming-knowledge put forth in this study. The use of certain terms in a particular context, for example the use of prepositions in reference to procedures or of pronouns in reference to interactive processes may be correlated to important knowledge about programming. A linguistic analysis of expert programmers vs. programmers at other levels might reveal this.

Conclusion

As discussed in Chapter 1, the concept of a variable in Logo programming is a complex but an interesting knowledge construct, that can provide insight into the nature of programming-skill acquisition. While the

particular distinctions of Logo variable-use, cited in that chapter, did not necessarily transfer to other intellectual domains such as science or mathematics, the general concept of a variable is important to these and other disciplines. While the literature reviewed in Chapter 2 examined the general nature of programming knowledge, of the concept of variable in mathematics, and of other programming constructs (such as control structures), the paucity of work focused on programming variables called for an exploratory approach.

What were seen in Chapter 2 were a number of general themes, reflected in the literature on programming skill development, that acted as filters, providing alternative ways to view the results of the current study. At the highest level, these either depicted programming knowledge as a concept (e.g., as a "runnable mental model", internalized in the expert programmer), or as a production, built up from programming experience and stored as procedural knowledge (e.g., Anderson et al, 1984). The descriptive approach of this study, specified in Chapter 3, was chosen to allow a wide view of the complex concepts and skills associated with programming in general and variable-use in particular. Diagramming techniques were taken from other studies and combined to allow the representation of both conceptual knowledge and procedural knowledge and the interaction between them after careful microanalysis.

In Chapter 4, the results of microanalysis revolved around the diagrams of six student and expert protocols on a selected problem. Concepts and misconceptions, inferred in microanalysis, were collected from these diagrams and summarized in a Table 9. Highlights of all subjects on all nine problems (which represented diverse examples of variable use) were discussed in that chapter and summarized in Table 10. In general, the results emphasized the qualitative difference between expert and novice behavior. This was primarily categorized in Chapter 5 as a difference in the way these two groups integrated new programming knowledge and experience with high-level, general knowledge, and was termed "meta-programming knowledge". Meta-programming knowledge was hypothesized to be a secondary reorganization of both descriptive and procedural knowledge into integrated, general knowledge structures. It was recommended that meta-programming knowledge be emphasized in teaching programming and examined in greater detail in further studies.

FOOTNOTES
Chapter 5

Clement, J., Lochhead, J. & Soloway, E. "Positive effects of computer programming on the students understanding of variables and equations", Proceedings of the Association for Computing Machinery, National Conference, 1980.

Brown, J.S. & VanLehn, K. "Towards a Generative Theory of 'Bugs'", Cognitive and Instructional Series # 2, Xerox Palo Alto Research Center, 1979.

Bonar, J. & Soloway, E., "Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers", Human-Computer Interaction, Fall, 1985.

Adelson, B. "Problem solving and the development of abstract categories in programming languages", Memory and Cognition 9, 422-433, 1981.

Newell & Simon, Human Problem Solving, Prentice Hall, 1972.

Anderson, J., Farrell, R., Sauers, R. "Learning to Program in Lisp", Cognitive Science, 8:87-129, 1984.

Adelson, B. "Problem solving and the development of abstract categories in programming languages", Memory and Cognition 9, 422-433, 1981.

Hoc, J.M. "Developmental stages in learning to program" International Journal of Man-Machine Studies, 9, 87-105, 1977.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. "What do novices know about programming?", Technical Report, Yale University, 1982.

Turkle, S. The Second Self Simon & Schuster, 1984.

Davidson, L. "Logo Syntax: Another Story", (unpublished manuscript), 1986.

Friendly, M. Logo: A Language for Learning, Earlbaum Associates, 1987.

Dwyer, T. "Significance of Solo-mode Computing for Curriculum Design", in Computers in the Schools, Tutor, Tool, Tutee, (R. Taylor, Ed.), Teachers College Press, 1980.

duBoulay, B., O'shea, T. & Monk, J. "The black box inside the glass box: presenting computing concepts to novices", International Journal of Man-Machine Studies, 14, 237-249, 1981.

Appendix A Instructional Script

Here is a summary of some important features of the Logo computer language.

Every line of Logo is composed of procedures and inputs to procedures.

Print is a built-in or "primitive" procedure that prints its input on the screen. For example;

```
PRINT 5
```

puts the number 5 on the screen. The number 5 here is an input to the print procedure, i.e. it is an argument or parameter that alters the behavior of the command.

Print can take as input a number, a word or a list. For example,

```
PRINT "ONEWORD
```

prints out a word,

```
ONEWORD
```

and

```
PRINT [HERE IS A LIST]
```

prints out a list.

```
HERE IS A LIST
```

An input can also be an expression, i.e. a set of one or more operations the whole set having an explicit result. so you can print the result of numeric expressions, such as 4+5.

```
4+5  
RESULT: 9
```

When used as an input to PRINT...

```
PRINT 4+5  
9
```

the expression is, in effect, replaced by its result. The plus sign is a special procedure known as a numeric operator.

Here are some other numeric operators;

```
PRINT 3/2 ...division
```

```

1.5
  PRINT 2-10 ...subtraction
-8
  PRINT 3*4 ...multiplication
12

```

You can also print the result of certain procedures that operate on words or lists. We say that such a procedure has an OUTPUT. FIRST outputs the first item of its input. If the input is a list, FIRST will output its first item, usually a word.

```
FIRST [ RED GREEN BLUE ]
```

```
RESULT: RED
```

the first of a word is always a word with one letter.

```
FIRST "CATAPULT
```

```
RESULT: C
```

Since FIRST has an explicit result, its output can act as the input for another procedure, such as PRINT

```
PRINT FIRST "CATAPULT
```

In other words, procedures with explicit results can act as expressions for other procedures. Such a procedure is sometimes called an "operation".

RANDOM is a procedure that outputs a random positive integer, from 0 to one less than its input. RANDOM 3, for example, might output 0, 1 or 2

```
RANDOM 3
```

You can also print something that has previously been stored as a variable. A variable can be created with a MAKE statement, such as

```
MAKE "N 6
```

Once created, you can use a variable by typing a colon immediately before its name.

```
:N
```

```
RESULT: 6
```

The colon here literally means "the value of the thing named 'n'". another way to say this in Logo is with the procedure THING.

THING "N Notice that :n RESULT: 6 and THING "N are
identical operations.

:N is shorter, but THING can be useful. for example, sometimes
programmers store one name inside another:

```
MAKE "VARNAME "N
```

In this example, you could use either the colon or THING to get the
value one of the variables

```
:VARNAME
RESULT: N (the value of
          VARNAME)
```

```
THING "N
RESULT: 6
```

, but you would need THING to get the 6 directly out of VARNAME.

```
THING :VARNAME
RESULT: 6
```

(:VARNAME is "N ... this is the value of the thing named by "n)

```
THING THING "VARNAME
RESULT: 6
```

, (identical to the previous line), but not;

```
::VARNAME
```

Make is sometimes a useful way to count things. If you start by
making 1 the value of C

```
MAKE "C 1
```

, you can increment C by one by MAKE-ing C its present value plus 1

```
MAKE "C :C+1
:C
RESULT: 2
```

This can be done repeatedly;

```
MAKE "C :C+1
:C
RESULT: 3
```

Several commands are useful for doing graphics.

DRAW

All drawing is done with a triangular object known as "the turtle". FORWARD moves the turtle in the direction that it is presently facing;

FD 30

(The back of the turtle is indicated by the unshaded bar). The turtle is conceived of as having a pen, that can be picked up to prevent it from tracing its path

PU FD 10

and put down again

PD FD 15

LEFT and RIGHT turn the turtle a given number of degrees.

RT 45

LT 90

The turtle has certain properties that you can inquire about at any time. It has an absolute heading, like a compass heading of West North-west

HEADING
RESULT: 315

It has Cartesian coordinate, including an x-coordinate

XCOR
RESULT: 0

And a y-coordinate

YCOR
RESULT: 55

Other procedures can change the turtle state.

The heading can be changed with the SETHEADING command

SETH 180

(...to point due South).

SETX can be used to position the turtle at a specific x-coordinate

```
SETX -50
```

SETY specifies a y-coordinate

```
SETY 5
```

SETXY can be used to move directly to a position identified by an x,y pair.

```
SETXY 100 45
```

Any of these commands can be used as part of a programmer-defined procedure. To define a procedure, type the word "TO", followed by a procedure title.

```
TO LINETURN
```

You will immediately go into the Logo editor. Type in commands in the order in which you want them executed;

```
FD 10
RT 30
```

When you are done, type END, and then "CTRL"- "c", (typing both keys at once).

```
LINETURN DEFINED
```

To execute the procedure, type its title.

```
LINETURN
```

(Draws a line and turns)

Your procedure can have an output.

```
TO PI
OUTPUT 3.14
END
```

```
PI DEFINED
```

Running the procedure...

```
PI
RESULT: 3.14
```

...causes it to output its specified value.

It should be noted here that as soon as a procedure outputs something, it immediately stops. In this case, that didn't have much effect, since PI had nothing left to do after it outputted, but latter it will

matter.

One programmer-defined procedure can use other programmer-defined procedures;

```
TO PARA
  LINETURN
  FD 50 RT 150
  LINETURN
  FD 50 RT 150
END
```

PARA DEFINED

PARA

(Draws a parallelogram)

If you provide one or more variable names on the top line when you define it, Your procedure can take inputs:

```
TO C.SQUARED :A :B
  PRINT [C SQUARED IS ]
  PRINT :A * :A + :B * :B
END
```

C.SQUARED DEFINED

```
C.SQUARED 5 10
C SQUARED IS
125 Here, A becomes 5 and B becomes 10.
```

Each of the input names in the top line is given a value when the procedure is used. Unlike variables created with MAKE, these variables only exist while the procedure is running.

```
:A
A HAS NO VALUE
```

Another way to take in a value is with the REQUEST command. RQ, as REQUEST is abbreviated, is usually used in conjunction with a MAKE statement to assign a variable "on the fly". For example, if you wanted to ask a person for their name, you might write a procedure like this.

```
TO INQUIRE
  PRINT [WHAT IS YOUR NAME, ANYWAY?]
  MAKE "PLAYER1 RQ
  PRINT :PLAYER1
  PRINT [ THAT'S A NICE NAME!]
END
```

INQUIRE DEFINED

```
INQUIRE
WHAT IS YOUR NAME, ANYWAY?
```

Here the computer pauses for the REQUEST; I'll type in my name.
RICK

```
(The computer types...)
RICK
THAT'S A NICE NAME!
```

Any variables created with MAKE, either inside or outside of a procedure, are permanent. Logo remembers the value of PLAYER1

```
:PLAYER1
RICK
```

, even though the INQUIRE procedure has finished running.

Sometimes you want a procedure to behave differently in different situations. For example, you might want a procedure to stop running when your friend types in her name. This can be accomplished with an IF statement.

```
(
  TO INQUIRE
    PRINT [WHAT IS YOUR NAME, ANYWAY?]
    MAKE "PLAYER1 RQ
```

Its form is IF, followed by a condition, THEN, and an action or set of actions to be carried out if and only if the condition is true.

```
  IF :PLAYER1="LISA THEN STOP

  PRINT :PLAYER1
  PRINT [THAT'S A NICE NAME!]
  END
```

```
)
```

STOP is the command that makes a procedure stop running.

```
INQUIRE
WHAT IS YOUR NAME, ANYWAY?
LISA
```

```
INQUIRE
WHAT IS YOUR NAME, ANYWAY?
RICK
RICK
THAT'S A NICE NAME!
```

A procedure can be written that uses itself. This is a technique known as recursion. I'll change LINETURN into a recursive procedure.

TO LINETURN

(Change procedure to read:

```

      TO LINETURN :L
    (I'll change LINETURN to take an input)
      IF :L < 3 THEN STOP
    (This if statement will stop the procedure if :L gets too small)
      FD :L
      RT 30
      LINETURN :L-3
    (Here I add the recursion)
      END
  )

```

LINETURN DEFINED

```

DRAW
LINETURN 300
(Draws a spiral-like figure)

```

This new version stops if its input is very small. Otherwise it moves and turns, as before, but it then does another LINETURN with a 5 smaller input. That LINETURN does the same thing, including a call of another LINETURN. This continues until some LINETURN has an input of less than 3. When that procedure stops, its parent procedure is finished, and then its grandparent, and so on until all LINETURNS can end.

Logo also allows you to create recursive operations. For example, the operation of exponentiation (i.e., raising to a power) is sometimes defined recursively as follows:

TO EXP :BASE :POW

If you are raising the base to the 0 power, then the answer is 1, because anything raised to 0th power is 1.

IF :POW = 0 THEN OUTPUT 1

Otherwise, the result is defined as the base times the base raised to one-less power.

```

OP :BASE * EXP :BASE :POW-1
END

```

EXP DEFINED

EXP 2 4

RESULT: 16

This concludes the summary of Logo.

Appendix B Problem Set

The follow problem set is ordered from the simplest and most interesting problem (as determined by a pre-analysis) to the least. The first four problems are meant to establish a baseline measure for the four variable classes established earlier, global variable (Problem E-2), explicitly-read variables (Problem A-2), procedural inputs (Problem D) and operations (Problem B-2). The remaining problems involve the use of one or more of these same variable classes but differ in their greater complexity due to several different factors, (see comments for each problem).

Problem E-2

Create a variable called NUMBER, such that

```
PRINT :NUMBER
```

prints out the number 7.

Problem E-2; Comments

A problem in global variable assignment, probably the simplest example of a variable. Besides requiring an understanding of the basic syntax of the MAKE statement, subjects should appreciate such limitations as the "lifespan" of a variable and the distinction between its name and value. In a mature concept, one would understand the function of both the quotes in the MAKE statement and the colon ("dots") used to retrieve the value.

Alternate Problem A-2

Write a procedure that first prints the message,

GIVE ME A NUMBER

, and then prints

THE NUMBER SQUARED IS...

, followed by the square of the number supplied by the person using the program. As an example, after the program is used once, the screen might look like this;

GIVE ME A NUMBER

2

THE NUMBER SQUARED IS...

4

Problem A-2; Comments

This problem requires that the use of variable, and that the variable name and value be defined at different times (a "temporal offset"). The wording was intentionally left ambiguous as to exactly how and when the procedure would collect the value, although the author expected most subjects to use the RQ command (Terrapin dialect only) or the RW command (LCSI dialect only). Both commands have a complex syntax; they take no input, but pause for a user-input and output something as soon as the user hits the return key. RQ outputs the entire user-input as a list, RW outputs the everything the user had typed, up to the first space or carriage return, as a word.

Original the problem was given to subjects using an LCSI dialect. This version of the problem statement was retained for those subjects who used an LCSI dialect (the minority). For the majority, who needed to use RQ, the Interviewer utilized a "teaching probe" to aid them in overcoming any difficulties they had extracting the first word of user-input.

Problem D

Write a procedure called MOVE that takes two numbers as inputs, an X and a Y coordinate. The procedure should move the turtle to that point on the screen. For example;

MOVE 100 -5

should move the turtle to that point on the screen with an x-coordinate of 100 and a y-coordinate of negative 5.

Problem D; Comments

Involves use of a local variable and requires correlation between variable defined in the header line (as a variable name), in the program body (as a parameter) and the value provided at the time of the procedural call.

In the pre-instruction I will show an example of definition of a procedure with INPUT IN THE HEADER LINE and Demonstrate SETPOS. During the session, I will verify that the subject understands the Cartesian coordinate system for both positive and negative numbers and answer any questions about SETPOS or Cartesian coordinates.

Problem B-2

Write a procedure called R100 that outputs a random number from 0 to 99, such that if you then type FD R100 the turtle will draw a line segment, but PRINT R100 prints a random number from 0 to 99.

Prob. B-2; Comments

RANDOM and OUTPUT will be explained, with examples, during pre-instruction. Any questions about RANDOM will be answered immediately. Some of likely difficulties include confusing PRINT ("screen output") with OP.

Problem E

Write a procedure called COUNTER that takes no inputs, and that prints out how many times it has been used. For example, the first time you type COUNTER, it will print "1", the second time "2", etc.

Problem E; Comments

Before presenting, demonstrate the MAKE statement. Answer any questions about the longevity of global and/or local variables. Subject should understand the term "inputs" as it is used in the problem. This problem requires that subjects have or develop a variable-counter plan. To be successful, the programmer must understand that global variables are accessible within a procedure as well as outside.

Problem F

Type in the following commands;

```
MAKE "BILL "TEACHER
MAKE "GEORGE "PROGRAMMER
MAKE "SALLY "PROGRAMMER
MAKE "PROGRAMMER [$20 PER HOUR]
MAKE "TEACHER [$15 PER HOUR]
```

Now write a procedure called WAGE, that takes one input. If the input is a person's name (e.g. SALLY), the procedure should print out that person's salary. For example,

```
WAGE "SALLY
```

should print

```
$20 PER HOUR
```

, (Sally being a Programmer).

Prob. F; Notes

Like E-2 & E, this problem requires a distinction between name and value, but has a much greater complexity, requiring several levels of indirection. Subjects may have difficulty keeping track of this many levels. Using an object alternately as both value AND variable name probably requires greater ability to deal of abstraction.

As in E, the programmer must understand that global variables are accessible within a procedure as well as outside.

Notes: Explain, with example, MAKE and PRINT :NAME, and answer questions about same.

Problem A

Write a procedure or procedures that repeatedly read in integers until it reads the integer 99999. After seeing 99999, it should print out the correct average. That is, it should not count the final 99999.

Problem A; Comments

This problem was originally used to study conceptualization of flow of control in Pascal. Without convenient looping constructs in Logo (Logo does have a REPEAT function, but the repetition cannot be conditioned), the easiest solution is probably a recursive procedure with an IF/THEN/ELSE conditional.

Problem B

Write a procedure or procedures that compute(s) the factorial of a number. Try to put it in as brief a form as possible.

Prob. B; Comments

Notes: Solutions may be either iterative or recursive. I start by providing a comprehensive definition of factorial, including an example. During the recursive definition, I will use the word "recursive".

Beside using input and output, subjects must be able to utilize or develop plans for either a recursive operation or a looping product in order to solve this problem.

Problem C

Write a procedure that, when run, finds the turtle's present compass-heading and points the turtle to a new heading, one-half of the starting heading. The procedure should operate correctly, no matter what was the starting position or heading of the turtle.

Prob. C; Comments

Notes: This problem tests use of machine state variables, specifically the subjects understanding of the SETH and HEADING commands, and of the concept of absolute heading. Start the turtle at heading 0. Interviewer should verify that the subject understands the term "heading", and define it if she does not. Subjects for this problem should previously have been exposed to the SETH command, but it should NOT be explicitly mentioned during presentation of the problem, unless the subject specifically asks for a tool to set the turtles absolute heading. (The same is true for the HEADING command).

When the subject says that she has finished, have her test at least twice, first with at a start heading 0 and a second time after the interviewer has turned the turtle with the command, "SETHEADING 270", (NOT the command LT or RT something)! This state-change should be explained to the subject as it is done, and she should be allowed to alter his procedure if she wishes, but if the solution is not heading specific, the turtle heading should be secretly reset, for a third test.

This problem requires an appreciation for certain "state-variables". To do it, a subject must recognize that for each compass heading is a distinct SETH input and a particular value for HEADING.

Appendix C
H; Problem A-2

00:00:00 H: (takes written prob. from I. and begins to read)

"Write a procedure that prints the message...ok..."

COMMENT: From 00:05 on, H's overall approach shows an unexpected pattern. He does not begin by classifying the task as a prototype and then actualize the plan, completing one step at a time, as Anderson claims is typical. Rather, his approach is to carefully read through the problem from start to finish, stopping to code each problem element as he recognizes it, and carefully proofreading each block of code before moving on to the next problem element. We treat this as general plan, MAPPING THE PROBLEM AS A SEQUENCE OF STEPS. This plan contains four, sequential steps, I. READ A PROBLEM ELEMENT, II. CODE THE PROBLEM ELEMENT III. PROOFREAD THAT CODE, IV. ONCE THE LAST PROBLEM ELEMENT IS CODED, TEST THE ENTIRE PROCEDURE. Our assumption is that H's selection of this approach indicates a piece of pragmatic knowledge. We assume that, in his rapid, initial reading, H has determined that this plan is appropriate for this problem, i.e., that problem steps are sufficiently independently to be coded individually. Such a plan may serve to optimize programmer efficiency by minimizing errors in the mapping aspect of the task. His emphasis seems to be on insuring the accuracy of each step before going on to the next.

00:05 well, we'll do this,...

(types ED)

(a recently written proc. appears in the Logo editor))

...I like procedures; my favorite; oh jeepers;...

(types (ctrl.)-C)

(types: ED [GLUB])

(TO GLUB

END

...appears in the Ed. screen)

...my favorite procedure is...

I: This is A 2.

00:14 H: ...:A 2, I'm sorry; my favorite name for procedures that don't...

(moves cursor to r. of top line)

...do anything that makes any sense is "glub", so...

00:18 (looks at problem)

...it prints, um, give me a number, and then prints, the number squared is;

00:27 (looks up at screen)

Ahhh, "TO GLUB",...

(types: (ret.))

...so we'll say, "PRINT GIVE..."

(types: PR [GIVE ...])

"...GIVE ME A NUMBER"...

(types: ...ME A NUMBER])

Ahh...

00:37 (looks at problem)

'n' It prints "THE NUMBER SQUARE IS"...

(turns toward screen)

00:40 OK, make
 (types: (ret.))
 (types: MAE...)
 (hits DEL key to erase last letter)

00:44 (turns to problem)
 (reads, rapidly and inaudibly)

00:47 Ok, "MAKE...", ahh...
 (types: ...KE)
 (2 sec. pause)

00:51 ...oh well; "MAKE...NUM..."
 (types: ... "NUM RL ...)

00:54 RL"; So I want to pick it off the keyboard.
 (types: ... (ret.))
 And then it says "PRINT, um,..."
 (looks at problem for 1.5 sec.)

01:04 ...sentence of "THE NUMBER SQUARED IS,..."
 (types: PR SE [THE NUMBER SQUARED IS] ...)

01:12 Ah; now...
 (looks down at prob. for 1 sec.)
 (looks up at screen)
 (Immediately types: ... SQ :NUM)
 I'm saying square dots number...

I: "SQ"

H: SQ...I'm gonna write that,...

I: "Dots NUM", ok.

01:24 H: (looks down at problem)
 Um; number squared is that number supplied, blah, might look
 like the number squared, OK,...

01:29 ..."THE NUMBER SQUARED IS";...
 (looks up; reading off screen)
 "PRINT SENTENCE of; THE NUMBER SQUARED IS square dots number"; ok
 (moves cursor down 2 lines, (below END))

COMMENT: The above seems to be an attempt by H to verify the coding of this line.

01:36 ...and we will write, "TO SQUARE dots N, output dots N times dots N, END"...

(types: TO SQ :N
 OP :N * :N
 END)

01:47 (looks up at screen, reading)
 ...um, OK, "GLUB PRINT GIVE ME A NUMBER", It'll print give me a number
 , make number (unintelligible) pick a number, ... OK.
 The number squared is..., but I've made a mistake.

COMMENT: In proofreading this line, H recognizes an error and shifts his attention to the correction of this error.

01:56 (moves cursor up to GLUB, line 3, just I. of the last word on that

```

line (:NUM) )
I need to say the number squared is the square of, I used RL, and I
have to get, ah, first...
02:06 (types : ...FIRST )
...of the list that I caught, so it'll make number squared, is square
to the first of the list; TO SQUARE the list, blah, blah blah blah.
(still looking at screen)
02:15 (types: (ctrl)-C )
Ok, that ought to work, we'll try it out.
Um, "GLUB...
(types: GLUB
GIVE ME A NUMBER )
GIVE ME A NUMBER". I'll give the number, ah... let me give a number I
recognize, 3.
02:25 (types: 3 (ret.)
THE NUMBER SQUARED IS 9 )
The number squared is 9. "GLUB,...
02:31 (types: GLUB (ret.)
GIVE ME A NUMBER )
GIVE ME A NUMBER"...Minus 2...
(types: -2 )
THE NUMBER SQUARED IS 4". So far it's right, I think it's right.
02:38 (turns to I.)

I: Ok

```

Appendix D
P; Problem A-2

0:00:00 P: (reading) "Alternate problem A-2. Write a procedure that prints the message, 'Give me a number,' then prints 'The number squared is' followed by the square of the number supplied by the person using the program. As an example, after this program is used once the screen might look like this: GIVE ME A NUMBER, 2, THE NUMBER SQUARED IS 4."

00:19 (places hands on keyboard, looking from problem to screen to prob. to screen)

00:20 OK. Well it says write a procedure, so I'm gonna...write a procedure let's call it "Square".

00:27 (crans neck to bring head closer to problem)
(speaking slowly)
You don't...ask...me;
(2 sec. pause, looking at problem)

00:30 (returns head to normal position, looking at keyboard)
...OK, you don't specify a title for me...
(gestures w. l. hand to l. of k.b., fingers fully extended, palm up, and looks to screen)

00:34 ...so I'm going to type "To Square",...
(types: TO SQUARE ...)

00:38 (turns to problem)
...and...uh...it's going to ask for the number...

00:42 (turns to screen, pointing w. l. i. finger to r. side of top line)
...so I'm not going to put any input; I'm not going to give it any input hopper here;

00:45 (types (ret.))
...it's "To Square".
(1 sec. pause, looking at problem)

00:48 And, um, the first thing it's supposed to do is print the message, so we're going to say "Print"...

00:53 (types: PRINT [...])
...and the message I'm going to put in a list; "Give me a number."
(types: GIVE ME A NUMBER]...
(2 sec. pause, looking at problem)
(types: ... (ret.))

01:03 And...then it is supposed to...um...
(5 sec. pause, (before and after "...um...", still looking at problem)
(removes hands from keyboard, looking at problem)

01:11 ...Do you want me to use exactly this format? That is, um, a four line format,...

01:17 (slicing gesture w. r. hand, makes 4 parallel lines in air to r. of screen)
...just the way this is printed?
(points to several lines of problem w. l..l. finger)

01:18 You say, as an example, after the program is used once, the screen might look like this.

I: You could give an alternative if you think it would be better.

01:32 P: Ok, well, let me do exactly this one first, and then I'll tell you what I was thinking of.

I: Okay.

01:38 P: (hands to keyboard, looking at screen)
Print "Give me a number." So, ah, now I want to do, what is it, request in this; RQ; in this dialect?

I: Yea.

01:48 P: So, um...

(2 sec. pause, looking at problem)

01:52 ...since I want to get the number here,...

(on "since", P slides l. l. finger l. to r., below line 2)

01:56 ...and...then I want to print something else,...

(on "then", uses same sliding gesture, just below previous one)

01:59 ...and then I want to use it,...

(on "then", uses same gesture, 1 line lower still)

02:01 (begins tapping gesture w. l. l. finger in the 1st (unwritten) "line" below line 2)

...I'm inclined either to do a "Make" here, you know to grab hold of that number...

(grabbing gesture w. cupped l. hand, palm up, to l. of screen)

02:08 ...'cause I'm going to have to defer the use of it later on,...

(holding l. hand, palm down, in front of his body, P slides r. hand, palm down, from l. palm down 6 inches. Repeats gesture.)

...maybe I'll use a subprocedure or something like that.

02:12 Um; why...

(6 sec. pause, w. fixed gaze to the r. of the screen)

02:20 ...I'll use a subprocedure.

(2 sec. pause)

02:23 The problem is, to find; no...I'm gonna...yes, I'll use a subprocedure...

(the above is punctuated w. a pair of gestures done twice, in quick succession. P first points to the problem w. r. i. finger (on "The problem..." and "...I'm gonna...") followed by a gesture of dismissal, literally throwing up his hands, on "...no..." and "...yes...")

... 'cause I'm going to keep to exact format that you've got here and then I'll show what difference I might have done.

(slaps the problem w. all fingers of r. hand "...keep...")

02:32 Um; (2 sec. pause, looking at screen) Print answer...

(types: PRINTANSWER(space) ...)

...request.

(types: ...RQ (ret.)

END (ret.)

(ret.))

COMMENT: The second return indicates that at this point P had it in mind to follow up with a sub procedure.

02:39 (points w. l. l. finger to line 3)

I'm using a subprocedure called "Printanswer"...

I: Um huh.

02:43 P: (points to RQ, on r. of 3rd line)
...and i'm giving it the input of, um, a request, which is going to ask for the number here.
(on "...ask...", P turns to point at problem w. r. i. finger)

I: Okay

02:49 P: So-we-say, "TO PRINTANSWER",...
(types:
TO PRINTANSWER(space)...)
...and, you know, some number in list...
(types:
...:NUMBERINLIST(ret.))

I: Number in list?

03:00 P: Number in list...
(points at line 0 of PRINTANSWER w. i. i. finger)
...To "Print answer, colon, number in list, um...

I: Uh huh

03:06 P: (2 sec. pause, looking down at problem)
...print,...
(types: PRINT(space)...)
...because you want a separate line.
(points to prob. w. r. i. finger)

03:10 (types: ...[...]
Print a message "The number squared is"...
(types: ... THE NUMBER SQUARED IS...]
(ret.))

03:17 ...and then "Print",...
(types: PRINT(space)...)
(2 sec. pause, looking at screen)

03:21 ...and now i; i need to multiply the number that's in this list...
(points w. i. i. finger to :NUMBERINLIST on top line of PRINTANSWER)
...times itself. Ah...
(1.5 sec. pause, looking down at keyboard)

03:29 If i don't care about robustness in this program i could simply try taking the first of that request,...
(points to "RQ" in line 2 of SQUARE w. i. i. finger)
(3 sec. pause, looking at screen)

03:39 ...so, ah...but let's; let's do it...
(moves cursor up to line 2 of SQUARE)

03:44 So, i'm going to go back up the first program,...
(points to line 2 of SQUARE)
...so...
(moves cursor r., to the "R" of RQ)

03:47 ...the second line of the first program is going to be "Printanswer...
(types: FIRST)
(line now reads:

```

                                PRINTANSWER FIRST RQ      )
...first of the request."
(points w. l. l. finger to line 2 of SQUARE      )

I:  Uh huh.

03:56 P:  So that "To square, print give me an answer",...
        (pointing at line 1 of SQUARE)
        "Printanswer first of request."...
        (pointing to line 2 of SQUARE )
04:00 ...and then "Printanswer"...
        (points w. l. l. finger to :NUMBERINLIST on line 0 of PRINTANSWER )
        ...is not going to be a number in a list anymore but;
        (moves cursor down to r. side of line 0 of PRINTANSWER)
        (Child crying in background),
        Poor child...
        (deletes 6 chars. on write of that line)
        (line now reads:
                                TO PRINTANSWER :NUMBER      )
04:12 "Number". And then it says, "Print the number squared
        is"...
        (points w. l. l. finger at line 1 of PRINTANSWER)
04:18 "...number...times...number."
        (types (at the end of line 2):
                                ...:NUMBER * :NUMBER
                                END (ret.)      )
        (proc. now reads:
                                TO PRINTANSWER :NUMBER
                                PRINT [THE NUMBER SQUARED IS...]
                                PRINT :NUMBER * NUMBER
                                END      )
I think I believe this.
(2 sec. pause, looking at screen)
(types: (ctrl)-C      )
04:28 So, let's see; "Square",...
        (types:          SQUARE(ret.)
                                GIVE ME A NUMBER      )
        ..."Give me a number", 2,...
04:33 (types:          2
                                THE NUMBER SQUARED IS ...4      )
        "The number squared is; 4."
        (nods his head in the affirmative)

I:  Okay

04:39 P:  Okay, well, um,...
        (types:          ED (ret.)      )
        ..."Ed", what I had wanted to do, I mean this
        sort of forced me to go;...
        (points up and down screen w. l. l. finger)
        (makes 4 slicing motions in front of the screen & turns to problem)
04:46 ...getting that extra line in there, making it a four line program
        made it actually slightly harder for me to write than if I had

```

allowed myself a three line program.
 (turns toward screen)
 04:57 What I really wanted to do;
 (4 sec. pause, looking at screen)
 05:03 ...well I still have to use that number twice, don't I; I still have
 to use that number twice....
 (moves cursor down to line 1 of PRINTANSWER)
 05:08 Well, what I had wanted to to was put "The number squared is" and
 the answer all on the same line,...
 (points w. l. l. finger to line 1 of PRINTANSWER on "the number..."
 and to line 2 on "...the answer...")
 ...but actually it doesn't save me any room...
 (points w. l. l. finger to RQ in line 2 of SQUARE)
 ...'cause I still have to hold on to that number...
 ("holding gesture" w. l. hand to l. of screen, as before, on
 "...hold...", then points to RQ again)
 ...to use it twice, so that's about as good as...

I: You were thinking you could write it in fewer lines?

05:27 P: Well, what I didn't want to have to do is name this request.
 (pointing w. l. l. finger to line 2 of SQUARE)

I: How do you mean?

05:34 P: Um, well, as it stands, I get a request, so the number is
 typed in from the keyboard,...
 (still pointing at 2nd line of SQUARE)
 05:40 ...um, and I either have to name it so that I can type this line
 in between the "the number squared is",...
 (pointing to line 1 of PRINTANSWER)
 05:48 ...or, and I could have done that with the local, or just with the
 make up here or something, but I chose to use a subprocedure.
 (points w. l. l. finger to 2nd line of SQUARE, then to 0th line
 of PRINTANSWER)
 05:55 But that also names it, it's naming it "Number".
 (pointing w. l. l. finger to ":ANSWER" in line 0 of PRINTANSWER)
 05:58 And I guess in my head I was thinking I could get away without that,
 and I can't get away without that.
 (turns to look at l. for 2 sec. pause)
 I...I...I...

06:08 ...it just felt like it was extra baggage to have to give it a
 name, but obviously I have to multiply it times itself so it has
 to be somewhere for me to use.

06:16 I'm using it twice, at least,...
 (pointing to 2nd line of PRINTANSWER)
 ...and in this case only twice, um, so I don't save very much by
 avoiding this intervening step,...
 (pointing to line 1 of PRINTANSWER)

06:27 ...that's what I was hoping to do, to you know, print "The number
 squared is," and I was thinking maybe I could put that all on one
 line but it doesn't really save me anything at all.

06:35 Here's another version, but it's (unintelligible).

(inserts SE in front of "[THE NUMBER ...]" on line 1 of PRINTANSWER)
(line now reads:

PRINT SE [THE NUMBER SQARED IS...])

COMMENT: Apparently suggesting merging line 2 with line 1, though he does not remove the carriage return separating the two lines.

I: Put a sentence in front of that and have "Print"; have "NUMBER COLON NUMBER"?

06:43 P: Yeah. At the end.

I: Okay. Let me just ask you quickiy, you changed NUMBERINLIST to NUMBER...

P: Yes

06:52 I: ...in "Print answer" and you also changed the second line of "Square" from "Print answer request" to "Print answer, first request."

P: Yes

I: ...and if you could maybe just explain that quickiy.

07:06 P: Okay, well, originally, when I said "Print answer request,"... (points w. I. I. finger to that line of SQUARE)
...what I was thinking to myself is, um, I was thinking of robust programming.
(turns toward I.)

07:15 Ah, when I take a request, the person could type anything, including nothing, he could give an empty line.

07:23 Um, with an empty line, and this thing is asking for a number,... (points w. I. I. finger to line 2 of SQUARE)
...cleariy in a good robust program it ought to be able to, you know, do the right thing if you don't give it a number, you give it nothing, or you give it a word.

I: Um hm.

P: And, ah; ah;
(4 sec pause, looking at screen)

07:42 So my sort of standard thing to do when I hit a line like that is just to go ahead and take it, and let "Print answer" worry about filtering out...figuring out whether it got the right stuff.

07:53 Well, I decided I wasn't going to build in all of these bells and whistles, to try to figure it out. Um, and realized that, in here;

07:59 the point is the first of request is going to be bad news if somebody just hits an empty line.

I: Uh huh.

08:05 P: Okay. And I figured I'd take care of that in here,...
(scratches fingers of I. hand in front of PRINTANSWER procedure)

...but then when I realized in here I would,
(points w. l. l. finger to :NUMBER * :NUMBER in SQUARE)
08:12 ...If I wasn't going to do that building, then by the time I got down
to "Print number times number," it would be "Print first of number in
list times first of number in list," and that's repeating a
computation here,...

08:25 ...there's no point to repeating that computation
unless I were planning on doing something clever with it, and
since I'd already given up on that idea, I just decided to put the
first out here, give "Print answer" only the number,...

(pointing w. l. l. finger to line 2 of SQUARE)

08:38 ...and then I changed the name here because I always like
my names to refer to what I've actually got. Before it was going to
be a number in a list, because that's what "Request" does.

08:47 So there you are.

Appendix E
B, Problem A-2

00:00 B: Write a procedure that first prints the message "Give me a number" and then prints "The number squared is" followed by the square of the number supplied by the person using the program. So, I've gotta write the program. ...The screen might look like this "Give me a number, 2, the number...square...is".

00:16 I: Yea.

00:17 B: Okay. So now do you want me to just do it or do you want me to tell you what I'm doing?

00:22 I: Right, I should make it clear, I do want you to talk about um, you know alot of things so I'd like you to talk about,..uh...the process as you go through it as much as possible, and um, yeah, I mean that's a good enough answer for you because you're familiar with loud thinking.

00:39 B: Okay, um...

I: And I'll be a listener

B: Bright, okay. Um...

I: By the way, that's a problem for the camera, that's in part for me too.

00:50 B: So, I would write a procedure called, probably, "Square" or something...

00:57 Um, and it's going to be interactive with the user because I want to, basically request them to give me input, which will be the number two.

COMMENT: Note the use of "them" to denote the user, "it" and "me" for the procedure.

01:05 Um, actually if..., there are a couple ways I would do this problem, I'll show you one way. Um...

I: Okay.

01:11 B: It would be nice if, um, if I had a little procedure called "Square" that just squares a number and; outputs a square of a number.

01:22 Okay, so I can say to "Square dots num",...

(types: TO SQUARE :NUM(ret.))
...where Num is my variable, and I'll just output, you want me to write that out? Output...
(types: OUTPUT...)
..."dots num times dots num"...

(types: ...:NUM * :NUM(ret.))
 END(ret.))

...and there's my little square procedure.

I: Okay.

B: (types (ctrl.)-C)

01:44 Okay. So now when I type "Square four",...

(types: SQUARE 4(ret.))

RESULT: 16

...um, I'll get the result "sixteen", okay?

I: Okay.

01:51 B: So I'm going to use that in my, um, procedure, F00, I don't know what I'll call it, um, you want me to...

(types: TO ...)

...give it a real name?

I: Anything you want. Anything you like.

02:00 B: To do "SQ dot Num"...

(types ...SQ.NUM(ret.))

02:05 ...Okay? Um, so the first thing I want to do

is have it ask the person, um, a question, so I'm going to say

02:13 "Print, uh..."

(types: PRINT ...)

"Print, uh, give me a number",...

(types: ...[GIVE ME A NUMBER]...)

02:22 ...um and I always like things to look exactly the way they will on the screen,...

(types: ...(ret.))

02:28 ...um,...

(3 sec. pause)

02:32 ...I suppose you want me to have it print the number two there, right?

02:39 I: Um, well let's see...

B: Bright.

I: Why do you say that?

02:46 B: Well, just because of the way it's printed there as a "2".

I: Oh the...

B: And that will...

I: I meant that to be, um, that's a, that's a...

02:56 B: Will that be something that the person types...

I: Just what the person types in, you know.

B: Okay, that makes it easier.

03:00 I: This should probably have a question mark, "Give me a number" should probably have a question mark before it.

03:04 B: Bright, I'll put in a question mark. Um, or a colon, how about that since it's a statement and all that...
(moves cursor to r. end of line 1 and inserts a colon before "]")

I: Okay, right.

03:11 B: Um, and then what I'll do is I'll "Print" um;...
(types: PRINT ...
(4 sec. pause)

03:19 ...this is interesting, this is interesting, 'cause I always; what's nice about these problems is as I'm doing them I can think of, like, three different branches or something about how I would, sort of, solve it, and then I try and think of the most concise way to do it;

03:35 ...um, really, what I would do is I'd say "Print a sentence...
(types: ...SENTENCE ...

03:43 ...made up of "The number squared" um, "is",...
(as she types: ...[THE NUMBER SQUARED IS]...)

03:55 ...and then, um, "SQ first request".
(types: ...SQ FIRST REQUEST(ret.)
END)

I: Uh huh.

04:08 B: Now, that's not necessarily the; if I were doing this for, um, kids to use or something like that, I wouldn't necessarily do this because I don't think it's necessarily clear what's going on.

I: You mean the internals of that program are not as clear as a demonstration?

04:29 B: Right, right, it's not as, sort of linear; sequential, um, if;
(1 sec. pause)

04:36 ...you know, another way I could have done it was to create a variable called "Number" but since I learned LOGO from Brian Harvey, I always learned that you never use MAKE, so I try to avoid it whenever possible when it wasn't...the right thing to do.

I: So, let me see if I understand the advantage of using a variable would be what? For someone, you said, as a sort of exemplary program if you're using it to teach with, uh...

05:02 B: Well, like here,...
(points w. r. l. finger procedure on the screen with a single, quick stroke going from header to last line)

05:04 ...um, if kids are trying to figure out what's going on here,...
(turns r. hand, now holding it palm up in front of screen)
...it's not clear where the person has really typed in the number

that the...

(rotates r. hand clockwise at wrist, w pinching gesture of fingers)

...programmer's asked them for.

COMMENT: The distinction between "the programmer" and the user ("them") is made concisely.

05:14 In other words, I haven't specified a real container for that number,...

(r. hand palm up, as before, then pinches her fingers on "container")

I: Uh huh.

05:19 B:...and it's sort of magically included in this last, ah, line here,...

(points w. r. l. finger to 3rd line)

...um, so that it's actually doing the requesting,...

(circles the word "REQUEST" on the screen w. r. l. finger)

...getting the number...

(clenches r. hand into a fist in a "grabbing" gesture)

... and then automatically passing it down to that, uh, Square procedure that I have as a tool hanging around.

(on "...automatically passing it down..." B. pulls fist away from screen and down, retaining clenched hand)

I: Yes.

05:38 B: Um, now of course if I have a problem, if really what I wanted to say instead of just the "Number squared is"...

(slides her pointing r. l. finger along 2nd line)

...such and such,...

(bounces r. hand, held w. palm up, three times, moving from before screen to r. of screen)

05:47 ...um, if I wanted to say "The number, (whatever the number is) squared is",...

(slides pointing r. l. finger to same line, pausing between "NUMBER" and "SQUARED" during the phrase "... (whatever the number is) ...")

...then I would have to reorganize my thinking because I haven't...

(circles REQUEST on line 2 w. r. l. finger)

...given that number sort of a label; or a name.

06:00 I: Uh huh, uh huh. How would you do that; you might want to test this....

06:04 B: (laughs)

I'd better test this and make sure the thing works.

(1 sec. pause)

06:09 Um, so "SQ dot num"...

(types: SQ.NUM...)

I doubt it, ah,...

(types: ...(ret.)

GIVE ME A NUMBER:

THERE IS NO PROCEDURE SQ IN LINE

PRINT [THE NUMBER SQUARED IS] SQ REQUEST

AT LEVEL 1 OF SQ.NUM

...there's no procedure named "SQ".

06:14 What did I call it? Square?
 (types: ED(ret.))
 I guess I called it square.

I: Um...

06:18 B: Yes, I did.
 (moves cursor to line 2 of SQ.NUM and changes SQ to SQUARE)
 (types: (ctrl.)-C
 06:25 POTS(ret.)
 TO SQUARE :NUM
 TO SQ.NUM

B: There. Yeah I called it "Square". So "SQ dot one",...

06:30 (types: SQ.NUM)
 GIVE ME A NUMBER: ...)

..."Give me a number", I don't know, twelve,...

06:33 (types: ...12(ret.))
 THE NUMBER SQUARED IS 144)
 ..."The number squared is 144".

I: Okay.

06:39 B: Okay?

I: Let me, some, there's a lot of...(LAUGHS), I'd love to have branching videotapes so we could pursue some of these branches you're talking about, um...

06:43 B: Oh, I know...

I: You got an error message there, I think the error message was something like "There is no procedure SQ in line something of SQ dot NUM, and you..."

06:56 B: Do you want me to go, I can go make the error again, if you want.

I: Well, I think,...

07:00 ...does that sound right to you?

07:04 B: (types: ED(ret.))

I: Um, or maybe, or just as you "Make" it if you talk about, I mean you quickly...

B: (moves cursor to 2nd line and changes SQUARE to SQ)

07:11 (types: (ctrl.)-C)

I: It looked like that, ok.

07:12 B: Right, so I give "SQ dot num"...
 (types: SQ.NUM(ret.)
 (duplicates earlier error message)
 ...and it said "Give me a number there is no procedure named "SQ in
 line print" It tells me what line the error is in...
 (pointing to the middle of the error message)

I: Uh huh,

07:20 B: And at what level that's in case I made a recursive procedure
 that's in some other level down there or whether it's called a
 subprocedure...
 (pointing to the following line of the error message)

I: You know this version of LOGO very well.

B: (laughs)

07:33 I: Um, but if you could, you then asked about what you called
 another procedure you'd written and you did a "POTS", what was, um;
 do you remember anything of the thought processes that very quickly
 led you to fix something that made everything work?

07:50 B: Oh, I mean at this point this is sort of, um, rote, I know it
 says "There is no procedure named Square", it,...
 (points to first line of error message)

08:00 ...I know that, I'm trying to call a procedure by a name, and I don't
 have a procedure by that name. So by saying "Print out titles",...
 (types: POTS(ret.))

08:11 ...I get the titles of the names of the procedures I have, and so I
 can look at that and I can say, oh yeah, I don't have one called "SQ"
 I remember what I did was I called it "Square".

I: Uh huh. And you recognized right away that the, well the line is
 printed out there for you...

08:22 B: Right, right,...

I: Print sentence...

08:25 B: ...but, I mean, I didn't even look at the rest of the line, I
 just said when it said "There is no procedure named SQ" I knew
 automatically that I made, that I could call it "Square" instead of
 "SQ".

I: Yeah, yeah.

08:37 B: 'Course there's not a lot of stuff in my workspace, so, that was
 easy.

I: Okay. Well thanks, that's helpful to have you talk about; um,...
 what about any of these other branches, any of them that are worth;
 uh, noteworthy...

08:49 B: Well, so, for example, suppose, um, instead of it saying um,
"The number squared is such and such" that I wanted it to say, um,

09:00 "The number twelve squared is 144"?

COMMENT: Had this line in the problem been worded this way, B. would
apparently have selected this alternative. While this seems a subtle
distinction in execution, B. recognizes this final declarative sentence
would strongly favor one approach to coding over another.

I: Um hum...

09:04 B:

(types: ED)

Then, probably what I would do since I like having a little
procedure "Square" around as a tool, um, is I'd probably do something
like, um, "Make quote number first request",...

(moves cursor just r. of SQ in line 2 of SQ.NUM)

(deletes SQ and inserts: MAKE "NUMBER FIRST (ret.)

In front of REQUEST on that line;)

(Procedure now reads:

TO SQ.NUM

PRINT [GIVE ME A NUMBER:]

MAKE "NUMBER FIRST REQUEST

PRINT SENTENCE (etc.))

...first because request; you really want to know all this?

I: Yes, yeah.

09:25 B: "Request" um, will output a list and I, a list of one, ah, it'll
output a list, since what I'm doing is typing in number, that's a one
element list. I need to get that number out of the list so I say
"First of request" and that will, the first of course in a one
element list is that element. So I want this...

(circles "NUMBER w. r. I. finger)

...to be, ah, a number. Of course I don't need to do that here
because it's not going to know that anyway;...

(points to [THE NUMBER SQUARED IS] w. r. I. finger)

09:58 ...when I put it; but I put it in; its just interesting but, um...

I: When you print it.

B: Yeah. So...

I: It won't matter if it's a list or a number?

10:08 B: Right. I have to put a parenthesis here,...

(inserts "(" before SENTENCE in line 3 of SQ.NUM)

...because I'm now going to have more than two inputs to sentence,
sentence is going to take, ah, well, other than two inputs, it's
going to take three in this case.

I: Um hum.

10:20 B: So I'm going to print a sentence out of the number sq... "The number"...

(moves cursor to r. of "[THE NUMBER" and inserts "]" ")

...dots number",...

(inserts: ...:NUMBER)

...okay, because my number will be in that variable...

(pointing w. vague waving motion of r. l. finger, approx. 18 in. In front of screen)

I: Um hum.

10:32 B: Um, then I guess I'm having four inputs to sentence,...

(types: "[" between NUMBER(space) and SQUARED on line 3 of SQ.NUM)

..."SQUARED IS",...

(moves cursor r. to SQ on that line)

...and now here,...

(points w. r. l. finger to FIRST REQUEST in line 3)

...If I leave this here "Request" is going to ask me,...It's going to wait for me to type in something again, I already have, have my something...

(adds "ARE" to r. of SQ)

(deletes FIRST REQUEST)

...and it's "dots number".

(inserts: :NUMBER))

So that would be my new version and...

11:00 ...of course up here, just to make things nice, I'd say something like cleartext.

(inserts: CLEARTEXT as new line 1)

(proc. now reads:

TO SQ.NUM

CLEARTEXT

PRINT [GIVE ME A NUMBER:]

MAKE "NUMBER FIRST REQUEST

PRINT (SENTENCE [THE NUMBER] :NUMBER [S!

QUARED IS SQUARE :NUMBER)

END

I: Um hum.

B: You want to see it?

I: Yeah.

B: (types: (ctrl.)-C)

Alright, "SQ NUM",...

(types: SQ.NUM(ret.)

GIVE ME A NUMBER:)

"Give me a number", twelve,...

(types: 12(ret.)

THE NUMBER 12 SQUARED IS 144)

"The number twelve squared is 144".

I: I see, I see. Do you use the term "GREEDY" for procedures that use parentheses, the ACORN manual uses the term, calls those "GREEDY" procedures...

11:25 B: Well, I wouldn't call them "GREEDY" because, um, in actuality, um, primitives(?) that take parentheses mean that you use parentheses any time the number of inputs is anything other than the default number. So for example, I could use sentence with one input, and in that case I would still need parentheses. I can use it with no inputs.

I: Um hum.

12:00 B: I don't know why you'd want to but, very often times when you'd only use it with one input, and so that isn't really being 'greedy' that's kind of the other side of greediness...

I: Um hum, yeah, that's true, that's true, um, but not all procedures can be so parenthesized so, to operate that flexibly.

B: Right, right.

I: Okay. Um, anything else?

B: Nope. You have another problem?

I: (guffaws)

B: Alright, I'm going to say goodbye, you want me to say goodbye, or do you want me to leave all my stuff here?

I: Um, whatever you like.

B: Going to lend my my "SQ" procedure again?

I: No, none of these should use each other, um, well thank you, 'course you might find a way to use that "Square" procedure again.

Appendix F
R, Problem A-2

- 00:00 R: Problem A-2, alternate Problem A-2. Write a procedure that first prints the message "Give me a number" and then prints "The number squared is" followed by the square of the number supplied by the person using the program. As an example, after the program is used once, the screen might look like this: "Give me a number...
...2...The number squared is...4".
- 00:32 I: Okay, I should mention and probably should specify in this sheet, or should have specified on this sheet that "Give me a number" is printed by the computer. The "2" would be printed by the user; would appear, you know, ah...
- R: Uh huh.
- I:...as a result of what the user did. The computer would print the next two lines, "the number squared is"...
- 00:48 R: (nods his head up and down)
Okay.
- I: ...and "4".
- 00:50 R: Okay. So first we'll;
00:52 (types: TO SQUARE)
- I: You say "TO SQUARE", can I stop you for just a second?
- 01:01 R: Yeah?
- I: I'm interested in something, you read the problem, I would say going over it carefully, I think, um, and very quickly, in all cases, I believe, moved into typing something, um, can you tell me about that, was there any gap in time at all, was there an instantaneous...
- 01:24 R: Yeah, yeah, I thought about the crucial several lines that would do what the program use...was...
- I: What was that thought process like? Was it like, um, let me see where the few crucial lines... Or was it like, obviously the crucial lines are...
- 01:42 R: Yeah, like I sort of dismissed the print and input parts because those weren't very hard to do. And then I focused on like what would actually accomplish what the procedure is supposed to accomplish.
- I: Uh huh, and what were the, maybe, you haven't written it yet, but what were the crucial, what did you decide were the crucial...

02:03 R: Okay, the crucial line would be what...the one that printed the square of the number, because all the other lines were just "PRINT" or Input..., or Inputting the...that number...

I: Could you perhaps, and I'll stop disturbing you from, from doing the solution, could you perhaps talk about the crucial;.. it's a wonderful idea, I'm really glad I asked. And if you could maybe point to it when you hit the crucial part of the problem.

02:27 R: Okay.

I: Very interesting; um, well, I'll shut up for, but, you know, that idea of, there, locating a crucial part of the problem is a fascinating idea.

02:36 R: Alright.

I: Hum... 'cause, you know, well I should shut up...

02:43 R: (laughs)
Okay.
(briefly looks down at the problem).

I: I'd love to talk to you a little bit about this after we're done.

02:45 R: (types:
PRINT ...)
Sure.

I: By the way, I should have mentioned earlier that I'm,...

02:48 R: (looks down at written problem for 1.5 sec.)
(types:
...[GIVE ME A NUMBER] (ret.))

I: ...while I'm, you know, particularly trying to avoid certain sorts of questions of discussions, um...

02:51 R:
(types:
MAKE...
(looks down at written problem for 1.5 seconds)
02:54 ..."NUM RW..."

I: ...I don't know whether that's come up with us or not...

R: (types:
...(ret.))
(turns to look at I for 17 secs.)

03:04 I: ...Um, I'm very happy to talk and would love and very much enjoy, by the way, talking with you about any of this, including your own, you know, personal reaction to it if...

03:13 R: Okay.

I: ...ah...I would find that very, I would just be very interested in that and your're welcome, you know, please keep that in mind, um.

03:23 R: Okay.
(turns toward screen; 3 second pause, looking at screen)
So...

03:26 (types: PRINT [THE NUMBER SQUARED IS...] (ret.))

I: Allright, you typed "To square print give me a number" then "Make quotes NUM RW" then "Print the number squared is", dots and a bracket, then "PRINT..."

03:43 R: (types: PRINT...)
Okay, now this is the crucial part.
(types: ... :A*:A ...)

I: "Print colon A times colon A". Okay.

03:50 R: Which, um, each "colon A"...
(points with right thumb to first :A, then second)
(turns toward I)
...tells the computer to use the value of
the number which would be, in effect, that number times itself.

03:59 I: I see.

04:05 R: (types: ... (ret.)
END (ret.)
SQUARE DEFINED)

I: And you type "End", okay.

04:07 R: (types: SQUARE (ret.)
GIVE ME A NUMBER)

I: "SQUARE DEFINED, you type "Square", the program says "Give me a number"...

04:10 R: (types: 4 (ret.)
THE NUMBER SQUARED IS...
1521)
Hmm.

I: You type "4", and it says "The number squared is 1521".

04:17 R: Which is because I was careless...
(points with right thumb to line 4, then line 2; back and forth

several times)
...and changed the name of my variable in between without thinking about it.

I: Okay, pointing at "MAKE NUM READWORD" and "PRINT DOTS A TIMES DOTS A"?

04:26 R: (types: ED [SQUARE]
Yeah, I should have used either "NUM" or "A", but not both.

I: Hmm. Okay.

04:34 R: (moves cursor down to line 4)
I was just thinking about what you said.
(changes line 4 to read:

PRINT :NUM * :NUM)

I: I should keep my mouth shut probably (laughs).

04:42 R: (types:

(ctrl.)C
SQUARE

GIVE ME A NUMBER)

I: I'm sure, it's a lot easier to program than it is to program and listen to me babble on about this, eh?

04:47 R: (types: 7(ret.)
THE NUMBER SQUARED IS...
49)

I: You say "Square give me a number 7" it says "The number squared is 49".

04:55 R: (types:

SQUARE

GIVE ME A NUMBER

14

THE NUMBER SQUARED IS...
196)

I: Square, give me a number, 14, the number squared is 196".

05:01 R: One more.
(types:

SQUARE

GIVE ME A NUMBER

25

THE NUMBER SQUARED IS...
625)

I: "Square, give me a number, 25, the number squared is 625".

05:08 R: Okay.

I: Why the three tests?

05:10 R: Oh, just checking, um, I was sure it worked before, I was just, like, playing around with things I didn't know.

I: Um hum. Do you usually check?...

05:19 R: Yes...

I: As you go through a process?

R: ...to make sure, like, it would work in any case.

I: Okay.

Appendix G
M, Problem A-2

0:00:00 I: (Reads the written problem)

00:35 Ok? So is it clear which parts the person types in and which parts the person types in here? The computer types...

00:44 M: Ah, you type in the name of the procedure and then the computer takes over.

I: Uh-huh. And then the computer types, "Give me a number", ok, and you have to type...

00:56 M: The number.

I: ...the person there has to type, the number, and then the square of it is four.

01:02 M: Um-hmm...Ok.

(4 sec. pause, looking at screen)

01:07 (types: TO)

I: Ah, one thing you should do, there, is type "GOODBYE", to clear the screen; ...ah, clear the procedures.

01:15 M: (erases TO and types GOODBY(ret.))

(M corrects this misspelling to GOODBYE to restart Logo).

01:44 Ah, we'll call this, "To num"...
(types:

TO NUM(ret.))

01:56 (looks down to written problem)
Ok;

(4 sec. pause, looking at problem)

02:02 Well first we want to give it a print...
(types: PR...)

(turns to I.)

Now if you want it to print a list you put it in parentheses? Ah, brackets, I mean?

I: Brackets, right.

02:12 M: (types:

...[...)

(turns to keyboard)

(4 sec. pause, eyes wandering over keyboard)

Hmm..."G"

I: You don't have to type the whole thing if you don't want. You can just type "Give number"...

M: Thank you.

(types:

...GIVE NUMBER?](ret.))

02:39 O.K.(7 sec. pause, looking at the screen, then down at the written problem)

02:46 Now to take this number and to reprint it would you put "Output dots x"; ah... output...ah...how did you do that, again? I forgot."
(laughs)

COMMENT: This is an example of the same "language confound" for OUTPUT observed in other protocols, (OUTPUT may mean "screen-output"). In other words, M is using the term "output" to mean "screen output", and uses this association as an (incorrect) formal interpretation of Logo's OUTPUT command.

03:05 I: Tell me more specifically. . .

.
(I tells M that she may ask questions freely)

03:21 ...So you're asking now?

03:23 M: Ah, I'm going to be given a number here;
(points to screen, line 1 of procedure).
I want to put that number in a variable, so I can use it whenever I want it, and the first way I want to use that variable is to reprint it back on the screen"

I: OK. OK. Um.

03:41 M: So do you use a MAKE statement?"

I: You can use a MAKE statement to create a variable, yeah.

03:47 M: But I wanna output it. So the first thing I want to do is output it back to the person. So if they type in a 2
(points above top l. of keyboard, then sweeps finger in big arc, to point to her right side, keyboard high)
I type back; give them back the 2. I guess if they do a 2
(points to top, left keyboard, then top, computer screen)
It will go anyway, won't it?"

I: It will go where?

04:03 M: If they type in 2,
(points to top left of keyboard)
It will be on the screen anyway.
(points to top left screen)

I: It will be on the screen yea. You don't have to echo the 2 back.

04:13 M: Ok. So I just want to make this...
(points to screen)
...a variable. So I can use a MAKE statement?

I: Yea.

04:22 M: (begins typing)

Now I've probably forgotten how to use it correctly at this point.
(types:

MAKE ...)

Make...now let's see.

(types: ..."X)

04:42 (3 sec. pause, looking at the screen)

Now how do I make make dots x that number,...

(points to screen w. l. l. finger)

...that they just typed in?

I: Which number.

04:47 M: You say give number, ...

(points to screen with left index finger)

...they give you a number,...

(points to mid-air, 12 inches in front of screen)

...how do you make dots x...

(points to screen w. l. l. finger and slides finger to r.)

...the number which they give you?

COMMENT: M. seems here to show an excellent understanding of the function of a line that assigns user-input to a variable, (including the notion of a variable as an "alias", standing for a value), even though she does not at this point remember the last part of it, the REQUEST statement, nor does she yet demonstrate a full understanding of the concepts related to it.

I: Ok, you remember; (sighs) thats...If you ask me...

04:57 M: They give you inputs....

COMMENT: This comparison to parameter input, a construct more familiar to M., is the first indication that she has some understanding of the function of her "missing word". At the heart of the comparison between parameter input and the REQUEST statement is the fact that both techniques allow a programmer to bring into his program information from the outside.

I:...a more specific question about something we talked about before... They give you inputs. So would it be something that you define in the top line as inputs?

05:09 M: (5 sec. pause, looking at screen)

Um...I guess so.

I: Ok. So maybe...what's...tell me; tell me what it is you're looking for in a little more detail...I know you're looking for something to just get that number, but if you could explain to me;...I think I know something of what you're looking; you're looking for something to type right now, right?

05:38 M: Um hm.

I: Should it be one thing you type, or a long line of things?

05:44 M: Its probably 2 or 3 words.

I: Two or three words?

05:50 M: You write give number and they give you a number, I want to know how to transfer the number that they give you, such that that number, whatever it may be, the computer knows that number is now x.

I: Right. Ok, there was a command I showed you at the beginning, when I talked about this thing, do you remember what that was? That you could use to have someone specifically type in a number?

06:22 M: Well MAKE makes the variable ...

I: That's right.

06:25 M: ...in the middle, and that's what I'm doing. An interactive variable, yea?

I: Uh-huh. Well, if I'd asked you, "MAKE"...if I'd asked you to type to the person, give me a number, and then have the program print the number 10, no matter what number they gave you print the number 10, you could say make x what?

06:49 M: Um...(8 sec. pause, looking at the screen)
MAKE X PRINT, or MAKE X NUMBER? I don't know. I forgot. I've forgotten what you did, you did this exact same thing as an example, you said make, um, PRINT, um, WRITE YOUR NAME, and then, you know, name, MAKE NAME the variable, or whatever; I mean you did the same thing. If I had written it down, I would have it now. (laughs)

COMMENT: In an apparent reference to the line she had seen in the introductory presentation, (MAKE "PLAYER1 REQUEST), M. correctly uses the term "interactive variable" (at 6:25) and is searching for that idiom, has at this point shown no clear understanding of the functional role of the part of the idiom that she cannot recall (REQUEST). Notice that the use of quotes as punctuation within the idiom poses no apparent problem at this time (although see 11:12).

I: There was a command to read something from outside of the program back into it...

07:32 M: Right.

I: ...and that was request, R, Q.

07:35 M: OK. That was the only one that I didn't really know at the time. (types ...RQ)

- .
- . (M. makes several procedural suggestions, including a suggestion to
- . provide subjects with a reference of Logo commands, with examples).

NOTE: During M's interview, subjects were given an oral review of Logo, and there was neither an instructional videotape nor a written script for her to refer to.

08:25 I: You hadn't seen REQUEST before, right?

08:27 M: I hadn't seen request before.

COMMENT: M. must mean that she has not seen and used REQUEST before this interview (she has already acknowledged that REQUEST was part of the instructional example given at the start of this session).

I: That's a good suggestion, that's an issue I'm gonna have to think about, because I don't want to make this a quiz of memory. But let me think about that.

08:42 M: Because I just learned MAKE. I mean, I wouldn't have known that one either.

I: You just learned that when you sat down?

08:49 M: No, um, I learned that just the other day in making my final project.

I: What were you doing?

08:58 M: What did I do?

I: Yea, what were you doing when you used MAKE?

09:02 M: I used it several times. I was using the law of sines and the law of cosines. And I needed to take my answer from the law of cosines and put it in the law of sines, and I needed it for a variable later on. But I never would have remembered from what you did at the beginning to use make here, except if I had used it before.

(pointing quickly to the center of the screen)

COMMENT: The experience to which M refers helps to explain her clear understanding of the concepts of variable-as-container and variable-as-alias. This raises some important questions about the role of rehearsal in the development of predictive theories.

09:26

(Discussion of methodology, including the idea of using an instructional videotape)

09:52 Ok. Now, what this will do; now tell me, what this will do is, the REQUEST...

(points with left index finger to line 2, then slides left index finger up to line 1)

...will, um, take that number?"

(pointing to line 2 with left index finger)

NOTE: By pointing first at the REQUEST line and the PR [NUMBER?] line, and alternating between those two lines several times, M seems to be referring to the number that will eventually be input by the user. It is not clear why she does not point to the X in the same line as the REQUEST; she may be attributing meaning to the data being printed, (i.e., assuming the use of "NUMBER?" as an argument to PRINT will be semantically interpreted by Logo, a phenomenon was that was observed later in M's work in a follow-up problem,

Is...very similar to what you just had.

12:12 M: (moves cursor back to next, blank line)
Ok. Dots x.

.
.
.
.
.
.

(M spends some time searching the keyboard, first for ":", then for
times ("*") (with questions for I.)).

(types:

:X * :X)

12:48 OK. (5 second pause, looking at screen) So MAKE x; does this have to
be MAKE...

(pointing to screen with left index finger)

...dots x? I mean make quotes dots X? Or is it just x?

13:01 I: No, that's just fine.

13:04 M: Just fine, Ok. Um...

(2 second pause, looking at the screen)

...and then I get an answer for that, and then I
need to output that answer.

(2 second pause, looking at the screen)

Um...

(7 second pause, looking at the screen)

13:21 ...so first I have to do this

(pointing down to the written problem)

(2 second pause, looking at problem)

13:26 I wanna go up...

(with some difficulty, M. puts blank line between lines 2 and 3)

13:46 Ok. Now I wanna do, "Print", P, R.

(types: PR...)

I: You can just type NUMBER, if you want.

COMMENT: There appears to be established here a clear agreement that "NUMBER"
will stand as an abbreviation for "THE NUMBER SQUARED IS..." (see 17:45)

M: (types:

...[NUMBER])

14:14 ..well, then I don't really need this bracket, because it's not a list.

I: Well...that's ok. It won't hurt.

14:21 M: Um; and then ; um...

(moves cursor down to next line)

14:32 Now; I want to output the answer, so...

14:43 (types OP before :X * :X)

If I put OUTPUT that, will it output this,...

(pointing to the screen)

...or will it output dot x times dot x?

I: Will it output 4 or will it output dot x times dot x, or will it output?

15:01 M: If the number was two,...

(points to the screen)

...would it output 4?

(points down and towards center of screen)

COMMENT: The first alternative is that the procedure would print, literally, :X * :X (see 15:38).

I: Well I don't want to run it for you.

15:09 M: Try it.

I: Well, yeah, why don't you just try it.

15:13 M: (types: (ctrl.-C)

NUM DEFINED

NUM...)

I: Your question was, would it output 2 or would it output 4?

15:23 M: (types: ...(ret.)

GIVE NUMBER

2(ret.)

* DOESN'T LIKE [2] AS INPUT IN LINE

OP :X * :X

AT LEVEL 1 OF NUM)

I: Hang on a second. Your question, before, you thought it might output 4 but what was the other thing it might have output if you gave it a 2?

15:38 M: Um...well, actually it wouldn't have, it might have outputted dot x times dot x.

I: You mean just put those letters on the screen?

15:53 M: Right. But I realized it wouldn't 'cause it's not quotes.

.

.

.

(I. explains that RQ "always puts brackets around" what it takes in; suggests, as a fix to this problem with multiplication, putting FIRST just before RQ on line 2. M. makes that correction)

COMMENT: In the Logo dialect used during the design phase of this study (and used in several of these interviews), input could be taken in as a word with the READWORD command whereas the dialect used here permits only list-input. To adapt to this unavoidable complication, the researcher adapted the instructional presentation depending on the dialect being used and determined to avoid the issue of "preparing" number input in

this problem by immediately suggesting the "fix" suggested above.

.
.
.

17:33 M: Now let's try.

(types:

```
                NUM(ret.)  
GIVE NUMBER  
                2  
NUMBER  
RESULT: 4      )
```

17:45 I: "Number", "THE NUMBER IS" and it says, "RESULT: 4". OK? You happy with that?

17:50 M: Sure.

COMMENT: The last line is technically not correct since "RESULT: 4" being in reality a gently worded error message. Had the procedure contained additional lines they would not have been executed, making this more obvious. Unfortunately, M's missing this point probably only reinforces her misconception that OUTPUT-means-"screen-output".

I: Fine.

Appendix H
A, Problem A-2

-1:10 (I. shows Andrea the Instruction Script, not available during the her previous session).

00:00 I: Reads the problem ("Write a procedure that first prints the message, , 'Give me a number'...)

00:37 I: Allright so what do you think?

A: So what do I think...um

00:45 (types: TO A (ret.)
...ok...
(types: PRINT...)

00:55 A: Um...Your saying print; I don't know what punctuation to use.

I: Ok, If you looked back on the sheet you'd see brackets...

A: Brackets.

I: ...print uses brackets as punctuation.

01:07 A: So, (looks at u.r. keyboard) they're up here, right?

I: Unshifted, yea.

01:10 A: (types [GIVE ME A NUMBER...])

01:23 Can I put please? (smiles) Just make it a polite computer.
(types ...PLEASE] (ret.))

01:31 (4 sec. pause, looking at screen)

01:35 A: Can you; Is there such a thing, you know, like how you have output
(waves with hand at center of screen)
...In a program, so you can have Input?

I: Um-hm

01:46 A: ...I don't know if that would do it.

I: Have the word Input?

A: Yea

I: Well there certainly is a word output, um; there's not really a word input; um...so that doesn't; where does the word input come from; why did think of the word input?

02:07 A: Just because I remember that there's an output before...

I: Uh-huh.

02:11 A: ...saying that like if an if, IN...AN...IF statement or whatever saying if its built on an output, saying that...I was just wondering if you could have an input that's just the opposite.

02:25 I: Uh-huh, uh-huh; You know there's another computer language that has an input statement, uh, that's BASIC that has an input statement. You ever done BASIC?

A: Yea.

Comment: The idea for a primitive INPUT command may come from any of a number of sources, or from a combination thereof. The word "input" in the problem statement is the probably the most direct influence, probably interacting with A.'s experience with an INPUT statement in BASIC (see the following interaction).

I: You ever do an input statement in BASIC?

02:40 A: Yea, and also I've done Logo on Atari.

I: Did that have anything like an input statement

02:51 A: Yea, I think so...I think so. (Said while staring at the screen).

I: Well, how would you write it if you had an input statement? Or, you know, alternatively if you know another way.

03:00 A: Well, cause I just wanna say that its gonna get a variable. That's what I want to tell it.

Comment: "It (the computer) is gonna get a variable", (what it actually gets from the user a value to be stored in the variable) shows that A. does not make a clear distinction between variable name and value. On the other hand, A. seems to understand very well the roles of the computer, prompting for input and using a variable to hold same, and the user ("the person") as the supplier of this input (see the following).

I: What's the it that's gonna get a variable?

03:11 A: The computer's gonna get a variable

I: The computer's gonna get a variable? From where?

03:17 A: From the person typing it in.

I: Ok. The computer's gonna get a variable from the person typing it in.

A: Um-hm

03:25 I: Now you're inside procedure A

A: Um-hm

I: What, uh.. you know what...why are you in procedure a, I mean why

not, uh... Is there a reason why you're in procedure A?

03:43 A: What, you mean, like, the name?

I: Yea, or why did you write a procedure, you know.

03:50 A: Just because, so, like, you could have a procedure and you could do with all types of different numbers.

04:00 I: So this has to take place inside a procedure, huh?

04:03 A: Well yea, also this says "write a procedure".

I: Yea, thats...and...that's a good clue.

A: Yea (laughs).

04:10 I: Alright, well, what about this input statement? There's no input statement, in fact I don't think Atari Logo's got an input; do you remember how to use an input in BASIC?

04:19 A: I don't know. I don't want to think about it (laughs) but, um... (looks at screen for 5 secs.).

I: Would it help to look at these things?

04:30 A: Yea, it would. (takes a copy of the Script)
(Looking at page 1 of Script)
Print...ok...

(turns to page 2 of Script)

04:50 A: What...ok...One thing that I could try to do is, um...it probably won't work ...

04:58 (types: PRINT :S (ret.))
but I can't think of any way other to do it, because, um...well I could probably tell you what the computer's gonna say to me, but I don't care. (Smiles, briefly)

05:10 (types: PRINT [THE NUMBER SQUARED IS]
PRINT :S*2)

(Procedure now reads:

TO A

PRINT [GIVE ME A NUMBER PLEASE]

PRINT :S

PRINT [THE NUMBER SQUARED IS]

PRINT :S*2

05:55 I:Ok, well what is...just tell me what the S star 2 means.

05:59 A: It means to, like, times it by 2.

Comment: "It" must describe the value input by the user. Her use of multiplication rather than expotentialtion is a conceptual error, not a mistaking of * as an sign for expotentialtion.

I: Ok. Good, yea, that's right

06:08 A: (types: (ctrl)-C)
 (types: A)
 GIVE ME A NUMBER PLEASE)
 (types: 2
 THERE IS NO NAME S IN LINE
 PRINT :S AT LEVEL 1 OF A)

I: It says, "give me a number please...There is no name S in 'Print S' at level one of A". Why does it say that?

06:21 A: Because; I KNEW it was gonna say this, because I don't have, like in the definition, I don't have dots s, so it doesn't know, you know it ... doesn't know.

Comment: Here begins a notable SHIFT from the immediate goal of exhaustively testing keywords to fill the role that she has so eloquently described to the development of heuristics to aid her search through a review of descriptive knowledge about this aspect of the problem.

I: Doesn't know what? Fill in the "you know".

06:41 A: (Smiles) It doesn't know what that dots s is, because I haven't told it the right thing.

06:56 I: So what do you need in your program? What would fix that?

07:04 A: Um...
 (4 sec. pause)
 (Types: ED (ret.))

07:11 I need something right there (points to center of screen), instead of "print", I just need, I don't know...I mean...

Comment: A here demonstrates a functional understanding of a line to accept user input, at the location where one would expect to find it. This is probably based on her experience with an INPUT statement in BASIC, notable in that it indicates some TRANSFER of a concept of variable from one computer language to another.

I: I mean, what...well, what would that thing do, that you put there instead; so you mean the line instead of print s, you'd put something there instead of print s?

07:27 A: Yea, it probably, it might still have an s, but, like, it would get it so the computer would take the number, and so you wouldn't have to; it was like, if you had a variable (points to top r. of screen)

I: Uh-huh

A: ...an' you could say; I could say "to A dots s", but then; when it says "a defined" and you had to type in a, you had to type in a number as well, and that's not what it wants....

Comment: Again, A. seems honed in on the problem, (here explaining why adding a parameter to A would not be appropriate), but neither the instructional

presentation nor the Script are apparently sufficient for her to refine an implementation plan.

I: Uh-huh, uh-huh

07:50 A: ...It wants it to print out give me a number please, and then have you be able to, like, type in a number,...so, um...(8 sec. pause)

08:03 I: Ok, so you need something to let you get a number; Is that what you said?

A: Yea.

I: And it would have an s in it somewhere, in the line it would have an s in it somewhere; and it would remember; what; what would; lets say the person typed in 2, what would that be, you know, what would that have to do with the rest of the program. Lets say the person typed in 2.

08:27 A: OK, it said, give me a number please and it typed in 2?

I: Yea

A: Then it'd print, (points to screen) "the number squared is" and it would take the 2 times 2...

I: Uh-huh

A: ...which is four.

I: Right.

08:41 A: (Looks at Script). This is to type lines and stuff.
(10 sec. pause, reading Script).

08:55 Does it, like...does it understand when you say like... does it understand words like exponent and stuff like that?

Comment: We discount this question of a primitive as a distraction, possible due to something A. saw during her examination of the Script.

09:06 I: No. You have to type that in.

A: Ok, so it doesn't...ok.

I: Those are procedures you'd have to type in.

09:11 A: Ok (looks at Script for 7 secs.)

09:16 I: Um...airlight; well...there's a section here, (takes Script from A.) that, you know I think you've described at least a lot of the behavior of the command that you need. (Sorting Script, then hands it back to A). Alright, just; why don't you look at the section from here down.

09:48 A: (Reading Script)
10:12 Oh, ok! Ok I remember. Ok (smiles), you see 't's
because i had never seen this before.

I: Yea, I know.

10:23 A: (deletes PRINT :S)
Ok. Then i can say, like...
(types, in its place:
MAKE ...)
make; do i have to use? (looks down at Script); yea.
(types: ... " ...)

Comment: The quotes are seen as part of a frame for accepting input from the user. The form of this frame is:

MAKE "(variable name) REQUEST

10:45 (stares at screen; blows hair out of eyes)
(2 sec pause) So, um...(3 sec. pause) Its still a variable, though;
right? So why don't I...do this, If not i can just...(unintelligable)
(types: ...:S RQ)

Comment: A. sees DOTS (:) as part of the variable name, rather than as an abbreviation of 'THING "'. This generates a bug, (":S rather than "S) that hinders A.'s efforts for the remainder of this session.

I: Ok, what have you just typed, there?

A: Ok, i'm saying, alright, to make dots s, which is a variable, um,
a request.

I: Ok, but you've got quotes in front of the colon, right?

A: Yeah.

I: ...which is fine, you say make quotes dots s; rq is request,
now what does request do?

A: Its, like, its; Its asking the computer to the; or, you know, its;
its an input kind of thing.

11:34 I: Ok.

A: ...Indefinite Its says, like, player1; but you could; but
I mean; put anything.

I: Yeah. So that's what a name is?

A: Yeah.

I: So, for; In a name you can use any word you want to.

A: Yeah.

17:44 I: And in a variable; how does that work.

A: Then you usually say, dots. Or you could have a colon, and its, like a letter.

I: Uh huh.

17:55 A: (looks at screen; pause)
...Um...

18:09 I: Does a name start with any punctuation or end with any punctuation?

A: No (looks down at script)

I: Ok, like, for examp'; Is there a name in here, in INQUIRE?

A: Yeah.

18:21 I: What's; what is it?

A: Its, um, in the make statement...

I: Yeah

A: ...It...It just says...
(short pause, looking at script)
, well it has a quote around it, one, but, like...

18:32 I: Which, which; read it to me. MAKE...

A: Quote Player1.

I: Ok, so; which is the name

A: Player1

I: Player1. Ok. So, player1 is a name, and its got quotes in front of it. And is there a variable in there?

18:45 A: No. There isn't
(moves cursor to line 2)
So I'm gonna eat this.
(changes line 2 to read, MAKE "S RQ)

I: You take out the colon from the make statement.

18:56 A: (moves to line 4, removes : so that line reads: MAKE S * 2)
...take out that.
(types: ctrl-C

A

GIVE ME A NUMBER, PLEASE

19:06 I: Alright, and you took the; now you've got the bottom so that the
...the make statement in there says, "make quotes s something"...

A: (Types: 2

THERE IS NO PROCEDURE NAMED S IN LINE
MAKE S * 2

I: ...and then the bottom line says "make no-quotes s dots s time 2",
I think?

19:23 A: Um; well, I took out the dots.

I: Oh, just s; MAKE S times 2

A: (Smiling); Yeah (under her breath)

I: Allright, so it says give me a number please, and you type 2, it
says, "the number squared is", and "there is no procedure s in line
make s times 2.

19:40 A: (types: ED) Its just this stupid (unintelligible)

I: (unintelligible)

A: (moves cursor to 4th line)

(unintelligible) Uh...(looks at script)...um (looks at screen; moves
lips).

20:05 I: What should that line do? I mean 's...

A: (Moves cursor r. 4 spaces)

It should; ah...

(looks down; adds ' ' after MAKE)

20:26 It should...this line (points to upper 1/3 of screen) should take the
number you put in up there and times it by two.

(moves cursor to right side of line 4)

But its not because it doesn't like me.

I: Uh huh

A: (unintelligible)

(adds RQ to end of line, to read:

MAKE "S * 2 RQ)

(types: ctrl-C

A

GIVE ME A NUMBER, PLEASE)

(quietly) A; Give me a number please.

(types: 6

THE NUMBER SQUARE IS

* DOESN'T LIKE S AS INPUT IN LINE

MAKE "S * 2 RQ)

20:54 I: Ok, It says give me a number please, type 6, It says the number
square is, and it says "star doesn't like s as input in line
MAKE "S * 2 RQ".

A: (types ED)
 Ummm...
 (moves down to 4th line; deletes S)
 21:19 Oops! I really didn't mean to do that. I wanted to get um closer.
 (replaces the S)
 I don't know what to do.
 (deletes the spaces on either side of the *)
 (line 4 now reads:
 MAKE "S*2 RQ)
 (types: ctrl-C
 A
 GIVE ME A NUMBER PLEASE
 3
 THE NUMBER SQUARED IS
 (pause in program)
 (2 sec. pause, lips move; laughs)

I: Allright, It says "Give me a number please", you type 3, It types
 "the number square is",
 What's going on here?

21:44 A: Because its just like the other one, I mean, its just like
 (points to top 1/3 of screen)
 ...wait a ml'
 (types: ED
 ?)
 Ohps. Excuse me.

I: What?

A: (types: ED)
 21:57 No! because this is just; this (points to top 1/3 screen) line is
 just like that line (points up 1-2 inches), Its waiting for...

I: Its just like the make s request

A: Yes

I: Its waiting for what?

A: Its waiting for somebody to type in a number
 (moves cursor down to 4th line)
 , and that's not what its supposed to do.

I: Um hm.

22:14 A: But it; It doesn't know what its s'posed to do.
 (Begins looking at script)
 And I don't know what its s'posed to do. Or I do, but...
 (still reading script)

I: What's, what's; or, you know what its supposed to do but you don't
 know what you're; how to do it in Logo, is that the...?

A: Yeah.

I: What's it supposed to do once you; something about the role of that line?

A: What, this line? (pointing to top 1/3 of screen) This line...

I: Yeah; the bottom line, just before the END, yeah.

22:37 A: This line is supposed to take the input up here (moves finger up, 1 to 2 in.), of the first make statement, and times it ... multiply it by 2, and print out the answer.

I: Ok.

22:52 A: (looks at screen)

22:57 And obviously,

(moves to end of line 4)

I knew this before, it wouldn't be request,

(deletes RQ).

because, ...um (stares at screen for 3 sec.)

23:07 (looks down at script for 3 sec.; up at screen for 4 sec.)

(yawns) ...um (looks down at script for 8 sec.)...

23:28 I don't know.

I: Do you want to, you know, work on it some more, or do you want to go on to the next one?

23:34 A: I don't know. I; I'd like to work on it but I; I don't know... (staring at screen)

23:39 I don't know what to do...

23:45 (Looks at script)

Um; because...

23:57 (looks up at screen) Ok.

(changes line 4 to:

OP S*2

ctrl-C

(laughs) I don't know, I just put that there, 's (shrugs)

24:19 I: Ok, OP?

A: (types: A

GIVE ME A NUMBER, PLEASE

I: So that's; that line says OP...

A: (types 3 ;

THE NUMBER SQUARE IS

THERE IS NO PROCEDURE S IN LINE

OP S*2

AT LEVEL 1 OF A)

I: ..., no colon s times 2 right?

A: 2...yea.

24:27 I: Alright, you type A, it says "give me a number please", 't sez 3, "the number squared is", and then it says "there is no procedure S...

A: S...

I: ...in line op s times 2".

A: (types ED)

24:36 I: Why does it think s is a procedure?

A: Um; because; it just does.
(moves cursor to 4th line, deletes OP)
No, I don't know.

I: Um hm

24:44 A: (blows hair out of eyes)
so...(3 sec. pause, looking at screen)
(changes line to RQ*2)
(laughs; shakes head l. to r.)
(unintelligible)
(types ctrl-C)

24:56 I: Alright, now you change that line to RQ * 2

A: (types A
GIVE ME A NUMBER, PLEASE)

24:58 Yeah. It won't work either, but I just...
(types: 5
THE NUMBER SQUARE IS ...)

25:01 I: Give me a number, 5, you say, the number square is...

A: (4 sec. pause, looking at screen)
(types: 5)

I: You type 5, it says, "star doesn't like BRACKET 5 as input
in line, request times 2 at level 1 of a.

A: (types: ED)

25:18 I: Do you understand that? You know, that's something...

A: Yea.
I: ...that I don't know if I...Ok.

A: (moves cursor to line 4, deletes RQ)

25:24 I do, but, I mean, I just don't know what to do to...to make it
better; or to; you know; to make it correct

25:32 I: Ok. Do you want to just pass on this one? Let it; let it lie?
And if something comes to you, you can come back to it.

A: Yeah, I might as well. I mean, I don't want to, but...

25:43 I: Ok. Well its no; we'll be able to over these later, too.

BIBLIOGRAPHY

Adelson, B. "Problem solving and the development of abstract categories in programming languages", Memory and Cognition, 9, 422-433, 1981.

Allen, J. & Davis, R. "In praise of fingertips", unpublished paper, 1984.

Anderson, J. "Acquisition of Cognitive Skill", Psychological Review 89, 4:369-406, 1982.

Anderson, J., Farrell, R., Sauers, R. "Learning to Program in Lisp", Cognitive Science, 8, 87-129, 1984.

Bonar, J. & Soloway, E. "The Bridge From Non-Programmer to Programmer", Technical Report, University of Massachusetts, Amherst, 1983.

Bonar, J. & Soloway, E. "Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers", Human-Computer Interaction, Fall, 1985.

Brown, J.S. & VanLehn, K. "Towards a Generative Theory of 'Bugs'", Cognitive and Instructional Series # 2, Xerox Palo Alto Research Center, 1979.

Clement, J. "Quantitative Problem Solving Processes in Children", Doctoral Dissertation, University of Massachusetts, 1977.

Clement, J. "Cognitive Microanalysis: An Approach to Analyzing Intuitive Mathematical Reasoning Processes", Cognitive Process Research Group, 1980.

Clement, J., Lochhead, J. & Soloway, E. "Positive effects of computer programming on the students understanding of variables and equations", Proceedings of the Association for Computing Machinery, National Conference, 1980.

Clements, D., Gullo, F. "Effects of Computer Programming on Young Children's Cognition", Journal of Educational Psychology, 76, 6:1051-1058, 1984.

Confrey, J. "An Examination of the Conceptions of Mathematics of Young Women in High School", unpublished paper, 1984.

Davidson, L. "Logo Syntax: Another Story", (unpublished manuscript), 1986.

Dijkstra, E.W. "Goto statement considered harmful", Communications of the Association for Computing Machinery, 1968, 11, 3:147-148.

Driver, R., The Pupil as Scientist?, Open University Press, 1983.

duBoulay, B., O'shea, T. & Monk, J. "The black box inside the glass box: presenting computing concepts to novices", International Journal of Man-Machine Studies, 14, 237-249, 1981.

Duckworth, E. "The Having of Wonderful Ideas", in Piaget in the Classroom, (Schwebel, M. & Raph, J., Editors), Basic Books, 1973.

Dwyer, T. "Significance of Solo-mode Computing for Curriculum Design", in Computers in the Schools, Tutor, Tool, Tutee, (R. Taylor, Ed.), Teachers College Press, 1980.

Easley, J.A. "The Structured Paradigm in Protocol Analysis" Cognitive Process Instruction: Research on Teaching Thinking Skills, (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.

Eisenberg, M. & Peelle, H.A. "APL learning bugs", APL Quote Quad, 13, (3):11-16, 1983.

Elliott, P. "Computer 'glass-boxes' as advance organizers in mathematics instruction", International Journal of Mathematics in Science and Technology, 9, 1:79-87, 1978.

Floyd, R. "The Paradigms of Programming" Communications of the Association for Computing Machinery, 455-460, 1979.

Gardner, M. aha! Insight Scientific American Inc., 1978.

Gruber, H.E., "Courage and Cognitive Growth in Children and Scientists", in Piaget in the Classroom, (Schwebel, M. & Raph, J., Editors), Basic Books, 1973.

Harvey, B. Computer Science Logo Style: Intermediate Programming, MIT Press, 1985.

Hoc, J.M. "Developmental stages in learning to program" International Journal of Man-Machine Studies, 9, 87-105, 1977.

Inhelder, B. & Piaget, J. Logical Thinking from Childhood to Adolescence, Basic Books, 1958.

- Kamii, C. "Pedagogical Principles Derived from Piaget's Theory: Relevance for Educational Practice", in Piaget in the Classroom, (Schwebel, M. & Raph, J., Editors), Basic Books, 1973.
- Smith, A. Karmiloff & Inhelder, B., "If You Want to Get Ahead, Get a Theory", Cognition, 3, 195-212, 1975.
- Kuchemann, D. "Children's Understanding of Numerical Variables", Math in School, September, 1978.
- Kurland, M. & Pea, R. "Children's Mental Model of their Own Recursive LOGO Programs", Technical Report, Bank Street College, 1983.
- Lin, H., "Approaches to Clinical Research in Cognitive Process Instruction", in Cognitive Process Instruction: Research on Teaching Thinking Skills, (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.
- Lemos, R. "An Implementation of Structured Walk-throughs in Teaching COBOL Programming", Communications of the Association for Computing Machinery, 22, 6, 1979.
- Louie, S. "A Report of a Pilot Study", Tucson Learning Center, Tucson, Arizona, 1985.
- Luehrmann, A. "Should the Computer Teach the Student or Vice-versa?", in Computers in the Schools, Tutor, Tool, Tutee, (R. Taylor, Ed.), Teachers College Press, 1980.
- Mawbry, R., Clement, C., Pea, R. & Hawkins, J. "Structured Interviews on Children's Conceptions of Computers", Technical Report, Bank Street College, 1983.
- Mayer, R.E. "A Psychology of Learning BASIC" Communications of the Association for Computing Machinery, 22, 589-594, 1979.
- Mayer, R.E. "The Psychology of How Novices Learn Computer Programming" Computing Surveys, 13, (1), 1981.
- Miller, L.A. "Programming by Non-programmers" International Journal of Man-Machine Studies, 6, 237-260, 1974.
- Miller, M.L. "A Structured Planning and Debugging Environment for Elementary Programming", in Intelligent Tutoring Systems, (Sleeman, D. & Brown, J.S., Editors), Academic Press, 1982.
- Minsky, M. "Form and Content in Computer Science", Communications of the Association for Computing Machinery, 17, 2:197-215, 1970.

Murray, T. & Clement, J. "Progress Report: Evidence for Building Blocks Contributing to a Robust Concept of Variation and Covariation in Two-Variable Algebra Word Problems", Technical Report, Cognitive Process Research Group, 1986.

Newell & Simon, Human Problem Solving Prentice Hall, 1972.

Papert, S. "Teaching Children to be Mathematicians vs. Teaching Children About Mathematics", International Journal of Mathematics Education in Science & Technology, 1972.

Papert, S. Mindstorms Basic Books, 1980.

Pea, R. "LOGO Programming and Problem Solving", Technical Report, Bank Street College, 1983.

Pea, R. & Kurland, M. "On the Cognitive and Educational Benefits of Teaching Children Programming: a Critical Look", Technical Report, Bank Street College, 1984.

Peelle, H.A. "Learning Mathematics with Recursive Computer Programs", Journal of Computer-Based Instruction, 3, 3:97-102, 1977.

Peelle, H.A. "Alternative Algorithms in APL: Implications for Education", Proceedings, APL'80, 1980.

Polya, G., How to Solve It: A New Aspect of Mathematical Method, Princeton University Press, 1957.

Rodgers, J. "Teaching Beginners to Program: Some Cognitive Considerations", unpublished paper, University of Oregon.

Rosnick, P. & Clement, J. "Learning Without Understanding: The Effect of Tutoring Strategies on Algebra Misconceptions", Journal of Mathematical Behavior, 3, 1:3-27, 1980.

Rosnick, P. "Some Misconceptions Concerning the Concept of Variable", The Mathematics Teacher, September, 1981.

Schoenfeld, A. "Can Heuristics be Taught?", in Cognitive Process Instruction: Research on Teaching Thinking Skills, (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.

Schon, D., The Reflective Practitioner: How Professionals Think in Action, Basic Books, 1983.

Sheil, B.A. "Coping with Complexity" in Cognitive and Instructional Sciences Series, Xerox Palo Alto Research Center, 1981.

Silver, E., Branca, N., Adams, V. "Metacognition: The Missing Link in Problem Solving?", Proceedings of the 4th International Conference for the Psychology of Mathematics Education, 1980.

Sime, M.E., Green, T.R. & Guest, D.J. "Scope Marking in Computer Conditionals- A Psychological Evaluation", International Journal of Man-Machine Studies, 5, 105-113, 1977.

Soloway, E., Bonar, J. & Ehrlich, K. "Cognitive Strategies and Looping Constructs: An Empirical Study", Technical Paper, Yale University, 1981.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. "What do novices know about programming?", Technical Paper, Yale University, 1982.

Stake & Easley, Case Studies in Science Education, Center for Instructional Research and Curriculum Education, University of Illinois, 1978.

Tate, M., Stanier, B., Harootunian, B., "Differences Between Good and Poor Problem Solvers", School of Education, University of Pennsylvania, 1959.

Temple, M., Goldenberg, E.P., Lewis, P. & Horlick, R. "Logo: A Laboratory For Intellectual Investigation" in Microcomputers Go To School, (Leggett, S, Editor), Teach'em Press, 1984.

Turkle, S. The Second Self Simon & Schuster, 1984.

Winograd, T. "Beyond Programming Languages" Communications of the Association for Computing Machinery, 22, 391-401, 1979.

Wertine, R. "Students, Problems and 'Courage Spans'", in Cognitive Process Instruction: Research on Teaching Thinking Skills, (Lochhead, J. & Clement, J., Editors), Franklin Institute Press, 1979.

Whimby, A. & Lochhead, J., Problem Solving and Comprehension: A Short Course in Analytical Reasoning, Franklin Institute Press, 1981.

Youngs, E.A. "Human Errors in Programming", International Journal of Man-Machine Studies, 6, 361-376, 1974.

